Ruby master - Feature #15408

Deprecate object_id and _id2ref

12/13/2018 12:53 AM - headius (Charles Nutter)

Status:	Open
Priority:	Normal
Assignee:	headius (Charles Nutter)
Target version:	

Description

Ruby currently provides the object_id method to get a "identifier" for a given object. According to the documentation, this ID is the same for every object_id call against a given object, and guaranteed not to be the same as any other active (i.e. alive) object. However, no guarantee is made about the ID being reused for a future object after the original has been garbage collected.

As a result, object_id can't be used to uniquely identify any object that might be garbage collected, since that ID may be associated with a completely different object in the future.

Ruby also provides a method to go from an object_id to the object reference itself: ObjectSpace._id2ref. This method has been in Ruby for decades and is often used to implement a weak hashmap from ID to reference, since holding the ID will not keep the object alive. However due to the problems with object_id not actually being unique, it's possible for _id2ref to return a different object than originally had that ID as object slots are reused in the heap.

The only way to implement object_id safely (with idempotency guarantees) would be to assign to all objects a monotonically-increasing ID. Alternatively, this ID could be assigned lazily only for those objects on which the code calls object_id. JRuby implements object_id in this way currently.

The only way to implement _id2ref safely would be to have a mapping in memory from those monotonically-increasing IDs to the actual objects. This would have to be a weak mapping to prevent the objects from being garbage collected. JRuby currently only supports _id2ref via a flag, since the additional overhead of weakly tracking every requested object_id is extremely high. An alternative for MRI would be to implement _id2ref as a heap scan, as it is implemented in Rubinius. This would make it entirely unpractical due to the cost of scanning the heap for every ID lookup.

I propose that both methods should immediately be deprecated for removal in Ruby 3.0.

- They do not do what people expect.
- They cannot reliably do what they claim to do.
- They eventually lead to difficult-to-diagnose bugs in every possible use case.

Put simply, both methods have always been broken in MRI and making them unbroken would render them useless.

History

#1 - 12/13/2018 12:58 AM - headius (Charles Nutter)

I should point out that even the monotonically-increasing ID will eventually break once enough objects have been created to roll over a 64-bit integer limit unless object_id can be a Bignum, which would mean holding references Bignums for every ID from that point on.

#2 - 12/13/2018 01:22 AM - shyouhei (Shyouhei Urabe)

100% agree. These methods show too much internals (bare C pointer values). Maybe I can compromise on them moving into ext/objspace, but at least they should be hidden from the core API.

#3 - 12/13/2018 01:36 AM - ko1 (Koichi Sasada)

I use object_id for debugging to compare identities.
comparison methods can be enough.

#4 - 12/13/2018 01:43 AM - headius (Charles Nutter)

> I use object_id for debugging to compare identities.
> comparison methods can be enough.

I believe you are saying that using equal? is an acceptable alternative to comparing object_id's, yes?

#5 - 12/13/2018 01:47 AM - headius (Charles Nutter)

> These methods show too much internals (bare C pointer values).

It is possible to implement them as lazy monotonically increasing integers, but you either have to accept that they will overflow into bignum or wrap around at some point. JRuby has opted for wrap-around currently, so even our impl is not truly idempotent.

> ...they should be hidden from the core API.

I'd love to see a hard break sooner than later, but I think deprecation will quickly get the Ruby community to clean up uses.

It would also be a good idea for WeakMap to become a standard, public class rather than one hidden inside ObjectSpace, since that's typically the use case for _id2ref. Libraries can assign their own keys for objects in whatever way they choose (e.g. monotonically increasing integer).

#6 - 12/13/2018 01:51 AM - headius (Charles Nutter)

Oh, I'd also rather not even see these features moved into ext/objspace since that just means people will start adding require 'objspace' so they can keep using the features. They should always warn, and ideally just disappear altogether.

Or as in JRuby, you have to enable them at the command line, with no alternative (to make it an explicit opt-in that no library can override).

#7 - 12/13/2018 02:00 AM - headius (Charles Nutter)

There are currently 64 references in ext+lib to object_id, __id__, or _id2ref.

```
[] ~/projects/ruby $ git grep object_id lib | wc -l
      40

[] ~/projects/ruby $ git grep __id__ lib | wc -l
       9

[] ~/projects/ruby $ git grep object_id ext | wc -l
      14

[] ~/projects/ruby $ git grep __id__ ext | wc -l
       0

[] ~/projects/ruby $ git grep _id2ref lib | wc -l
       1

[] ~/projects/ruby $ git grep _id2ref ext | wc -l
       0
```

#8 - 12/13/2018 02:11 AM - ahorek (Pavel Rosický)

https://github.com/rspec/rspec-expectations/blob/848fcb426cca1977b75909d0cb8a10f03a104b36/lib/rspec/matchers/built_in/equal.rb#L76
object_id is used even in rspec for debugging.
I also find it useful to estimate how many objects were created, but it may be wrong usage. Equal? can be enough for most cases.

_id2ref should be hidden.

#9 - 12/13/2018 02:17 AM - headius (Charles Nutter)

My proposal would make WeakMap a standard Ruby feature, so you can implement your own ID system and weakly track objects in the right way. Maybe we need IDMap that returns a guaranteed-unique generated key when you insert an object.

https://github.com/headius/weakling/blob/master/lib/weakling/collections.rb

#10 - 12/13/2018 02:21 AM - headius (Charles Nutter)

> object_id is used even in rspec for debugging.

All objects need to have a base hashcode; that's what we should be logging instead. In Java, this is accessible using System.identityHashCode(obj). JRuby uses this for the base object hashcode (i.e. we do not trigger object_id to be created just for inspect).

> I also find it useful to estimate how many objects were created, but it may be wrong usage. Equal? can be enough for most cases.

I'm not sure I understand this. It wouldn't be accurate to just track seen object_id's anyway since they'll get reused.

#11 - 12/13/2018 04:03 AM - spatulasnout (B Kelly)

On 12/12/2018 4:53 PM, headius@headius.com wrote:

> I propose that both methods should immediately be deprecated for removal in Ruby 3.0.

Agree on id2ref; *strongly* disagree on object_id.

- They do not do what people expect.

object_id does precisely what I expect.

- They cannot reliably do what they claim to do.

If so, let's fix the documentation of object_id.

- They eventually lead to difficult-to-diagnose bugs in every possible use case.

How?

Sources of the embedded ruby portion of a robust C++ desktop application continuously
developed for 15+ years:

```
$ git grep object_id | wc -l
111
```

> Put simply, both methods have always been broken in MRI and making them unbroken would render them useless.

object_id has never been broken. No need to tar it with id2ref's failings.

> I should point out that even the monotonically-increasing ID will eventually break once enough objects have been created to roll over a 64-bit integer limit

```
((2**64)/1_000_000_000)/60.0/60.0/24.0/365.25
=> 584.5420460681421
```

(Merely an aside, since I maintain object_id is fine as-is. But this
appears to be a billion objects per second for 584 years.)

tl;dr: id2ref can go; please leave object_id alone.

Regards,

Bill

#12 - 12/13/2018 04:13 AM - spatulasnout (B Kelly)

> ((2**64)/1_000_000_000)/60.0/60.0/24.0/365.25
> => 584.5420460681421
>
> (Merely an aside, since I maintain object_id is fine as-is. But this
> appears to be a billion objects per second for 584 years.)

Sorry, subtract the two bits needed to distinguish Fixnum from other
objects. Still: 146 years?

#13 - 12/13/2018 04:29 AM - duerst (Martin Dürst)

headius (Charles Nutter) wrote:

> I should point out that even the monotonically-increasing ID will eventually break once enough objects have been created to roll over a 64-bit integer limit unless object_id can be a Bignum, which would mean holding references Bignums for every ID from that point on.

I'd like to mention that in that case, the Bignum object will also need an object_id, which will also be a Bignum object,...

Fortunately, the numbers that spatulasnout (B Kelly) (B Kelly) has given show that this scenario is rather irrelevant.

On the other hand, if object_id is just a form of the pointer, then I wonder whether we're safe for transient heaps and the like, where objects are being moved around.

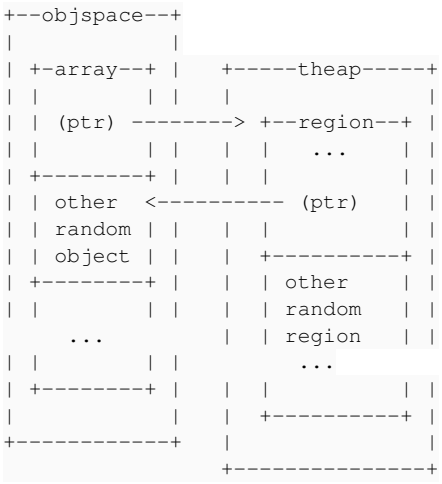
#14 - 12/13/2018 05:00 AM - shyouhei (Shyouhei Urabe)

duerst (Martin Dürst) wrote:

> On the other hand, if object_id is just a form of the pointer, then I wonder whether we're safe for transient heaps and the like, where objects are being moved around.

Just to tell you that transient heaps are for non-object allocations, never for objects. They include pointers to objects, though.

- ObjectSpace is a collection of objects
- An array is an object, which resides inside of an objspace, which has a pointer to a memory region allocated in a transient heap.
- That region inside of a transient heap has pointers to other objects, but never objects themselves.



Anyways this is an implementation detail that should never be visible from any ruby programs.

#15 - 12/13/2018 05:36 AM - headius (Charles Nutter)

> object_id does precisely what I expect.

Then your expectations do not include that it's actually an ID, since it's literally just a pointer into the heap. In the short term, that pointer will likely be occupied by other objects. Longer term for Ruby that pointer value will actually change as the heap gets compacted or objects are moved to other generations.

The problems of object_id could possibly be solved with a rename or better documentation. I think it needs to either be:

- A strictly-monotonically increasing value. As you pointer out, it would be difficult with current Ruby implementations on current hardware to blow out the 62-bit limit. However that integer needs to be atomically updated for every object that needs an ID. It would still be best to do this lazily only for objects where it's needed. This is exactly the JRuby implementation right now (though we don't lose those two bits).

OR

- A pseudo-random hash value never guaranteed to be unique but guaranteed to have a reasonable hash distribution. This is the JVM's identityHashCode which JRuby uses for the base hash value for all objects. MRI currently uses object_id both as a base hash and as an unreliable pseudo-ID.

If _id2ref goes away and object_id becomes one of the above, that's likely acceptable. I don't like changing how object_id works in such a drastic way without naming it something more appropriate, though.

> If so, let's fix the documentation of object_id.

Or we fix object_id to actually be an ID. Or we get rid of it and replace it with something more like a base hash. Both are better options than leaving it in place, since it's not an ID, it's not idempotent, and it doesn't do what most people expect.

> They eventually lead to difficult-to-diagnose bugs in every possible use case.
> How?

Nearly all uses of object_id I have seen treat it as a reliable alias for the object itself. All such code is broken. Exceptions include object_id used solely for base object hash calculation or inspect output, neither of which really require uniqueness.

> Sources of the embedded ruby portion of a robust C++ desktop application continuously developed for 15+ years

This proves nothing without knowing how object_id is being used. What are you using those object_ids for? Show us please.

> object_id has never been broken. No need to tar it with id2ref's failings.

One of the primary use cases of object_id is pairing it with _id2ref. As I've said a couple times, _id2ref most definitely needs to go away. Once it does, we have to decide what object_id is really supposed to be, because it can't be what it is now and be safely usable for more than logging or debugging.

> Sorry, subtract the two bits needed to distinguish Fixnum from other objects. Still: 146 years?

Assuming 64-bit systems, you're right, it would take a long time with current Ruby implementations. That's why we thought it acceptable to implement it this way in JRuby many years ago, since JRuby has always had 64-bit Fixnums.

> On the other hand, if object_id is just a form of the pointer...

Hopefully it's clear by now that it can't just be a form of the pointer, since the pointers are reused today and will be reused even more in the future.

#16 - 12/13/2018 07:11 AM - ko1 (Koichi Sasada)

headius (Charles Nutter) wrote:

> > I use object_id for debugging to compare identities.
> > comparison methods can be enough.
>
> I believe you are saying that using equal? is an acceptable alternative to comparing object_id's, yes?

Yes. The problem is I always forget which is identity comparison, equal?, eql?, eq?, ==.
:p

#17 - 12/13/2018 11:32 AM - Eregon (Benoit Daloze)

Agreed we should remove ObjectSpace._id2ref, since it's fundamentally broken on MRI with the current MRI object_id semantics (could get another object if that memory address is reused after GC'ing the old object).

Removing #object_id doesn't make sense to me if we remove _id2ref: #object_id has the same semantics as System.identityHashCode(), and it is idempotent, but not unique (at least on MRI).
The Ruby documentation doesn't guarantee uniqueness:

> The same number will be returned on all calls to object_id for a given object,
> and no two active objects will share an id.

That's the same as System.identityHashCode().
I think we do need a identity-based hash for hashtables and hashing arbitrary objects which do not redefine #hash.

So, how about deprecating _id2ref in 2.6 or 2.7, and removing it in 3.0?

It seems only drb is using _id2ref in the standard library. We will need to find a replacement for that usage.

Aside: I wonder how _id2ref works in MRI if objects are moved by the GC (since #object_id is just the address), or maybe objects are never moved in MRI?

#18 - 12/13/2018 11:54 AM - shevegen (Robert A. Heiler)

I originally wanted to write a very long reply but I think it becomes too difficult
to keep track of what is being said, so just my opinion in a somewhat more condensed
form than before:

- I have no strong opinion on _id2ref. I think I used it only once or twice in 13 years.
- I am not convinced that .object_id should be removed. We would lose some introspection
  here, wouldn't we?

I would recommend to postpone deprecation after ruby 3.x IF it is decided to remove

object_id. But I am not convinced that it should be removed - it is not only a question of implementation details, or how people use it "incorrectly", but it is whether we should be able to query object ids, find them in ObjectSpace etc...

I would suggest this to ask matz in an upcoming developer meeting since part of the question is the intent of how matz may suggest or think that ruby users may use object_id. I don't have much code that relies on object_id, so an any change would not affect me that much - but I am in disfavour of removing it eventually without really knowing what the alternatives are. Or whether we would just lose functionality without any real gain, with which I would disagree very strongly. So my own opinion is much closer as to what spatulasnout wrote.

PS: It is a bit difficult to reason in pro/con when it comes to "incorrect" usage of ruby code. Ultimately the parser allows something or does not; and how matz designed ruby + ruby's philosophy plays in, which of course determines what the parser allows (compare to python requiring () for method calls and ruby not "caring" that much, unless it is ambiguous). Anything aside from that is heavily subject to a personal opinion, and this becomes difficult to reason about.

The world is not going to end if object_id is removed; but it is not going to end when object_id remains, either.

For sake of completion, I would also like to point out that the pickaxe mentioned object_id several times, so I assume quite some people know and have used object_id (and probably not many used _id2ref); and the name used to be .id if I recall correctly before it was changed to object_id. But anyway, I highly recommend to have this be discussed some time in 2019 at a developer meeting.

PSS: Perhaps for alternatives, such as WeakMap, it could be tested extensively, perhaps as a separate gem, if only to provide a proof-of-concept.

#19 - 12/13/2018 01:32 PM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote:

> It seems only drb is using _id2ref in the standard library. We will need to find a replacement for that usage.

I've heard that pycall is also using _id2ref. This API seems to be useful to create a remote wrapper of a Ruby object out of the Ruby process; 1) Ruby passes object_id of some object to another process (say, Python), 2) Python creates a wrapper object that contains the Ruby object_id, and 3) Python asks the Ruby process to invoke a method of the object that has the object_id (_id2ref is actually useful here). I'm unsure how to garbage-collect the wrapped objects, though. Maybe they are assumed to be always marked during the session. Or a kind of distributed GC may be needed. (I'm not familiar with this area.)

> Aside: I wonder how _id2ref works in MRI if objects are moved by the GC (since #object_id is just the address), or maybe objects are never moved in MRI?

Objects are never moved in MRI.

#20 - 12/13/2018 01:56 PM - headius (Charles Nutter)

I'm glad most of us are in agreement about _id2ref!

> object_id ... is idempotent,

Yeah you're right here...I realize the lack of idempotency applies to using it in _id2ref, since it will eventually return a different result over time, but a given object currently does maintain a consistent object_id.

> > and no two active objects will share an id.
>
> That's the same as System.identityHashCode().

System.identityHashCode makes no uniqueness guarantees at all. It's absolutely possible for two objects to have the same identityHashCode, especially because it's only a 32-bit signed integer.

object_id guarantees uniqueness against other currently-alive objects, since it's the pointer to each object.

I mentioned above, I'd be mostly satisfied if object_id were reduced to an identity hash code OR if it were generated and guaranteed unique to the lifetime run of the process. It's somewhere in the middle right now and that's where the problems come from.

> I wonder how _id2ref works in MRI if objects are moved by the GC

ko1 and tenderlove know current status of this, but up until recently no objects were moved in MRI ever. Now with generational GC and compaction, they'll absolutely be moved around, so object_id *must* change, regardless of how much people love it the way it is. The two options I've spelled out are reasonable alternatives.

At this point, my main concern is having it be called anything like "ID" without uniqueness.

If it remains "object_id" I think it needs to use the monotonically-increasing value.

If it will be reduced to an identity hashcode, it should not be named "object_id". "identity_hash" (mirrors its use in Hash) or something similar would be better/more accurate/more descriptive.

#21 - 12/13/2018 01:59 PM - headius (Charles Nutter)

> The world is not going to end if object_id is removed; but it is not going to end when object_id remains, either.

I don't think the world is going to end regardless of what we do. But object_id is going to become even less ID-like as we improve MRI's GC, so it really does have to change now. And obviously it can't be based on the pointer to the object once objects can move on the heap (compaction, generational), so that forces the issue right there.

#22 - 12/13/2018 02:03 PM - headius (Charles Nutter)

> I've heard that pycall is also using _id2ref. This API seems to be useful to create a remote wrapper of a Ruby object out of the Ruby process...

This isn't a problem. pycall needs to have its own unique ID generator for each object passed out to python, and a weak map to recover the object reference from that ID. Making WeakMap official largely covers this use case.

FWIW this is one reason why Java's native interface (JNI) requires you to explicitly request a handle you can save globally for a given object reference. Maintaining this mapping from a handle or ID back to a movable object reference is tricky and often expensive.

#23 - 12/13/2018 02:08 PM - headius (Charles Nutter)

> I'm unsure how to garbage-collect the wrapped objects, though

Oh this leads to another item Ruby really needs to add, related to the _id2ref removal: reference queues.

On the JVM, when you create a WeakReference, you can register it with a ReferenceQueue. When the object associated with the WeakReference is collected, the WeakReference is emptied and pushed onto the ReferenceQueue (by the GC). Later on, or in another thread, you can pull from that queue to clean up resources like wrappers, native pointers, or Hash entries.

Without ReferenceQueue, Ruby has no efficient way of cleaning up evacuated WeakRef objects (you have to scan for empty ones). I fixed this for JRuby in the weakling gem by exposing the ReferenceQueue implementation of the JVM:
https://github.com/headius/weakling/blob/master/ext/org/jruby/ext/RefQueueLibrary.java#L56

With proper weak references and a reference queue, *anyone* can implement WeakMap efficiently on their own, and that covers most uses of _id22ref. We should still ship an official supported WeakMap, though.

#24 - 12/13/2018 02:52 PM - mrkn (Kenta Murata)

Endo-san, thank you for describing the case of pycall.
But, unfortunately, I'm not using object_id and _id2ref for managing Ruby object references in Python.

Rather, in pycall, I use object_id and _id2ref for managing wrappers of Python classes and Python modules in Ruby.
The reason why I use them is I want to reuse existing wrappers, while I don't want to protect the wrappers from garbage collection.
For this purpose, pycall has mappings from a Python object pointer to an ID of Ruby object, which is a wrapper of the Python object.
The existing WeakRef can be used for this purpose, but I didn't want to use it because I want to reduce the frequency of object generation.

#25 - 12/13/2018 03:05 PM - headius (Charles Nutter)

> For this purpose, pycall has mappings from a Python object pointer to an ID of Ruby object, which is a wrapper of the Python object.

Could you use WeakMap to simply map the Python object pointer to the wrapper? I'm guessing Python doesn't move objects either, since they have to reference-count everything.

#26 - 12/13/2018 03:23 PM - Eregon (Benoit Daloze)

mrkn (Kenta Murata) wrote:

> Rather, in pycall, I use object_id and _id2ref for managing wrappers of Python classes and Python modules in Ruby.
> The reason why I use them is I want to reuse existing wrappers, while I don't want to protect the wrappers from garbage collection.
> For this purpose, pycall has mappings from a Python object pointer to an ID of Ruby object, which is a wrapper of the Python object.
> The existing WeakRef can be used for this purpose, but I didn't want to use it because I want to reduce the frequency of object generation.

But then this is exposed to the _id2ref bug, i.e., it could return another wrapper if the wrapper is GC'd and a new wrapper gets the same address.
Then that would probably be a very serious bug exposed to pycall users.

#27 - 12/13/2018 03:36 PM - mrkn (Kenta Murata)

> Could you use WeakMap to simply map the Python object pointer to the wrapper?

Yes, I think I can use WeakMap for my purpose, maybe.

If my memory serves me correctly, when I implemented such a table, I investigated WeakMap. I found the following note in the comment, and then I decided to write it myself.

```
*  This class is mostly used internally by WeakRef, please use
*  +lib/weakref.rb+ for the public interface.
```

> i.e., it could return another wrapper if the wrapper is GC'd and a new wrapper gets the same address.

In this case, I can check whether the Python object pointer in the returned wrapper is equal to the expected Python object pointer.

#28 - 12/13/2018 03:46 PM - headius (Charles Nutter)

> I found the following note in the comment

Yes, I want WeakMap to become part of the public API. I think that needs to happen to smoothly replace _id2ref.

> I can check whether the Python object pointer in the returned wrapper is equal to the expected Python object pointer.

Or you can just wrap the whole thing with StandardError like the newrelic_rpm gem

https://github.com/newrelic/rpm/blob/c529a7dce6afb520a587079a609d39afa83368aa/lib/new_relic/agent/transaction_time_aggregator.rb#L105

(Please don't do this)

#29 - 12/13/2018 04:01 PM - Hanmac (Hans Mackowiak)

i hope you don't plan to mess with the VALUE type too

i jumped through many loops to get it right. I connected the Ruby GC with the Object handing from wxWidgets.

- the wx Object holds a Holder Type which has the Ruby value inside
- the Ruby value holds a pointer to the wx Object inside
- then there is a RefCounter added, (so some are extra protected)
- and i added the RubyHash as holder marked as global so as long as the Values stays in this map, they are not freed

=> Result:

- For as long as the wx Object lives, the Holder Object inside it is alive too
- When the wx Object gets deleted, the older object is deleted too, that does decrease the ref count, which when reaching zero does remove it from the Hash.
- this does allow the ruby side allow the object to be freed too.

And yeah there is protection in Case the wx Object is deleted before the ruby Object, in that case the functions throw an Exception

#30 - 12/13/2018 04:11 PM - headius (Charles Nutter)

> i hope you don't plan to mess with the VALUE type too

Again, ko1 can explain better, but my understanding is that any direct references to VALUE (pointers to actual heap objects) will mark those objects as "shady" and they will not be moved. I don't think that protects the pointer from eventually pointing at another object, though.

object_id will have to be modified to not return the pointer value, so if you're relying on that you might have issues. But otherwise I think object_id is unrelated to VALUE or how objects are managed in C extensions.

#31 - 12/15/2018 02:29 PM - matz (Yukihiro Matsumoto)

I agree with removing _id2ref (gradually for not breaking existing code, of course).
I am against removing object_id since it is used widely. But by removing _id2ref, the implementation of object_id can be separated from pointer values.

Matz.

#32 - 12/16/2018 10:49 AM - Eregon (Benoit Daloze)

naruse (Yui NARUSE)matz (Yukihiro Matsumoto) Is it still time to deprecate ObjectSpace._id2ref for 2.6, or is it too late already in the release cycle?

#33 - 12/16/2018 10:57 AM - naruse (Yui NARUSE)

Eregon (Benoit Daloze) wrote:

> naruse (Yui NARUSE)matz (Yukihiro Matsumoto) Is it still time to deprecate ObjectSpace._id2ref for 2.6, or is it too late already in the release cycle?

For 2.6, it's too late because people don't have a chance to check the breakage with preview/rc.
Try on 2.7.

#34 - 12/16/2018 11:00 AM - Eregon (Benoit Daloze)

- Target version set to 2.7

Let's deprecate ObjectSpace._id2ref in 2.7 then.
I think we also need to address the usage in DRb. cc seki (Masatoshi Seki)

#35 - 01/09/2019 05:46 PM - Eregon (Benoit Daloze)

- Assignee set to headius (Charles Nutter)

headius (Charles Nutter) Would you like to start the deprecation of ObjectSpace._id2ref since you started this issue?

#36 - 01/14/2019 07:35 AM - naruse (Yui NARUSE)

- Target version deleted (2.7)

Target version is used by release engineering; don't use this as just a goal.
In this case set target version after drb removes ObjectSpace._id2ref.

#37 - 03/19/2019 04:02 PM - headius (Charles Nutter)

Sorry for the delay folks, I was intermittently blocked from accessing bugs.ruby-lang.org, but things are working now!

I'll move forward with fixing DRb using the logic JRuby ships in https://bugs.ruby-lang.org/issues/15711.

Remaining work here then would be to mark the method deprecated in the preferred way (for 2.7, hopefully, since it relates to https://bugs.ruby-lang.org/issues/15626 which it sounds like we want to land in 2.7).