

This is a ColdFusion 8.0.1 only feature

Contents

- [Using TransferObject Proxies](#)
 - [Example Scenario](#)
 - [Safe Methods](#)
 - [Useful Methods](#)
 - [getIsLoaded\(\)](#)
 - [getLoadedObject\(\)](#)
 - [getIsProxy\(\)](#)
 - [Caveats](#)

Using TransferObject Proxies

This attribute is found on *onetomany*, *manytoone* and *manytomany* elements, and tells Transfer to load up a collection of objects as Object Proxies, rather than the actual TransferObjects.

TransferObject Proxies use `onMissingMethod` to proxy method calls from themselves to whatever object it is they are proxying, which is why it is a 8.0.1 only feature.

Example configuration:

```
<object name="Foo">
  <id name="idbasic" type="numeric" />
  <property name="bar" />
  <onetomany name="Child" proxied="true">
    <link to="Bar" column="FKBarID" />
    <collection type="array">
      </collection>
    </onetomany>
  </object>
```

Example Scenario

For example - you have a *onetomany*, say of *Users-o2m->Orders*.

Normally this would be a pretty bad thing, as users keep making orders, and they could start ranging into the hundreds of orders, and then you have to create and populate 400 objects when the you want to get the User.

So first level, is obviously, lazy loading. So we can configure that collection to be lazy loaded, in which case, we can request the User, and the 400 Orders, don't get

loaded.

BUT when we do request the Orders, all 400 objects get loaded at once, which brings the server to its knees.

Now we have a second level, we can write:

```
<onetomany lazy="true" proxied="true">
...
</onetomany>
```

Now, instead of querying for all the data of the 400 objects, it just queries for the IDs of the Orders, and then creates a TransferObjectProxy for each record it finds.

Only when a method is called on the Proxy that requires data it doesn't know about is called on it, will it load up the TransferObject it represents.

This means you could do something like:

```
foo = transfer.get( "Foo" , 1 );
len = ArrayLen(foo.getChildArray());
```

And there would be no loading of the underlying TransferObjects. You could also use find() or contains(), without worrying about the objects being loaded.

This should be a huge performance boost for many applications.

Now that all being said, lazy, and proxied can be used separately, or together, totally at your discretion.

Safe Methods

Now, there are several things the TOP (TransferObjectProxy) knows about, and therefore won't load the underlying object when requested:

- getIsPersisted() - it knows it's true, so unless it's loaded, it will return 'true' without loading.
- getIsDirty() - it knows this is false, so unless it's loaded, it will return 'false' without loading.
- getClassName() - it knows what class it represents, so it won't be loaded if this is called.
- getid[@name]() - the TOP stores the primary key value, so it knows about it, so it won't be loaded.
- with the combination of this, and the above, you can call equalsTransfer() without worrying about loading

Finally, the TOP knows about any properties that are used on it either as a struct key, or as an ordering property, so if the TOP is used in a collection that is ordered by 'Foo', calling getFoo() on the TOP will not cause a load, as it already has that information.

Useful Methods

getIsLoaded()

([API](#))

This method returns whether or not the Proxied object has been loaded. This can be useful for avoiding unnecessary TransferObject loads inside a Proxy.

This method can also be found on general [TransferObjects](#).

getLoadedObject()

([API](#))

This returns the Proxied object, and forces a load of the object if it not yet loaded.

This is particularly useful if you are doing metadata based operations on TransferObjects. Since the proxy doesn't have any of the methods that exist on the proxied object, due to the use of onMissingMethod, this makes it very hard. By having access to the Proxied object directly via getLoadedObject(), it makes these sort of operations possible.

This method can also be found on general [TransferObjects](#).

getIsProxy()

([API](#))

This method returns whether or not the given object is a Proxy. This can be useful in determining at runtime if a given object is a proxy, as it may, or may not be, depending on whether it was loaded through a collection, or directly.

This method can also be found on general [TransferObjects](#).

Caveats

It should be noted that the Proxy will proxy both the generated TransferObject and the Decorator.

So, if you are just using a TransferObject, the structure is:

TransferObjectProxy -loads->TransferObject


If you are using a Decorator:

TransferObjectProxy -loads->Decorator -contains->TransferObject

This means that when you are proxying objects, the type of the CFC won't always be the type as found in the decorator, it can end up being `transfer.com.TransferObjectProxy`.

Therefore, you may wish to set your argument and return types to `transfer.com.TransferObject` or any for object's that are being proxied.

Do realise that CFC creation and population does have its expense, so if you are creating 1,000,000 TransferObjectProxies, it may well be slow. So still load test, and make sure that your application performs well, etc. There is no silver bullet ;o)

 Categories:

- [Configuration](#)
- [Performance](#)