Ruby master - Feature #4168

WeakRef is unsafe to use in Ruby 1.9

12/18/2010 01:57 AM - bdurand (Brian Durand)

Status:	Closed	
Priority:	Normal	
Assignee:	nobu (Nobuyoshi Nakada)	
Target version:	2.0.0	

Description

=begin
I've found a couple issues with the implementation of WeakRef in Ruby 1.9.

1. WeakRef is unsafe to use because the ObjectSpace will recycle object ids for use with newly allocated objects after the old objects have been garbage collected. This problem was not present in 1.8, but with the 1.9 threading improvements there is no guarantee that the garbage collector thread has finished running the finalizers before new objects are allocated. This can result in a WeakRef returning a different object than the one it originally referenced.

2. In Ruby 1.9 a Mutex is used to synchronize access to the shared data structures. However, Mutexes are not reentrant and the finalizers are silently throwing deadlock errors if they run while new WeakRefs are being created.

I've included in my patch a reimplementation of WeakRef called WeakReference that does not extend Delegator. This code is based on the original WeakRef code but has been rewritten so the variable names are a little clearer. It replaces the Mutex with a Monitor and adds an additional check to make sure that the object returned by the reference is the same one it originally referred to. WeakRef is rewritten as a Delegator wrapper around a WeakReference for backward compatibility.

I went this route because in some Ruby implementations (like Ruby 1.8), Delegator is very heavy weight to instantiate and creating a collection of thousands of WeakRefs will be slow and use up far too much memory (in fact, it would be nice to either backport this patch to Ruby 1.8 or the delegate changes from 1.9 to 1.8 if possible). I also don't really see the value of having weak references be delegators since is very unsafe to do anything without wrapping the call in a "rescue RefError" block. The object can be reclaimed at any time so the only safe method to be sure you still have it is to create a strong reference to the object at which point you might as well use that reference instead of the delegated methods on the WeakRef. It also should be simpler for other implementations of Ruby like Jruby or Rubinius to map native weak references to a simpler interface.

Sample code with WeakReference

```
orig = Object.new
ref = WeakReference.new(orig)
# ...
obj = ref.object
if obj
  # Do something
end
```

I also have a version of the patch which just fixes WeakRef to work without introducing a new class, but I feel this version is the right way to go.

Also included are unit tests for weak references but the test that checks for the broken functionality is not 100% reliable since garbage collection is not deterministic. I've also included a script that shows the problem in more detail with the existing weak reference implementation.

```
ruby show_bug.rb 100000     # Run 100,000 iterations on the current implementation of WeakRef and
report any problems
ruby -I. show_bug.rb 100000 # Run 100,000 iterations on the new implementation of WeakRef and rep
ort any problems
```

=end

History

#1 - 12/18/2010 02:43 AM - bdurand (Brian Durand)

- File weakref.rb added

=begin
Rdoc comments on the WeakReference class didn't get synced with the code. Updated file attached that just fixes the comments.
=end

#2 - 12/21/2010 03:10 AM - headius (Charles Nutter)

=begin
JRuby already maps WeakRef to the native JVM construct. In addition, the "weakling" gem provides reference queues and a simple weak ID hash based on them. Weak references without reference queues are much more difficult to use, since you have to constantly be scanning weak lists/maps/etc for dead references. I would recommend that weakling's API be incorporated into Ruby's weakref support.

In addition, there are other types of references from the JVM we would like to expose:

- Soft references, which are weak but do not collect unless under GC pressure or if they are untraversed for a certain number of GC cycles
- Phantom references, which are only enqueued and cannot be traversed, providing a much cleaner and safer way to implement finalization =end

#3 - 01/06/2011 03:00 AM - bdurand (Brian Durand)

- File weakref.rb added

- File weakref_patch.diff added

=begin
I found what may be a bug in my patch. When testing similar code on Rubinius, I found the process deadlocking. I believe what was happening is that a thread was stopped in order to run garbage collection when it was inside a synchronize block on a monitor. On garbage collection the finalizers were run which locks on the same monitor to cleanup weak references which resulted in the deadlock and hung process. Can anyone shed some light on if this might be a problem with the garbage collector in YARV? I've attached updated patch files that removes the synchronization in the finalizers and is should still be thread safe.

Also, I believe there are really two issues and that #2 could use some more discussion (and maybe a ticket of its own):

1. BUG - WeakRef is broken on YARV 1.9
2. ENHANCEMENT - The Ruby specification should include a WeakReference and SoftReference implementation

Because soft and weak referencers are fairly tightly tied to the VM's memory manager, having them as part of the specification would make it easier for gem authors to count on them being present and having a consistent interface. I'm not as sure about the utility of phantom references or porting reference queues straight from Java. I'm not sure they fit into Ruby the same way they fit into Java. Specifically, is there a difference in Ruby between a weak reference and a phantom reference with the way Ruby finalizers work and is there a better Rubyesque interface for reference queues given the way finalizers can be attached to objects?

I don't think WeakRef should be the standard going forward because it encourages buggy code by mixing in the delegator pattern. If you want to safely use a weakly referenced object, you should first establish a strong reference to it. Once you have this strong reference, there is no reason to use a delegate anymore.
=end

#4 - 01/11/2011 12:43 AM - bdurand (Brian Durand)

- File weakref.rb added

- File weakref_patch.diff added

=begin
After further review, I found the synchronization issue is definitely with Rubinius and that not synchronizing breaks on other Rubies where hash operations are not atomic. Reverted patch files attached.
=end

#5 - 01/11/2011 04:57 AM - headius (Charles Nutter)

=begin
Some clarification about Weak/Soft/Phantom references.

- Weak: not considered as references by the GC. Objects that are only weakly referenced could be garbage collected at any time. Weak references can be traversed to reach the object as long as it is referenced.
- Soft: Like weak references, but with softer GC characteristics. Specifically, they may survive some number of GC runs before being collected. Useful for weakly-referenced resources that you may leave unreferenced for a while but don't want to end up recreating frequently, such as expensive-to-create data structures. On the JVM, soft references are collected on a schedule based on the size of the heap (i.e. they are not eligible or collection until there's memory pressure or they have been unreferenced for some number of milliseconds per MB of heap).
- Phantom: Similar to weak references, but cannot be traversed to get to the original object. Used to implement finalizers in a "pull" model. Where normal finalizers must be executed by the VM, potentially blocking other operations like GC (and usually blocking other finalizers), phantom references can be used with reference queues to implement user-pulled finalization, reducing load on the VM. You associated finalization code with a phantom reference, and when the object it was associated with has been collected it will be pushed onto a reference queue. Periodically, you can pull from that queue to perform finalization in userland. =end

#6 - 01/11/2011 12:52 PM - kstephens (Kurt Stephens)

=begin
Maybe it's time to implement weakrefs in C that work well with MRI's collector.

http://www.haible.de/bruno/papers/cs/weak/WeakDatastructures-writeup.html

=end

#7 - 01/11/2011 03:54 PM - kstephens (Kurt Stephens)

=begin
I'm working on a patch to implement weak refs in C.
=end

#8 - 01/11/2011 06:08 PM - kstephens (Kurt Stephens)

=begin
See commit in branch from trunk:

http://github.com/kstephens/ruby/commit/0942a955c649c39d3be4db13dbfb0bacfd634994

Probably not the same API as WeakReference.

=end

#9 - 01/11/2011 07:13 PM - kstephens (Kurt Stephens)

=begin
Topic branch: trunk-weakref-c

Fixed: Free cached WeakRef instances after they are unreferenced.

https://github.com/kstephens/ruby/commit/3d4ab8de8231f7d0bd0e9543a41f0c45e08dbfc8

=end

#10 - 01/11/2011 07:28 PM - kstephens (Kurt Stephens)

=begin
This patch could be written as a C ext/*.so if MRI had an API for callbacks functions before and after each of the GC phases.

=end

#11 - 01/12/2011 12:04 AM - headius (Charles Nutter)

=begin
On Tue, Jan 11, 2011 at 3:08 AM, Kurt Stephens redmine@ruby-lang.org wrote:

> Issue #4168 has been updated by Kurt Stephens.
>
> See commit in branch from trunk:
>
> http://github.com/kstephens/ruby/commit/0942a955c649c39d3be4db13dbfb0bacfd634994
>
> Probably not the same API as WeakReference.

It's more similar to Java's WeakReference than Ruby's. Ruby's uses
Delegate, so that method dispatches can go through the WeakRef object
as though it weren't there. Personally, I find this a *terrible* idea,
since code could get a weak reference error at any time, when not
expecting it. Requiring a user to actually traverse the weakref before
invoking makes it clear that there's an intermediate result involved.

Obviously your version doesn't use Delegate at all. ruby-core would
need to decide whether this would be an official external class upon
which the more "feature-rich" WeakRef would build, or whether this
should actually become WeakRef, Delegate and all.

- Charlie

=end

#12 - 01/12/2011 04:09 AM - kstephens (Kurt Stephens)

=begin

Charles:

I agree particularly with the weak reference error problem in the Delegate API. It forces the programmer to rescue exceptions when a simple nil test is enough. Weakrefs cannot be used without evasive action anyway -- the simplest API is probably the best API.

The C implementation is in alignment with Brian's original suggestion. And should it be adopted into Ruby core, it's a simple enough API to be implemented in other Rubies without additional baggage; probably add WeakRef#== and WeakRef#hash methods but that's about all that's necessary.

However this leaves proper weak key Hashes unsolved, as discussed in Bruno Haible's paper. Ruby has the additional problem that its Hash keys are subject to the latent typing of the key values: sometimes object identity, sometimes object equality (String#==, String#hash). However since MRI has a non-copying GC, maybe it's not an issue. How do JRuby and Rubinius maintain EQ hash invariance between collections?

There are fixes in branch to deal with uncollectable immediates (e.g.: Fixnum, Boolean).

- KAS

=end

#13 - 01/12/2011 04:48 AM - headius (Charles Nutter)

=begin
Given proper WeakReference and ReferenceQueue implementations (the latter we still don't for C, I believe), implementing weak-keyed or weak-valued hashes becomes trivial. See my "weakling" gem for an example that uses Java WeaReference/ReferenceQueue.

Without ReferenceQueue, WeakReference is much less useful. You are forced to constantly scan lists/maps/etc for dead links.
=end

#14 - 01/12/2011 10:19 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:34353] Re: [Ruby 1.9-Bug#4168] WeakRef is unsafe to use in Ruby 1.9"
on Wed, 12 Jan 2011 00:03:50 +0900, Charles Oliver Nutter headius@headius.com writes:

|It's more similar to Java's WeakReference than Ruby's. Ruby's uses
|Delegate, so that method dispatches can go through the WeakRef object
|as though it weren't there. Personally, I find this a *terrible* idea,
|since code could get a weak reference error at any time, when not
|expecting it. Requiring a user to actually traverse the weakref before
|invoking makes it clear that there's an intermediate result involved.

It might be worth considering adding new WeakHash or something like
that, leaving weakref.rb as it is broken now, and mark it obsolete.

```
                          matz.
```

=end

#15 - 01/12/2011 11:17 AM - kstephens (Kurt Stephens)

=begin
Matz: are you suggesting renaming this C implementation "WeakPtr" (or something), make it part of Ruby core, and let the old WeakRef class die off? Should I take a crack at designing a GC callback API and make this an ext/*.so?

Charles: If MRI will support a GC callback API, maybe I could add this to weakling and standardize a WeakPtr API for both JRuby, MRI and maybe Rubinius, Evan willing. ReferenceQueue seems to make more sense with generational/incremental GCs (as in many JREs) and it might be overkill/painful in MRIs straight-forward stop-mark-sweep GC. Would a standard Hash::WeakKey class suffice since everything else can be implemented in terms of WeakPtr? Or are you thinking that ReferenceQueue would make weak array, sets, lists, pairs, etc. easier in the long run? I'd rather not design something that isn't gonna jive with other Rubies.

Anybody from the Rubinius team care to comment?
=end

#16 - 01/12/2011 02:15 PM - headius (Charles Nutter)

=begin
A GC callback API? There is no possibility of implementing such an API on JRuby. All the JVMs have different GC implementations, and there's no standard way to have GC callbacks into user code. Is that what you mean?

ReferenceQueue is not related to GC implementation at all. A ReferenceQueue is used by the GC to indicate WeakReferences that have had their referenced object collected. They are generally used to do some cleanup logic once WeakReferences no longer reference any object. For example, a WeakHash would register all weak keys with a reference queue, and on each call check to see if any references had been collected. It can then efficiently clean them up, rather than either letting the hash grow endlessly (filling up with dead WeakReference instances) or scanning all instances

periodically to see if they are dead. I believe ReferenceQueue would make weak collections trivial to implement from Ruby. See the implementation of IdHash in weakling:

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

ReferenceQueue can also be paired with PhantomReference to implement more efficient finalization, as I described above.
=end

#17 - 01/13/2011 12:12 AM - bdurand (Brian Durand)

=begin
My 2 cents on reference queues is that they aren't critical to Ruby weak references because of the way Ruby allows finalizers to be defined for objects external to the objects class definition. A ruby implementation of a weak key hash can simply add finalizers the referenced objects as they are added. The finalizer can take care of removing the reference from the hash.

See https://github.com/bdurand/ref/blob/master/lib/ref/abstract_reference_key_map.rb for an implementation.

That being said, I could still them being useful, but they are not critical and a standard implementation could be nice to have. I think a standard class could be provided as a pure ruby implementation which the various VM's could implement in native code if desired. See https://github.com/bdurand/ref/blob/master/lib/ref/reference_queue.rb
=end

#18 - 01/13/2011 12:59 AM - matz (Yukihiro Matsumoto)

=begin
Hi,

In message "Re: [ruby-core:34400] [Ruby 1.9-Bug#4168] WeakRef is unsafe to use in Ruby 1.9"
on Wed, 12 Jan 2011 11:17:12 +0900, Kurt Stephens redmine@ruby-lang.org writes:

|Matz: are you suggesting renaming this C implementation "WeakPtr" (or something), make it part of Ruby core, and let the old WeakRef class die off?

I am not sure if we use your C implementation. But at least I agree to replace the old WeakRef by one with better API, and let it die gradually. Right now I am thinking of something like WeakMap that does not represent an object, but collection of objects, so that each implementation choose the best implementation for the job.

```
                        matz.
```

=end

#19 - 01/13/2011 05:57 AM - kstephens (Kurt Stephens)

=begin
Charles writes:

| A GC callback API? There is no possibility of implementing such an API on JRuby. All the JVMs have different GC implementations, and there's no standard way to have GC callbacks into user code. Is that what you mean?

No. I'm not proposing a GC API for JRuby because JREs already support WeakReferences, etc natively.

To make C WeakRef an MRI ext/*.so, it needs a callback from MRI's GC after the mark phase. This will only be for MRI (or other Rubies that can/want to implement it in their C APIs). There maybe a couple functions/macros that will help make C WeakRef work with REE's COW-friendly mark bitmaps. A very minimal API.

A few other reasons for exposing GC phases/internals to extensions: stats, efficient instance caches for value objects, etc.

=end

#20 - 01/13/2011 04:03 PM - headius (Charles Nutter)

=begin
Brian: ReferenceQueue is unrelated to finalization. It is not possible to do what a RefQueue does using finalizers, unless you define them on every object you put into a weak collection (and likely slow down GC significantly as a result). RefQueue is basically a way to allow user code to do a much lighter-weight poll than constantly scanning and checking weakrefs for "deadness". I wish I knew another way to demonstrate how much easier it is to build weak collections when you have RefQueues available...

Matz: I'd really like to see a really solid WeakReference added to Ruby rather than just a weak hash. People have needs for weak-keyed hashes, weak-valued hashes, weak-valued lists and sets, and many other collections. If Ruby had a solid, reliable WeakReference (and ReferenceQueue), all these would be possible.

I think some of the basic collections could also be added, but with a standard WeakReference/ReferenceQueue, it could be trivially implemented entirely in Ruby.

Kurt: Ok, I understand.
=end

#21 - 01/14/2011 01:05 AM - bdurand (Brian Durand)

=begin
Charles: The logic to enqueue a weak reference onto a reference queue needs to happen at some point in the garbage collection cycle and something needs to keep track of which weak references need to be enqueued on which queues. My point was that they aren't critical to using weak references and they can be implemented in pure ruby. Don't get me wrong, I like reference queues. I brought it up because I want to make sure that references are thoroughly thought out in the standard library and alternative considered since the current WeakRef implementation was not.

Also, I'd vote against adding any sort of weak hash that extends Hash in the standard library. There are so many methods on Hash that would not be safe to call with weak reference in play. In my opinion it would be better to implement only the safe methods rather than risk failures.
=end

#22 - 01/14/2011 03:29 AM - headius (Charles Nutter)

=begin
Can you describe how you would implement reference queues in pure Ruby?
=end

#23 - 01/14/2011 03:40 AM - bdurand (Brian Durand)

=begin
Ruby reference queue implementation:

http://github.com/bdurand/ref/blob/master/lib/ref/reference_queue.rb
=end

#24 - 01/15/2011 12:02 AM - headius (Charles Nutter)

=begin
Ahh sure, the finalizer way...I thought you meant some other way.

Yes, a finalizer-based queue would work. There are a few issues:

- Much slower than a native queue
- All those finalizers would slow down garbage collection as well
- It does not allow another key use of weak references and ref queues: avoiding finalizers :)

Other than that, yes, it would work. I would not recommend any heavy use, though.
=end

#25 - 01/15/2011 04:21 AM - kstephens (Kurt Stephens)

=begin
Maybe we should take this approach:

1) A standard set of "Weak" classes/API/namespace: WeakReference, ReferenceQueue, WeakKeyHash, maybe even a WeakPair, etc. Brian's ref gem and Charles' weakling gem are an excellent start.
2) Rubies should have some native support for WeakReference due to GC interactions and performance considerations. The pure Ruby object_id/id2ref hack cannot work because GC can occur at any time.

3) The remaining Weak classes/APIs can either be implemented natively in VMs or in pure Ruby using native WeakReference and finalization as primitives.
4) A small MRI-specific GC API to enable WeakReferences as a C extension. This should be portable between MRI and REE mark bit schemes.

As a proof-of-concept, I'll create an MRI GC API and repackage weakref.c as a ext/*.so and show how it can work under MRI and REE.

=end

#26 - 01/15/2011 10:59 AM - headius (Charles Nutter)

=begin
Kurt: I'd say at least WeakReference and ReferenceQueue, with the former as a GC-aware native builtin. The latter could be pure Ruby if nobody wants it to be native, but it would be native in JRuby. I really believe it needs to be there, though, since even though it's possible to implement a ReferenceQueue with finalizers, we'd want something built into all Ruby impls going forward.

A weak hash would be useful, but it's definitely further down my list. If we could guarantee that Ruby impls have WeakReference and ReferenceQueue out of the box, the collections that utilize them could be done entirely from gems (a la weakling).

Goal 1 should be to eliminate the use of object_id/_id2ref from "recommended" weak reference logic. Goal 2 should be to provide the reference-related tools anyone would need to build the other libraries. I believe ReferenceQueue is needed. Anything after that would just be gravy.
=end

#27 - 01/19/2011 03:26 AM - kstephens (Kurt Stephens)

=begin
Progress:

- Moved weakref.c into ext/weak_reference/weak_reference.c via a simple gc_api.c callback manager. Class is now named WeakReference.
- Suggestions/comments on gc_api.c and its semantics are greatly appreciated. FL_MARK tests are abstracted into RB_GC_MARKED(obj) for REE compatibility.
- Any suggestions on a standard Ruby namespace for Weak-related classes? I like Brian's Ref:: namespace (https://github.com/bdurand/ref/blob/master/lib/ref.rb).
- weak_reference.c is not native thread-safe; anybody have suggestions/tips on the best way to do this portably under the YARV VM? I don't think a plain-old Ruby Mutex can be used since parts of the weak_reference.c code are active between critical GC mark and sweep phases.
- TODO: native MRI ReferenceQueue.
- TODO: backporting to MRI 1.8.7 and REE.

Working commit: https://github.com/kstephens/ruby/commit/83b96068f6b3ada4f31057f56254f0190ec54e69
Branch: https://github.com/kstephens/ruby/tree/trunk-weakref-c

=end

#28 - 01/19/2011 06:54 AM - headius (Charles Nutter)

=begin
FWIW, "ref" is the namespace Java uses (java.lang.ref.WeakReference). Since Ruby's namespaces are shorter, I'd almost prefer Reference::WeakReference, but I'll abstain from the naming discussion.
=end

#29 - 01/19/2011 01:33 PM - kstephens (Kurt Stephens)

=begin
Progress:

- Prototype for C SoftReference as requested by Brian.
- Refactored code to use a rb_reference data structure.

https://github.com/kstephens/ruby/commit/c5ab0bf789089d08ddc549d5d9950addeeb41943

=end

#30 - 01/19/2011 10:42 PM - kstephens (Kurt Stephens)

=begin
Progress:

- Implementation of reference queue core protocol in C. Works with any object that respond_to?(:push).
- Nasty HACKS due to MRI GC clobbering of RBASIC(obj)->flags to handle deferred free lists. Would love to understand this more.
- Sample Reference::ReferenceQueue class based on Brian's https://github.com/bdurand/ref/blob/master/lib/ref/reference_queue.rb.
- Changed namespace to Reference::.
- SoftReferences probably need more tuning controls.
- C extensions and GC API are probably *NOT NATIVE THREAD-SAFE*. Need some help in this area.

https://github.com/kstephens/ruby/commit/6d5d783e8983b4f916e5a83fa2add3b665a3e6cc

=end

#31 - 01/20/2011 07:02 AM - headius (Charles Nutter)

=begin
FYI, on the JVM it is not possible to use an arbitrary collection for reference enqueuing, and I think there's a good reason for it: calling back into user code while enqueuing a reference is probably not a good idea.

The JVM ReferenceQueue's backing store is implemented natively, and it is not possible to provide your own queue object. I believe this is to ensure references are enqueued quickly and without potentially triggering memory or GC events (or allowing any thread to completely block the GC). You can extend ReferenceQueue, but you cannot override or otherwise change the enqueuing logic. Perhaps it's presumptive of me, but I assume that if it were reasonable and safe to allow arbitrary user code to fire when enqueuing objects, the JVM designers would have allowed it.

So I would recommend that there not be an open door policy allowing any collection to be used as target reference queue. There should be a single built-in reference queue implementation, and users must be required to use that structure and that structure alone.

In any case, if this feature isn't removed, I believe JRuby will be unable to support it.
=end

#32 - 01/20/2011 07:18 AM - headius (Charles Nutter)

=begin
More comments:

- On JVM, you can only specify a ReferenceQueue to the constructor of a Reference. You cannot change or add queues on already-created references. JRuby would not be able to support adding new reference queues to existing references.
- Soft reference TTL is not configurable from Java APIs directly, and I'm not sure it's possible to query them. Different JVMs use different heuristics for collecting soft references, so there's no consistent definition of what "TTL" actually means. JRuby would not be able to support the gc_ttl method on SoftReference.
- It is not possible to find out how many references have been enqueued on the JVM. You must poll the ReferenceQueue to deal with them as they arrive. I'm not sure of the reason for this limitation, but I assume it's because elements could get enqueued at any time, so the size would be meaningless. JRuby would not be able to support the queued_reference_count method.
- It is also not possible to find out (via normal means) how many weak or soft references are in flight because it does not actually track all of them. JRuby would not be able to support either the weak or soft cached_instance_count methods.

I'd strongly recommend limiting features to those we could implement with the Java reference APIs, since adding APIs we can't support will just cause a lot of code that won't work across implementations.

The docs for the JVM java.lang.ref package are here: http://download.oracle.com/javase/1.5.0/docs/api/java/lang/ref/package-summary.html

I'm very glad to see this API coming together, but I believe care should be taken to avoid exposing details of MRI's GC implementation and avoid adding features that can't be supported on other Ruby VMs.
=end

#33 - 01/21/2011 04:17 AM - kstephens (Kurt Stephens)

=begin
Charles:

I've been using Brian's ref gem as a starting point for a proposed standard API; for example: WeakReference and SoftReference are subclasses of Reference.

I probably should have done this at the start, but I will make changes to hide MRI implementation details (SoftReference#gc_ttl, #cached_instance_count, Reference.queued_reference_count, ReferenceQueue#push, etc.) from a proposed standard API. The lack of distinction is due to programmer laziness.

The multiple ReferenceQueues per Reference and ReferenceQueue#push protocol "features" are really only implementation details: the former is necessary because Reference.new(obj) may return the same cached instance for any obj, the latter is just an internal design/implementation decision. #cached_instance_count, #queued_reference_count, and even #gc_ttl, etc are intended mostly as testing/debugging hooks.

I'll move some of these class methods into a separate "MRI-only" module and prefix instance methods so that it's clear they not part of a proposed standard API.

My intent is to keep this API simple, efficient and portable. You might want to review Brian's ref gem for features that can/cannot be efficiently supported in JRuby; since we're starting from scratch in MRI, it makes sense to follow JRuby as prior-art. Ultimately an API test suite should pass on MRI, JRuby and others, along with some implementation-dependent tests.

Thanks for the excellent feedback and suggestions. Feedback from other Ruby implementers (anybody else out there?) is very valuable at this time.

=end

#34 - 01/21/2011 08:38 AM - headius (Charles Nutter)

=begin
I'm not sure I see any real benefit to caching the WeakReference and SoftReference objects themselves. Indeed, it may be undesirable to not be guaranteed you're going to have a fresh reference, since there may be multiple places in code where users want to construct references around the same object. What's the justification for caching the instances?

I'll have another look at Brian's gem. Basically I'm looking for a minimal implementation similar to "weakling", since that mimics the Java API (sans soft and phantom references).
=end

#35 - 01/21/2011 09:41 AM - headius (Charles Nutter)

=begin
Ok, I see why Brian's reference queue doesn't have the same restrictions as the JVM's: it isn't triggered by the reference, it's triggered by the finalizer attached to the object.

There's a lot of other stuff here, so I'm not sure how much I ought to review. The basic comments I made above about your (Kurt) current patchset are good enough comments for now. And as I mentioned in previous comment, you can also use Weakling's feature set as a pretty good indication of the limitations of JVM weakreferences (and what we'd be able to support in a new official API).

Matz: You said previously you'd like to see a new weak reference API added and the old weakref.rb allowed to die. Couldn't we also just make the new API be the backend for weakref.rb? Or is your intent to actually phase that library out (perhaps due to using Delegate, which most folks seem to agree is a bad idea for a weak reference).
=end

#36 - 01/25/2011 02:43 PM - kstephens (Kurt Stephens)

=begin
Progress: https://github.com/kstephens/ruby/commit/6f3f2a538455187d67ef504cfa266cf5789ca099

- Support for multiple Reference subclass instance caches.
- Working WeakReference, SoftReference and ReferenceQueue in ext/reference/reference.c
- SoftReference responds to new GC heap allocations as "memory pressure" by globally reducing all SoftReference#_mri_gc_left counts.
- Sample WeakKeyHash class using WeakReference and ReferenceQueue.
- Support for JRuby-compatible WeakReference.new(object, reference_queue) protocol.
- Removed ReferenceQueue#monitor method which cannot be supported in JRuby.
- Renamed MRI-specific methods to mri*.
- Optimized gc_api.c callback lists.
- Clean up the C stack after gc_api.c callbacks to avoid conservative GC pinning (might be overkill).
- Removed the ugly (and stupid) flags hacks.
- Added some basic RDOC-style comments.

Charles:

I'm caching live Reference instances to minimize the number of References objects that must be scanned before the sweep phase.

I'd either need to keep a doubly-linked list of References or scan all the GC heap for References (which I did not want to expose from gc.c). Using a lookup table (st_table in MRI) to track of live References was not much more work than a doubly-linked list and it collapses Reference.new(object) requests into a single instance. This is beneficial for other reasons: it reduces the total number of References to common objects and keeps the Reference#== operator simple for live instances (it's simply Reference#object_id#==).

```
Once Reference#object is sweeped, all the References to #object are no longer "live"; and are removed from cac
he and scheduled for ReferenceQueues after GC.  References to immediates are never cached.  Dead References wi
ll probably will be removed from collections or instance variables shortly thereafter (i.e.:
```

via ReferenceQueues).

Why would a user want to insure that Reference.new(object) always returns a new instance, since References are/should be immutable? JRuby and other Rubies do not need to implement the Reference cache.

Aside:

Q: Should we have a trivial HardReference class that always keeps its #object pinned, just for API consistency?

=end

#37 - 01/25/2011 04:40 PM - headius (Charles Nutter)

=begin
On Mon, Jan 24, 2011 at 11:44 PM, Kurt Stephens redmine@ruby-lang.org wrote:

> - Support for multiple Reference subclass instance caches.
> - Working WeakReference, SoftReference and ReferenceQueue in ext/reference/reference.c
> - SoftReference responds to new GC heap allocations as "memory pressure" by globally reducing all SoftReference#_mri_gc_left counts.

Can you provide a short description of the complete "soft" heuristic
as of this commit? Even on JVM, it's unspecified, but each JVM does
spell out what constitutes a "soft" reference for them.

> - Sample WeakKeyHash class using WeakReference and ReferenceQueue.
> - Support for JRuby-compatible WeakReference.new(object, reference_queue) protocol.
> - Removed ReferenceQueue#monitor method which cannot be supported in JRuby.
> - Renamed MRI-specific methods to mri*.
> - Optimized gc_api.c callback lists.
> - Clean up the C stack after gc_api.c callbacks to avoid conservative GC pinning (might be overkill).
> - Removed the ugly (and stupid) flags hacks.
> - Added some basic RDOC-style comments.

It's looking pretty clean. The look-alike API in JRuby will be a thin,
direct layer over Java's reference logic, similar to that found in
weakling.

> Charles:
>
> I'm caching live Reference instances to minimize the number of References objects that must be scanned before the sweep phase.

As far as I know, JVMs do not cache weak/soft references internally.
The GC impact of having many weak references around is a frequent but
accepted problem.

> I'd either need to keep a doubly-linked list of References or scan all the GC heap for References (which I did not want to expose from gc.c).
> Using a lookup table (st_table in MRI) to track of live References was not much more work than a doubly-linked list and it collapses Reference.new(object) requests into a single instance. This is beneficial for other reasons: it reduces the total number of References to common objects and keeps the Reference#== operator simple for live instances (it's simply Reference#object_id#==).

In MRI, where an object can be uniquely identified, I suppose this is fine. I'm not familiar with how other conservative GCs implement efficient weak/soft reference logic, so I'll defer.

> Once Reference#object is sweeped, all the References to #object are no longer "live"; and are removed from cache and scheduled for ReferenceQueues after GC. References to immediates are never cached. Dead References will probably will be removed from collections or instance variables shortly thereafter (i.e.:
> via ReferenceQueues).
>
> Why would a user want to insure that Reference.new(object) always returns a new instance, since References are/should be immutable?
> JRuby and other Rubies do not need to implement the Reference cache.

Provided that there's no specified requirement that the same object get the same Reference, and as many users as want to can have their own separate reference queues receive a reference for the same object, I have no objection.

> Aside:
>
> Q: Should we have a trivial HardReference class that always keeps its #object pinned, just for API consistency?

I always thought this was a gap in java.lang.ref; there's an abstract Reference class, but anyone who wants to mix hard and soft references in a collection must always write their own HardReference subclass. I wanted several times for there to just be a built in HardReference I could use as a one-element indirection mechanism alongside weak, soft, and phantom references.

But that's Java with static types, where I wanted to commingle weak and non-weak references in the same logic. Would this be a useful thing to have in Ruby? I don't know.

When you think your tests are stabilizing, I will implement the same API in JRuby and we'll see how it looks.

- Charlie

=end

#38 - 01/26/2011 09:52 AM - kstephens (Kurt Stephens)

=begin

> 01/25/2011 04:40 PM - Charles Nutter
> Can you provide a short description of the complete "soft" heuristic as of this commit? Even on JVM, it's unspecified, but each JVM does spell out what constitutes a "soft" reference for them.

- SoftReference#_mri_gc_left and SoftRefernce#_mri_gc_ttl are initialized to some arbitrary value: SoftReference._mri_gc_ttl (default = 10).
- Before sweep phase: If SoftReference#object was traversed since last GC, SoftReference#_mri_gc_left is reset to SoftRereference#_mri_gc_ttl.
- If SoftReference#object was not traversed since last GC, SoftReference#_mri_gc_left is decremented by 1.
- If memory pressure occurred, SoftReference#_mri_gc_left is also decremented by 0.5 * average of all SoftReference#_mri_gc_left.
- "Memory pressure" is whenever MRI allocates a new GC heap.
- When SoftReference#_mri_gc_left reaches 0, SoftReference#object= nil.
- When SoftReference#object == nil or #object is not reachable, ReferenceQueues are notified.

The values of 10 and 0.5 above are off-the-cuff tuning parameters. I have no empirical data to warrant the values, but the test shows that the algorithm works.
If the test also passes under JRuby, awesome! :)

See the pseudo-code here: https://github.com/kstephens/ruby/blob/trunk-weakref-c/ext/reference/lib/reference/mri/soft_reference.rb
I'm not sure I like the heuristics. I think others could come up with something better in pure Ruby, thus the pseudo-code may become the implementation.

> When you think your tests are stabilizing, I will implement the same
> API in JRuby and we'll see how it looks.

The unit tests have MRI-specific assertions in them -- I'll abstract them out.

Aside:

Q: How should HardReference behave when associated with a ReferenceQueue?

=end

#39 - 01/26/2011 05:19 PM - headius (Charles Nutter)

=begin
On Tue, Jan 25, 2011 at 6:52 PM, Kurt Stephens redmine@ruby-lang.org wrote:

- Each SoftReference#_mri_gc_left and SoftRefernce#_mri_gc_ttl are set to some arbitrary value: SoftReference._mri_gc_ttl (default = 10).
- If SoftReference#object was traversed since last GC, SoftReference#_mri_gc_left is reset to SoftRereference#_mri_gc_ttl.
- If SoftReference#object was not traversed since last GC, SoftReference#_mri_gc_left is decremented by 1.
- If memory pressure occurred, SoftReference#_mri_gc_left is also decremented by 0.5 * average of all SoftReference#_mri_gc_left.
- "Memory pressure" is whenever MRI allocates a new GC heap.
- When SoftReference#_mri_gc_left reaches 0, SoftReference#object= nil.
- When SoftReference#object == nil or #object is not reachable, ReferenceQueues are notified.

> The values of 10 and 0.5 above are off-the-cuff tuning parameters. I have no empirical data to warrant the values, but the test shows that the algorithm works.
> If the test also passes under JRuby, awesome! :)
>
> See the pseudo-code here: https://github.com/kstephens/ruby/blob/trunk-weakref-c/ext/reference/lib/reference/mri/soft_reference.rb
> I'm not sure I like the heuristics. I think others could come up with something better in pure Ruby, thus the pseudo-code may become the implementation.

This sounds similar to the JRockit JVM, which will definitely clear soft references when out of heap memory and "probably" clean them after some number of GC runs ("probably" as in it's not configurable and is based on a rough LRU system).

Hotspot (Sun/OpenJDK) uses absolute time, measured from the time of the GC prior to the last traversal of the soft reference (in other words, every GC run records a timestamp, and when you traverse a soft reference its internal "clock" variable is set to the previous GC's timestamp). The time-based clearing is configured to be some number of milliseconds per MB of heap. If you have a 500MB heap and ms-per-mb is set to 10, you expect untraversed soft references to survive for roughly 5 seconds (depending on GC schedules, of course. Memory pressure will also trigger a flush of soft references.

I couldn't find information on the J9 JVM's soft reference heuristic.
That's where I stopped :)

> The unit tests have MRI-specific assertions in them -- I'll abstract them out.

Looking forward to it. I should be able to try out the test cases the day you tell me they're ready.

> Aside:
>
> Q: How should HardReference behave when associated with a ReferenceQueue?

It should do nothing. References are added to their queue when the object they reference has been collected (or when it's about to be collected). A HardReference by definition prevents the referenced object from being collected. So registering a HardReference with a ReferenceQueue has no meaning.

I'd almost say it should be illegal to construct a HardReference with a ReferenceQueue, so nobody's confused about expectations. Or perhaps we should have no HardReference at all, since it doesn't fit the rest of the namespace's utility...

- Charlie

=end

#40 - 01/27/2011 10:19 AM - kstephens (Kurt Stephens)

=begin
Progress: https://github.com/kstephens/ruby/commit/8058894024c9bac07e6e44c91f5984a1e347074f

- Refactored MRI-specifics out of tests.
- Implemented SoftReference heuristics in Ruby *and* C.
- Trivial HardReference class ignores ref_queue argument, class is probably pointless anyway.

Charlie:

I think it makes sense to move my ext/reference code into a fork of Brian's ref gem and port gc_api.c to MRI 1.8, so the supporting change in MRI 1.9 and 1.8 is minimal.
This should also make it easier for you to test a JRuby impl. A WeakReference port for Rubinius should be trivial.

I need MRI adoption of gc_api.c soon to keep moving forward;
Matz can adopt Reference::* into MRI core whenever he feels it has matured.

- Kurt

=end

#41 - 01/29/2011 04:54 AM - headius (Charles Nutter)

=begin
If you move the code into a gem, I can add the same bits for JRuby so people can gem install it in 1.8.7 and 1.9.2 modes. I will probably just implement all in Ruby for now.
=end

#42 - 02/24/2011 03:46 PM - kstephens (Kurt Stephens)

=begin
Progress:

I've implemented gc_api.[ch] source patches for MRI 1.9, 1.8.7 and REE 1.8.7 2011.02 and moved the MRI reference.c code into a fork of Brian's ref gem:

https://github.com/kstephens/ref/tree/master-mri-gc_api

Patches can be applied to MRI and variants using "rake *:patch src_dir=XXX".

The tests under test/reference pass on the above MRI platforms.

The basic Reference::* API classes are as follows:

```
class Reference::Reference # (abstract)
def initialize(object, reference_queue = nil); ...
def object; ...
def referenced_object_id; ...
end
class Reference::WeakReference < Reference
class Reference::SoftReference < Reference
class Reference::HardReference < Reference # pure ruby
class Reference::ReferenceQueue # pure ruby under MRI w/ reference.c
def pop; ...
def shift; ...
def empty?; ...
end
```

This API could be the core to be supported on all platforms.

For MRI: It's a mix of C and Ruby code. MRI-specific methods are prefixed with _mri.

ReferenceQueue is implemented in MRI using a simple protocol; the notification and management is implemented
in reference.c. I cribbed Reference::SafeMonitor from Brian's Ref::SafeMonitor.

TODO:

- It just occurred to me that Reference::Reference is solely implemented in reference.c, I'll write a boilerplate reference.rb impl.
- try re-implementing the rest of Brian's Ref::* classes using the core Reference::* classes.
- Implement the core Reference::* classes in Rubinius.

Charles: can you review the API? Perhaps you can implement the Reference::* core classes in JRuby? Trim down the ReferenceQueue methods down to what can be implemented in JRuby.

Sorry this took a while...

-- Kurt

=end

#43 - 06/10/2011 04:23 AM - ko1 (Koichi Sasada)

- ruby -v changed from ruby 1.9.2p0 (2010-08-18 revision 29036) [x86_64-darwin10.4.0] to -

Hi,

Sorry for late response.

I can't find out the conclusion of this problem. Is it solved? Is
there summary of this thread?

(2010/12/18 1:57), Brian Durand wrote:

> Bug #4168: WeakRef is unsafe to use in Ruby 1.9
> http://redmine.ruby-lang.org/issues/show/4168
>
> Author: Brian Durand
> Status: Open, Priority: Normal
> Category: lib, Target version: 1.9.x
> ruby -v: ruby 1.9.2p0 (2010-08-18 revision 29036) [x86_64-darwin10.4.0]
>
> I've found a couple issues with the implementation of WeakRef in Ruby 1.9.
>
> 1. WeakRef is unsafe to use because the ObjectSpace will recycle object ids for use with newly allocated objects after the old objects have been garbage collected. This problem was not present in 1.8, but with the 1.9 threading improvements there is no guarantee that the garbage collector thread has finished running the finalizers before new objects are allocated. This can result in a WeakRef returning a different object than the one it originally referenced.
> 2. In Ruby 1.9 a Mutex is used to synchronize access to the shared data structures. However, Mutexes are not reentrant and the finalizers are silently throwing deadlock errors if they run while new WeakRefs are being created.
>
> I've included in my patch a reimplementation of WeakRef called WeakReference that does not extend Delegator. This code is based on the original WeakRef code but has been rewritten so the variable names are a little clearer. It replaces the Mutex with a Monitor and adds an additional check to make sure that the object returned by the reference is the same one it originally referred to. WeakRef is rewritten as a Delegator wrapper around a WeakReference for backward compatibility.
>
> I went this route because in some Ruby implementations (like Ruby 1.8), Delegator is very heavy weight to instantiate and creating a collection of thousands of WeakRefs will be slow and use up far too much memory (in fact, it would be nice to either backport this patch to Ruby 1.8 or the delegate changes from 1.9 to 1.8 if possible). I also don't really see the value of having weak references be delegators since is very unsafe to do anything without wrapping the call in a "rescue RefError" block. The object can be reclaimed at any time so the only safe method to be sure you still have it is to create a strong reference to the object at which point you might as well use that reference instead of the delegated methods on the WeakRef. It also should be simpler for other implementations of Ruby like Jruby or Rubinius to map native weak references to a simpler interface.
>
> Sample code with WeakReference
>
> ```
> orig = Object.new
> ref = WeakReference.new(orig)
> # ...
> obj = ref.object
> if obj
>   # Do something
> end
> ```
>
> I also have a version of the patch which just fixes WeakRef to work without introducing a new class, but I feel this version is the right way to go.
>
> Also included are unit tests for weak references but the test that checks for the broken functionality is not 100% reliable since garbage collection is not deterministic. I've also included a script that shows the problem in more detail with the existing weak reference implementation.
>
> ```
> ruby show_bug.rb 100000     # Run 100,000 iterations on the current implementation of WeakRef and report a
> ny problems
> ruby -I. show_bug.rb 100000 # Run 100,000 iterations on the new implementation of WeakRef and report any p
> roblems
> ```
>
> ---
>
> http://redmine.ruby-lang.org

--
// SASADA Koichi at atdot dot net

#44 - 06/26/2011 04:35 PM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

- Assignee set to shyouhei (Shyouhei Urabe)

#45 - 06/26/2011 04:44 PM - shyouhei (Shyouhei Urabe)

- Status changed from Assigned to Open

- Assignee deleted (shyouhei (Shyouhei Urabe))

I have to say this is an open problem. No one is responding comment # 43.

#46 - 06/26/2011 11:54 PM - rosenfeld (Rodrigo Rosenfeld Rosas)

I'm particularly worried if current documentation of WeakRef states it is currently unreliable until this can be fixed.

I mean, 1.9.3 should change the documentation to reflect this issue so people know they shouldn't use it, while keeping it for avoiding code breakage (which would be a bit broken/buggy anyway).

#47 - 07/01/2011 04:29 AM - kstephens (Kurt Stephens)

I have a working solution for MRI 1.8.7, 1.9 and REE. See https://github.com/kstephens/ref/tree/master-mri-gc_api. It is performant and reliable. The details of the APIs need to be formalized.

#48 - 07/25/2011 08:16 PM - naruse (Yui NARUSE)

- Tracker changed from Bug to Feature

This issue is not a bug, but needs a discussion about new ideal feature.

#49 - 03/18/2012 06:49 PM - nahi (Hiroshi Nakamura)

- Status changed from Open to Assigned

- Assignee set to nobu (Nobuyoshi Nakada)

Nobu, can we close this now?

#50 - 04/09/2012 09:51 PM - nobu (Nobuyoshi Nakada)

- Status changed from Assigned to Closed

I believe so.
If something goes wrong, please reopen this or file a new ticket.

#51 - 04/10/2012 01:02 AM - headius (Charles Nutter)

I'd like to clarify the fix here.

- ObjectSpace::WeakMap was added as a "proper" weak reference map
- WeakRef uses WeakMap instead of _id2ref, mapping the WeakRef instance weakly to the object

Correct?

#52 - 04/10/2012 02:36 PM - nobu (Nobuyoshi Nakada)

Correct.

Do you have any suggestion?

#53 - 04/17/2012 05:25 PM - headius (Charles Nutter)

The implementation looks fine to me.

I still believe that most interesting uses of WeakRef would be more efficient with support for a reference queue. I filed a bug to add reference queues to Ruby here: https://bugs.ruby-lang.org/issues/6309

I'd also like to see other reference types similar to JVM...I can file a bug for them if there's a chance they might be added:

- Soft references: not as weak as weak references, referred object is collected only if there's memory pressure or if it has only been softly referenced for some time. Useful for soft caches that age out old data. On OpenJDK, soft referenced objects are only collected during a full GC or if they have been only softly referenced for a certain amount of time multiplied by heap size.
- Phantom references: not traversible so they are less impact on GC, but still added to reference queue when their object is collected. Often used

to implement more controlled finalization (e.g. finalization by a user thread rather than a VM thread).

#54 - 04/17/2012 06:09 PM - headius (Charles Nutter)

A video of Bob Lee explaining the various JVM references, and why using finalizers is bad: http://www.youtube.com/watch?v=KTC0g14lmPc

Files

weakref.rb	5.2 KB	12/18/2010	bdurand (Brian Durand)
test_weakref.rb	2.39 KB	12/18/2010	bdurand (Brian Durand)
show_bug.rb	1.71 KB	12/18/2010	bdurand (Brian Durand)
weakref_patch.diff	3.95 KB	12/18/2010	bdurand (Brian Durand)
weakref.rb	5.12 KB	12/18/2010	bdurand (Brian Durand)
weakref.rb	4.97 KB	01/06/2011	bdurand (Brian Durand)
weakref_patch.diff	3.99 KB	01/06/2011	bdurand (Brian Durand)
weakref.rb	5.09 KB	01/11/2011	bdurand (Brian Durand)
weakref_patch.diff	4.08 KB	01/11/2011	bdurand (Brian Durand)