Advanced Rails Recipes

Recipe 60

Cache Data Easily

By Mark Bates (http://www.markbates.com)
Mark is currently the director of architecture for Helium (http://helium.com). He spends his days fighting the establishment and wishing he'll be called up as the next front man for Van Halen. In addition to knowing the true meaning of Arbor Day, Mark also knows who let the dogs out, where the beef is, and who shot J.R.

Problem

You need a cheap and easy way to speed up slow parts of your application by keeping the results of calculations, renderings, or database calls around for subsequent requests.

Ingredients

- Rails 2.1 or higher
- Optionally, the memcached daemon.[105] On a Mac we install it like this:

  ```
  $ sudo port install memcached
  ```

  There is also a port for memcached on Windows.[106]

Solution

Rails 2.1 gives us four cache storage mechanisms right out of the gate—memory, file, DRb, and memcached—with minimal configuration in the environment file:

```
ActionController::Base.cache_store = :memory_store
ActionController::Base.cache_store = :file_store, "/path/to/cache/directory"
ActionController::Base.cache_store = :drb_store, "druby://localhost:9192"
ActionController::Base.cache_store = :mem_cache_store, "localhost"
```

Action and fragment caching now both transparently use the currently configured cache store. Nothing has changed in how you use them—simply in how you configure them. This is certainly good news, but (as they say) there's more! It turns out you can cache anything you want.

105.http://www.danga.com/memcached
106.http://jehiah.cz/projects/memcached-win32/

For our example, let's imagine we have a little online store with a marketing mind of its own. It can tally up all the books, T-shirts, and coffee mugs you've already purchased, and it can recommend other goodies you might like. The trouble is, the database query to do that is sort of expensive, despite our best optimization efforts. We need a cache.

The cache store API is really easy to use—it's basically like a hash. The best way to see how it works is in the console:

```
$ ruby script/console
>> products = Product.related_products("fred")
=> [#<Product id: 17, ...]

>> ActionController::Base.cache_store.write("fred", products)
=> [#<Product id: 17, ...]

>> ActionController::Base.cache_store.read("fred")
=> [#<Product id: 17, ...]

>> ActionController::Base.cache_store.delete("fred")
=> [#<Product id: 17, ...]

>> products = Product.related_products("barney")
>> ActionController::Base.cache_store.fetch("barney") { products }
=> [#<Product id: 21, ...]
```

The write method takes a key and value, and the read and delete methods take just the key. Finally, the fetch method does dual duty to perform "just-in-time" caching. When it doesn't find your key in the cache, it'll run the block, save its results into the cache, and give them back to you.

Let's go ahead and put this to use in our ProductsController and introduce a couple handy new things while we're at it:

```
def show
  @product = Product.find(params[:id])

  user_id = session[:user_id]
  @related_products = cache(['related_products', user_id],
                            :expires_in => 15.minutes) do
    Product.related_products(user_id)
  end

end
```

In a controller or view, you can simply use the cache convenience method. It wraps fetch but performs the caching only if caching is enabled. We've passed in an array for the key, which will get expanded into a namespaced key of controller/related_products/fred, for example. We've also used the :expires_in option[107] to invalidate this cache entry fifteen minutes from now.

So, here's what happens: the first time a particular user views a product, the heavyweight database query is run to haul in her related products. Each subsequent time the user views a product, we get a cache hit on the related products for a significant performance boost. If a particular user doesn't show a product within fifteen minutes, her related products are automatically kicked out of the cache.

To test this in development, you'll need to enable caching by updating your development.rb environment file as follows:

```
config.action_controller.perform_caching = true
```

Restart your app, and you're off to the races!

Discussion

memcached is by far the preferred store for this kind of stuff. The other options are there, but if you have the opportunity to use memcached, it's just about always the right answer. The nice thing about this new cache store API is it's all transparent. By default it's using the memory_store cache store, which is a great option in your development environment. In production, however, you can switch to using the memcached store by adding the following to your production.rb environment file:

Download CheapEasyCaching/config/environments/production.rb

```
config.cache_store = :mem_cache_store, 'localhost'
```

Then on your production box you'll need to fire up the memcached daemon:

```
$ memcached -vv
```

If you'd prefer not to use the currently configured cache store, you can instead instantiate your own store:

```
ActiveSupport::Cache.lookup_store(:drb_store, "druby://localhost:9192")
```

Using this method makes it very easy to segregate your caches either to use different types of cache stores for particular situations or to better namespace them.

107.The :expires_in option works only with the memcached store, as of this writing.

We could have buried the caching calls back in the related_products method of our Product model. But when you cache at the model layer, you don't have an easy way of sidestepping the cache to get fresh data. The caching mechanism built into Rails is intended to be used in controllers or views.

Also See

If you need a caching solution with a few more bells and whistles (or you're not using Rails 2.1 yet), see Recipe 63, *Cache Up with the Big Guys*, on page 307.