

- [RSS](#)

<input type="text" value="Search"/>
» RSS ▼

- [Blog](#)
- [Archives](#)

Active Support中的callback

May 12th, 2012 | [Comments](#)

在rails中提供了很多callback，model里有before_save, after_save等，controller里有before_filter, after_filter等，他们都是基于activesupport的callback来实现的，callback主要功能实现的代码也在activesupport中，其他只是一层封装，所以研究了下activesupport中callback的实现，记录一下。

从一个简单的用例开始吧。

```
1 # coding: utf-8
2
3 require 'active_support/callbacks'
4
5 class Record
6   include ActiveSupport::Callbacks
7   define_callbacks :save
8
9   def save
10     run_callbacks :save do
11       puts "- save"
12     end
13   end
14
15   set_callback :save, :before, :saving_message
16   def saving_message
17     puts "saving..."
18   end
19 end
```

```

18   return false
19 end
20
21 set_callback :save, :after do |object|
22   puts "saved"
23 end

```



```

28 person.save
29
30 # ---output---
31 # saving...
32 # - save
33 # saved

```

define_callbacks

先来看看define_callbacks

```

1 def define_callbacks(*callbacks)
2   config = callbacks.last.is_a?(Hash) ? callbacks.pop : {}
3   callbacks.each do |callback|
4     class_attribute "_#{callback}_callbacks"
5     send("_#{callback}_callbacks=", CallbackChain.new(callback, config))
6     __define_runner(callback)
7   end
8 end

```

首先提取最后一个hash作为config，然后为每个callback定义class_attribute，用class_attribute定义类变量，子类继承时会拥有自己的一份，修改不会影响父类。然后通过send命令把新建了CallbackChain对象赋予了刚刚定义的类变量。

```

1 class CallbackChain < Array #:nodoc:#
2   attr_reader :name, :config
3
4   def initialize(name, config)
5     @name = name
6     @config = {
7       :terminator => "false",
8       :rescuable => false,

```

```

9      :scope => [ :kind ]
10    }.merge(config)
11  end
12
13  #...

```



一月 三月 四月

[Close](#)

[18 captures](#)

25 八月 12 - 20 三月 16

20
2015 2016 2017

[Help](#)

回到define_callbacks，继续看__define_runner，顺便说一下，以__开头的都是内部使用的。

```

1 def __define_runner(symbol) #:nodoc:
2   runner_method = "_run_#{symbol}_callbacks"
3   unless private_method_defined?(runner_method)
4     class_eval <<-RUBY_EVAL, __FILE__, __LINE__ + 1
5       def #{runner_method}(key = nil, &blk)
6         self.class.__run_callback(key, :#{symbol}, self, &blk)
7       end
8       private :#{runner_method}
9     RUBY_EVAL
10  end
11 end

```

__define_runner定义了一个名字叫”_run_#{symbol}_callbacks”的私有方法，本例中也就是”_run_save_callbacks”，该方法接受2个参数key和block，方法很简单，直接调用自己的类方法__run_callback。

set_callback

```

1 def set_callback(name, *filter_list, &block)
2   mapped = nil
3
4   __update_callbacks(name, filter_list, block) do |target, chain, type, filters, options|
5     mapped ||= filters.map do |filter|
6       Callback.new(chain, filter, type, options.dup, self)
7     end
8
9     filters.each do |filter|
10      chain.delete_if {|c| c.matches?(type, filter) }
11    end
12
13    options[:prepend] ? chain.unshift(*(mapped.reverse)) : chain.push(*mapped)

```

```

14
15     target.send("_#{name}_callbacks=", chain)
16 end
17 end

```



```

1 def __update_callbacks(name, filters = [], block = nil) #:nodoc:
2   type = filters.first.in?([:before, :after, :around]) ? filters.shift : :before
3   options = filters.last.is_a?(Hash) ? filters.pop : {}
4   filters.unshift(block) if block
5
6   ([self] + ActiveSupport::DescendantsTracker.descendants(self)).reverse.each do |target|
7     chain = target.send("_#{name}_callbacks")
8     yield target, chain.dup, type, filters, options
9     target.__reset_runner(name)
10  end
11 end

```

首先检查`filters`的一个元素是不是`[:before, :after, :around]`中的一个，否则就默认为`:before`，赋予`type`。

然后检查`filters`的最后一个元素是不是`Hash`，将其赋给`options`。

如果定义了`block`，就把`block`放到`filters`的第一个。

`[self] + ActiveSupport::DescendantsTracker.descendants(self)`这句是把自己和继承自己的子类组成一个数组，关于`ActiveSupport::DescendantsTracker`可以看看我另外一篇文章[Rails源码——ActiveSupport::DescendantsTracker](#)。

倒置该数组循环，先通过`send`调用先前用`class_attribute`定义的`__save_callbacks`，获取到`CallbackChain`，通过`yield`回到`__set_callbacks`。

```

1 mapped ||= filters.map do |filter|
2   Callback.new(chain, filter, type, options.dup, self)
3 end

```

想想对于本例此时`filters`里包含什么呢？根据前面的代码，可以分析出，本例有2个`set_callback`，第一次`filters`里应该是一个`symbol`，`:saving_message`，第二次是一个`block`，也就是一个`Proc`对象。

在这个`map`迭代器里，创建了`Callback`对象。

```

1 class Callback #:nodoc:#
2   @@_callback_sequence = 0
3
4   attr_accessor :chain, :filter, :kind, :options, :per_key, :klass, :raw_filter
5 end

```



[18 captures](#)

25 八月 12 - 20 三月 16

一月 三月 四月

[Close](#)

20
2015 2016 2017

[Help](#)

```

10   @per_key          = options.delete(:per_key)
11   @raw_filter, @options = filter, options
12   @filter           = _compile_filter(filter)
13   @compiled_options = _compile_options(options)
14   @callback_id       = next_id
15
16   _compile_per_key_options
17 end
18
19 #...
20 end

```

Callback的构造函数事情就多一点，`normalize_options`通过`Array.wrap`将`options`中的参数包装成一个数组。

```

1 def normalize_options!(options)
2   options[:if] = Array.wrap(options[:if])
3   options[:unless] = Array.wrap(options[:unless])
4
5   options[:per_key] ||= {}
6   options[:per_key][:if] = Array.wrap(options[:per_key][:if])
7   options[:per_key][:unless] = Array.wrap(options[:per_key][:unless])
8 end

```

本例中我们没有用到这些参数，所以可以先不关注。

随后看`_compile_filter`，这个方法比较重要，从这个方法可以看出，`Callback`支持多少类型的参数来作为`filter`。

```

1 def _compile_filter(filter)
2   method_name = "_callback_#{@kind}_#{@next_id}"
3   case filter
4   when Array
5     filter.map {|f| _compile_filter(f)}

```

```
6  when Symbol
7    filter
8  when String
9    "(#{filter})"
10 when Proc
11   ...

INTERNET ARCHIVE
WayBackMachine
http://kenbeit.com/blog/2012/05/12/callback-in-active-support/ Go
18 captures
25 八月 12 - 20 三月 16
一月 三月 四月
20
2015 2016 2017 Close Help

16  @klass.send(:define_method, "#{method_name}_object") { filter }
17
18  _normalize_legacy_filter(kind, filter)
19  scopes = Array.wrap(chain.config[:scope])
20  method_to_call = scopes.map{ |s| s.is_a?(Symbol) ? send(s) : s }.join("_")
21
22  @klass.class_eval <<-RUBY_EVAL, __FILE__, __LINE__ + 1
23    def #{method_name}(&blk)
24      #{method_name}_object.send(:#{method_to_call}, self, &blk)
25    end
26  RUBY_EVAL
27
28  method_name
29 end
30 end
```

首先定义了一个方法名字符串，本例中是”_callback_before_#{next_id}“，”_callback_after_#{next_id}“，next_id是返回的是一个不断自增的整数，为了使方法名不重复。

可以看到filter支持Array，Symbol，String，Proc，和任何Object，我们一个个来看看。

Array和Symbol很简单，是Array就递归循环处理，是Symbol不做处理。

我被String的括号迷惑了几分钟，后来发现还蛮简单的，这里最后返回的结果都是字符串，因为最后都是通过class_eval来处理的。String类型的参数，可以直接用ruby code，而不只是方法名的字符串形式，所以像这样的filter也是支持的。


```
set_callback :save, :before, "puts 'validating data...'"
```

为了确保String的语句的正常执行及优先级，所以加了括号。

Proc的处理也很简单，先通过define_method，为Record定义了名字是_callback_after_#{next_id}的方法，方法的逻辑就是你写的block，然后判断这个block是否有参数，根据参数个数把self(运行时就是Record的实例)，和一个新的Proc对象拼接 method_name后面。

当参数是其他对象时就复杂了，继续讲下去可能有点迷茫，先拿个用object最为filter例子来看看，再继续往后讲比较好。

```
1 class Audit
2   def before(caller)
3
4
5
6
7     puts "Audit: before_save is called"
8   end
9
10  def save(caller)
11    puts "Audit: save is called"
12  end
13 end
14
15 class Account
16   include ActiveSupport::Callbacks
17
18   define_callbacks :save
19   #define_callbacks :save, :scope => [:kind, :name]
20   #define_callbacks :save, :scope => [:name]
21   set_callback :save, :before, Audit.new
22
23   def save
24     run_callbacks :save do
25       puts 'save in main'
26     end
27   end
28 end
29
30 Account.new.save
```



你可以通过取消不同的注释来看看最终的输出结果有何不同。

回到`_compile_filter`，我们继续看源代码。

```
1 @klass.send(:define_method, "#{method_name}_object") { filter }
2
3 _normalize_legacy_filter(kind, filter)
4 scopes = Array.wrap(chain.config[:scope])
5 method_to_call = scopes.map{ |s| s.is_a?(Symbol) ? send(s) : s }.join("_")
6
```

```

7 @class.class_eval <<-RUBY_EVAL, __FILE__, __LINE__ + 1
8   def #{method_name}(&blk)
9     #{method_name}_object.send(:#{method_to_call}, self, &blk)
10  end
11 RUBY_EVAL
12

```



通过define_method为Record定义了一个#{method_name}_object的方法，该方法返回inter对象本身。

_normalize_legacy_filter我就跳过了，这个应该这是向后兼容的处理。

然后到了chain.config[:scope]，你现在可以往前看看，创建CallbackChain时默认scope的值是多少了，应该是[:kind]。

scope可选的是:kind和:name的一种或者其组合。

通过scopes的组合最后得到一个方法名，在我给的例子中，根据你选择的scope可能是before，before_save，或者是save。

最后通过class_eval定义了个方法，该方法最后委托给filter对象本身，调用根据scopes转换成的方法。

离开这个方法，回到Callback的构造函数，接着是_compile_options和_compile_per_key_options，这2个方法处理的都是同一件事，就是可以通过if和unless来为你的callback增加执行条件判断，不同的是_compile_per_key_options是预判断一次，所以性能会更好。before_filter的only和except就是用per_key来定义的条件。

```

1 def _compile_options(options)
2   conditions = ["true"]
3
4   unless options[:if].empty?
5     conditions << Array.wrap(_compile_filter(options[:if]))
6   end
7
8   unless options[:unless].empty?
9     conditions << Array.wrap(_compile_filter(options[:unless])).map {|f| "!#{f}" }
10  end
11
12  conditions.flatten.join(" && ")
13 end

```

可以看到其内部仍然是调用_compile_filter来处理条件，所以if和unless也可以是Array, Symbol, String, Proc, Object。最后返回值类似与ture && foo_method && _callback_before_100这样的条件字符串，后面的eval会用到。


```

1 def _compile_per_key_options
2   key_options = _compile_options(@per_key)
3
4   @klass.class_eval <<-RUBY_EVAL, __FILE__, __LINE__ + 1

```



[18 captures](#)

25 八月 12 - 20 三月 16

一月 三月 四月

[Close](#)

20
2015 2016 2017

[Help](#)

首先调用`_compile_options`来完成`per_key`的处理，然后定义了一个`”_one_time_conditions_valid_xxx”`方法，后面预处理会用到。

离开`Callback`的构造方法，回到`set_callback`，继续往下：

```

1 filters.each do |filter|
2   chain.delete_if {|c| c.matches?(type, filter) }
3 end
4
5 options[:prepend] ? chain.unshift(*(mapped.reverse)) : chain.push(*mapped)
6
7 target.send("_#{name}_callbacks=", chain)

```

从`CallbackChain`中找到相同的`filter`，并删除。然后将新生成的`Callback`对象数组放入`chain`中。

`prepend`参数表示把新的`Callback`放到`CallbackChain`的最前面，默认是往后`push`。

最后再把`chain`赋值回`target`的类变量。

run_callbacks

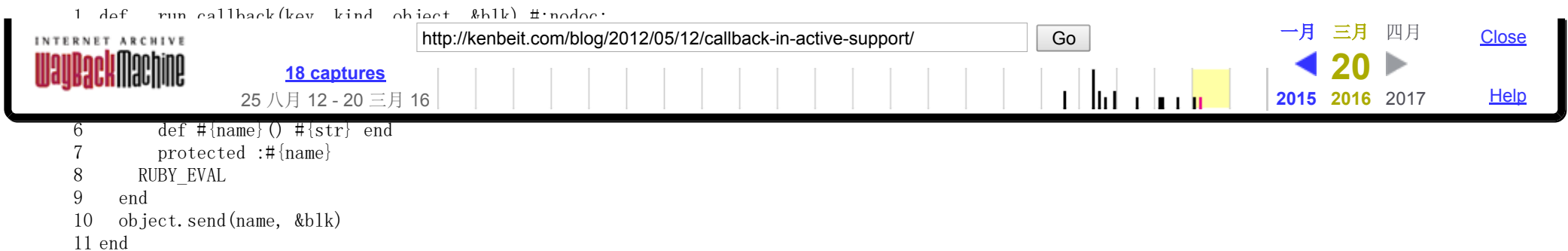
定义都完成了，现在就到了执行了，我们调用了`Record`实例的`save`方法，看到`before`和`after`的`callback`都执行了。来看看`run_callbacks`都干了什么。

```

1 def run_callbacks(kind, *args, &block)
2   send("_run_#{kind}_callbacks", *args, &block)
3 end

```

通过send，调用”_run_save_callbacks”方法。该方法在define_callbacks时定义，如果你忘记了，可以到最前面再看看。_run_save_callbacks方法调用了类方法self.class.__run_callback(key, :save, self, &blk)。



利用__callback_runner_name生成一个方法名

```
1 def __callback_runner_name(key, kind)
2   "__run__#{self.name.hash.abs}__#{kind}__#{key.hash.abs}__callbacks"
3 end
```

很简单通过类名的hash值和key的hash值生成了一个方法名。key这个参数在你使用了per_key时，才需要传入，我们这里没有用到，所以它是nil。如果用了per_key，会为每一个key生成一个方法，里面包含了条件预处理的判断结果，不需要每次调用去if和unless的逻辑。

回到__run_callback，检查object是否已经有了name对应的方法，有就直接调用，从这里可以看到只有第一次调用是会动态生成方法，后面就会调用真是存在的方法，这样可以提高性能。

object从_save_callbacks中取得CallbackChain，调用compile方法，得到出方法的逻辑。

```
1 def compile(key=nil, object=nil)
2   method = []
3   method << "value = nil"
4   method << "halted = false"
5
6   each do |callback|
7     method << callback.start(key, object)
8   end
9
10  if config[:rescuable]
```

```
11 method << "rescued_error = nil"
12 method << "begin"
13 end
14
15 method << "value = yield if block_given? && !halted"
16
```



http://kenbeit.com/blog/2012/05/12/callback-in-active-support/

Go

18 captures

25 八月 12 - 20 三月 16

一月 三月 四月

[Close](#)

20
2015 2016 2017

[Help](#)

```
21 end
22
23 reverse_each do |callback|
24   method << callback.end(key, object)
25 end
26
27 method << "raise rescued_error if rescued_error" if config[:rescuable]
28 method << "halted ? false : (block_given? ? value : true)"
29
30 method.compact.join("\n")
31 end
```

compile最后返回的是字符串，因为最后是交给class_eval去动态生成方法。CallbackChain是继承自数组的，所以有数组的方法，数组的元素就是Callback对象。

我们看到compile中依此调用了callback对象的start方法。代码太长了，我还是贴github的源代码地址吧。。。哦，我早该这么做了。 [start](#)

start方法只用before和around，也就是处理前置过滤器。

第一行就是per_key的处理，如果"_one_time_conditions_valid_#{@callback_id}"返回false，表示不需要执行callback，否则就是要执行。

如果不用per_key，直接用if和unless，就会在后面用到@compiled_options，这个实例变量包含了if和unless的条件判断字符串，那么每次执行都要判断一次，效率就比较低了。

代码看上去挺复杂的，抛开其他部分，看result = #{@filter}，还记得@filter是什么吗，本例中是saving_message的symbol，也就是在这里执行的我们的saving_message的代码。

顺便说一下config[:terminator]，这个参数可以给你一次机会来决定是否终止后面的callback，假如你有一个4个before filter，当第一个不满足某个条件是，你希望其他几个就不要执行了，你就可以用它。在define_callbacks时，你可以用它来定义一些条件判断，从而达到你的目的，具体使用可以直接看API文档。

around的部分很搞脑子，有几个来回yield，这里就先不讲了，后面我会给个我从代码里了解的最简单的around的实现。

回到compile方法，我们看到了config[:rescuable]，这个参数是给你寄回是否捕获异常的，默认情况下如果其他callback或者最终代码抛出了异常，剩下的callback就不执行了，通过设定该参数，可以替你暂时捕获异常，使其他callback顺利完成，然后再把异常抛出。

又是关键的一句method << "value = yield if block_given? && !halted"，这句的yield就是返回到你自已写run_callbacks的block中执行你自已的业务逻辑



这里就简单多了，很容易看懂，不解释了。

回到__run_callback，返回compile后的字符串，在class_eval中动态定义了该方法，最后通过send调用。

终于，callback顺利的可以运行了。哦，耶。

reset_callbacks和skip_callback

除了define_callbacks，set_callback，run_callbacks，剩下的就是reset_callbacks和skip_callback，从名字就可以看出是用来重置和跳过callback的。

这2个就不讲了，代码比较简单，如果你到现在为止的都明白了，那自己看一下这个方法也没什么问题。

结束语

看源代码真累，写出来更累，花了我一下午。你能看到这里也算是给面子了，希望对你有用。

看了元编程的书，也看了一些流行框架的源码，所谓元编程，其实就是那几招，关键看你会不会用，想不想的到要用，和怎么用。

这就好像以前数学题一样，有些题你解不出来，别人会，其实说知识点还是那些，只不过别人比你熟练，别人比你更会灵活组织和运用，所以别人最后答出来了。

编程也一样，光知道用法没用的，你要通过长时间使用后总结的经验和多阅读高手的代码，体会其用意后慢慢的转化为自己的能力。

附录

对around有兴趣的人，这是我简单总结的一个小例子，实际上around最后也是处理成这样的，以下是代码：

```
1 class Record
2
3   def around
```

```
4   p "before"
5   yield
6   p "after"
7 end
8
```



一月 三月 四月

[Close](#)

[18 captures](#)

25 八月 12 - 20 三月 16

20
2015 2016 2017

[Help](#)

```
14
15 def run
16   value = nil
17   internal do
18     value = yield
19   end
20 end
21
22 def save
23   run do |a|
24     p "save"
25   end
26 end
27
28 end
29
30 Record.new.save
```

Posted by kenshin54 May 12th, 2012 [callback](#), [ruby on rails](#)

[Tweet](#)

« [rspec 笔记](#) [Flush DNS cache in Mac OS](#) »

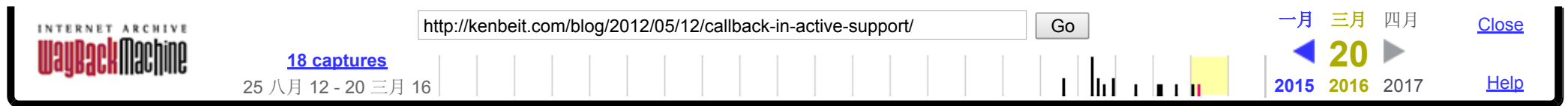
Comments

We were unable to load Disqus. If you are a moderator please see our [troubleshooting guide](#).

Categories

- [Mac](#), [callback](#), [javascript](#), [linux](#), [octopress](#), [rspec](#), [ruby on rails](#), [shell](#)

Recent Posts



- [mac在终端切换网络设置](#)

GitHub Repos

- [popline](#)

Popline is an HTML5 Rich-Text-Editor Toolbar

- [crane](#)

A mini linux container.

- [blog](#)

- [devlang](#)

My implementation of devlang.

- [logbin](#)

A lightweight centralized logging utility.

- [powersql](#)

A lightweight ActiveRecord DSL extension.

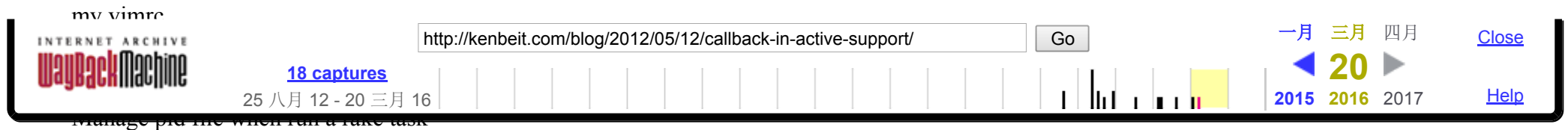
- [jquery-koala](#)

Koala is a jQuery plugin that capture user's continuous input and execute callbacks of keyboard events delayed.

- [alfred2-rubygems-workflow](#)

Alfred 2 Workflow for search and install rubygems

- [vimfiles](#)



- [top_notify](#)

TopNotify use to process notification stream from TOP.

- [am_alipay](#)

[@kenshin54](#) on GitHub

Latest Tweets

- Status updating...

[Follow @kenshin54](#)



Copyright © 2014 - kenshin54 - Powered by [Octopress](#)