Ruby master - Feature #6688

Object#replace

07/03/2012 05:05 PM - prijutme4ty (Ilya Vorontsov)

Status:	Rejected	
Priority:	Normal	
Assignee:	matz (Yukihiro Matsumoto)	
Target version:	3.0	

Description

I suggest that #replace works not only on Enumerables but on any Object. It can make use the same object in different places more consistent. It makes it possible to write
class Egg; end
class Hen; end
class HenHouse; attr_accessor :species; end
class Incubator; def incubate(egg) Hen.new; end

Here it is!

class IncubatorWithReplace;
def incubate(egg)
egg.replace(Hen.new)
end
end

e1,e2,e3 = Egg.new, Egg.new, Egg.new
h1, h2 = HenHouse.new, HenHouse.new

One egg is shared between hen houses

h1.species = [e1, e2]
h2.species = [e1, e3]
p h1 # ==> ,#]
p h2 # ==> ,#]

# First option. It's bad choise because it makes two "data structures" HenHouse inconsistent:
# they have different object while must have the same
h1[0] = Incubator.new.incubate(h1[0])
p h1 # ==> ,#]
p h2 # ==> ,#]

# Second option is ok - now both shared objects're changed.
IncubatorWithReplace.new.incubate(h1[0])
h1 # ==> ,#]
h2 # ==> ,#]

# Third option is bad - it wouldn't affect HenHouses at all
e1 = Incubator.new.incubate(e1)
p h1 # ==> ,#]
p h2 # ==> ,#]

while Fourth option is ok and works as second do
IncubatorWithReplace.new.incubate(e1) ## would affect both HenHouses
p h1 # ==> ,#]
p h2 # ==> ,#]

I can't imagine how it'd be realized, it looks like some dark magic with ObjectSpace needed to replace one object at a reference with another at the same reference. But I didn't found a solution.

About ret-value. I think it should be two forms:
Object#replace(obj, retain = false)
If retain is false #replace should return a reference to a new object (in fact the same reference as to old object but with other content)

If retain is true, old object should be moved at another place and new reference to it is returned, so:
e1 # ==>
e1.replace(Hen.new, true) # ==>
e1 # ==>

History

#1 - 07/03/2012 06:02 PM - shugo (Shugo Maeda)

prijutme4ty (Ilya Vorontsov) wrote:

> I suggest that #replace works not only on Enumerables but on any Object. It can make use the same object in different places more consistent.

Smalltalk has such a method called "become:", which swaps the receiver and the argument.
For example:

```
| x y z |

x := 'foo'.
y := #(1 2 3).
z := y.
x become: y.

^ {x. y. z}  "=> #(#(1 2 3) 'foo' 'foo')"
```

But I'm against your proposal. become: is too dangerous because it breaks object identity.

#2 - 07/03/2012 11:04 PM - trans (Thomas Sawyer)

=begin
I don't think #replace works on Enumerables, rather it works on Array and Hash.

It does seem a rather powerful notion to be able to replace an object with another wherever is may be referenced. Though @shudo may be right that it is too dangerous. With good design, a proper interface can handle the reassignment through the usual mechanisms, although admittedly it may entail many more compute cycles to do it.

On the other hand, if Smalltalk supports it then perhaps it's worth consideration. Always respect the Smalltalk :) But I agree with it, better name is #become.

Also note that a less dangerous notion of a generic replace is simply to copy common instance variables.

```
class X
def initialize(a)
@a = a
end
end

class Y
def initialize(a,b)
@a, @b = a, b
end
end

x = X.new(1)
y = Y.new(2,3)

x.a  #=> 1
x.replace(y)
x.a  #=> 2
```

Although a more appropriate name for this is probably #instance_replace. This preserves object identity, but can only be used to "copy" an object of the same type --which, if you think about it, is what Array#replace and Hash#replace does too actually.
=end

#3 - 07/12/2012 07:38 AM - prijutme4ty (Ilya Vorontsov)

shugo (Shugo Maeda) wrote:

> prijutme4ty (Ilya Vorontsov) wrote:
>
> > I suggest that #replace works not only on Enumerables but on any Object. It can make use the same object in different places more consistent.

Smalltalk has such a method called "become:", which swaps the receiver and the argument.
For example:

```
| x y z |

x := 'foo'.
y := #(1 2 3).
z := y.
x become: y.

^ {x. y. z}  "=> #(#(1 2 3) 'foo' 'foo')"
```

But I'm against your proposal. become: is too dangerous because it breaks object identity.

#become is a good name
I can't understand your fears. Monkey patching is also too dangerous. I just propose that one could use all advantages of duck typing.
Imagine that you want to make method hash_with_indifferent_access! which not just create a new object but is also substituted instead of an old object. All structures which use old object will now unobtrusively use new object.
I can't understand why is this more dangerous than e.g. def somefunc(arr); arr.shift; end
When you give an array object that can be changed in method, you know what are you doing. If you don't want to change initial array - you just use somefunc(myarray.dup). Analogical you can write x.dup.became(y.dup) Certainly it makes nothing just like myarray.dup.shift or str.dup.upcase!

#4 - 07/12/2012 08:03 AM - prijutme4ty (Ilya Vorontsov)

trans (Thomas Sawyer) wrote:

=begin
I don't think #replace works on Enumerables, rather it works on Array and Hash.

It does seem a rather powerful notion to be able to replace an object with another wherever is may be referenced. Though @shudo may be right that it is too dangerous. With good design, a proper interface can handle the reassignment through the usual mechanisms, although admittedly it may entail many more compute cycles to do it.

On the other hand, if Smalltalk supports it then perhaps it's worth consideration. Always respect the Smalltalk :) But I agree with it, better name is #become.

Also note that a less dangerous notion of a generic replace is simply to copy common instance variables.

```
class X
def initialize(a)
@a = a
end
end

class Y
def initialize(a,b)
@a, @b = a, b
end
end

x = X.new(1)
y = Y.new(2,3)

x.a  #=> 1
x.replace(y)
x.a  #=> 2
```

Although a more appropriate name for this is probably #instance_replace. This preserves object identity, but can only be used to "copy" an object of the same type --which, if you think about it, is what Array#replace and Hash#replace does too actually.
=end

I've answered Shugo about dangers. I suppose that it's not more dangerous than bang-methods.

Also when one changes constants for stubbing a class it's almost the same dangerous but particulary useful.

Try to write Hash#with_indifferent_access! that gives an object ability to understand both symbols and strings without defining singleton methods (because you can't dump them).

What for me, good design is a design that gives me ability to write flexible and concise code. And nothing more. Implicit typing and duck-typing obliges us to write tests for all. But result worth spent efforts. You can use tests you've already wrote to test that everything works. And you don't need to write and test complex data structres in order to support multiple structres consistency (here you can make much more bugs).

#5 - 07/12/2012 08:24 AM - prijutme4ty (Ilya Vorontsov)

trans (Thomas Sawyer) wrote:

> =begin
> I don't think #replace works on Enumerables, rather it works on Array and Hash.
>
> It does seem a rather powerful notion to be able to replace an object with another wherever is may be referenced. Though @shudo may be right that it is too dangerous. With good design, a proper interface can handle the reassignment through the usual mechanisms, although admittedly it may entail many more compute cycles to do it.
>
> On the other hand, if Smalltalk supports it then perhaps it's worth consideration. Always respect the Smalltalk :) But I agree with it, better name is #become.
>
> Also note that a less dangerous notion of a generic replace is simply to copy common instance variables.
>
> class X
> def initialize(a)
> @a = a
> end
> end
>
> class Y
> def initialize(a,b)
> @a, @b = a, b
> end
> end
>
> x = X.new(1)
> y = Y.new(2,3)
>
> x.a #=> 1
> x.replace(y)
> x.a #=> 2
>
> Although a more appropriate name for this is probably #instance_replace. This preserves object identity, but can only be used to "copy" an object of the same type --which, if you think about it, is what Array#replace and Hash#replace does too actually.
> =end

Object copying is useful but far less powerful. Imagine that you want to redefine files ARGF is linked to. Instead of
ARGV = ['filename','filename2']; puts ARGF
you should write
ARGV.replace ['filename','filename2']; puts ARGF

But now imagine that you want to use lazy enumerator to print content of all files in a directory. You don't want to create an array of all files, so you just try to write
ARGV.replace Dir.to_enum(:glob,'**/*.txt'); puts ARGF
...and fails because you can't use enumerable here, only an array. Why we should put up with it if it can be improved not to break duck-typing paradigm?

#6 - 07/12/2012 09:54 AM - kstephens (Kurt Stephens)

How would it behave for immediate VALUEs (Fixnum, Symbol, etc.)?
Or value classes that are immutable or often occur as literals (Float, Rational, Time, etc.)?

This would probably be very difficult to implement in JRuby (and other Ruby implementations) without adding indirection handles between object references and their memory locations; and would reduce performance.

Without indirection, it's difficult to remap references on the stack and other structures that may be opaque to the CRuby GC.

If I recall correctly, early versions of Objective-C (pre-NeXTSTEP) used indirection handles to reduce memory fragmentation, thus it may have been supported there.

This "feature" would constrain low-level memory management decisions in a fundamental way.

#7 - 07/13/2012 07:03 AM - prijutme4ty (Ilya Vorontsov)

I suppose it'd work in such a way:
x.became(1) makes x equal to 1 and returns x as a new object
1.became(x) raises an exception

Unfortunately I don't clearly realise, what ruby interpreter makes under hood, so I really appreciate any corrections in my reasoning:

Here I suppose that if ruby yields all values by references it already have such level of indirection. Am I right that when I call f(y), y internally doesn't contain value, but a pointer to a value and that pointer can be derived from object_id. If so - there is a place where data live and we can rearrange

pointers to data

Imagine code with two variables (object_id values are symbolical):

y.object_id == 1

```
def f(y)
# x.object_id == 2; ++x.ref_conter;
x = X.new

# y.object_id still == 1, x.object_id still == 2
# internal data for x and y swaps.
#ref_counters for particular object_ids(!) didn't changed. This action does nothing GC should worry about if it indexes objects by their object_id.
y.become x

#here we return nil so x.ref_counter decreases and object at x.object_id can now be garbage collected. It's internal data is data that was in y before became call
nil
end
```

Where I am wrong if anywhere?

#8 - 07/13/2012 10:50 AM - nobu (Nobuyoshi Nakada)

```
=begin
Imagine what would happen with:

String.become("")
=end
```

#9 - 07/13/2012 03:14 PM - prijutme4ty (Ilya Vorontsov)

nobu (Nobuyoshi Nakada) wrote:

> =begin
> Imagine what would happen with:
>
> String.become("")
> =end

Something very similar to String = "" (legal code now, but interpreter crashes in a pair of actions after that). There're such many possibilities to shot own leg in ruby, but I hardly can guess who makes a shot in such obviously stupid way.

#10 - 07/13/2012 06:18 PM - alexeymuranov (Alexey Muranov)

=begin
I like that in Ruby (({1})) is a unique object, i will not appreciate a feature (({1.become(2)})).

However i was wondering myslef about an ability to replace not ((*objects*)), but contents of ((*variables*)) (so it would be a syntactic change), and an ability to pass variables by reference. Ilya, maybe this is what you want?

Something like:

```
n = 1
n <- &:to_s   # invented syntax, weird alternative to n = n.to_s

replace_1_with_2 = proc do |(v)|  # invented syntax, denotes a variable passed by reference
v = 2 if v = 1
end

v = 1
replace_1_with_2.call(v)
v   # => 2

def replace_2_with_3(|u|)   # invented syntax :)
u = 3 if u = 2
end

replace_2_with_3(v)
v  # => 3
```

((*Update*)). How would the addition of integers be implemented in Ruby to take into account all the previous calls to (({3.become(5)})), etc.? Taking the quotient by the ideal generated by (({2 = 5 - 3}))? ((({3.become(5)})) should be equivalent to (({2.become(0)})), right?)

=end

#11 - 07/13/2012 09:38 PM - alexeymuranov (Alexey Muranov)

=begin
After some thinking, this can be realized with a new class :) :

```
class Reference
attr_reader :true_identity

def initialize(object)
@true_identity = true_identity_of(object)
end

def replace_with(object)
@true_identity = true_identity_of(object)
end

private
def true_identity_of(object)
object.is_a?(self.class) ? object.true_identity : object
end

end

a = Reference.new(1)
a.true_identity # => 1
a.replace_with(2)
a.true_identity # => 2
b = Reference.new(3)
a.replace_with(b)
a.true_identity # => 3
```
=end

#12 - 07/18/2012 08:15 AM - prijutme4ty (Ilya Vorontsov)

1.become(2) should certainly raise an exception as I meant. But I caught what the problem have you probably meant:
a=1
a.become(2)
I don't know if there're simple workarounds to understand how to behave with constant classes such as Numeric and Symbols. It really can be a great problem in realizing such a feature.

#13 - 08/05/2012 07:50 PM - alexeymuranov (Alexey Muranov)

=begin
Ilya, according to how Ruby works (as far as i understand), there should be no difference, as far as exceptions concerned, between

```
1.become(2)
```

and

```
a = 1
a.become(2)
```

so this should be forbidden, i think.

However, i was thinking recently about other problems associated with OO programming, and in my opinion there should be a clear distinction between mutable and immutable objects, maybe even the latter in addition to a class should also have a "type" (in a language like Ruby).

To me, (({Object#become!!})) would make sense for mutable objects, which have a ((*state*)) not determined by their ((*identity*)). I can imagine that it might require changes in the current implementation, to allow all properties of every mutable object be replaced. Then i can imagine, for example, type conversion for mutable objects done like this:

```
class Object
def to!!(new_class)
become!!(new_class[self]) # raises an exception if self is immutable
end
def class=(new_class)
to!!(new_class)
new_class
end
end

a = "abc"
a.to!!(Array)
```

a # => ["a", "b", "c"]

Of course calling (({#become!!})) on objects that are already referenced by other objects can break many things, (({like with String.become!!("")})).

I think that every time an attribute writer is called on a mutable object, it is like executing a partial form of (({#become!!})).

Update: In fact, i do not know if strings are sufficiently mutable in Ruby, or if only their contents is. What i mean by mutable objects is something like instances of (({Struct})) classes.

Maybe, to encourage only good OO practices, a mutable object should only be allowed to become an object of a superclass or of a subclass of the object's class. Maybe with this restriction (({become})) can even encourage the good practice of constructing class inheritance hierarchies which are "(({become}))-safe" :).
=end

#14 - 10/28/2012 12:17 AM - yhara (Yutaka HARA)

- Category set to core

- Target version set to 2.6

#15 - 10/30/2012 03:29 AM - nobu (Nobuyoshi Nakada)

- Target version changed from 2.6 to 3.0

#16 - 11/18/2012 12:27 PM - Anonymous

I support Shugo Maeda, who is against this method. Non-essential feature creep.

#17 - 11/25/2012 02:33 AM - headius (Charles Nutter)

This is an awful, awful idea. Objects should never change their basic type in-place, or the very essence of OO is destroyed. Matz has also said in the past he did not like #become and would not add it to Ruby.

It is bad enough that we currently have methods like IO#reopen that change the class of the object. This method is nearly impossible to support in JRuby, because once an object has been constructed we can no longer change its essential type.

No no, a thousand times no to #become in Ruby.

#18 - 11/25/2012 09:32 PM - charliesome (Charlie Somerville)

This is a very dangerous proposal and has the potential to cause all sorts of crashes and other nastiness.

Consider this case:

```
o = {}
m = o.method :[]=
o.replace []
m.call("foo", "bar")
```

And this case:

```
a = Class.new
b = Class.new a
a.replace b
b.send :boom
```

What should happen here?

I'm of the opinion that it should *never* be possible to segfault Ruby from within Ruby, but this proposal makes it possible to violate all sorts of invariants and expectations within MRI.

#19 - 02/21/2013 03:00 AM - trans (Thomas Sawyer)

=begin
I have idea for less dangerous form of become.

I am working with a parser that supports object references. It is difficult b/c children of a given object might reference the parent, creating circular structures. So you can't just parse the children without first adding the parent to an alias table in case a child references it, but you also can't create the parent without the children. So the first thought is to allocate the parent, put the allocation in the alias table and then comeback and add the children tp the parent afterwards. Unfortunately not all classes support allocate, nor do I like the idea of depending on the mutability of the allocated object. So I thought maybe using a Proxy for the object while the children are parsed would work, and it almost does, except now I must replace all the proxies with the original object. But guess what? While I have a reference to the proxy, there is no one way to traverse the all children of an arbitrary object. I can't say where the proxy ended up.

So that problem led me the idea. Instead of any object supporting #become, only an instance of a Proxy class could do so. e.g.

```
p = Proxy.new
a = p
p.become(1)
a #=> 1
```

Allowing only this one type of object to do this would limit the dangers to controlled usage, while providing a very powerful technique when tricky situations like the one I described above arise. I guess this Proxy class is like the concept of a Future, but one that's supported more deeply in the language via #become.
=end

#20 - 02/21/2013 11:35 AM - trans (Thomas Sawyer)

=begin
Even if this idea of Proxy/Proxy#become isn't deemed suitable for core, is it still possible to do it as a 3rd party extension? I took a stab at it with:

```
#include <ruby.h>

VALUE cProxy;

/* call-seq:
 *   proxy.become(newobj)
 */
static VALUE proxy_become(VALUE self, VALUE newobj) {
  VALUE name = rb_mod_name(rb_obj_class(newobj));
  char * c_name = StringValueCStr(name);
  struct RObject *robj1 = ROBJECT(self);
  struct RObject *robj2 = ROBJECT(newobj);

  rb_secure(1);

  if(OBJ_FROZEN(self) || OBJ_FROZEN(newobj)) {
    rb_error_frozen("object");
  }

  switch(TYPE(newobj)) {
    case T_NIL:
    case T_FALSE:
    case T_TRUE:
    case T_SYMBOL:
    case T_FIXNUM:
      /* rb_raise(rb_eTypeError, "cannot become %s", c_name); */
      break;
    default:
      *robj1  = *robj2;
  }

  return newobj;
}

void Init_proxy(void)
{
  cProxy = rb_define_class("Proxy", rb_cObject);
  rb_define_method(cProxy, "become", proxy_become, 1);
}
```

This ((*almost*)) works. But sometimes I get a core dump --something about (({glibc detected ruby: double free or corruption (fasttop)})).

Is it possible to fix this? Or is this just something outside the realm of possibility?
=end

#21 - 02/22/2013 09:05 AM - ko1 (Koichi Sasada)

- Assignee set to matz (Yukihiro Matsumoto)

#22 - 02/22/2013 12:11 PM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

I think Smalltalk experience has proven that Object#replace is a bad bad idea.

Matz.

#23 - 11/20/2015 06:03 PM - nobu (Nobuyoshi Nakada)

- Related to Feature #6721: Object#yield_self added

#24 - 11/20/2015 06:04 PM - nobu (Nobuyoshi Nakada)

- Related to deleted (Feature #6721: Object#yield_self)