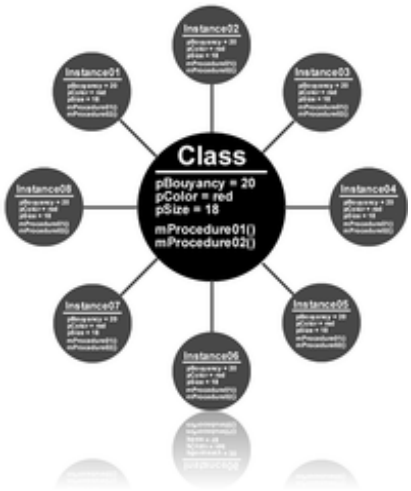
Object-Oriented Value Caching

Posted At : May 14, 2009 4:30 PM | Posted By : Jon Hartmann
Related Categories: ColdFusion, Examples & Tutorials



Like every other developer out there, I often find myself repeating the same kind of functionality over and over again, and like every other programmer (or at least a lot of you), I apply the DRY principal and abstract it away into a function for reuse. Recently though, I've noticed that its not just specific bits of code that I'm reusing, but patterns for performing actions, and I've wanted to start looking for better solutions.

The Problem

I've been applying the DRY principal to my programming ever since I understood code reuse, but I've started to notice that some times I end up repeating a particular solution to a problem, but not in a way that can simply be wrapped in a function for reuse. For example, I often need to cache small queries pulled in from a database to cut done on the number of times I perform that query in a given operation. If I query the system for the user information for user 1243, I shouldn't need to ask the database for it again during this operation. I'd usually pull this bit of functionality out into a function like getUser(*userID*) that handles keeping a cache of the data together and only querying the database if given a novel user ID. Unfortunately, I often end up with things like getUser(), getBook(), and getLocation() each handling a cache for their own information types. Not much I can do about it though, because they each have their own way of looking up the data, their own cache, and their own keys for the cache, right? I can't combined much more can I?

The Solution

Actually, I can do one better than having different cache functions for each cache I need. Applying a little object-oriented know-how I can create a single base object, Cache.cfc, that handles all of the basic methods of a cache that I would need, and if I provide a method that is meant to be overridden by its inheritors, all I need to do to make a cache for a new object type is extend Cache.cfc and put one new method in place to handle how to get the value of a given key if it isn't in the cache.

I start by creating Cache.cfc, and adding the following as a starting point:

```
<cfcomponent output="false">

    <cffunction name="init" access="public" output="false" returntype="any">

        <!--- Create Cache --->
        <cfset variables.cache = StructNew() />

        <cfreturn this />
    </cffunction>

</cfcomponent>
```

This is really basic, but the initialization function just creates a private variable cache that will hold our cached values and then returns itself. Now we need a way to put some information into the cache and get some out:

```
<cffunction name="set" access="public" output="false" returntype="void"
    hint="Sets a cache item to the specified value."
>
    <cfargument name="item" type="string" required="true" />
    <cfargument name="value" type="any" required="true" />
```

```
    <!--- Set the value into the cache --->
    <cfset variables.cache[arguments.item] = arguments.value />

    <cfreturn />
</cffunction>

<cffunction name="delete" access="public" output="false" returntype="void"
    hint="Removes an item from the cache."
>
    <cfargument name="item" type="string" required="true" />

    <!--- Delete the value from the cache --->
    <cfset StructDelete(variables.cache, arguments.item) />

    <cfreturn />
</cffunction>
```

Now that we're putting information into the cache, it would probably be good to have a way to check on what we have in the cache:

```
<cffunction name="exists" access="public" output="false" returntype="boolean"
    hint="Checks to see if a given item exists in the cache."
>
    <cfargument name="item" type="string" required="true" />

    <!--- Return if the value exists in the cache --->
    <cfreturn StructKeyExists(variables.cache, arguments.item) />
</cffunction>

<cffunction name="inspect" access="public" output="true" returntype="void"
    hint="Dumps the cache contents."
>
    <cfdump var="#variables.cache#">

    <cfreturn />
</cffunction>
```

None of thats really doing any magic though is it? So lets add the last few pieces of the puzzle:

```
<cffunction name="get" access="public" output="false" returntype="any"
    hint="Gets a value from the cache, even if its not in the cache yet."
>
    <cfargument name="item" type="string" required="true" />

    <!--- See if this value Doesn't exist --->
    <cfif NOT exists(arguments.item)>
        <!--- Call the lookUp() to find the value --->
        <cfset variables.cache[arguments.item] = lookUp(arguments.item) />
    </cfif>

    <!--- Return the cached value --->
    <cfreturn variables.cache[arguments.item] />
</cffunction>

<cffunction name="lookUp" access="private" output="false" returntype="any"
    hint="Internal function, overridden by classes that extend Cache"
>
    <cfargument name="item" type="string" required="true" />
```

```
    <!--- Loop up the value of the item --->

    <cfreturn item />
</cffunction>
```

Now with the addition of get() and lookUp() we are ready to make some magic. The basic idea is that any object that extends Cache will define a method called lookUp() that handles how to look up a given key and return the value to be placed into the cache. All of the other methods of the cache work invisibly with the new version of lookUp(), so they require no changes.

I've only got one more function to add, one thats really more of a utility than a requirement:

```
<cffunction name="refresh" access="public" output="false" returntype="any"
    hint="Looks up the item again, places it into the cache, and deletes it"
>
    <cfargument name="item" type="string" required="true" />

    <!--- Delete the cached value --->
    <cfset delete(arguments.item) />

    <!--- Call to get the value --->
    <cfreturn get(arguments.item) />
</cffunction>
```

The method refresh() just takes advantage of two existing functions to handle the process of forcing the cache to update a value.

Now that we've got our base Cache object, we need to define an extended, usable version. For simplicity sake, I'll create a cache for calculating factorial numbers, called FactorialCache.cfc.

FactorialCache.cfc

```
<cfcomponent extends="Cache" output="false">

    <cffunction name="lookUp" access="private" output="false" returntype="string">
        <cfargument name="item" type="numeric" required="true" />

        <cfset Sleep(2000) />

        <cfreturn Factorial(arguments.item) />
    </cffunction>

    <cffunction name="Factorial" access="private" output="false" returntype="numeric">
        <cfargument name="factor" type="numeric" required="true" />

        <cfif arguments.factor lte 1>
            <cfreturn 1 />
        <cfelse>
            <cfreturn arguments.factor * factorial(arguments.factor-1) />
        </cfif>
    </cffunction>

</cfcomponent>
```

The only problem with a factorial cache as an example is that calculating factorials that ColdFusion can handle is too quick to really notice, and anything large enough to take some time exceeds CF's built in numeric type. Rather than trying to use some Java numeric type, I threw in sleep(2000) to simulate a task that takes a while to complete. Now lets look at some test code:

```
<cfset Cache = CreateObject("component", "FactorialCache").init() />

<cfset Cache.inspect() />

<cfset Cache.get(35) />
```

```
<cfdump var="#now()#">

<cfset Cache.inspect() />

<cfset Cache.get(127) />
<cfdump var="#now()#">

<cfset Cache.inspect() />

<cfset Cache.get(96) />
<cfdump var="#now()#">

<cfset Cache.inspect() />

<cfset Cache.get(35) />
<cfdump var="#now()#">

<cfset Cache.inspect() />

<cfset Cache.get(127) />
<cfdump var="#now()#">

<cfset Cache.inspect() />

<cfset Cache.refresh(96) />
<cfdump var="#now()#">

<cfset Cache.inspect() />
```

If you run this test you should see that the first three calls to the cache take 2 seconds each, as they must call lookUp() to get the value, while the second two calls take almost no time, accessing the cache directly. Finally, the sixth call takes 2 seconds, because the refresh() function makes the cache look up the value again.

The Conclusion

While handling a cache is a relatively trivial task, this more OO way of thinking can offer me a number of benefits in other areas of my application as well, and services as a good basic lesson in leveraging some very basic object oriented design to solve a problem. The downside of this design is that I'll need a new object for every different cache type I have, but I do have the benefit that if I discover a faster way to handle caching, I can realize the benefits in every single cache I have by updating a single CFC, rather than however many times I had to write a new caching function. Its also much more clean during execution, with each object maintaining its own internal cache, and thats something I like as well.

The Code

The final version of Cache.cfc

Cache.cfc

```
<cfcomponent output="false">

    <cffunction name="init" access="public" output="false" returntype="any">

        <!--- Create Cache --->
        <cfset variables.cache = StructNew() />

        <cfreturn this />
    </cffunction>

    <cffunction name="get" access="public" output="false" returntype="any"
        hint="Gets a value from the cache, even if its not in the cache yet."
    >
        <cfargument name="item" type="string" required="true" />
```

```
        <!--- See if this value Doesn't exist --->
        <cfif NOT exists(arguments.item)>
            <!--- Call the lookUp() to find the value --->
            <cfset variables.cache[arguments.item] = lookUp(arguments.item) />
        </cfif>

        <!--- Return the cached value --->
        <cfreturn variables.cache[arguments.item] />
    </cffunction>

    <cffunction name="set" access="public" output="false" returntype="void"
        hint="Sets a cache item to the specified value."
    >
        <cfargument name="item" type="string" required="true" />
        <cfargument name="value" type="any" required="true" />

        <!--- Set the value into the cache --->
        <cfset variables.cache[arguments.item] = arguments.value />

        <cfreturn />
    </cffunction>

    <cffunction name="exists" access="public" output="false" returntype="boolean"
        hint="Checks to see if a given item exists in the cache."
    >
        <cfargument name="item" type="string" required="true" />

        <!--- Return if the value exists in the cache --->
        <cfreturn StructKeyExists(variables.cache, arguments.item) />
    </cffunction>

    <cffunction name="delete" access="public" output="false" returntype="void"
        hint="Removes an item from the cache."
    >
        <cfargument name="item" type="string" required="true" />

        <!--- Delete the value from the cache --->
        <cfset StructDelete(variables.cache, arguments.item) />

        <cfreturn />
    </cffunction>

    <cffunction name="inspect" access="public" output="true" returntype="void"
        hint="Dumps the cache contents."
    >
        <cfdump var="#variables.cache#">

        <cfreturn />
    </cffunction>

    <cffunction name="refresh" access="public" output="false" returntype="any"
        hint="Looks up the item again, places it into the cache, and deletes it"
    >
        <cfargument name="item" type="string" required="true" />

        <!--- Delete the cached value --->
        <cfset delete(arguments.item) />

        <!--- Call to get the value --->
        <cfreturn get(arguments.item) />
```

```
    </cffunction>

    <cffunction name="lookUp" access="private" output="false" returntype="any"
        hint="Internal function, overridden by classes that extend Cache"
    >
        <cfargument name="item" type="string" required="true" />

        <!--- Loop up the value of the item --->

        <cfreturn item />
    </cffunction>

</cfcomponent>
```

FactorialCache.cfc

```
<cfcomponent extends="Cache" output="false">

    <cffunction name="lookUp" access="private" output="false" returntype="string">
        <cfargument name="item" type="numeric" required="true" />

        <cfset Sleep(2000) />

        <cfreturn Factorial(arguments.item) />
    </cffunction>

    <cffunction name="Factorial" access="private" output="false" returntype="numeric">
        <cfargument name="factor" type="numeric" required="true" />

        <cfif arguments.factor lte 1>
            <cfreturn 1 />
        <cfelse>
            <cfreturn arguments.factor * factorial(arguments.factor-1) />
        </cfif>
    </cffunction>

</cfcomponent>
```