Tercera semana de trabajo [14 - 18 de abril]

Milton Inostroza Aguilera

20 Abril de 2008

Abstract

En la reconstrucción del capturador de huella ha quedado claro en que lugar se deben registrar los distintos objetos [clases, metodos, funciones, variables locales, atributos].

Se ha interiorizado en el funcionamiento de threads y sockets en Python.

1 Desarrollo

La semana comenzó por escribir la nueva estructura del capturador de huella. Para esto se trabajó en las siguientes etapas:

- Momento en que una clase debe ser registrada.
- Momento en que un método debe ser registrado.
- Momento en que una función debe ser registrada.
- Momento en que una variable local debe ser registrada.
- Momento en que un atributo debe ser registrado.

1.1 Registro de clase

El mejor momento para registrar la clase y sus métodos asociados es cuando el programador termina de definirla en su programa. Dentro de nuestro capturador cuando se gatille un evento textitreturn preguntaremos en *locals* si es que la función *__init__* esta en este diccionario, si es así, estamos en la presencia de la definición de una clase. El siguiente ejemplo describe este caso:

```
class myclass:

    def __init__(self):
        self.a = 10
        x = 8
        return

    def mymethod(self, q):
        self.b = q
        return
```

Al momento de definir la clase el *return* nos interesa:

```
{
    '__module__':'__main__',
    'mymethod':<function mymethod>,
    '__init__':<function __init__>
}
```

Como se puede observar en locals están todas las funciones que representan los métodos de *myclass*.

1.2 Registro de método

Para registrar un método el mejor lugar para hacer esto es cuando nos encontramos con un evento del tipo *call*. Preguntamos por el atributo *self* en el diccionario *locals*, si es que este está debemos registrar el método.

Buscamos en nuestras clases registradas con anterioridad, utilizando el atributo *self* del diccionario *locals* y obtenemos la clase asociada a self de la siguiente manera:

```
type(self).__name__
```

Si la clase es encontrada se añade al registro de métodos utilizando el id global que este mismo método tiene en la clase que fue definida.

No he pensado en el caso que el objeto sea un método y su clase no sea encontrada, hasta el momento sólo se sale *__trace__* sin hacer nada.

1.3 Registro de función

Para registrar una función el mejor lugar para hacer esto es cuando nos encontramos con un evento del tipo *call*. Preguntamos por el atributo *self* en el diccionario *locals*, si es que este no está y la función *isfunction* nos entrega un valor TRUE, debemos registrar la función.

1.4 Registro de atributos

Para registrar los atributos de una clase se hace a través de la clase *Descriptor*. El programador en la definición de su clase debe explicitar una herencia con la clase *Descriptor*.

1.5 Registro de variables locales

Para registrar las variables locales se debe tener la siguiente consideración:

Cuando el programador pone explicitamente la instrucción *return* en su trozo de código, las variables locales sólo seran capturadas cuando se genere el evento *line*. En el caso que el programador no ponga la instrucción *return*, las variables locales seran capturadas cuando generen los eventos *line* y *return*.

Es importante señalar que por un asunto de optimización y precisión por cada evento se analiza el bytecode de la instrucción para ver si es que realmente corresponde a una modificación de una o varias variables locales.

1.6 Análisis de eventos generados

Antes de revisar los ejemplos de código es importante señalar que los mensajes están compuestos por [register] y los eventos estan compuestos por [set, call, return]. Después de esta aclaración se revisan los mensajes que el capturador de huellas envía para estos ejemplos:

```
def prueba():
    x = 10
    a = 0
    x = 15
    y = x = a = 1

prueba()
```

Genera los siguientes eventos/mensajes:

1. register prueba , id = 5 , args= {}
2. call prueba , id = 5 , args = ,llamado id = -1
3. register x = 0
4. set x = 10 , id= 0, ...
5. register a = 1
6. set a = 0 , id= 1 , ...
7. set x = 15 , id= 0 , ...
8. register y = 2
9. set y = 1 , id= 2 , ...
10. set x = 1 , id= 0 , ...
11. set a = 1 , id= 1 , ...
12. return

Es importante señalar que *register* indica el nombre de la variable local y su identificador local.

```
class clasePrueba(Descriptor):

    def __init__(self):
        x = 1
        self.w = 20
        self. h = 50
        x = self.w

    def impresion(self):
        print 'hola'

    def asignacion(self):
        j = 4
```

Genera los siguientes mensajes:

1. register clasePrueba , id = 1 , code = code...
2. register asignacion = 2
3. register impresion = 3
4. register __init__ = 4
5. return

Los mensajes *register* se generan cuando el programador termina de escribir su clase. Note que no es necesario que el programador cree una instancia de la clase.

```
class clasePrueba(Descriptor):

    def __init__(self):
        x = 1
        self.w = 20
        self. h = 50
        x = self.w

    def impresion(self):
        print 'hola'

    def asignacion(self):
        j = 4

objeto = clasePrueba()
objeto.asignacion()
```

Genera los siguientes eventos/mensajes:

1. register __init__ , id = 4 , idClass = 1 , args= 'self': 0
2. register self = 0
3. call __init__ , id = 4 , target = 1 , args = 'self': 0 ...
4. register x = 1
5. set x = 1 , id= 1 ...
6. register w = 6
7. set w = 20 id = 6
8. register h = 7
9. set h = 50 id = 7
10. set x = 20 , id= 1
11. return
12. register asignacion , id = 2 , idClass = 1 , args= 'self': 0
13. register self = 0
14. call asignacion , id = 2 , target = 1 , args = 'self': 0 ...
15. register j = 1
16. set j = 4 , id= 1 ...
17. return

Se han omitido algunos atributos de los eventos para facilitar la ya complicada lectura de los mismos.

1.7 Modificaciones

Al momento de pensar que teniamos nuestro capturador de eventos listo, surgieron diversas necesidades que son comentadas en esta sección.

Se debe identificar el objeto desde el cual se esta llamando el objeto actual. Esto es posible de hacer con la instrucción:

```
frame.f_back
```

Se debe implementar la profundidad que tiene el evento generado. Esto es posible de hacer agregando el atributo *__depthFrame__* a *frame.f_locals*.

Se debe implementar un time stamp por cada evento. Se utiliza la libreria time y la función time para generar un timestamp con precisión del orden de los nanosegundos.

Se debe marcar cada frame con su propio timestamp en el momento que sea requerido. Se agrega el atributo *__timeStampFrame__* a *frame.f_locals*.

Se implementa un identificador global de Probe.

Por cada generación de evento se imprime la estructura de Probe, que es la siguiente: (ProbeId, f_lasti,MethodId—FunctionId—LocalId—AttributeId).

Se debe agregar un dato más llamado ThreadId, el cual indica en que thread estamos. Para esto se investigó un poco la librería que implementa threads en Python y se logra que por cada thread que lanze el programador de forma transparente cada thread tome *__trace__*como su función para su propio *sys.settrace*.

References

[1] Module of Python: http://lfw.org/python/inspect.py

[2] Bytecode of Python: http://docs.python.org/lib/bytecodes.html