

Contents

- [Managing the Cache](#)
 - [Memory Management](#)
 - [Configuration](#)
 - [Caching Types in Transfer](#)
 - [Caching methods](#)
 - [Transfer.discard\(object\)](#)
 - [Transfer.discardByClassAndKey\(className, key\)](#)
 - [Transfer.discardByClassAndKeyArray\(className, keyArray\)](#)
 - [Transfer.discardByClassAndKeyQuery\(className, keyQuery, columnName\)](#)
 - [Transfer.discardAll\(\)](#)
 - [Transfer.recycle\(object\)](#)
-

Managing the Cache

Transfer has an in built Object Caching system that is highly configurable.

This means that for every [TransferObject](#) that is created, or retrieved via Transfer is also stored within the configured cache. This speeds up subsequent retrievals greatly as TransferObjects can be fetched from memory, and a database hit is not required.

For example, if I was to write;

```
<cfscript>
  user = getTransfer().get( "user.User" , 1 );
</cfscript>
```

The first time that this is run, Transfer will retrieve it from the database. However, the next time that Transfer is asked for user.User with the ID of '1', it will fetch it from the cache.

Also, if I write:

```
<cfscript>
  post = getTransfer().get( "post.Post" , 1 );
</cfscript>
```

And the post.Post object contains the user.User that was already retrieved above, it would fetch the User TransferObject from the cache, rather than from the database.

This means that there is only ever one TransferObject for a given entity (i.e. a particular User, or a Post) stored inside the cache at any given moment.

While this can create an extra degree of complexity to an application, it also allows a great deal of control over data within the system.

Memory Management

The cache is configured such that it discards TransferObjects when the underlying Java engine requests memory to be freed.

This runs separately to all other configuration options, and cannot be turned off.

This is to ensure that the server's memory is not over run, and to ensure the cache doesn't become a memory leak.

Configuration

All the aspects of Caching in Transfer is configured via the [Transfer Configuration File](#)

By default, all TransferObjects are cached in the 'instance' scope, and are stored indefinitely.

Caching Types in Transfer

There are several different types of caching available with Transfer, each one having it's own capabilities.

- **instance**

Instance caching is the default caching for Transfer Objects. Instance caching is where the caching mechanism is stored as a property of the Transfer library itself, and is specific to that instance of library.

- **application**

Application caching is where the caching mechanism is stored in the application scope of the ColdFusion application, under the key specified in the [Transfer Configuration File](#).

This can be used to share the cache across multiple instances of Transfer within an application, as long as the configuration is the same between them.

- **session**

Session caching is where the caching mechanism is stored in the session scope of the user, under the key specified in the [Transfer Configuration File](#). Sessions must be enabled for session scope caching to work. If sessions are disabled, the request scope is used.

- **transaction**

Transaction caching is where the caching mechanism is stored in the session scope of the user, under the key specified in the [Transfer Configuration File](#), however, when a create(), update() or save() call is made on the object, it is discarded from the cache.

- **request**

Request caching is where the caching mechanism is stored in the request scope of the user, under the key specified in the [Transfer Configuration File](#).

- **server**

Server caching is where the caching mechanism is stored in the server scope of the ColdFusion server, under the key specified in the [Transfer Configuration File](#).

This can be used to share the cache across multiple instances of Transfer within an server, as long as the configuration is the same between them.

- **none**

None caching is where there is no caching.

Setting the cache settings for objects that have composites (i.e. onetomany, manytoone, manytomany) such that they are in difference scopes could cause indeterminate behaviour.

Example:

```
<transfer>
  <objectCache>
    <defaultCache>
      <scope="instance" >
    </defaultCache>
    <cache class="Foo" >
      <scope="request" >
    </cache>
  </objectCache>
  <objectDefinitions>
    <object name="Bar" >
      ...
    <manytoone name="Foo" >...</manytoone>
  </object>
  </object name="Foo" >...</object>
  </objectDefinitions>
</transfer>
```

This would cause indeterminate behaviour, as the *Bar* objects would retain a copy of *Foo*, as there is no way the system can track when a request has timed out and act accordingly.

If this sort of set up is necessary, it may be better to have a short *accessedminutestimeout*, or *maxminutestimeout* cache setting on the *Foo* object, to allow for short caching periods.

Caching methods

There are several methods that can be used with caching.

Transfer.discard(object)

([API](#))

Removes an object from its designated caching mechanism.

This will also discard any objects that have a reference to this object via composition.

It should be noted that `Transfer.save()` and `Transfer.update()` on a discarded object will update the object currently in cache to the state of the saved object

Transfer.discardByClassAndKey(className, key)

[\(API\)](#)

Removes an object from its designated caching mechanism by its class and its key, if it exists.

It should be noted that [Transfer.save\(\)](#) and [Transfer.update\(\)](#) on a discarded object will update the object currently in cache to the state of the saved object.

Transfer.discardByClassAndKeyArray(className, keyArray)

[\(API\)](#)

Removes an object from its designated caching mechanism by its class and each key in the array, if it exists.

It should be noted that [Transfer.save\(\)](#) and [Transfer.update\(\)](#) on a discarded object will update the object currently in cache to the state of the saved object.

Transfer.discardByClassAndKeyQuery(className, keyQuery, columnName)

[\(API\)](#)

Removes an object from its designated caching mechanism by its class and each key in the query, if it exists.

It should be noted that [Transfer.save\(\)](#) and [Transfer.update\(\)](#) on a discarded object will update the object currently in cache to the state of the saved object.

Transfer.discardAll()

[\(API\)](#)

Discards all objects that are currently stored in the Transfer cache.

It should be noted that [Transfer.save\(\)](#) and [Transfer.update\(\)](#) on a discarded object will update the object currently in cache to the state of the saved object.

Transfer.recycle(object)

[\(API\)](#)

Resets the state of the TransferObject, and puts it back into a pool of TransferObjects to be populated with data.

This is very good for performance, as it means the system does not need to create a new TransferObject when requested to, it can recycle an old one.

The `Recycle` method should only be utilised once a `TransferObject` has either (a) not been inserted in the database, (b) discarded from the cache or (c) deleted from the system, as the `TransferObject` loses all state once it has been recycled.

Due to the performance increases provided by ColdFusion 8, recycling provides little, if no benefit on that platform.

Categories: • [Cache](#)