Crashing Your Server in an Infinite Number of Steps

Posted At : February 15, 2006 2:24 PM | Posted By : Mark Kruger
Related Categories: Coldfusion MX 7, Coldfusion Tips and Techniques

Infinite loops are great fun. Technically an *infinite* loop is one with no hope of ever stopping. In the old days(1995) I worked for an outfit with a database product for salvage yards. The whole thing was written in MUMPS and everything was done through a terminal. The first program I ever wrote myself was a "spinner" program. It ran on the terminal and produced the following characters 1/2 second apart - "/,|,-,\,|" the result was a little spinning widget on the terminal. I added some text that said "rebuilding dataset, please wait". Whenever we were working on something and needed a way to keep folks from hassling us with new issues we would put it up on the screen. It looked like it was really doing something. With an http request it's a different story however...

In web requests, infinite loops are generally bad. A web request is designed to run for short duration - in keeping with the stateless nature of http. Even though I've seen requests that run an hour or more *by design* (like a data import script) you should try to keep your request down below a few hundred milliseconds.

An infinite loop isn't just a really really long request however. Usually they are like that Chinese dragon that eats himself starting at the tail. They are created by folding a variable or object into itself - or by creating a condition that is never met. This often creates the unwelcome condition of memory growing and resources being consumed as the loop races to *catch up* with itself. This can be a great deal of fun and cause server administrators to run wildly through the data center throwing their hair back. I don't mean tossing their head like Fabio. I mean actually pulling out their hair and throwing it back behind them as they run. You can crash your server faster than a Hollywood marriage if you know what you are doing - or (especially) if you *don't* know what you are doing.

Other languages are actually more susceptible to infinite loops than Coldfusion. Why? One reason is the "cfloop" and "cfoutput" tag. Consider this ASP code:

```
<%
   'rs2 is a result set from a query
   do while not rs2.eof
      caclTotals = calcTotals + rs2("subtotal") + rs2("tax")
   loop
   %>
```

Can anyone spot why this is an infite loop? Easy right? There is no "moveNext" indicator inside the loop. This code will "stick" on the first row of the query and stay there until it uses up all the resources it can or generates a buffer overflow. The correct code is:

```
<%
   'rs2 is a result set from a query
   do while not rs2.eof
      caclTotals = calcTotals + rs2("subtotal") + rs2("tax")
      rs2.movenext
   loop
   %>
```

The "movenext" indicator makes sure that the next row is selected. Of course the

equivalent code in Coldfusion simply takes that possibility out of the equation and makes an infinite loop far less likely.

```
<Cfloop query="rs2">
      <cfse calcTotals = calcTotals + subtotal + tax>
   </CFLOOP>
```

Since there is no need to increment a variable CF makes it more difficult to fall into this trap. That's not to say you can't generate an infinite loop in CF. obviously you can do it with a "do while" or "conditional" loop that does not resolve the condition.

```
<Cfset calcTotals = 0>
<cfset maxAmt   =   500.00>
<Cfset qty = 0>
<cfloop condition = "calcTotals LE maxAmt">
   <cfset caclTotals = calcTotals + amt>
   <Cfset qty = qty + 1>
</cfloop>
```

Look carefully. This is an infinite loop because I have misspelled "calcTotals" inside the loop. This code will set a variable called "caclTotls" to the same amount over and over again. While this probably won't crash the server, it's has no hope of stopping.

Here's one that *will* crash the server and it's harder to spot. In this code I want to run a series of calculations and store them in the "variables" scope. Then, I want to copy the values from the variables scope into an array, loop back and do the whole thing over again until I have an array of 10 containing all the variables from the variables scope. Sounds easy - right?

```
<cfset tblArr = arrayNew(1)>
<cfloop from="1" to="10" index="loopcount">
<cfscript>
   // run some calculations    val1   =   1.5;
   val2    =   val1 * 4 + 10;
   val3   =   val2 + val1 / 100;
   // set this index as a structure so I can    //contain a bunch of primitive variables.
   tblArr[loopCount]   =   structNew()
   // loop over my calculations and insert
   //them into the array as keys for the struct.
   for(each in variables) {
         tblArr[loopcount][each] = variables[each];
   }
</cfscript>
</cfloop>
```

Why do it this way? I like this approach for outputting a variable number of columns (as apposed to rows). In most query driven output you know the number of columns and the rows are variable - but sometimes (like in an loan payment chart) you might now all the rows and it's the columns that are variable. In that case an array with defined structures and structure keys is a better choice because my output code can set the left hand label and loop *n* number of times based on the length of the array.

In any case, not only will this code *not run*, but it will crash the server. You may have spotted the flaw in my loop already. The variable "tblArr" is an array that exists *in the variables scope*. When the loop reaches the variable "tblArr" it is going to try to set a reference to itself (tblArr[1][tblArr] = tblArr). This is going to grind the code to a halt and set the processor spinning. Remember that Chinese snake?

Instead, the code should have been written like this:

```
<cfset tblArr = arrayNew(1)>
<cfloop from="1" to="10" index="loopcount">
<cfscript>
   // run some calculations    val1   =   1.5;
   val2    =   val1 * 4 + 10;
   val3   =   val2 + val1 / 100;
   // set this index as a structure so I can
   //contain a bunch of primitive variables.
   tblArr[loopCount]   =   structNew()
   // loop over my calculations and insert    //them into the array as keys for the struct.
   for(each in variables) {
      // check for self referencing
      if(each IS NOT 'tblArr')
         tblArr[loopcount][each] = variables[each];
   }
</cfscript>
</cfloop>
```

Now the code will succeed because it is no longer trying to reference itself. hard to spot, but easy to correct.