Ruby master - Feature #4264

General type coercion protocol for Ruby

01/12/2011 12:38 AM - headius (Charles Nutter)

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

Description

=begin
Justification: Ruby objects variously define to_ary, to_int and others for either explicit (before call) or implicit (during call) coercion. However, the set of methods is limited, and adding more names just clutters namespaces and isn't feasible long-term anyway. This proposal will hopefully start a discussion toward adding a general type-coercion protocol via a #to or #to_any method. To blunt the naming discussion a bit, I will refer to it as #to_x.

Description: The #to_x method will be a "supermethod" of sorts that can be used to coerce a given object to an arbitrary type. Where currently there are specific methods for coercing to specific types (to_ary, to_str), and other more general methods intended not for coercion but for explicitly re-structuring an object's data (to_a, to_s), there's no one protocol for doing general coercion. #to_x would fill the roles of the coercion methods, accepting a target class and responding appropriately.

The response will depend on whether the target object can be coerced to the given type. The result for success should obviously be an instance of the target type. The result for failure could either be "soft": returning nil, or "hard": raising an error. There could also be an optional boolean flag that specifies hard or soft.

Existing coercion methods could (but need not be) implemented in terms of #to_x

def to_ary
to_x(Array)
end

def to_str
to_x(String)
end

Prior art: JRuby supports coercing Ruby objects to arbitrary Java types in this way. Currently only a set of hard-coded target types are supported for various core Ruby classes, but this is intended to eventually be part of the invocation protocol when calling Java. In other words, if the object being passed is not the exact type of the target parameter, JRuby will invoke to_java(target_param_type) to do the coercion. Performance implications in this are obvious...so there may need to be discussions about modifying this protocol to make it easier to optimize.
=end

History

#1 - 01/12/2011 01:07 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin

> The result for failure could either be "soft": returning nil, or "hard": raising an error.

You mean #to_x and #to_x! ? By the way, I prefer obj.to(Array) instead of obj.to_any(Array)

Great idea! +1
=end

#2 - 01/12/2011 01:26 AM - headius (Charles Nutter)

=begin
I like #to better as well, but isn't it too generic a name, sure to conflict with libraries out there?

Groovy uses #as

obj.as String

...though that's just as generic.
=end

#3 - 01/12/2011 01:39 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin
Although I don't like Groovy very much (but still use it at a daily basis as part of my current job - at least it is not as bad as Grails - ARGH!), I think 'as' is also a good name and maybe even better than 'to'.
=end

#4 - 01/12/2011 01:46 AM - rkh (Konstantin Haase)

=begin
Both #as (Parslet, Sequel, MetaWhere, hacketyhack, Facets, Ruport, CouchPotato, ...) and #to (RSpec, Ramaze, ActiveSupport) are use in DSLs. However, I really prefer #to.

Konstantin

On Jan 11, 2011, at 17:39 , Rodrigo Rosenfeld Rosas wrote:

> Issue #4264 has been updated by Rodrigo Rosenfeld Rosas.
>
> **Although I don't like Groovy very much (but still use it at a daily basis as part of my current job - at least it is not as bad as Grails - ARGH!), I think 'as' is also a good name and maybe even better than 'to'.**
>
> http://redmine.ruby-lang.org/issues/show/4264
>
> ---
>
> http://redmine.ruby-lang.org

=end

#5 - 01/12/2011 02:09 AM - headius (Charles Nutter)

=begin
Other thoughts after talking with tenderlove:

1. double-dispatched protocol

The #to protocol could double-dispatch against the class, as in:

```
def to(type)
type.coerce(self)
end
```

This would allow defining coercions on the target class, rather than on the #to method. It does incur the hit of doing a double-dispatch though.

1. symbolic types

Allowing a symbol could make it possible to define coercions based on name alone, such as obj.to :yaml or obj.to :json. A default impl of coerce could be added to Symbol:

```
class Symbol
def coerce(obj)
obj.send "to_#{self}"
end
end
```

Given the double-dipatched protocol, the above may be left for users to implement, but it serves a similar purpose to the now-standard Symbol#to_proc.
=end

#6 - 01/12/2011 02:11 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin
Whatever what name is chosen, shouldn't we also define another method
like Object#from so that Object#to would be implemented as below?

```
def to(klass)
klass.from(self)
end
```

Rodrigo

Em 11-01-2011 14:46, Haase, Konstantin escreveu:

> Both #as (Parslet, Sequel, MetaWhere, hacketyhack, Facets, Ruport, CouchPotato, ...) and #to (RSpec, Ramaze, ActiveSupport) are use in DSLs.
> However, I really prefer #to.
>
> Konstantin
>
> On Jan 11, 2011, at 17:39 , Rodrigo Rosenfeld Rosas wrote:

=end

#7 - 01/12/2011 02:11 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin
Sorry, I didn't see this message when I posted the last one.

Em 11-01-2011 15:09, Charles Nutter escreveu:

> Issue #4264 has been updated by Charles Nutter.
>
> Other thoughts after talking with tenderlove:
>
> 1. double-dispatched protocol
>
> The #to protocol could double-dispatch against the class, as in:
>
> def to(type)
> type.coerce(self)
> end
>
> This would allow defining coercions on the target class, rather than on the #to method. It does incur the hit of doing a double-dispatch though.
>
> 1. symbolic types
>
> Allowing a symbol could make it possible to define coercions based on name alone, such as obj.to :yaml or obj.to :json. A default impl of coerce could be added to Symbol:
>
> class Symbol
> def coerce(obj)
> obj.send "to_#{self}"
> end
> end
>
> Given the double-dipatched protocol, the above may be left for users to implement, but it serves a similar purpose to the now-standard Symbol#to_proc.

=end

#8 - 01/12/2011 02:14 AM - headius (Charles Nutter)

=begin
On Tue, Jan 11, 2011 at 10:27 AM, Jim Weirich jim.weirich@gmail.com wrote:

> (1) This is a extending the concept of to_ary and to_str to arbitrary
> classes, (as opposed to extending the concepts of to_a and to_s). Correct?

Correct.

> (1a) Do we need a general solution for the to_a/to_s situation too?

Maybe? The big problem with to_ary/to_str versus to_a/to_s is that there's no clear understanding of the differences. Up through 1.8.7, to_a was defined on all objects, with the default behavior just returning a one-element array of that object. to_s is still defined on all objects. On the contrary, to_ary and to_str are specifically implemented only on objects that have a "natural" conversion to those types.

I guess I see the difference being that to_ary/to_str are like slightly softer type-casting operations and to_a/to_s are hard, unconditional conversions. What would the equivalent be for "to"?

(2) Will there be a way of specifying the conversion in both directions?
For example, if I add my own container type (JimContainer), will I be able
to specify the coercions Array->JimContainer and JimContainer->Array?

I added to the bug a proposal for double-dispatch; I think that would
address it neatly:

class JimContainer
def to(type)
case type
when Array
(your array conversion logic)
...
end

and Array would simply dispatch to JimContainer.coerce(ary).

- Charlie

=end

#9 - 01/12/2011 02:30 AM - headius (Charles Nutter)

=begin
FWIW, JRuby's to_java does support real types and symbolic types, as in the following equivalent calls:

'foo'.to_java # defaults to java.lang.Object, so we produce a java.lang.String
'foo'.to_java :string # known "core" type name, so we know to do java.lang.String
'foo'.to_java java.lang.String # specific type
=end

#10 - 01/12/2011 02:35 AM - headius (Charles Nutter)

=begin
Tom Enebo proposed (in an offhand way) an additional #to? that simply returns whether the coercion would succeed. So the following code:

obj.to_str if obj.respond_to? :to_str

Would become

obj.to(String) if obj.to?(String)

Some backstory on why I like this idea: In JRuby we often have to choose from multiple method overloads with different types of parameters. Having a to? protocol would be necessary for us to allow users to define their own java coercions, since without to? we won't know which types they support (without potentially forcing them to coerce, an expensive operation before we've decided that we have found the right overload to invoke).
=end

#11 - 01/12/2011 03:11 AM - headius (Charles Nutter)

=begin
A question was raised about whether #to should understand context, as in knowing a top-level object being coerced to YAML should include the --- header. The answer seems to come from Marshal.

Marshal.dump(object) knows how to produce the marshal header, linking, and so on. Marshal is the master of this format. But Marshal defers to the objects themselves for actual *content* to go into that marshaled output, calling _dump or marshal_dump on each object in turn. With the #to protocol, Marshal.dump could be implemented as:

def Marshal.dump(obj)
emit_header
obj.to(Marshal)
end

In the same way that marshal_dump is used today.

Another concern raised is whether #to should be expected on all objects all the time, rather than just always calling MyClass.coerce(obj). The answer is simple: you want individual source types to control how they coerce to a target type, rather than expecting the target type to know about all possible sources. The default protocol where #to calls type.coerce would simply be a default behavior.
=end

#12 - 01/12/2011 05:00 AM - headius (Charles Nutter)

=begin
On Tue, Jan 11, 2011 at 12:23 PM, Jim Weirich jim.weirich@gmail.com wrote:

> On Jan 11, 2011, at 1:17 PM, James Edward Gray II wrote:
>
> > I understood it as to_a/to_s are for conversions. They basically mean change this object into an Array or a Sring, however it makes sense to do that for the current object.
> >
> > In contrast, to_ary/to_str are for objects that essentially are enhanced versions of Array or String. They mean, it's OK to pretend this object is an Array or a String.
> >
> > Perhaps I'm wrong though.
>
> That's my understanding as well.

The logic is inconsistently applied in Ruby right now. Sometimes to_str means treat me as a string without any additional calls, sometimes it's called blindly, sometimes it's checked and called...and sometimes to_s is used instead. However to_str and to_ary are used much more heavily during call protocols (usually by the receiver) than to_a and to_s. The latter two are more heavily used for user-driven conversions.

Whether what's described above is correct or not (or desirable), I think the current real-world uses of to_ary/to_str fit my description better.

The lack of a formal statement on coercion/conversion protocols is probably responsible for this confusion in the first place.

- Charlie

=end

#13 - 01/12/2011 06:05 AM - kstephens (Kurt Stephens)

=begin
Below is the simple, extensible technique used in Oaklisp.
Give each Module a coercer Symbol.
Apply the target type's coercer to the object to be coercer.
Cost is two message sends.

```
class Module
IDENTITY_PROC = lambda { | obj | obj }
def coercer
unless @coercer
@coercer = :"#{self.name}_coercer"
define_method(@coercer, &IDENTITY_PROC)
end
@coercer
end
end

class Object
def as(mod)
send(mod.coercer, self)
end
end

########################################

class String
define_method(Float.coercer) { | obj | obj.to_f }
end

class Float
define_method(String.coercer) { | obj | obj.to_s }
end

########################################
require 'pp'

x = 1.23
pp [ x, String.coercer, x.as(String) ]
```

pp [x, Float.coercer, x.as(Float)]

x = "2.34"
pp [x, String.coercer, x.as(String)]
pp [x, Float.coercer, x.as(Float)]

##########################

[1.23, :String_coercer, "1.23"]
[1.23, :Float_coercer, 1.23]
["2.34", :String_coercer, "2.34"]
["2.34", :Float_coercer, 2.34]

=end

#14 - 01/12/2011 08:35 AM - sorah (Sorah Fukumori)

=begin
On Wed, Jan 12, 2011 at 2:10 AM, Rodrigo Rosenfeld Rosas
rr.rosas@gmail.com wrote:

> Whatever what name is chosen, shouldn't we also define another method like
> Object#from so that Object#to would be implemented as below?
>
> def to(klass)
> klass.from(self)
> end
>
> Rodrigo
>
> Em 11-01-2011 14:46, Haase, Konstantin escreveu:
>
>> Both #as (Parslet, Sequel, MetaWhere, hacketyhack, Facets, Ruport,
>> CouchPotato, ...) and #to (RSpec, Ramaze, ActiveSupport) are use in DSLs.
>> However, I really prefer #to.
>>
>> Konstantin
>>
>> On Jan 11, 2011, at 17:39 , Rodrigo Rosenfeld Rosas wrote:

+1. Good Idea.

- Object#to(klass, *args)
- Class.from(obj, *args) ##

"10".to(Integer) #=> 10
Integer.from("10") #=> 10

10.to(String, 8) #=> "12"
String.from(10, 8) #=> "12"

##
--
Shota Fukumori a.k.a. @sora_h - http://codnote.net/

=end

#15 - 01/12/2011 10:19 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:34366] Re: [Ruby 1.9-Feature#4264][Open] General type coercion protocol for Ruby"
on Wed, 12 Jan 2011 02:14:24 +0900, Charles Oliver Nutter headius@headius.com writes:

|> (1a) Do we need a general solution for the to_a/to_s situation too?
|
|Maybe? The big problem with to_ary/to_str versus to_a/to_s is that
|there's no clear understanding of the differences.

I think I have explained before but I couldn't find the reference, I
will restate here.

- to_s, to_i, to_f, to_a etc. are conversion method. For example, if a object can be converted into a string, it would have to_s method, but it doesn't have to behave like a string.

- to_str, to_int, to_ary are implicit conversion method. Some objects mimic built-in objects do not have same internal structures, but C defined methods requires the structure, so to_str and the likes work as hooks to retrieve C structure like RString. The object that provide to_str should behave as a string.

- String() and Array() are conversion methods, so (in case of String) it first try to_str to check if they are string, then try to_s to get string object.

If there's any usage of to_x methods that does not fit above principles, I consider it as a bug.

I think we are discussing general conversion way to replace to_s, etc. here, not to_str and alike.

```
                          matz.
```

=end

#16 - 01/12/2011 02:10 PM - headius (Charles Nutter)

=begin
On Tue, Jan 11, 2011 at 7:18 PM, Yukihiro Matsumoto matz@ruby-lang.org wrote:

> * to_s, to_i, to_f, to_a etc. are conversion method. For example, if
> a object can be converted into a string, it would have to_s method,
> but it doesn't have to behave like a string.
>
> * to_str, to_int, to_ary are implicit conversion method. Some
> objects mimic built-in objects do not have same internal
> structures, but C defined methods requires the structure, so to_str
> and the likes work as hooks to retrieve C structure like RString.
> The object that provide to_str should behave as a string.

I guess I'm confused what the difference is. For example, sometimes to_ary is used as a marker, and not invoked. Sometimes it is invoked and expected to return an array. There are many cases in RubySpec showing both behaviors. So it seems to be a more narrow conversion method, but it still seems like a conversion method to me.

> * String() and Array() are conversion methods, so (in case of String)
> it first try to_str to check if they are string, then try to_s to
> get string object.
>
> If there's any usage of to_x methods that does not fit above
> principles, I consider it as a bug.
>
> I think we are discussing general conversion way to replace to_s,
> etc. here, not to_str and alike.

Based on your above description, I agree.

Can you provide more clarification:

- Are to_ary/to_str/to_int intended only for use from core methods?
- Are they expected to always return the an instance of the appropriate type?

- Should users be defining these methods to produce new structures in the same way as they define to_a, or should they only be doing so if their class can duck-type as an array/string/integer for all array/string/integer methods?

- Charlie

=end

#17 - 01/12/2011 07:05 PM - manveru (Michael Fellinger)

=begin
On Wed, Jan 12, 2011 at 1:46 AM, Haase, Konstantin
Konstantin.Haase@student.hpi.uni-potsdam.de wrote:

> Both #as (Parslet, Sequel, MetaWhere, hacketyhack, Facets, Ruport, CouchPotato, ...) and #to (RSpec, Ramaze, ActiveSupport) are use in DSLs.
> However, I really prefer #to.

Just wanted to point out that the #to method is only defined on two specific instances in Ramaze, so it will not cause any conflicts to worry about.

> Konstantin
>
> On Jan 11, 2011, at 17:39 , Rodrigo Rosenfeld Rosas wrote:
>
> > Issue #4264 has been updated by Rodrigo Rosenfeld Rosas.
> >
> > **Although I don't like Groovy very much (but still use it at a daily basis as part of my current job - at least it is not as bad as Grails - ARGH!), I think 'as' is also a good name and maybe even better than 'to'.**
> >
> > http://redmine.ruby-lang.org/issues/show/4264
> >
> > ---
> >
> > http://redmine.ruby-lang.org

--
Michael Fellinger
CTO, The Rubyists, LLC

=end

#18 - 01/12/2011 10:01 PM - zimbatm (zimba tm)

- File typecast.rb added

=begin
I have this version from 2004 that is lying around that does approximately the same. It was included in the factets gem but got pulled out at some time. It uses #cast_to and #cast_from as the methods of coercion. Each conversion is defined as a method either on the instance of the object that calls #cast_to (#cast_to_#{target_class}), or on the target class as #cast_from_#{origin_klass}. I'm not totally happy about it, since it has a strong dependence on the class name, which means that inheriting classes cannot inherit the conversion mechanisms.
=end

#19 - 01/13/2011 01:12 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:34404] Re: [Ruby 1.9-Feature#4264][Open] General type coercion protocol for Ruby"
on Wed, 12 Jan 2011 14:09:47 +0900, Charles Oliver Nutter headius@headius.com writes:

|I guess I'm confused what the difference is. For example, sometimes
|to_ary is used as a marker, and not invoked. Sometimes it is invoked
|and expected to return an array. There are many cases in RubySpec
|showing both behaviors. So it seems to be a more narrow conversion
|method, but it still seems like a conversion method to me.

Now the world of Ruby is huge, so I could not be responsible for all, but the original intention of to_ary and alike are implicit conversion to retrieve C structure for C defined methods. Using it as a mere marker is not a good idea, I think.

|Can you provide more clarification:
|
|* Are to_ary/to_str/to_int intended only for use from core methods?

Basically they are for C implemented methods. Ruby implemented method

should based on duck typing.

|* Are they expected to always return the an instance of the appropriate type?

They are expected to return an instance of appropriate type all the time, no exception. If you cannot provide corresponding object, you should not define to_ary for the class.

|* Should users be defining these methods to produce new structures in
|the same way as they define to_a, or should they only be doing so if
|their class can duck-type as an array/string/integer for all
|array/string/integer methods?

I strongly suggest that users should define to_ary only when the object behaves just like an array.

Did I make myself clear?

```
                        matz.
```

=end

#20 - 01/13/2011 03:45 PM - headius (Charles Nutter)

=begin
On Wed, Jan 12, 2011 at 10:11 AM, Yukihiro Matsumoto matz@ruby-lang.org wrote:

> Did I make myself clear?

Perfectly, thank you. This should be preserved for all time :) Bottom line, don't define those methods unless you are 100% compatible with the type they reference, and generally they're only intended for use from core methods that need to get those types of objects.

So the proposal is intended for userland conversion methods like to_a, to_s, to_yaml, and friends.

- Charlie

=end

#21 - 01/13/2011 04:27 PM - headius (Charles Nutter)

=begin
Jonas: Cool! This is basically the exact protocol we're talking about here, just with different names. It actually also does Jim's suggestion, having the class-side to_ methods include the class name, as in your example of to_string. I like it.
=end

#22 - 01/15/2011 05:58 AM - kstephens (Kurt Stephens)

=begin
The:

def to(target_module, *args)
send("to_#{target_module.name}", *args)
end

protocol requires the dynamic construction of a Symbol (and/or String), whereas the:

def to(target_module, *args)
send(target_module.coercer, *args)
end

protocol does not.

They both behave the same way, but the latter is easier to implement efficiently.

Some Rubies could inline target_module.coercer and perhaps even inline a specialization of #to(target_module).

Module#coercer can return *any* Symbol (too bad Ruby doesn't have anonymous Symbols).

Some syntactic sugar/module macros could abstract the coercing method away from the protocol:

class MyClass

define_to_method(String) do
"yo!"
end
end
MyClass.new.to(String)

=end

#23 - 01/19/2011 07:06 PM - meta (mathew murphy)

=begin
On Tue, Jan 11, 2011 at 10:26, Charles Nutter redmine@ruby-lang.org wrote:

> Issue #4264 has been updated by Charles Nutter.
>
> I like #to better as well, but isn't it too generic a name, sure to conflict with libraries out there?

Two-letter method names make me uneasy. How about obj.cast(Class)?

Or cast_to / cast_from, as in Jonas Pfenniger's patch.

mathew
[Not a big fan of attr_* either]

=end

#24 - 01/19/2011 08:55 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin
On 19-01-2011 08:05, mathew wrote:

> On Tue, Jan 11, 2011 at 10:26, Charles Nutterredmine@ruby-lang.org wrote:
>
> > Issue #4264 has been updated by Charles Nutter.
> >
> > I like #to better as well, but isn't it too generic a name, sure to conflict with libraries out there?
> > Two-letter method names make me uneasy. How about obj.cast(Class)?
>
> Or cast_to / cast_from, as in Jonas Pfenniger's patch.
>
> mathew
> [Not a big fan of attr_* either]
>
> I like the cast_to/cast_from names and I guess it avoids conflict with most libraries.

Rodrigo.

=end

#25 - 01/20/2011 06:57 AM - headius (Charles Nutter)

=begin
On Wed, Jan 19, 2011 at 5:54 AM, Rodrigo Rosenfeld Rosas
rr.rosas@gmail.com wrote:

> I like the cast_to/cast_from names and I guess it avoids conflict with most libraries.

They're not bad, but casting to me means something altogether different: treating a an object as a different type, while still referencing the same object.

I don't know what "cast" means generally in language design/type theory, but it seems wrong for what's happening here. In our case, we're asking the object to convert itself to a specific type...not casting it to a type it already implements.

Perhaps convert_to and convert_from would be more in line with what's happening?

- Charlie

=end

#26 - 01/20/2011 09:16 AM - rosenfeld (Rodrigo Rosenfeld Rosas)

=begin
Em 19-01-2011 19:53, Charles Oliver Nutter escreveu:

> On Wed, Jan 19, 2011 at 5:54 AM, Rodrigo Rosenfeld Rosas
> rr.rosas@gmail.com wrote:
>
> > I like the cast_to/cast_from names and I guess it avoids conflict with most
> > libraries.
> > They're not bad, but casting to me means something altogether
> > different: treating a an object as a different type, while still
> > referencing the same object.
>
> I don't know what "cast" means generally in language design/type
> theory, but it seems wrong for what's happening here. In our case,
> we're asking the object to convert itself to a specific type...not
> casting it to a type it already implements.
>
> Perhaps convert_to and convert_from would be more in line with what's happening?
>
> - Charlie

convert_to/from is ok to me too. Does someone else object this name?

Rodrigo

=end

#27 - 01/25/2011 11:14 AM - meta (mathew murphy)

=begin
On Wed, Jan 19, 2011 at 15:53, Charles Oliver Nutter
headius@headius.com wrote:

> They're not bad, but casting to me means something altogether
> different: treating a an object as a different type, while still
> referencing the same object.

Casting refers to treating a value as a value of a different type,
that's true; however, it may or may not reference the same storage
(object, memory or register) after casting.

Consider the C code

int i = 4;
float j = (float) i;

which depending on the machine architecture and compiler, could result
in the second variable referring to a different register, perhaps even
on a different processor.

Having said that, I prefer "convert_to" and "convert_from" because
"cast" in this sense is specialist jargon, and why use jargon when
it's not necessary?

mathew

=end

#28 - 03/25/2012 02:48 PM - mame (Yusuke Endoh)

- Description updated

- Status changed from Open to Assigned

- Assignee set to matz (Yukihiro Matsumoto)

#29 - 03/25/2012 02:48 PM - mame (Yusuke Endoh)

- Assignee changed from matz (Yukihiro Matsumoto) to mame (Yusuke Endoh)

#30 - 04/02/2012 10:44 PM - mame (Yusuke Endoh)

- Assignee changed from mame (Yusuke Endoh) to matz (Yukihiro Matsumoto)

Hello,

I can't remember why I assigned this to myself...
I think I can do nothing about this ticket.
So I assign this to matz.

Sorry I did not follow the discussion, but I couldn't understand this issue at all.
I guess it is helpful to restate:

- what is the problem you think?
- what are you proposing?
- how does your proposal solve the problem?

by using sample code?

--
Yusuke Endoh mame@tsg.ne.jp

#31 - 04/03/2012 02:14 AM - headius (Charles Nutter)

Mostly I'd like to see some consistency in coercion supported by a base protocol in Ruby itself.

The rough proposal was that BasicObject could have something like #convert_to(cls) that attempts to call a conversion method on the target class, passing self.

```
class BasicObject
def convert_to(cls)
cls.convert(self)
end
end
```

This double-dispatching protocol could allow simple conversions to be implemented in terms of Class#convert:

```
class BasicObject
def to_a
convert_to(Array)
end
end

class Array
def self.convert(obj)
[obj]
end
end
```

And so on. But the more valuable benefit of the protocol would be that instead of having everyone add methods to BasicObject for their own conversion type, they'd add convert impls to their custom classes

```
class MyDataType
def self.convert(obj)
# logic for converting arbitrary objects to MyDataType
end
end
```

There's many dimensions to this discussion, of course, but I wanted to start the discussion about making Ruby's coercion protocols a bit more formal and supported by standard APIs.

#32 - 04/03/2012 03:50 AM - trans (Thomas Sawyer)

This may be of some interest to the conversation:

http://rubyworks.github.com/platypus/

It includes a derivation of typecast system originally developer by Jonas Pfenniger. Interestingly I came across this too:

http://bugs.ruby-lang.org/attachments/1412/typecast.rb

#33 - 04/03/2012 03:52 AM - trans (Thomas Sawyer)

Oh, I just noticed that attachment is actually from earlier in the conversion.

#34 - 04/03/2012 07:59 AM - mame (Yusuke Endoh)

Hello,

Thank you, but still I'm not sure...
If we are writing the following code:

class Foo
def to_a
...converting Foo to Array...
end
end
class Bar
def to_a
...converting Bar to Array...
end
end
class Baz
def to_a
...converting Baz to Array...
end
end

we will rewrite the following in your proposal, right?

class Array
def convert(obj)
case obj
when Foo
...converting Foo to Array...
when Bar
...converting Bar to Array...
when Baz
...converting Baz to Array...
end
end
end

I can't understand why this is consistent.

--
Yusuke Endoh mame@tsg.ne.jp

#35 - 04/03/2012 12:20 PM - trans (Thomas Sawyer)

I really don't see how this can be about the "light" conversions, to_a, to_i, to_s, etc. It only makes sense to me as being for the strict conversions to_ary, to_int, to_str. Why? Because many different methods can result in array, integer, or string, etc. These common method names are just terms used to produce a common or typical form.

Also the use of case statement in the double dispatch is not a good idea. We need to be able to define these per class. Maybe something like:

class Array
define_conversion(Foo) do |obj|
...converting Foo to Array...
end

```
define_conversion(Bar) do |obj|
  ...converting Bar to Array...
end

define_conversion(Baz) do |obj|
   ...converting Baz to Array...
end
```

end

#36 - 04/05/2012 12:28 AM - headius (Charles Nutter)

Yusuke: Individual classes can (and often should) still define their own to_a, etc. There's no reason to move the current logic into a slower case/when statement.

The convert_to(cls) method would just be a standard way to allow new libraries to handle arbitrary object types without monkey-patching. For

example, rather than monkey-patching a to_xml into Object or BasicObject, an XML library could just define XML.convert(obj) that contains that logic. Users would then just call obj.convert_to(XML).

#37 - 04/05/2012 03:53 AM - Anonymous

- File noname added

On Thu, Apr 05, 2012 at 12:28:38AM +0900, headius (Charles Nutter) wrote:

> Issue #4264 has been updated by headius (Charles Nutter).
>
> Yusuke: Individual classes can (and often should) still define their own to_a, etc. There's no reason to move the current logic into a slower case/when statement.
>
> The convert_to(cls) method would just be a standard way to allow new libraries to handle arbitrary object types without monkey-patching. For example, rather than monkey-patching a to_xml into Object or BasicObject, an XML library could just define XML.convert(obj) that contains that logic. Users would then just call obj.convert_to(XML).

So implementation would just be something like this:

```
class Object
def convert_to obj
obj.convert self
end
end
```

It seems very nice. Hopefully it would discourage people from monkey patching things like to_json on to object. This type of monkey patching is a real world problem we (the Rails team) has to deal with. Several JSON libraries would implement to_json, and when people had them both activated at the same time, things would explode.

Fortunately we've worked around those issues.

I think this will help mitigate the "multi-monkey patch" problem, but as long as people can monkey patch, those types of bugs will happen. :(

--
Aaron Patterson
http://tenderlovemaking.com/

#38 - 04/05/2012 07:45 AM - mame (Yusuke Endoh)

=begin
Hello, headius

Thank you. Is your problem about just monkey-patching?
Please check the following summary.

== Problem
Many monkey-patching like below is used in some libraries, such as xml and json:

```
class Integer
def to_json
...
end
end
class Array
def to_json
...
end
end
...
```

The motivation of this proposal is to remove these monkey-patching.

== Proposal
You want Ruby to provide the following simple one method:

```
class Object
def convert_to(kls)
kls.convert(self)
end
end
```

The method name is still arguable, though.

== How the problem is solved
The libraries only have to define the following one method, instead of a lot of monkey-patching.

```
class JSON
def convert(obj)
case obj
when Integer
...
when Array
...
end
end
end
```

Note that a user should write convert_to(JSON) instead of to_json.
=end

#39 - 04/05/2012 07:47 AM - mame (Yusuke Endoh)

And I'd like to add some question.

== Why do you want to remove the monkey-patching?

If you are worried about method name conflict, this proposal does not solve the problem because the class names still collide.

If you dislike monkey-patching just because it is monkey-patching, refinement (#4085) may be more general solution.

== Method name

I understand you want to avoid bikeshed, but I think it is inevitable in this proposal.

== Performance

I don't think this is so big problem :-)

--
Yusuke Endoh mame@tsg.ne.jp

#40 - 11/20/2012 10:30 PM - mame (Yusuke Endoh)

- Target version set to 2.6

#41 - 12/07/2017 07:18 AM - mame (Yusuke Endoh)

- Status changed from Assigned to Feedback

#42 - 12/25/2017 06:14 PM - naruse (Yui NARUSE)

- Target version deleted (2.6)

Files

typecast.rb	6.15 KB	01/12/2011	zimbatm (zimba tm)
noname	500 Bytes	04/05/2012	Anonymous