CommonRuby - Feature #8556

MutexedDelegator as a trivial way to make an object thread-safe

06/21/2013 10:27 PM - headius (Charles Nutter)

Status:	Rejected	
Priority:	Normal	
Assignee:	ko1 (Koichi Sasada)	
Target version:	Ruby 2.1.0	

Description

I propose adding MutexedDelegator as a simple way to wrap any object with a thread-safe wrapper, via existing delegation logic in delegate.rb.

Delegator provides a way to pass method calls through to a wrapped object. SimpleDelegator is a trivial implementation that just holds the object in an instance variable. MutexedDelegator would extend SimpleDelegator and only override initialize and method_missing as follows:

```
  class MutexedDelegator < SimpleDelegator
    def initialize(*)
      super
      @mutex = Mutex.new
    end

    def method_missing(m, *args, &block)
      target, mutex = self.__getobj__, @mutex
      begin
        mutex.lock
        target.__send__(m, *args, &block)
      ensure
        mutex.unlock
      end
    end
  end
```

The only changes here are:

- Mutex#lock and unlock logic wrapping the send
- No respond_to? check; I'm not sure why it's there to begin with, since if we're in method_missing the super() call will fail just like a normal method_missing failure anyway
- No backtrace manipulation. This does not work on JRuby and Rubinius anyway, and in this case I feel that the delegator should not hide itself, since there's real behavior change happening.

This is a trivial addition to stdlib that would make it simple to synchronize all calls to a given object in the same way as the JDK's Collections.synchronizedSet/Map/List calls.

Related issues:

Related to CommonRuby - Feature #8961: Synchronizable module to easily wrap m...	**Open**	**09/27/2013**

History

#1 - 06/22/2013 01:37 AM - headius (Charles Nutter)

A few updates...

- After experimenting, I realize that the respond_to? check is to trigger "private" error for private methods. It may also have something to do with respond_to_missing? methods.
- It is probably better to delegate the dispatching logic to the super impl (in Delegator).
- The name "SynchronizedDelegator" was suggested to me as being more indicative of the intent of this logic.

So I have a revised implementation:

```
class SynchronizedDelegator < SimpleDelegator
  def initialize(*)
    super
    @mutex = Mutex.new
  end
```

```
  def method_missing(m, *args, &block)
    begin
      mutex = @mutex
      mutex.lock
      super
    ensure
      mutex.unlock
    end
  end
end
```

#2 - 09/27/2013 07:59 PM - headius (Charles Nutter)

Any comments here? This would be pretty easy to add to delegate.rb for 2.1.

#3 - 09/27/2013 08:13 PM - simonx1 (Szymon Kurcab)

That would be a useful feature.
+1

#4 - 09/27/2013 08:35 PM - headius (Charles Nutter)

Similar in nature to the "synchronized" module method proposed in https://bugs.ruby-lang.org/issues/8961. I like that proposal as well, but it does not help the case where you have a concurrency-unsafe object in hand that you would like to make concurrency-safe.

Another commenter on Twitter suggested that this is not the best way to go about making an object thread-safe, and he's right. It would be better to use immutable collections or explicitly concurrency-friendly collections. However, this is a simple pattern that works for all types of objects and makes it possible to start writing better threaded Ruby code today.

I would also like to note that this is helpful in MRI too, since the bodies of Ruby methods can context switch at any time. MRI also needs a better mechanism for saying "this class's methods should only be executed by one thread at a time".

#5 - 09/28/2013 03:42 AM - headius (Charles Nutter)

Formatting issue... the "synchronized" proposal is in https://bugs.ruby-lang.org/issues/8961

#6 - 09/30/2013 10:40 PM - naruse (Yui NARUSE)

- Target version set to Ruby 2.1.0

#7 - 09/30/2013 11:53 PM - avdi (Avdi Grimm)

On Fri, Sep 27, 2013 at 6:59 AM, headius (Charles Nutter) <
headius@headius.com> wrote:

> I propose adding MutexedDelegator as a simple way to wrap any object with
> a thread-safe wrapper, via existing delegation logic in delegate.rb.

I think that's a wonderful idea. What do you think of the name
SynchronizedDelegator?

--
Avdi Grimm
http://avdi.org

I only check email twice a day. to reach me sooner, go to
http://awayfind.com/avdi

#8 - 10/01/2013 06:13 AM - headius (Charles Nutter)

SynchronizedDelegator is a better name, and the code should use Monitor instead of Mutex so it can be reentrant. I'll do that now.

#9 - 10/01/2013 07:23 AM - avdi (Avdi Grimm)

On Mon, Sep 30, 2013 at 5:13 PM, headius (Charles Nutter) <
headius@headius.com> wrote:

> and the code should use Monitor instead of Mutex so it can be reentrant.

I'm trying to think of a case in which this would matter. Since it's
wrapping another object, we don't have to worry about the case where a
synced method calls another synced method on self.

Hmmm... I guess there's the case where synced_obj.foo receives a block, and
someone calls synced_obj.bar within that block.

I only bring this up because Monitor introduces a (small?) performance hit
over Mutex.

--
Avdi Grimm
http://avdi.org

I only check email twice a day. to reach me sooner, go to
http://awayfind.com/avdi

#10 - 10/01/2013 08:02 AM - headius (Charles Nutter)

I implemented this and a simple test in https://github.com/ruby/ruby/pull/405

If approved, I can merge that or commit to trunk directly.

The performance impact of Monitor is a separate issue; Monitor should probably be implemented natively to get maximum performance. I'm considering doing that for JRuby as well. As you point out, reentrancy is needed for any code that might call out to a block which might call back in.

There's not a great deal we can do to speed up Monitor as it is currently written, but perhaps you could file a bug for that and we can see about improving things.

#11 - 10/01/2013 02:11 PM - nobu (Nobuyoshi Nakada)

- Description updated

#12 - 10/01/2013 04:55 PM - naruse (Yui NARUSE)

- Status changed from Open to Assigned

- Assignee set to ko1 (Koichi Sasada)

ko1 will write objection.

#13 - 10/02/2013 03:12 AM - headius (Charles Nutter)

naruse (Yui NARUSE) wrote:

> ko1 will write objection.

I look forward to reading that objection :-)

#14 - 10/22/2013 09:23 PM - headius (Charles Nutter)

Still waiting to read ko1's objection. I am prepared to commit a monitor-based delegator if we go forward.

#15 - 10/22/2013 11:55 PM - ko1 (Koichi Sasada)

Sorry for late.

Summary: I believe we need more experience before including this library as standard.

(1) Try gem first

Basically, libraries written in Ruby can be provided by a gem easilly.
We can prove the usefulness with real experiece by this approach.
In other words, we shouldn't provide any libraries without such proof.

(2) Misleading design

I'm afraid that this library introduce bugs under misunderstanding.

For example, people consider that this object is without worry about
synchronization, people may write the following program.

```
# In fact, *I* wrote this program first without any question!!

####
require 'delegate'
require 'monitor'
```

```
class SynchronizedDelegator < SimpleDelegator
  def initialize(*)
    super
    @monitor = Monitor.new
  end

  def method_missing(m, *args, &block)
    begin
      monitor = @monitor
      monitor.mon_enter
      super
    ensure
      monitor.mon_exit
    end
  end
end

sdel_ary = SynchronizedDelegator.new([0])

ary = [0]
m = Mutex.new

ts = (1..2).map{
  Thread.new{
    100_000.times{
      sdel_ary[0] += 1 # -> 1
      sdel_ary[0] -= 1 # -> 0

      m.synchronize{
        ary[0] += 1
        ary[0] -= 1
      }
    }
  }
}

ts.each{|t| t.join}
p sdel_ary #=> [40] # or something wrong result
p ary      #=> [0]
####
```

At first I see this result, I can't understand why.
Of course, this program is completely bad program.
It is completely my mistake.

But I think this design will lead such misunderstanding and bugs easily.

To avoid a such bug, I define the inc() and sub() method in Array.

```
####
class Array
  def inc; self[0] += 1; end
  def sub; self[0] -= 1; end
end

sdel_ary = SynchronizedDelegator.new([0])

ts = (1..2).map{
  Thread.new{
    100_000.times{
      sdel_ary.inc
      sdel_ary.sub
    }
  }
}

ts.each{|t| t.join}
p sdel_ary[0] #=> 200000
####
```

This works completely.

But a person who assumes sdel_ary is free from consideration about locking,
can write the following program:

```
####
class << sdel_ary
  def inc; self[0] += 1; end
  def sub; self[0] -= 1; end
end

ts = (1..2).map{
  Thread.new{
    100_000.times{
      sdel_ary.inc
      sdel_ary.sub
    }
  }
}

ts.each{|t| t.join}
p sdel_ary[0] #=> 229
####
```

This doesn't work correctly (different from the person expect).

I feel we can find other cases.

Maybe professional programmers about concurrency program can avoid such silly bugs.
But if we introduce it as standard library, I'm afraid they are not.

(3) Lock based thraed programming

This is my opinion. So it is weak objection for this proposal.

I believe lock based thread programming introduced many bugs.
(Synchronized) Queue or more high-level structures should be used.

Or use Mutex (or monitor) explicitly for fing-grain locking.
It bothers programmers, so programmers use other approachs such as Queue (I hope).

Summary:

Mainly, my objection is based on (1) and (2).
Concurrency is a very difficult theme.
I feel 2.1 is too early to include this feature.
At least, we need more experience about this feature to introduce.

I'm not against that professionals use this libarary.

#16 - 10/23/2013 04:07 AM - headius (Charles Nutter)

ko1 (Koichi Sasada) wrote:

> (1) Try gem first

We could certainly put this into thread_safe gem, which is now a dependency of Rails and pretty widely deployed as a result. I am not opposed to testing this more in the wild before incorporation into stdlib.

So the rest of this may be moot, but I'll proceed anyway.

> (2) Misleading design
>
> I'm afraid that this library introduce bugs under misunderstanding.
>
> For example, people consider that this object is without worry about
> synchronization, people may write the following program.

Someone else raised a concern about += and friends, but there's *no* way in a library to ever make those operations thread safe (actually, atomic). That's what the "atomic" gem provides.

The only way to ever make +=, ||=, and others be atomic in Ruby proper would be to change the way they're parsed and potentially add a method that could be called. But this is even unpredictable because for variables, there's still no way to do it atomically.

FWIW, Java's ++ and -- and += and friends are *also* not atomic.

I don't believe these features being non-atomic is a good enough justification to prevent the addition of a synchronized delegator. The sync delegator explicitly just makes individual method calls synchronized; and += and friends require multiple method calls.

> At first I see this result, I can't understand why.
> Of course, this program is completely bad program.
> It is completely my mistake.
>
> But I think this design will lead such misunderstanding and bugs easily.

But this is not possible to fix in current Ruby and all other languages I know don't guarantee any atomicity here either.

> To avoid a such bug, I define the inc() and sub() method in Array.

This is an appropriate way to do it, indeed. However, anyone else still doing += mess up the results. If you want atomic mutation of individual elements, we need an AtomicArray or similar.

> This works completely.
>
> But a person who assumes sdel_ary is free from consideration about locking,
> can write the following program:

This is perhaps a valid concern. SynchronizedDelegate could use a method_added hook to wrap new methods, however. Is it warranted?

```
class << SynchronizedDelegator
  def method_added(name)
    unsync_name = :"__unsynchronized_#{name}"
    alias_method unsync_name, name
    define_method name do |*args, &block|+  def method_missing(m, *args, &block)
      begin
        monitor = @monitor
        monitor.mon_enter
        __send__ unsync_name, args, block
      ensure
        monitor.mon_exit
      end
    end
  end
end
```

Or something like that.

> Maybe professional about concurrency program can avoid such silly bugs.
> But if we introduce it as standard library, I'm afraid they are not.

I don't claim this solution solves all problems, obviously. But it solves many of them. It is an incremental tool to help improve concurrency capabilities of Ruby.

> (3) Lock based thraed programming
>
> This is my opinion. So it is weak objection for this proposal.
>
> I believe lock based thread programming introduced many bugs.
> (Synchronized) Queue or more high-level structures should be used.
>
> Or use Mutex (or monitor) explicitly for fing-grain locking.
> It bothers programmers, so programmers use other approachs such as Queue (I hope).

Getting explicitly concurrency-friendly collections into stdlib would be great. But this was intended as a small step given that 2.1 is close to finished.

Another data point: Java for years has had java.util.Collections.synchronized{List,Map,Set} for doing a quick and easy wrapper around those collection types. Sometimes it's the best simple solution for making a collection thread-safe.

#17 - 02/06/2017 03:04 AM - ko1 (Koichi Sasada)

- Description updated

#18 - 02/06/2017 03:17 AM - ko1 (Koichi Sasada)

- Status changed from Assigned to Rejected

Sorry for long absence.

> The only way to ever make +=, ||=, and others be atomic in Ruby proper would be to change the way they're parsed and potentially add a

method that could be called. But this is even unpredictable because for variables, there's still no way to do it atomically.

inc/dec is only an example.

But this is not possible to fix in current Ruby and all other languages I know don't guarantee any atomicity here either.

I believe ruby should move to non-sharing way. Performance on this approach is not so good, but enough I believe.