Primitive Variables Vs. Educated and Urbane Variables

Posted At : March 17, 2009 3:39 PM | Posted By : Mark Kruger
Related Categories: ColdFusion

One of the things that sometimes trips me up is the whole idea of references. I'm not talking about stuffy books in the library. I'm talking about the idea that setting one variable to another can sometimes create a pointer to that item rather than a copy. Now before we chat about it any further we should get square on the difference between a "primitive" or "simple" data type and a "complex" or "emotionally involved" data type.

Primitive Data Types

I like to think of "primitive" data types as "one level" members of any scope. I also like to think of them as little cave men in loin cloths running around and clubbing defenseless mammoths - but that is perhaps too much information. There's actually a pretty short list of primitive data types. In fact, "short" *actually is* one of the data types I believe. In the CF world a primitive Datatype would be a string, number or possibly a date. Of course the actual list is more like "short, long, float, double, int, string, byte" - but it is probably more useful (for the purpose of this post) to consider String, Number and Date. Check out this example.

```
<cfset inferior = 'Hello World'/>
<h4>I think I have a complex</h4>
<cfoutput>
    #isSimpleValue(inferior)#
<br>
</cfoutput><br>
<h4>It's not a complex - you really are inferior.</h4>
```

The "isSimpleValue()" function can help you sort out the type of variable. If it returns "YES" then you have a primitive (congratulations you can begin to look for remedial schooling). So, if we change our variable to a structure (for example):

```
<Cfset inferior = structNew()/>
<cfset inferior.item1 = true/>

<cfoutput>
#isStruct(inferior)#<br>
#isSimpleValue(inferior)#
<br>
<h4>You really are complex</h4>
</cfoutput>
```

Now we have a *non-primitive* value. Notice the "isStruct()" above. It's worth noting that there are a bunch of additional "is" functions for object typing (isQuery, isArray etc).

Setting Reference or Copy

Ok, here's the final piece of the puzzle. When you use cfset to create a variable on the left side that is assigning to another variable on the right one of 2 things happens. If the right side variable is a primitive the value is copied from the right to the left. If it is *not* a primitive but rather a complex data type or object then the value is *not* copied into your new variable container. Instead, CF assigns a pointer to it - a book

mark - based on your new variable name.

```
<cfset x = 'blue'/>
<h4>Copy from x to y</h4>
<cfset y = x/>
<cfset z = structnew()/>
<cfset z.sky = 'blue'/>
<H4>Set a pointer / bookmark</H4>
<cfset a = z/>
```

Why is This Important?

All righty, we get what is happening with primitives. A simple value is being passed back and forth. But what is happening with the complex types? Actually the data doesn't move at all. Instead, there are now 2 variables pointed to the same spot in memory and the members of one belong to the other (sort of like when Alamo and National merged). Try this:

```
<cfset y = structnew()/>
<cfset y.sky = 'Blue'/>

<cfset x = y>

<cfset x.sky = 'yellow'>

<h4>Dump out <strong>Y</strong></h4>
<Cfdump var="#y#"/>
```

See what I mean? when you change x then y has changed as well. The sky is yellow and there is a new member called "grass" that was never there before.

When is it a Problem?

Now admittedly the example above is probably not very useful - but there are some cases when it really comes into play. Consider the Application scope for example. Now be patient, this example requires some set up. First, we have the following Application scope code - note, I'm using application.cfm for simplicity here.

```
<cfapplication name="testStuff">

<Cfif NOT isDefined('application.testobj')>
    <cfset application.testobj = createobject("component","getuser")/>
</CFIF>
<!--- setting a reference from ap to variables --->
<cfset variables.testObj = application.testobj/>
```

In this case we are instantiating the "getuser" object and sticking it into the application scope, then creating a reference to it (for convenience I suppose) in the variables scope. So far so good. Let's look at "getuser.cfc".

```
<cfcomponent>
    <cffunction name="getUserInfo" ....>
        <cfargument name="permissions" .../>
        <cfquery name="getusers" ...>
            SELECT * ....
        </cfquery>

        <cfreturn getUsers/>
```

```
    </cffunction>

</cfcomponent>
```

The getuserinfo() Function returns the results of a query. Finally, we have some code on the user template - something like:

```
<cfset users = testobj.getUserInfo(permissions)/>
    <cfoutput query="users">
     #username# #password#<br/>
    </cfoutput>
```

This code will work and return users to the logged in user. Under a light load it will even return the correct users for a time. But as more and more requests are received *some users are going to see other users information*. Why? Simply because the variable "getusers" is going to persist in the application scope and be overwritten with each new request. Since the code testobj = application.testobj really just returns a reference to an object in the application scope, this will mean (eventually) that 2 users will hit at nearly the same time and get the same information (whichever thread was last to write to "result").

The Fix

The fix in this case is to make the "getusers" variable a member of the *function* so that it goes out of scope as soon as the function returns - like so:

```
<cffunction name="getUserInfo" ....>
    <cfargument name="permissions" .../>

    <!--- scope the var --->
    <cfset var getusers = ''/>

    <cfquery name="getusers" ...>
        SELECT * ....
    </cfquery>

    <cfreturn getUsers/>

</cffunction>
```

The lesson here is greater than the example however. When you set variables be sure you know what the underlying consequences are likely to be. Especially when dealing with the application and server scopes you should evaluate very carefully whether this reference approach has merit and is needed. If it *is* needed you will need to thoroughly test *under load* to be sure you have not created holes where you data can seep into unintended areas. Hope this helps - happy coding :).