Clément Béra ~ Smalltalk, Tips 'n Tricks

Smalltalk method and block models

29 *Monday* *Jul 2013*

Posted by Clement Bera in Cog, Pharo

≈ Leave a comment

Hey guys,

I had a little fun recently with Athens and NativeBoost. But this time I'll write again on my the main topic: compiler and runtime environment (also known as Virtual machine in the case of most smalltalk system).

Today I'd like to share ideas and knowledge about method and block models for a Smalltalk system. Let's first look at Pharo:

A method is represented by two objects in Pharo :

CompiledMethod
A compiled Method is generated from the source code of the method by the compiler (when you press Cmd + s after editing a method). It represents a method ready to be executed by the VM. Let's look at its state.

- *compiledMethodHeader:* low level information encoded in a smallInteger for the runtime method activation. These informations consists in:
  - *primitive number:* set to zero by default, set to the number in the pragma `primitive` if the method's a primitive, set to a quick method index in case of quick method. The VM knows how to activate the method by looking at this value. 0 means normal method, other index means either quick method or primitive.
  - *literals number:* basically permits to the VM to know where the bytecode to execute starts in the method (it starts after the literals).
  - *frame size:* On activation, a method context will be created. Now these contexts add a huge overhead in the Pharo runtime. So the VM recycles them. To recycle them easily, the VM has two pools of contexts, large and small ones. On compilation, the compiler find out if the method context need a large frame or if a small frame will be enough. Then on activation a recycled context of the correct size will be used. For the Stack and Cog VM, however, the contexts are handled differently (they're map to stack frames).
  - *number of temporaries:* number of temps declared in the method. This is used to manage the stack of variable in the activation context.
  - *number of arguments:* number of arguments declared in the method. This is used to know

how to access the receiver and the arguments from the outer context and this is also used for the cpu `ret` instruction when the context is mapped to a stack frame.

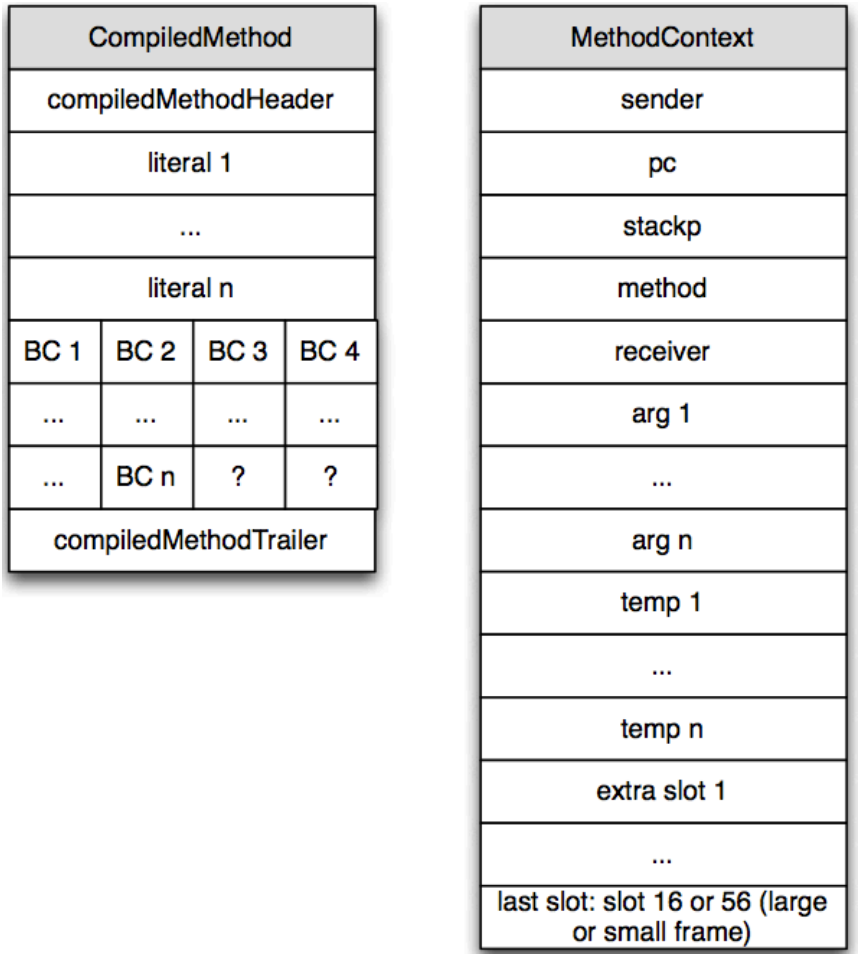
- *literals:* These consists in all the references from the method to movables objects. For example, a method can refer directly to literal such as a class or a symbol. However, these classes and symbols can be moved in the memory by the GC. Therefore, the literal array is used to keep track of the current address of the literal.
- *bytecode:* The byte code is a compact and easy to interpret representation of the high level source code of the method. The VM uses it to run the method's code.
- *compiledMethodTrailer:* SmallInteger that encodes the way to access the source code of the method. Is it in source file, in change file, not present at all ?

MethodContext

A method context is generated by the VM from the compiled method when it is activated. It represents the current execution state of the method. Let's look at its state.

- *method:* the compiled method (described before) that was activated to create this context.
- *receiver:* the object receiving the currently executed message, corresponding to `self` in the method source
- *stackp:* the current depth of the stack in the context. In the stack is put first a copy of the arguments, then slots for the temporaries, lastly extra slots for the stack. These extra slots are used for example when a message is sent, the sender context put the receiver and the argument on these slots.
- *sender:* the context that generating this context. It corresponds also to the context to return in case of a method return.
- *pc:* the program counter, which correspond to the currently executed bytecode index (so you can know what bytecode to execute next VM side)

Now let's look VM side to the representation of the objects.



Now contexts are currently map to stack frames, so we don't really need to discuss about it, because it is not relevant. Now CompiledMethod is a strange object. Why ?

The VM has to describe the format of an object to be able to access into their field. For example, when you do `myObject at: 3`, the VM has to access it differently if it is a ByteArray or an Array. For any object, there are different possible formats. There are 16 different ones in the CogVM, but some are not used we can consider there are 5 main ones:
– Fixed sized oop object (Object with only instance variables, for example, Point)
– Variable sized oop object (object with indexable fields, for example, Array)
– Weak oop object (object with weak indexable fields, for example, WeakArray)
– Word object (object with indexable fields that contains only 32 bits integers, for example, WordArray)
– Byte object (object with indexable fields that contains only 8 bits integers, for example, ByteArray)

But wait, to which object corresponds the compiled method. Well none. The VM now has a specific format for the compiled method because it is not a normal object. The alternative would be to externalize the bytecode to a byte array, which would result in having compiled method as normal objects. It is done like that in Visual Works. But there is a lot of compiled method in the system, and doing that will increase the size taken by each compiled method in the memory.

For example, in the current Pharo image there are 60 000 compiled methods. Putting the bytecode in an external byte array will add 12 bytes per method in the memory (8 for the new byte array header, 4 for the pointer towards it). 60 000 * 12 = 720 kb. The Pharo image is currently 15,3 Mb, so if we had compiled method as normal object, the image size would be around 16 Mb, which is 7% bigger.

Now a simpler VM could have compiled method as normal objects. Adding to that, when you look carefully, fixed-sized oop and variable-sized oop object behave the same. So in fact only 4 different formats are required for a Smalltalk VM :
– oop object
– weak oop object
– byte object
– word object

Another benefit of having CompiledMethod as normal objects in VisualWorks is that they can subclass it and add instance variables to them, that we cannot do in Pharo. So choosing to make an indirection over the bytecode is a trade between flexibility / simplicity and saving 720 kb.

Now let's look at the Block model of Pharo, implemented by the famous Eliot Miranda.

A block is represented by 3 objects in Pharo (on the contrary to the method that is represented by 2 objects).

- One of them is its enclosing compiled method, that holds inlined in its byte code the byte code of the block, and other information about the block closure instantiation, such as the block first byte code index and its number of arguments.
- Another one is the Block Context. In Pharo a Block context is an instance of MethodContext. However, there's an instance variable in method context, named `closureOrNil`, that is always nil in case of a method context (so I didn't talk about it) but contains the BlockClosure in case of a block activation.
- Lastly, the block is represented by a BlockClosure instance, which encapsulate certain behaviors that does not exist in a method, such as how to access enclosing context variables or where to return in case of a non local return.

Now let's look at when each object is created. Let's say that I write the method:

```
MyClass>>myMethod
| temp |
temp := [ 1 + 2 ].
temp value
```

When I will press Cmd+s in the class browser, the compiler will instantiate the compiledMethod holding in its byte code the byte code of the block (of 1 + 2).

No I evaluate:

```
MyClass new myMethod
```

The first statement of the method will be evaluated: `temp := [ 1 + 2 ].`. This statement creates the BlockClosure instance, which holds the first index of its byte code in the byte code of the method, and reference to its defining context to be able to handle non local return and access to outer variables.

Then the second statement of this method will be evaluated: `temp value`. This statement creates a block Context instance, which is similar to a method context except that it holds a reference to the blockClosure created previously in its variable `closureOrNil`.

Now let's look at the visual works block model. Without going into details, let's show their main difference:

- the compiled method byte code does not include the blocks byte code, the block byte code is stored in a compiled block, which is a different object from the compiled method. Then instead of having a reference to the index of the first byte code of the block in its method, the block closure holds a reference to its compiledBlock
- `BlockContext` and `MethodContext` are not the same objects. Adding to it, the variable `closureOrNil` does not exist at all. In Pharo, a block context holds a reference to 'self' in the receiver field and to the blockClosure in the closureOrNil field. In VW, they consider that [1 + 2] value means the block closure is the receiver of the message value, so the receiver of a block context is always a block closure, removing 1 field in the context (and also in the stack frame associated).
- VW has an optimization that we don't have in Pharo. I didn't talk about the optimizations we have in Pharo but we have some too. This optimization we don't have is named 'clean block', which means that a block that does not refer outer variables and does not use non local return is way faster (probably handled like a method).

Hope you enjoyed this post.