Managing Bi-directional Relationships in ColdFusion ORM - Array-based Collections

Posted At : March 29, 2010 6:00 AM | Posted By : Bob Silverberg
Related Categories: ColdFusion, CF ORM Integration

It's important to know that when you have a bi-directional relationship you should set the relationship on both sides when setting one side. There have been a number of discussions about this on the **cf-orm-dev google group**, including **this one** in which **Barney Boisvert** provides a very good explanation of why this is important. **Brian Kotek** has also written **two articles** on the subject in the past. If you're not already familiar with this topic I suggest you check out those links.

The general recommendation for addressing this requirement is to override the methods in your objects that set one side of the relationship (e.g., setX(), addX() and removeX()) so that they'll set both sides, rather than just the side of the object that was invoked. While doing some testing of the new CF9 ORM adapter for Model-Glue along with the new scaffolding mechanism that we're developing I needed to address this issue for a many-to-many bi-directional relationship. I found that there were a few wrinkles that made the task not quite as straightforward as I has originally imagined, so I figured I should share what I came up with.

The particular many-to-many in question used an array to store a collection of objects on one side, and a structure to store the collection of objects on the other side. I found that each of these implementations introduced their own wrinkles, so I'm going to start with a post about dealing with array-based collections and then follow up with a second post about struct-based collections. Let's start by looking at the cfcs in question. For this example I'm using Countries and Languages to experiment with many-to-manys. A County can have many Languages spoken in it and a Language can have many Countries in which it's spoken. Here's what the cfcs look like:

```
component persistent="true" hint="This is Country.cfc"
{
 property name="CountryId" fieldtype="id" generator="native";
 property name="CountryCode" length="2" notnull="true";
 property name="CountryName" notnull="true";
 property name="Languages" fieldtype="many-to-many" cfc="Language"
  type="array" singularname="Language" linktable="CountryLanguage";
}


component persistent="true" hint="This is Language.cfc"
{
 property name="LanguageId" fieldtype="id" generator="native";
 property name="LanguageName" notnull="true";
 property name="Countries" fieldtype="many-to-many" cfc="Country"
 type="array" singularname="Country" linktable="CountryLanguage"
 inverse="true";
}
```

In order to ensure that both sides of the relationship are set whenever one side is explicitly set I need to override *addLanguage()* and *removeLanguage()* in Country.cfc and I need to override *addCountry()* and *removeCountry()* in Language.cfc. The code in each is virtually identical, as both sets of collections are implemented as arrays, so let's just look at Language.cfc:

```
component persistent="true" hint="This is Language.cfc"
{
 property name="LanguageId" fieldtype="id" generator="native";
 property name="LanguageName" notnull="true";
 property name="Countries" fieldtype="many-to-many" cfc="Country"
 type="array" singularname="Country" linktable="CountryLanguage"
 inverse="true";

 public void function addCountry(required Country Country)
  hint="set both sides of the bi-directional relationship" {
  // set this side
  if (not hasCountry(arguments.Country)) {
   arrayAppend(variables.Countries,arguments.Country);
  }
  // set the other side
  if (not arguments.Country.hasLanguage(this)) {
   arguments.Country.addLanguage(this);
  }
 }


 public void function removeCountry(required Country Country)
  hint="set both sides of the bi-directional relationship" {
  // set this side
  var index = arrayFind(variables.Countries,arguments.Country);
  if (index gt 0) {
   arrayDeleteAt(variables.Countries,index);
  }

  // set the other side

  if (arguments.Country.hasLanguage(this)) {

   arguments.Country.removeLanguage(this);

  }

 }

}
```

Let's walk through the code and discuss some of the issues I had to address, starting with the *addCountry()* method. Because I'm overriding the implicit *addCountry()* method I have to implement it myself, which means that I have to add the Country object that was passed in to the current Language object. I first check to see if the Country is already present in the Countries collection using the*hasCountry()* method, and if it is not then I add it to the Countries collection using *arrayAppend()*. Next I have to set the other side, meaning I have to add the current Language object to the Country object that was passed in. This is a simple matter of calling *addLanguage()*on the Country object and passing in *this*, which is the current Language object. You'll notice that before I do that I first check to make sure that the current Language isn't already assigned to the Country. Do you know why that's necessary?

If I didn't do that check then this routine would call the *addLanguage()* routine in Country.cfc, which would turn around and call the *addCountry()* routine in Language.cfc, which would then call the *addLanguage()* routine in Country.cfc, ad infinitum, and we'd have a marvelous infinite loop. I personally don't like infinite loops creeping into my code, so I make sure that I only call the *addLanguage()* method if the Language has not already been assigned.

Let's move on the the *removeCountry()* method. Just as with the *addCountry()* method, because I'm overriding the implicit *removeCountry()* method I have to implement it myself, which means that I have to remove the Country object that was passed in from the Countries collection in the current Language object. Removing a specific item from an array is not as straightforward as adding an item to an array, so I have to use *arrayFind()* to first locate the Country in the array and then use*arrayDeleteAt()* to remove it. I can then set the other side exactly as I have done in the *addCountry()* method. I use *hasLanguage()* to see whether the current Language is assigned to the Country, and if it is then I use *removeLanguage()* to remove it.

This all works pretty well, but there is one situation in which errors can be thrown with the above code, and that's when we're working with a brand new object. When ColdFusion creates an instance of Language.cfc, for example when we call*entityNew("Language")*, all of the properties start off as nulls, including any collections. This means that if we create a new Language object and then try to add a Country object to it, we'll get an error on the line that reads:

```
arrayAppend(variables.Countries,arguments.Country)
```

because *variables.Countries* is not an array, it's a null. I've found the best way to deal with that is to default the collection to an empty array in the constructor. I add an*init()* method to my cfc that looks like this:

```
public Language function init() {

 if (isNull(variables.Countries)) {

  variables.Countries = [];

 }

 return this;

}
```

Now I can always count on *variables.Countries* being an array, and my code should work in all situations.

As I mentioned earlier, the approach that one must take when a collection is implemented as a struct, rather than an array, is a bit different and comes with its own set of wrinkles, so I plan to cover that in a future post.

Note also that an altogether different approach could be taken in which one creates new methods for managing the relationships. One might create *assignCountry()* and *clearCountry()* methods which, because they are not overriding the implicit methods, could simply make use of the implicit *addCountry()* and *removeCountry()* methods, which would eliminate much of the complexity required above. Another advantage of taking that approach is that one ends up programming to an interface rather than an implementation, which is always to be desired. The downside to that approach is that you are essentially changing the API of your object, and I wanted to avoid doing that in this specific case as I was trying to get code to work with a generic ORM adapter, which would have no idea that it should be calling *assignCountry()*rather than *addCountry()*. As with everything, design decisions are full of tradeoffs.

I am a little less than pleased with how complex this task seems to be and perhaps there are much better ways of tackling this than I have documented above. If anyone has any suggestions please leave them as comments.