Ruby - Feature #8961

Synchronizable module to easily wrap methods in a mutex

09/27/2013 08:23 PM - tobiassvn (Tobias Svensson)

Priority: Normal Assigne: Image:	Status: Open			
Assignee: Image: Second Se	Priority: Norma	I		
Target version: Description -begin Ipropose a Synchronizable mixin to easily wrap methods in a mutex which works together with Ruby 2.1's method name symbols returned from '((def))'. The Mixin adds a new '(((synchronized))') class method which would alias the referenced method and redefines the original method wrapped in a '((synchronized))' class method which would alias the referenced method and redefines the original method wrapped in a '((synchronized))' block. This is probably somewhat related and an alternative to gl8556. Proof of concept (I/ve used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module ClassMethods def synchronizable module diased writout_synchronization" alias method aliased, '*arge, sblock! monitor '.aynchronizable module ClassMethods def sonitor end end e	Assignee:			
Description -begin propose a Synchronizable mixin to easily wrap methods in a mutex which works together with Ruby 2.1's method name symbols returned from '(((dsynchronized)))' class method which would alias the referenced method and redefines the original method wrapped in a '(((synchronized) - and)))' block. The Mixin adds a new '(((synchronized)))' class method which would alias the referenced method and redefines the original method wrapped in a '(((synchronized) - and)))' block. This is probably somewhat related and an alternative to #8556. Proof of concept (I've used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module Synchronizable module ClassMethods def synchronizable motion' module ClassMethods def synchronized (method) alsa, method allased, method alsa, method allased, method and end	Target version:			
<pre>-begin lpropose a Synchronizable mixin to easily wrap methods in a mutex which works together with Ruby 2.1's method name symbols returned from (((def))). The Mixin adds a new (((synchronized)))' class method which would alias the referenced method and redefines the original method wrapped in a '((synchronized))' block. This is probably somewhat related and an alternative to #8556. Proof of concept (I've used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module Synchronizable module Synchronizable module Synchronizable module GlassMethods def synchronizable module GlassMethods def synchronizable module GlassMethods def fine_method method do (*args, sblock) monitor :gynchronization" aliase - "filmethod_winter.get & sblock) monitor []= Monitor.new end end end end end end end def self.included (base) base, extend (ClassMethods) end end end end end end end end end end</pre>	Description			
The Mixin adds a new ''((synchronized))' class method which would alias the referenced method and redefines the original method wrapped in a '((synchronize do end))' block. This is probably somewhat related and an alternative to (#8556. Proof of concept (I've used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module Synchronizable module ClassMethods def synchronizad(method) aliased = ''(method). without_synchronization* aliase_method aliased, method def ine_method method do (*args, sblock) monitor.synchronize do end def monitor @monitor[]= Monitor.new end def self.included(base) base.extend(classMethods) end end class Foo include Synchronizable synchronizable synchronizable synchronizable mod end class Foo include Synchronizable synchronizable mod end class Foo include Synchronizable synchronizable synchronizable synchronize do = = = = = = = = = = = = = = = = = =	=begin I propose a Synchronizable mixin to easily wrap methods in a mutex which works together with Ruby 2.1's method name symbols returned from '(({def}))'.			
This is probably somewhat related and an alternative to #2556. Proof of concept (Ive used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module Synchronizable module ClassMethods def synchronized(method) aliased = :"#(method)_without_synchronization" alias_method aliased, method def ine_method method do [*args, █] monitor.synchronize do end end def monitor f = Monitor.new end def self.included(base) base.extend(classMethods) end end class Foo include Synchronizable synchronizable synchronizable end end end end end end end end end en	The Mixin adds a new '(({synchronized}))' class method which would alias the referenced method and redefines the original method wrapped in a '(({synchronize do end}))' block.			
Proof of concept (I've used Monitor here so potential users won't have to worry about reentrancy): require 'monitor' module Synchronizable module ClassMethods def synchronized(method) aliased = 'ff(method), without_synchronization" aliased = 'ff(method), without_synchronization" end def monitor @monitor = Monitor.new end def self.included(base) base.extend(ClassMethods) end end class Foo include Synchronizable synchronizable synchronizable synchronizable synchronizable mod end end end Plated to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Bejected	This is probably somewhat related and an alternative to <u>#8556</u> .			
require 'monitor' module Synchronizable module ClassMethods def synchronizad(method) aliased =:'#f(method)_without_synchronization" alias_method aliased, method def fine_method method do [*args, █] monitor.aynchronized do end end end def monitor @monitor []= Monitor.new end def self.included(base) base.extend(ClassMethods) end end end class Foo include Synchronizable synchronizable synchronizable synchronizable synchronizable method for # * end	Proof of concept (I've used Monitor here so potential users won't have to worry about reentrancy):			
<pre>module Synchronizable module ClassMethods def synchronizationted(method) aliased = :*#(method) without_synchronization" alias_method aliased, method define_method method do [*args, █] monitor.synchronize do end end end end end end def monitor @monitor []= Monitor.new end def self.included(base) base.extend(ClassMethods) end end class Foo include Synchronizable synchronizable synchronized def bar # end E</pre>	require 'monitor'			
end end def monitor @monitor []= Monitor.new end def self.included(base) base.extend(ClassMethods) end end class Foo include Synchronizable synchronized def bar # end end end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an	<pre>module Synchronizable module ClassMethods def synchronized(method) aliased = :"#{method}_without_synchronization" alias_method aliased, method define_method method do *args, █ monitor.synchronize dosend(aliased, *args, █) end end</pre>			
<pre>def monitor @monitor = Monitor.new end def self.included(base) base.extend(ClassMethods) end end class Foo include Synchronizable synchronized def bar # end end end end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected</pre>	end			
def self.included (base) base.extend (ClassMethods) end end class Foo include Synchronizable synchronized def bar # end end end end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	<pre>def monitor @monitor = Monitor.new end</pre>			
end class Foo include Synchronizable synchronized def bar # end end =end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	<pre>def self.included(base) base.extend(ClassMethods) end</pre>			
class Foo include Synchronizable synchronized def bar # end end =end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	end			
synchronized def bar # end end =end Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	class Foo include Synchronizable			
Related issues: Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	<pre>synchronized def bar # end end =end</pre>			
Related to Ruby - Feature #8556: MutexedDelegator as a trivial way to make an Rejected	Related issues:			
- · · ·				

History

#1 - 09/27/2013 08:40 PM - headius (Charles Nutter)

I would like to see this in 2.1, as a standard Module method. The fact that "def" returns the method name now makes this really easy.

I think this would need to be implemented natively to work, however. The prototype above has a key flaw: there's no guarantee that only one Monitor will be created, so two threads could execute the same method at the same time, synchronizing against different monitors. Putting the synchronized wrapper into C code would prevent a potential context switch when first creating the Monitor instance (or it could simply use some other mechanism, such as normal Object monitor synchronization in JRuby).

This feature is similar to an extension in JRuby called JRuby::Synchronized that causes all method lookups to return synchronized equivalents.

Combined with https://bugs.ruby-lang.org/issues/8556 this could go a very long way toward giving Ruby users better tools to write thread-safe code.

#2 - 09/27/2013 09:45 PM - tobiassvn (Tobias Svensson)

Having this as a method on Module directly would of course be ideal. However, I believe the mutex/monitor used should still be exposed as a private method so it can be used without the 'synchronized' method.

#3 - 09/28/2013 03:45 AM - headius (Charles Nutter)

tobiassvn (Tobias Svensson) wrote:

Having this as a method on Module directly would of course be ideal. However, I believe the mutex/monitor used should still be exposed as a private method so it can be used without the 'synchronized' method.

Maybe. I don't like the idea of exposing this mutex/monitor, since it could be modified or locked and never released. I would be more in favor of a "tap" form that synchronizes against the same internal monitor, similar to Java's "synchronized" keyword.

obj.synchronized { thread-sensitive code here }

That would also open up the possibility of using a lighter-weight internal mutex/monitor rather than the rather heavy-weight Ruby-land version.

#4 - 09/28/2013 09:23 AM - nobu (Nobuyoshi Nakada)

headius (Charles Nutter) wrote:

Maybe. I don't like the idea of exposing this mutex/monitor, since it could be modified or locked and never released. I would be more in favor of a "tap" form that synchronizes against the same internal monitor, similar to Java's "synchronized" keyword.

obj.synchronized { thread-sensitive code here }

Use MonitorMixin.

#5 - 10/01/2013 02:14 PM - nobu (Nobuyoshi Nakada)

- Description updated

#6 - 10/02/2013 02:13 AM - headius (Charles Nutter)

nobu (Nobuyoshi Nakada) wrote:

headius (Charles Nutter) wrote:

Maybe. I don't like the idea of exposing this mutex/monitor, since it could be modified or locked and never released. I would be more in favor of a "tap" form that synchronizes against the same internal monitor, similar to Java's "synchronized" keyword.

obj.synchronized { thread-sensitive code here }

Use MonitorMixin.

Yeah, that's not a bad option from a pure-Ruby perspective. We could add "synchronized" to classes that include MonitorMixin, perhaps?

added to monitor.rb:

module MonitorMixin
module ClassMethods
def synchronized(method)
aliased = :"#{method}_without_synchronization"
alias_method aliased, method

```
define_method method do |*args, &block|
  mon_enter
  begin
```

```
__send__(aliased, *args, &block)
ensure
mon_exit
end
end
end
```

end

def self.included(base) base.extend(ClassMethods) end end

class Foo include MonitorMixin

synchronized def bar # ... end end

...

My suggestion to have it be native on Module opened up the possibility of implementing it in a faster, native way. MonitorMixin has a very large perf hit on all impls right now, but especially MRI. See my benchmarks in https://github.com/ruby/ruby/pull/405#issuecomment-25417666

#7 - 10/02/2013 10:15 PM - tobiassvn (Tobias Svensson)

I suppose if this is being added to MonitorMixin it should probably be in Mutex_m as well?

#8 - 10/03/2013 06:52 AM - headius (Charles Nutter)

tobiassvn (Tobias Svensson) wrote:

I suppose if this is being added to MonitorMixin it should probably be in Mutex_m as well?

I don't think so, since a Mutex is not reentrant and what we want is monitor semantics for #synchronized.

#9 - 12/23/2021 11:43 PM - hsbt (Hiroshi SHIBATA)

- Project changed from 14 to Ruby