How Does Caching Work in AFNetworking? : AFImageCache & NSUrlCache Explained

Feb 20th, 2014

If you are an iOS developer using Mattt Thompson's 'delightful networking framework' `AFNetworking` (and if you aren't, what are you waiting for?), perhaps you have been been curious or confused about the caching mechanism employed and how you can tweak it to your advantage.

`AFNetworking` actually takes advantage of **2** separate caching mechanisms:

- **AFImagecache**: a **memory-only** image cache private to `AFNetworking`, subclassed off of `NSCache`
- **NSURLCache**: `NSURLConnection's` default URL caching mechanism, used to store `NSURLResponse` objects : an **in-memory** cache by default, configurable as an **on-disk persistent cache**

In order to understand how each caching system works, let's look at how they are defined:

How AFImageCache Works

`AFImageCache` is a part of the `UIImageView+AFNetworking` category. It is a subclass of `NSCache`, storing `UIImage` objects with a URL string as its key (obtained from an input `NSURLRequest` object).

`AFImageCache` definition:

```
1  @interface AFImageCache : NSCache <AFImageCache>
2
3  // singleton instantiation :
4
5  + (id <AFImageCache>)sharedImageCache {
6      static AFImageCache *_af_defaultImageCache = nil;
7      static dispatch_once_t oncePredicate;
8      dispatch_once(&oncePredicate, ^{
9          _af_defaultImageCache = [[AFImageCache alloc] init];
10
11 // clears out cache on memory warning :
12
13     [[NSNotificationCenter defaultCenter] addObserverForName:UIApplicationDidReceiveMemoryWarningNotification object:nil queue:[NSOper
14         [_af_defaultImageCache removeAllObjects];
15     }];
16 });
17
18 // key from [[NSURLRequest URL] absoluteString] :
19
20 static inline NSString * AFImageCacheKeyFromURLRequest(NSURLRequest *request) {
21     return [[request URL] absoluteString];
22 }
23
24 @implementation AFImageCache
25
26 // write to cache if proper policy on NSURLRequest :
27
28 - (UIImage *)cachedImageForRequest:(NSURLRequest *)request {
29     switch ([request cachePolicy]) {
30         case NSURLRequestReloadIgnoringCacheData:
31         case NSURLRequestReloadIgnoringLocalAndRemoteCacheData:
32             return nil;
33         default:
34             break;
35     }
36
37     return [self objectForKey:AFImageCacheKeyFromURLRequest(request)];
38 }
39
40 // read from cache :
41
42 - (void)cacheImage:(UIImage *)image
43         forRequest:(NSURLRequest *)request {
44     if (image && request) {
45         [self setObject:image forKey:AFImageCacheKeyFromURLRequest(request)];
46     }
47 }
```

`AFImageCache` is a private implementation of `NSCache`. There is no customization that you can do outside of editing the implementation in the the

`UIImageView+AFNetworking` category, directly. It stores all accessed `UIImage` objects into its NSCache. The `NSCache` controls when the `UIImage` objects are released. If you wish to observe when images are released, you can implement `NSCacheDelegate`'s `cache:willEvictObject` method.

Edit (03.14.14) : Mattt Thompson has gratiously informed me that as of `AFNetworking` 2.1, `AFImageCache` is configurable. There is now a public **setSharedImageCache** method. Here's the full AFN 2.2.1 **UIImageView+AFNetworking** specification.

How NSURLCache Works

Since `AFNetworking` uses `NSURLConnection`, it takes advantage of its native caching mechanism, `NSURLCache`. `NSURLCache` caches `NSURLResponse` objects returned by server calls via `NSURLConnection`.

Enabled by Default, but Needs a Hand

An `NSURLCache` `sharedCache` is enabled by default and will be used by any `NSURLConnection` objects fetching URL contents for you.

Unfortunately, it has a tendency to hog memory and does not write to disk in its default configuration. To tame the beast and potentially add some persistance, you can simply declare a shared `NSURLCache` in your app delegate like so:

```
1 NSURLCache *sharedCache = [[NSURLCache alloc] initWithMemoryCapacity:2 * 1024 * 1024
2                                                         diskCapacity:100 * 1024 * 1024
3                                                             diskPath:nil];
4 [NSURLCache setSharedURLCache:sharedCache];
```

Here we declare a shared `NSURLCache` with 2mb of memory and 100mb of disk space

Setting the Cache Policy on NSURLRequest Objects

`NSURLCache` will respect the caching policy (`NSURLRequestCachePolicy`) of each `NSURLRequest` object. The policies are defined as follows :

- **NSURLRequestUseProtocolCachePolicy**: specifies that the caching logic defined in the protocol implementation, if any, is used for a particular URL load request. This is the default policy for URL load requests
- **NSURLRequestReloadIgnoringLocalCacheData**: ignore the local cache, reload from source
- **NSURLRequestReloadIgnoringLocalAndRemoteCacheData**: ignore local & remote caches, reload from source
- **NSURLRequestReturnCacheDataElseLoad**: load from cache, else go to source.
- **NSURLRequestReturnCacheDataDontLoad**: offline mode, load cache data regardless of expiration, do not go to source
- **NSURLRequestReloadRevalidatingCacheData**: existing cache data may be used provided the origin source confirms its validity, otherwise the URL is loaded from the origin source.

Caching to Disk with NSURLCache

Cache-Control HTTP Header

Either the `Cache-Control` header or the `Expires` header MUST be in the HTTP response header from the server in order for the client to cache it (with the existence of the `Cache-Control` header taking precedence over the `Expires` header). This is a huge gotcha to watch out for. `Cache Control` can have parameters defined such as `max-age` (how long to cache before updating response), public / private access, or `no-cache` (don't cache response). `Here` is a good introduction to HTTP cache headers.

Subclass NSURLCache for Ultimate Control

If you would like to bypass the requirement for a `Cache-Control` HTTP header and want to define your own rules for writing and reading the `NSURLCache` given an `NSURLResponse` object, you can subclass `NSURLCache`.

Here is an example that uses a `CACHE_EXPIRES` value to judge how long to hold on to the cached response before going back to the source:

(Thanks to Mattt Thompson for the feedback and code edits!)

```
1  @interface CustomURLCache : NSURLCache
2
3  static NSString * const CustomURLCacheExpirationKey = @"CustomURLCacheExpiration";
4  static NSTimeInterval const CustomURLCacheExpirationInterval = 600;
5
6  @implementation CustomURLCache
7
8  + (instancetype)standardURLCache {
9      static CustomURLCache *_standardURLCache = nil;
10     static dispatch_once_t onceToken;
11     dispatch_once(&onceToken, ^{
12         _standardURLCache = [[CustomURLCache alloc]
13                              initWithMemoryCapacity:(2 * 1024 * 1024)
14                              diskCapacity:(100 * 1024 * 1024)
15                              diskPath:nil];
16     }
17
18     return _standardURLCache;
19 }
20
21 #pragma mark - NSURLCache
22
23 - (NSCachedURLResponse *)cachedResponseForRequest:(NSURLRequest *)request {
24     NSCachedURLResponse *cachedResponse = [super cachedResponseForRequest:request];
25
26     if (cachedResponse) {
27         NSDate* cacheDate = cachedResponse.userInfo[CustomURLCacheExpirationKey];
28         NSDate* cacheExpirationDate = [cacheDate dateByAddingTimeInterval:CustomURLCacheExpirationInterval];
```

```
        if ([cacheExpirationDate compare:[NSDate date]] == NSOrderedAscending) {
            [self removeCachedResponseForRequest:request];
            return nil;
        }
    }
}

    return cachedResponse;
}

- (void)storeCachedResponse:(NSCachedURLResponse *)cachedResponse
                 forRequest:(NSURLRequest *)request
{
    NSMutableDictionary *userInfo = [NSMutableDictionary dictionaryWithDictionary:cachedResponse.userInfo];
    userInfo[CustomURLCacheExpirationKey] = [NSDate date];

    NSCachedURLResponse *modifiedCachedResponse = [[NSCachedURLResponse alloc] initWithResponse:cachedResponse.response data:cachedRes

    [super storeCachedResponse:modifiedCachedResponse forRequest:request];
}

@end
```

Now that you have your `NSURLCache` subclass, don't forget to initialize it in your AppDelegate in order to use it :

```
CustomURLCache *URLCache = [[CustomURLCache alloc] initWithMemoryCapacity:2 * 1024 * 1024
                                                             diskCapacity:100 * 1024 * 1024
                                                                 diskPath:nil];
[NSURLCache setSharedURLCache:URLCache];
```

Overriding the NSURLResponse before caching

The `-connection:willCacheResponse` delegate is a place to intercept and edit the `NSURLCachedResponse` object created by `NSURLConnection` before it is cached. In order to edit the `NSURLCachedResponse`, return an edited mutable copy as follows (code from NSHipster blog):

```
- (NSCachedURLResponse *)connection:(NSURLConnection *)connection
                  willCacheResponse:(NSCachedURLResponse *)cachedResponse {
    NSMutableDictionary *mutableUserInfo = [[cachedResponse userInfo] mutableCopy];
    NSMutableData *mutableData = [[cachedResponse data] mutableCopy];
    NSURLCacheStoragePolicy storagePolicy = NSURLCacheStorageAllowedInMemoryOnly;

    // ...

    return [[NSCachedURLResponse alloc] initWithResponse:[cachedResponse response]
                                                    data:mutableData
                                                userInfo:mutableUserInfo
                                           storagePolicy:storagePolicy];
}

// If you do not wish to cache the NSURLCachedResponse, just return nil from the delegate function:

- (NSCachedURLResponse *)connection:(NSURLConnection *)connection
                  willCacheResponse:(NSCachedURLResponse *)cachedResponse {
    return nil;
}
```

Disabling NSURLCache

Don't want to use the `NSURLCache`? Not Impressed? That's okay. To disable the `NSURLCache`, simply zero out memory and disk space in the shared `NSURLCache` definition in your appDelegate:

```
NSURLCache *sharedCache = [[NSURLCache alloc] initWithMemoryCapacity:0
                                                        diskCapacity:0
                                                            diskPath:nil];
[NSURLCache setSharedURLCache:sharedCache];
```

Summary

I wanted to write this blog post for the benefit of the iOS community, to summarize all of the information I found dealing with caching releated to `AFNetworking`. We had an internal app loading a lot of images that had some memory issues and performance problems. I was tasked with trying to diagnose the caching behavior of the app. During this exercise, I discovered the information on this post through scouring the web and doing plenty of debugging and logging. It is my hope that this post summarizes my findings and provides an opportunity for others with `AFNetworking` experience to add additional information. I hope that you have found this helpful.

Posted by Tim Brandt Feb 20th, 2014 AFImageCache, NSUrlCache, apple, ios

« One Simple Trick To Level Up Your Code TDD is BS** »

Comments

Recent Posts

- Releasing Git Town
- Managing Data Classes With Ids

- Android and CI and Gradle - a How-To
- Refactoring Git Branches - Part II
- iOS Checklists - Creating and Submitting Your App