源码篇： SDWebImage

时间 2015-02-07 23:30:45 南峰子的技术博客

原文 http://southpeak.github.io/blog/2015/02/07/yuan-ma-pian-:sdwebimage/

主题 SDWebImage

源码来源： https://github.com/rs/SDWebImage

版本: 3.7

SDWebImage是一个开源的第三方库，它提供了UIImageView的一个分类，以支持从远程服务器下载并缓存图片的功能。它具有以下功能：

1. 提供UIImageView的一个分类，以支持网络图片的加载与缓存管理
2. 一个异步的图片加载器
3. 一个异步的内存+磁盘图片缓存
4. 支持GIF图片
5. 支持WebP图片
6. 后台图片解压缩处理
7. 确保同一个URL的图片不被下载多次
8. 确保虚假的URL不会被反复加载
9. 确保下载及缓存时，主线程不被阻塞

从github上对SDWebImage使用情况就可以看出，SDWebImage在图片下载及缓存的处理方面还是很被认可的。在本文中，我们主要从源码的角度来分析一下SDWebImage的实现机制。讨论的内容将主要集中在图片的下载及缓存，而不包含对GIF图片及WebP图片的支持操作。

下载

在SDWebImage中，图片的下载是由SDWebImageDownloader类来完成的。它是一个异步下载器，并对图像加载做了优化处理。下面我们就来看看它的具体实现。

下载选项

在下载的过程中，程序会根据设置的不同的下载选项，而执行不同的操作。下载选项由枚举SDWebImageDownloaderOptions定义，具体如下

```
typedef NS_OPTIONS(NSUInteger, SDWebImageDownloaderOptions) {
  SDWebImageDownloaderLowPriority = 1 << 0,
  SDWebImageDownloaderProgressiveDownload = 1 << 1,
  // 默认情况下请求不使用NSURLCache，如果设置该选项，则以默认的缓存策略来使用NSURLCache
  SDWebImageDownloaderUseNSURLCache = 1 << 2,
  // 如果从NSURLCache缓存中读取图片，则使用nil作为参数来调用完成block
  SDWebImageDownloaderIgnoreCachedResponse = 1 << 3,
  // 在iOS 4+系统上，允许程序进入后台后继续下载图片。该操作通过向系统申请额外的时间来完成后台下载。如果后台任务
  SDWebImageDownloaderContinueInBackground = 1 << 4,
  // 通过设置NSMutableURLRequest.HTTPShouldHandleCookies = YES来处理存储在NSHTTPCookieStore中的cook
  SDWebImageDownloaderHandleCookies = 1 << 5,
  // 允许不受信任的SSL证书。主要用于测试目的。
  SDWebImageDownloaderAllowInvalidSSLCertificates = 1 << 6,
  // 将图片下载放到高优先级队列中
  SDWebImageDownloaderHighPriority = 1 << 7,
};
```

可以看出，这些选项主要涉及到下载的优先级、缓存、后台任务执行、cookie处理以认证几个方面。

下载顺序

SDWebImage的下载操作是按一定顺序来处理的，它定义了两种下载顺序，如下所示

```
typedef NS_ENUM(NSInteger, SDWebImageDownloaderExecutionOrder) {
  // 以队列的方式，按照先进先出的顺序下载。这是默认的下载顺序
  SDWebImageDownloaderFIFOExecutionOrder,
  // 以栈的方式，按照后进先出的顺序下载。
  SDWebImageDownloaderLIFOExecutionOrder
};
```

下载管理器

SDWebImageDownloader下载管理器是一个单例类，它主要负责图片的下载操作的管理。图片的下载是放在一个NSOperationQueue操作队列中来完成的，其声明如下：

```
@property (strong, nonatomic) NSOperationQueue *downloadQueue;
```

默认情况下，队列最大并发数是6。如果需要的话，我们可以通过SDWebImageDownloader类的**maxConcurrentDownloads** 属性来修改。

所有下载操作的网络响应序列化处理是放在一个自定义的并行调度队列中来处理的，其声明及定义如下：

```
@property (SDDispatchQueueSetterSementics, nonatomic) dispatch_queue_t barrierQueue;
- (id)init {
  if ((self = [super init])) {
    ...
    _barrierQueue = dispatch_queue_create("com.hackemist.SDWebImageDownloaderBarrierQueue", DIS
    ...
  }
  return self;
}
```

每一个图片的下载都会对应一些回调操作，如下载进度回调，下载完成回调等，这些回调操作是以block形式来呈现，为此在SDWebImageDownloader.h中定义了几个block，如下所示：

```
// 下载进度
typedef void(^SDWebImageDownloaderProgressBlock)(NSInteger receivedSize, NSInteger expectedSize
// 下载完成
typedef void(^SDWebImageDownloaderCompletedBlock)(UIImage *image, NSData *data, NSError *error,
// Header过滤
typedef NSDictionary *(^SDWebImageDownloaderHeadersFilterBlock)(NSURL *url, NSDictionary *heade
```

图片下载的这些回调信息存储在SDWebImageDownloader类的 **URLCallbacks** 属性中，该属性是一个字典，key是图片的URL地址，value则是一个数组，包含每个图片的多组回调信息。由于我们允许多个图片同时下载，因此可能会有多个线程同时操作URLCallbacks属性。为了保证URLCallbacks操作(添加、删除)的线程安全性，SDWebImageDownloader将这些操作作为一个个任务放到barrierQueue队列中，并设置屏障来确保同一时间只有一个线程操作URLCallbacks属性，我们以添加操作为例，如下代码所示：

```
- (void)addProgressCallback:(SDWebImageDownloaderProgressBlock)progressBlock andCompletedBlock:
    ...
    // 1. 以dispatch_barrier_sync操作来保证同一时间只有一个线程能对URLCallbacks进行操作
    dispatch_barrier_sync(self.barrierQueue, ^{
  ...
  // 2. 处理同一URL的同步下载请求的单个下载
  NSMutableArray *callbacksForURL = self.URLCallbacks[url];
  NSMutableDictionary *callbacks = [NSMutableDictionary new];
  if (progressBlock) callbacks[kProgressCallbackKey] = [progressBlock copy];
  if (completedBlock) callbacks[kCompletedCallbackKey] = [completedBlock copy];
  [callbacksForURL addObject:callbacks];
  self.URLCallbacks[url] = callbacksForURL;
  ...
    });
}
```

整个下载管理器对于下载请求的管理都是放在downloadImageWithURL:options:progress:completed:方法里面来处理的，该方法调用了上面所提到的addProgressCallback:andCompletedBlock:forURL:createCallback:方法来将请求的信息存入管理器中，同时在创建回调的block中创建新的操作，配置之后将其放入downloadQueue操作队列中，最后方法返回新创建的操作。其具体实现如下：

```
- (id <SDWebImageOperation>)downloadImageWithURL:(NSURL *)url options:(SDWebImageDownloaderOpti
  ...
  [self addProgressCallback:progressBlock andCompletedBlock:completedBlock forURL:url createCal
    ...
```

```
    // 1. 创建请求对象，并根据options参数设置其属性
    // 为了避免潜在的重复缓存(NSURLCache + SDImageCache)，如果没有明确告知需要缓存，则禁用图片请求的缓存操作
    NSMutableURLRequest *request = [[NSMutableURLRequest alloc] initWithURL:url cachePolicy:(op
    ...
    // 2. 创建SDWebImageDownloaderOperation操作对象，并进行配置
    // 配置信息包括是否需要认证、优先级
    operation = [[wself.operationClass alloc] initWithRequest:request
                                      options:options
                                     progress:^(NSInteger receivedSize, NSInteger expectedSize) {
                                       // 3. 从管理器的callbacksForURL中找出该URL所有的进度处理回调并调用
                                       ...
                                       for (NSDictionary *callbacks in callbacksForURL) {
                                         SDWebImageDownloaderProgressBlock callback = callbacks[kProgre
                                         if (callback) callback(receivedSize, expectedSize);
                                       }
                                     }
                                    completed:^(UIImage *image, NSData *data, NSError *error, BOOL fini
                                       // 4. 从管理器的callbacksForURL中找出该URL所有的完成处理回调并调用，
                                       // 如果finished为YES，则将该url对应的回调信息从URLCallbacks中删除
                                       ...
                                       if (finished) {
                                         [sself removeCallbacksForURL:url];
                                       }
                                       for (NSDictionary *callbacks in callbacksForURL) {
                                         SDWebImageDownloaderCompletedBlock callback = callbacks[kComple
                                         if (callback) callback(image, data, error, finished);
                                       }
                                    }
                                    cancelled:^{
                                       // 5. 取消操作将该url对应的回调信息从URLCallbacks中删除
                                       SDWebImageDownloader *sself = wself;
                                       if (!sself) return;
                                       [sself removeCallbacksForURL:url];
                                    }];
    ...
    // 6. 将操作加入到操作队列downloadQueue中
    // 如果是LIFO顺序，则将新的操作作为原队列中最后一个操作的依赖，然后将新操作设置为最后一个操作
    [wself.downloadQueue addOperation:operation];
    if (wself.executionOrder == SDWebImageDownloaderLIFOExecutionOrder) {
      [wself.lastAddedOperation addDependency:operation];
      wself.lastAddedOperation = operation;
    }
  }];
  return operation;
}
```

另外，每个下载操作的超时时间可以通过downloadTimeout属性来设置，默认值为15秒。

下载操作

每个图片的下载都是一个Operation操作。我们在上面分析过这个操作的创建及加入操作队列的过程。现在我们来看看单个操作的具体实现。

SDWebImage定义了一个协议，即 **SDWebImageOperation** 作为图片下载操作的基础协议。它只声明了一个cancel方法，用于取消操作。协议的具体声明如下：

```
@protocol SDWebImageOperation <NSObject>

- (void)cancel;

@end
```

SDWebImage自定义了一个Operation类，即 **SDWebImageDownloaderOperation** ，它继承自NSOperation，并采用了SDWebImageOperation协议。除了继承而来的方法，该类只向外暴露了一个方法，即上面所用到的初始化方法initWithRequest:options:progress:completed:cancelled:。

对于图片的下载，SDWebImageDownloaderOperation完全依赖于URL加载系统中的NSURLConnection类(并未使用7.0以后的NSURLSession类)。我们先来分析一下SDWebImageDownloaderOperation类中对于图片实际数据的下载处理，即NSURLConnection各代理方法的实现。

首先，SDWebImageDownloaderOperation在分类中采用了NSURLConnectionDataDelegate协议，并实现了该协议的以下几个方法：

```
- connection:didReceiveResponse:
- connection:didReceiveData:
- connectionDidFinishLoading:
- connection:didFailWithError:
- connection:willCacheResponse:
- connectionShouldUseCredentialStorage:
- connection:willSendRequestForAuthenticationChallenge:
```

我们在此不逐一分析每个方法的实现，就重点分析一下-connection:didReceiveData:方法。该方法的主要任务是接收数据。每次接收到数据时，都会用现有的数据创建一个CGImageSourceRef对象以做处理。在首次获取到数据时(width+height==0)会从这些包含图像信息的数据中取出图像的长、宽、方向等信息以备使用。而后在图片下载完成之前，会使用CGImageSourceRef对象创建一个图片对象，经过缩放、解压缩操作后生成一个UIImage对象供完成回调使用。当然，在这个方法中还需要处理的就是进度信息。如果我们有设置进度回调的话，就调用这个进度回调以处理当前图片的下载进度。

注：缩放操作可以查看SDWebImageCompat文件中的SDScaledImageForKey函数；解压缩操作可以查看SDWebImageDecoder文件+decodedImageWithImage方法

```
- (void)connection:(NSURLConnection *)connection didReceiveData:(NSData *)data {
  // 1. 附加数据
  [self.imageData appendData:data];
  if ((self.options & SDWebImageDownloaderProgressiveDownload) && self.expectedSize > 0 && self
    // 2. 获取已下载数据总大小
    const NSInteger totalSize = self.imageData.length;
    // 3. 更新数据源，我们需要传入所有数据，而不仅仅是新数据
    CGImageSourceRef imageSource = CGImageSourceCreateWithData((__bridge CFDataRef)self.imageDa
    // 4. 首次获取到数据时，从这些数据中获取图片的长、宽、方向属性值
    if (width + height == 0) {
      CFDictionaryRef properties = CGImageSourceCopyPropertiesAtIndex(imageSource, 0, NULL);
      if (properties) {
        NSInteger orientationValue = -1;
        CFTypeRef val = CFDictionaryGetValue(properties, kCGImagePropertyPixelHeight);
        if (val) CFNumberGetValue(val, kCFNumberLongType, &height);
        ...
        CFRelease(properties);
        // 5. 当绘制到Core Graphics时，我们会丢失方向信息，这意味着有时候由initWithCGIImage创建的图片
        //       的方向会不对，所以在这边我们先保存这个信息并在后面使用。
        orientation = [[self class] orientationFromPropertyValue:(orientationValue == -1 ? 1 :
      }
    }
    // 6. 图片还未下载完成
    if (width + height > 0 && totalSize < self.expectedSize) {
      // 7. 使用现有的数据创建图片对象，如果数据中存有多张图片，则取第一张
      CGImageRef partialImageRef = CGImageSourceCreateImageAtIndex(imageSource, 0, NULL);
#ifdef TARGET_OS_IPHONE
```

```
      // 8. 适用于iOS变形图像的解决方案。我的理解是由于iOS只支持RGB颜色空间，所以在此对下载下来的图片做个颜色
      if (partialImageRef) {
        const size_t partialHeight = CGImageGetHeight(partialImageRef);
        CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
        CGContextRef bmContext = CGBitmapContextCreate(NULL, width, height, 8, width * 4, colo
        CGColorSpaceRelease(colorSpace);
        if (bmContext) {
          CGContextDrawImage(bmContext, (CGRect){.origin.x = 0.0f, .origin.y = 0.0f, .size.widt
          CGImageRelease(partialImageRef);
          partialImageRef = CGBitmapContextCreateImage(bmContext);
          CGContextRelease(bmContext);
        }
        else {
          CGImageRelease(partialImageRef);
          partialImageRef = nil;
        }
      }
#endif
      // 9. 对图片进行缩放、解码操作
      if (partialImageRef) {
        UIImage *image = [UIImage imageWithCGImage:partialImageRef scale:1 orientation:orientat
        NSString *key = [[SDWebImageManager sharedManager] cacheKeyForURL:self.request.URL];
        UIImage *scaledImage = [self scaledImageForKey:key image:image];
        image = [UIImage decodedImageWithImage:scaledImage];
        CGImageRelease(partialImageRef);
        dispatch_main_sync_safe(^{
          if (self.completedBlock) {
            self.completedBlock(image, nil, nil, NO);
          }
        });
      }
    }
    CFRelease(imageSource);
  }
  if (self.progressBlock) {
    self.progressBlock(self.imageData.length, self.expectedSize);
  }
}
```

我们前面说过SDWebImageDownloaderOperation类是继承自NSOperation类。它没有简单的实现main方法，而是采用更加灵活的start方法，以便自己管理下载的状态。

在start方法中，创建了我们下载所使用的NSURLConnection对象，开启了图片的下载，同时抛出一个下载开始的通知。当然，如果我们期望下载在后台处理，则只需要配置我们的下载选项，使其包含SDWebImageDownloaderContinueInBackground选项。start方法的具体实现如下：

```
- (void)start {
  @synchronized (self) {
    // 管理下载状态，如果已取消，则重置当前下载并设置完成状态为YES
    if (self.isCancelled) {
      self.finished = YES;
      [self reset];
      return;
    }
#if TARGET_OS_IPHONE && __IPHONE_OS_VERSION_MAX_ALLOWED >= __IPHONE_4_0
    // 1. 如果设置了在后台执行，则进行后台执行
    if ([self shouldContinueWhenAppEntersBackground]) {
      __weak __typeof__ (self) wself = self;
      self.backgroundTaskId = [[UIApplication sharedApplication] beginBackgroundTaskWithExpirat
        ...
        }
      }];
    }
#endif
```

```
    self.executing = YES;
    self.connection = [[NSURLConnection alloc] initWithRequest:self.request delegate:self start
    self.thread = [NSThread currentThread];
  }
  [self.connection start];
  if (self.connection) {
    if (self.progressBlock) {
      self.progressBlock(0, NSURLResponseUnknownLength);
    }
    // 2. 在主线程抛出下载开始通知
    dispatch_async(dispatch_get_main_queue(), ^{
      [[NSNotificationCenter defaultCenter] postNotificationName:SDWebImageDownloadStartNotific
    });
    // 3. 启动run loop
    if (floor(NSFoundationVersionNumber) <= NSFoundationVersionNumber_iOS_5_1) {
      CFRunLoopRunInMode(kCFRunLoopDefaultMode, 10, false);
    }
    else {
      CFRunLoopRun();
    }
    // 4. 如果未完成，则取消连接
    if (!self.isFinished) {
      [self.connection cancel];
      [self connection:self.connection didFailWithError:[NSError errorWithDomain:NSURLErrorDoma
    }
  }
  else {
    ...
  }
#if TARGET_OS_IPHONE && __IPHONE_OS_VERSION_MAX_ALLOWED >= __IPHONE_4_0
  if (self.backgroundTaskId != UIBackgroundTaskInvalid) {
    [[UIApplication sharedApplication] endBackgroundTask:self.backgroundTaskId];
    self.backgroundTaskId = UIBackgroundTaskInvalid;
  }
#endif
}
```

当然，在下载完成或下载失败后，需要停止当前线程的run loop，清除连接，并抛出下载停止的通知。如果下载成功，则会处理完整的图片数据，对其进行适当的缩放与解压缩操作，以提供给完成回调使用。具体可参考-connectionDidFinishLoading:与-connection:didFailWithError:的实现。

小结

下载的核心其实就是利用NSURLConnection对象来加载数据。每个图片的下载都由一个Operation操作来完成，并将这些操作放到一个操作队列中。这样可以实现图片的并发下载。

缓存

为了减少网络流量的消耗，我们都希望下载下来的图片缓存到本地，下次再去获取同一张图片时，可以直接从本地获取，而不再从远程服务器获取。这样做的另一个好处是提升了用户体验，用户第二次查看同一幅图片时，能快速从本地获取图片直接呈现给用户。

SDWebImage提供了对图片缓存的支持，而该功能是由SDImageCache类来完成的。该类负责处理内存缓存及一个可选的磁盘缓存。其中磁盘缓存的写操作是异步的，这样就不会对UI操作造成影响。

内存缓存及磁盘缓存

内存缓存的处理是使用NSCache对象来实现的。NSCache是一个类似于集合的容器。它存储key-value对，这一点类似于NSDictionary类。我们通常用使用缓存来临时存储短时间使用但创建昂贵的对象。重用这些对象可以优化性能，因为它们的值不需要重新计算。另外一方面，这些对象对于程序来说不是紧要的，在内存紧张时会被丢弃。

磁盘缓存的处理则是使用NSFileManager对象来实现的。图片存储的位置是位于Cache文件夹。另外，SDImageCache还定义了一个串行队列，来异步存储图片。

内存缓存与磁盘缓存相关变量的声明及定义如下：

```
@interface SDImageCache ()
@property (strong, nonatomic) NSCache *memCache;
@property (strong, nonatomic) NSString *diskCachePath;
@property (strong, nonatomic) NSMutableArray *customPaths;
@property (SDDispatchQueueSetterSementics, nonatomic) dispatch_queue_t ioQueue;
@end
- (id)initWithNamespace:(NSString *)ns {
    if ((self = [super init])) {
  NSString *fullNamespace = [@"com.hackemist.SDWebImageCache." stringByAppendingString:ns];
  ...
  _ioQueue = dispatch_queue_create("com.hackemist.SDWebImageCache", DISPATCH_QUEUE_SERIAL);
  ...
  // Init the memory cache
  _memCache = [[NSCache alloc] init];
  _memCache.name = fullNamespace;
  // Init the disk cache
  NSArray *paths = NSSearchPathForDirectoriesInDomains(NSCachesDirectory, NSUserDomainMask, YES
  _diskCachePath = [paths[0] stringByAppendingPathComponent:fullNamespace];
  dispatch_sync(_ioQueue, ^{
      _fileManager = [NSFileManager new];
  });
  ...
    }
    return self;
}
```

SDImageCache提供了大量方法来缓存、获取、移除及清空图片。而对于每个图片，为了方便地在内存或磁盘中对它进行这些操作，我们需要一个key值来索引它。在内存中，我们将其作为NSCache的key值，而在磁盘中，我们用这个key作为图片的文件名。对于一个远程服务器下载的图片，其url是作为这个key的最佳选择了。我们在后面会看到这个key值的重要性。

存储图片

我们先来看看图片的缓存操作，该操作会在内存中放置一份缓存，而如果确定需要缓存到磁盘，则将磁盘缓存操作作为一个task放到串行队列中处理。在iOS中，会先检测图片是PNG还是JPEG，并将其转换为相应的图片数据，最后将数据写入到磁盘中(文件名是对key值做MD5摘要后的串)。缓存操作的基础方法是-storeImage:recalculateFromImage:imageData:forKey:toDisk，它的具体实现如下：

```
- (void)storeImage:(UIImage *)image recalculateFromImage:(BOOL)recalculate imageData:(NSData *)
  ...
  // 1. 内存缓存，将其存入NSCache中，同时传入图片的消耗值
  [self.memCache setObject:image forKey:key cost:image.size.height * image.size.width * image.s
  if (toDisk) {
    // 2. 如果确定需要磁盘缓存，则将缓存操作作为一个任务放入ioQueue中
    dispatch_async(self.ioQueue, ^{
      NSData *data = imageData;
```

```
      if (image && (recalculate || !data)) {
#if TARGET_OS_IPHONE
        // 3. 需要确定图片是PNG还是JPEG。PNG图片容易检测，因为有一个唯一签名。PNG图像的前8个字节总是包含以下
        // 在imageData为nil的情况下假定图像为PNG。我们将其当作PNG以避免丢失透明度。而当有图片数据时，我们检
        BOOL imageIsPng = YES;
        if ([imageData length] >= [kPNGSignatureData length]) {
          imageIsPng = ImageDataHasPNGPreffix(imageData);
        }
        if (imageIsPng) {
          data = UIImagePNGRepresentation(image);
        }
        else {
          data = UIImageJPEGRepresentation(image, (CGFloat)1.0);
        }
#else
        data = [NSBitmapImageRep representationOfImageRepsInArray:image.representations usingTy
#endif
      }
      // 4. 创建缓存文件并存储图片
      if (data) {
        if (![_fileManager fileExistsAtPath:_diskCachePath]) {
          [_fileManager createDirectoryAtPath:_diskCachePath withIntermediateDirectories:YES at
        }
        [_fileManager createFileAtPath:[self defaultCachePathForKey:key] contents:data attribut
      }
    });
  }
}
```

查询图片

如果我们想在内存或磁盘中查询是否有key指定的图片，则可以分别使用以下方法：

```
- (UIImage *)imageFromMemoryCacheForKey:(NSString *)key;
- (UIImage *)imageFromDiskCacheForKey:(NSString *)key;
```

而如果只是想查看本地是否在key指定的图片，则不管是在内存还是在磁盘上，则可以使用以下方法：

```
- (NSOperation *)queryDiskCacheForKey:(NSString *)key done:(SDWebImageQueryCompletedBlock)doneI
  ...
  // 1. 首先查看内存缓存，如果查找到，则直接回调doneBlock并返回
  UIImage *image = [self imageFromDiskCacheForKey:key];
  if (image) {
    doneBlock(image, SDImageCacheTypeMemory);
    return nil;
  }
  // 2. 如果内存中没有，则在磁盘中查找。如果找到，则将其放到内存缓存，并调用doneBlock回调
  NSOperation *operation = [NSOperation new];
  dispatch_async(self.ioQueue, ^{
    if (operation.isCancelled) {
      return;
    }
    @autoreleasepool {
      UIImage *diskImage = [self diskImageForKey:key];
      if (diskImage) {
        CGFloat cost = diskImage.size.height * diskImage.size.width * diskImage.scale * diskIma
        [self.memCache setObject:diskImage forKey:key cost:cost];
      }
      dispatch_async(dispatch_get_main_queue(), ^{
        doneBlock(diskImage, SDImageCacheTypeDisk);
      });
```

```
    }
  });
  return operation;
}
```

移除图片

图片的移除操作则可以使用以下方法：

```
- (void)removeImageForKey:(NSString *)key;
- (void)removeImageForKey:(NSString *)key withCompletion:(SDWebImageNoParamsBlock)completion;
- (void)removeImageForKey:(NSString *)key fromDisk:(BOOL)fromDisk;
- (void)removeImageForKey:(NSString *)key fromDisk:(BOOL)fromDisk withCompletion:(SDWebImageNoI
```

我们可以选择同时移除内存及磁盘上的图片。

清理图片

磁盘缓存图片的清理操作可以分为完全清空和部分清理。完全清空操作是直接把缓存的文件夹移除，清空操作有以下两个方法：

```
- (void)clearDisk;
- (void)clearDiskOnCompletion:(SDWebImageNoParamsBlock)completion;
```

而部分清理则是根据我们设定的一些参数值来移除一些文件，这里主要有两个指标：文件的缓存有效期及最大缓存空间大小。文件的缓存有效期可以通过maxCacheAge属性来设置，默认是1周的时间。如果文件的缓存时间超过这个时间值，则将其移除。而最大缓存空间大小是通过maxCacheSize属性来设置的，如果所有缓存文件的总大小超过这一大小，则会按照文件最后修改时间的逆序，以每次一半的递归来移除那些过早的文件，直到缓存的实际大小小于我们设置的最大使用空间。清理的操作在-cleanDiskWithCompletionBlock:方法中，其实现如下：

```
- (void)cleanDiskWithCompletionBlock:(SDWebImageNoParamsBlock)completionBlock {
  dispatch_async(self.ioQueue, ^{
    NSURL *diskCacheURL = [NSURL fileURLWithPath:self.diskCachePath isDirectory:YES];
    NSArray *resourceKeys = @[NSURLIsDirectoryKey, NSURLContentModificationDateKey, NSURLTotalI
    // 1. 该枚举器预先获取缓存文件的有用的属性
    NSDirectoryEnumerator *fileEnumerator = [_fileManager enumeratorAtURL:diskCacheURL
                               includingPropertiesForKeys:resourceKeys
                                                  options:NSDirectoryEnumerationSkipsHiddenFiles
                                             errorHandler:NULL];
    NSDate *expirationDate = [NSDate dateWithTimeIntervalSinceNow:-self.maxCacheAge];
    NSMutableDictionary *cacheFiles = [NSMutableDictionary dictionary];
    NSUInteger currentCacheSize = 0;
    // 2. 枚举缓存文件夹中所有文件，该迭代有两个目的：移除比过期日期更老的文件；存储文件属性以备后面执行基于缓存
    NSMutableArray *urlsToDelete = [[NSMutableArray alloc] init];
    for (NSURL *fileURL in fileEnumerator) {
      NSDictionary *resourceValues = [fileURL resourceValuesForKeys:resourceKeys error:NULL];
      // 3. 跳过文件夹
      if ([resourceValues[NSURLIsDirectoryKey] boolValue]) {
        continue;
      }
      // 4. 移除早于有效期的老文件
      NSDate *modificationDate = resourceValues[NSURLContentModificationDateKey];
      if ([[modificationDate laterDate:expirationDate] isEqualToDate:expirationDate]) {
        [urlsToDelete addObject:fileURL];
```

```
            continue;
        }
        // 5. 存储文件的引用并计算所有文件的总大小，以备后用
        NSNumber *totalAllocatedSize = resourceValues[NSURLTotalFileAllocatedSizeKey];
        currentCacheSize += [totalAllocatedSize unsignedIntegerValue];
        [cacheFiles setObject:resourceValues forKey:fileURL];
    }
    for (NSURL *fileURL in urlsToDelete) {
        [_fileManager removeItemAtURL:fileURL error:nil];
    }
    // 6.如果磁盘缓存的大小大于我们配置的最大大小，则执行基于文件大小的清理，我们首先删除最老的文件
    if (self.maxCacheSize > 0 && currentCacheSize > self.maxCacheSize) {
        // 7. 以设置的最大缓存大小的一半作为清理目标
        const NSUInteger desiredCacheSize = self.maxCacheSize / 2;
        // 8. 按照最后修改时间来排序剩下的缓存文件
        NSArray *sortedFiles = [cacheFiles keysSortedByValueWithOptions:NSSortConcurrent
                                usingComparator:^NSComparisonResult(id obj1, id obj2) {
                                    return [obj1[NSURLContentModificationDateKey] compare:obj2[NSUR
                                }];
        // 9. 删除文件，直到缓存总大小降到我们期望的大小
        for (NSURL *fileURL in sortedFiles) {
            if ([_fileManager removeItemAtURL:fileURL error:nil]) {
                NSDictionary *resourceValues = cacheFiles[fileURL];
                NSNumber *totalAllocatedSize = resourceValues[NSURLTotalFileAllocatedSizeKey];
                currentCacheSize -= [totalAllocatedSize unsignedIntegerValue];
                if (currentCacheSize < desiredCacheSize) {
                    break;
                }
            }
        }
    }
                if (completionBlock) {
        dispatch_async(dispatch_get_main_queue(), ^{
            completionBlock();
        });
    }
}
```

小结

以上分析了图片缓存操作，当然，除了上面讲的几个操作，SDImageCache类还提供了一些辅助方法。如获取缓存大小、缓存中图片的数量、判断缓存中是否存在某个key指定的图片。另外，SDImageCache类提供了一个单例方法的实现，所以我们可以将其当作单例对象来处理。

SDWebImageManager

在实际的运用中，我们并不直接使用SDWebImageDownloader类及SDImageCache类来执行图片的下载及缓存。为了方便用户的使用，SDWebImage提供了SDWebImageManager对象来管理图片的下载与缓存。而且我们经常用到的诸如UIImageView+WebCache等控件的分类都是基于SDWebImageManager对象的。该对象将一个下载器和一个图片缓存绑定在一起，并对外提供两个只读属性来获取它们，如下代码所示：

```
@interface SDWebImageManager : NSObject

@property (weak, nonatomic) id <SDWebImageManagerDelegate> delegate;

@property (strong, nonatomic, readonly) SDImageCache *imageCache;
@property (strong, nonatomic, readonly) SDWebImageDownloader *imageDownloader;
```

```
...

@end
```

从上面的代码中我们还可以看到有一个delegate属性，其是一个id<SDWebImageManagerDelegate>对象。SDWebImageManagerDelegate声明了两个可选实现的方法，如下所示：

```
// 控制当图片在缓存中没有找到时，应该下载哪个图片
- (BOOL)imageManager:(SDWebImageManager *)imageManager shouldDownloadImageForURL:(NSURL *)image

// 允许在图片已经被下载完成且被缓存到磁盘或内存前立即转换
- (UIImage *)imageManager:(SDWebImageManager *)imageManager transformDownloadedImage:(UIImage *
```

这两个代理方法会在SDWebImageManager的-downloadImageWithURL:options:progress:completed:方法中调用，而这个方法是SDWebImageManager类的核心所在。我们来看看它的具体实现：

```
- (id <SDWebImageOperation>)downloadImageWithURL:(NSURL *)url
                                         options:(SDWebImageOptions)options
                                        progress:(SDWebImageDownloaderProgressBlock)progressBlock
                                       completed:(SDWebImageCompletionWithFinishedBlock)completedBlock {
    ...
    // 前面省略n行。主要作了如下处理：
    // 1. 判断url的合法性
    // 2. 创建SDWebImageCombinedOperation对象
    // 3. 查看url是否是之前下载失败过的
    // 4. 如果url为nil，或者在不可重试的情况下是一个下载失败过的url，则直接返回操作对象并调用完成回调
    operation.cacheOperation = [self.imageCache queryDiskCacheForKey:key done:^(UIImage *image, S
        ...
    if ((!image || options & SDWebImageRefreshCached) && (![self.delegate respondsToSelector:@s
        // 下载
        id <SDWebImageOperation> subOperation = [self.imageDownloader downloadImageWithURL:url op
            if (weakOperation.isCancelled) {
                // 操作被取消，则不做任务事情
            }
            else if (error) {
                // 如果出错，则调用完成回调，并将url放入下载挫败url数组中
                ...
            }
            else {
                BOOL cacheOnDisk = !(options & SDWebImageCacheMemoryOnly);
                if (options & SDWebImageRefreshCached && image && !downloadedImage) {
                    // Image refresh hit the NSURLCache cache, do not call the completion block
                }
                else if (downloadedImage && (!downloadedImage.images || (options & SDWebImageTransfor
                    // 在全局队列中并行处理图片的缓存
                    // 首先对图片做个转换操作，该操作是代理对象实现的
                    // 然后对图片做缓存处理
                    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{
                        UIImage *transformedImage = [self.delegate imageManager:self transformDownloaded
                        if (transformedImage && finished) {
                            BOOL imageWasTransformed = ![transformedImage isEqual:downloadedImage];
                            [self.imageCache storeImage:transformedImage recalculateFromImage:imageWasTrans
                        }
                        ...
                    });
                }
                else {
                    if (downloadedImage && finished) {
                        [self.imageCache storeImage:downloadedImage recalculateFromImage:NO imageData:dat
                    }
                    ...
                }
            }
```

```
        // 下载完成并缓存后，将操作从队列中移除
        if (finished) {
          @synchronized (self.runningOperations) {
            [self.runningOperations removeObject:operation];
          }
        }
      }];
      // 设置取消回调
      operation.cancelBlock = ^{
        [subOperation cancel];
        @synchronized (self.runningOperations) {
          [self.runningOperations removeObject:weakOperation];
        }
      };
    }
    else if (image) {
      ...
    }
    else {
      ...
    }
  }];
  return operation;
}
```

对于这个方法，我们没有做过多的解释。其主要就是下载图片并根据操作选项来缓存图片。上面这个下载方法中的操作选项参数是由枚举SDWebImageOptions来定义的，这个操作中的一些选项是与SDWebImageDownloaderOptions中的选项对应的。我们来看看这个SDWebImageOptions选项都有哪些：

```
typedef NS_OPTIONS(NSUInteger, SDWebImageOptions) {
  // 默认情况下，当URL下载失败时，URL会被列入黑名单，导致库不会再去重试，该标记用于禁用黑名单
  SDWebImageRetryFailed = 1 << 0,
  // 默认情况下，图片下载开始于UI交互，该标记禁用这一特性，这样下载延迟到UIScrollView减速时
  SDWebImageLowPriority = 1 << 1,
  // 该标记禁用磁盘缓存
  SDWebImageCacheMemoryOnly = 1 << 2,
  // 该标记启用渐进式下载，图片在下载过程中是渐渐显示的，如同浏览器一下。
  // 默认情况下，图像在下载完成后一次性显示
  SDWebImageProgressiveDownload = 1 << 3,
  // 即使图片缓存了，也期望HTTP响应cache control，并在需要的情况下从远程刷新图片。
  // 磁盘缓存将被NSURLCache处理而不是SDWebImage，因为SDWebImage会导致轻微的性能下载。
  // 该标记帮助处理在相同请求URL后面改变的图片。如果缓存图片被刷新，则完成block会使用缓存图片调用一次
  // 然后再用最终图片调用一次
  SDWebImageRefreshCached = 1 << 4,
  // 在iOS 4+系统中，当程序进入后台后继续下载图片。这将要求系统给予额外的时间让请求完成
  // 如果后台任务超时，则操作被取消
  SDWebImageContinueInBackground = 1 << 5,
  // 通过设置NSMutableURLRequest.HTTPShouldHandleCookies = YES;来处理存储在NSHTTPCookieStore中的coo
  SDWebImageHandleCookies = 1 << 6,
  // 允许不受信任的SSL认证
  SDWebImageAllowInvalidSSLCertificates = 1 << 7,
  // 默认情况下，图片下载按入队的顺序来执行。该标记将其移到队列的前面，
  // 以便图片能立即下载而不是等到当前队列被加载
  SDWebImageHighPriority = 1 << 8,
  // 默认情况下，占位图片在加载图片的同时被加载。该标记延迟占位图片的加载直到图片已以被加载完成
  SDWebImageDelayPlaceholder = 1 << 9,
  // 通常我们不调用动画图片的transformDownloadedImage代理方法，因为大多数转换代码可以管理它。
  // 使用这个票房则不任何情况下都进行转换。
  SDWebImageTransformAnimatedImage = 1 << 10,
};
```

大家在看-downloadImageWithURL:options:progress:completed:，可以看到两个SDWebImageOptions与SDWebImageDownloaderOptions中的选项是如何对应起来的，在此不多做解释。

视图扩展

我在使用SDWebImage的时候，使用得最多的是UIImageView+WebCache中的针对UIImageView的扩展方法，这些扩展方法将UIImageView与WebCache集成在一起，来让UIImageView对象拥有异步下载和缓存远程图片的能力。其中最核心的方法是-sd_setImageWithURL:placeholderImage:options:progress:completed:，其使用SDWebImageManager单例对象下载并缓存图片，完成后将图片赋值给UIImageView对象的image属性，以使图片显示出来，其具体实现如下：

```
- (void)sd_setImageWithURL:(NSURL *)url placeholderImage:(UIImage *)placeholder options:(SDWebI
  ...
  if (url) {
    __weak UIImageView *wself = self;
    // 使用SDWebImageManager单例对象来下载图片
    id <SDWebImageOperation> operation = [SDWebImageManager.sharedManager downloadImageWithURL:
      if (!wself) return;
      dispatch_main_sync_safe(^{
        if (!wself) return;
        // 图片下载完后显示图片
        if (image) {
          wself.image = image;
          [wself setNeedsLayout];
        } else {
          if ((options & SDWebImageDelayPlaceholder)) {
            wself.image = placeholder;
            [wself setNeedsLayout];
          }
        }
        if (completedBlock && finished) {
          completedBlock(image, error, cacheType, url);
        }
      });
    }];
    [self sd_setImageLoadOperation:operation forKey:@"UIImageViewImageLoad"];
  } else {
    ...
  }
}
```

除了扩展UIImageView之外，SDWebImage还扩展了UIView、UIButton、MKAnnotationView等视图类，大家可以参考源码。

当然，如果不想使用这些扩展，则可以直接使用SDWebImageManager来下载图片，这也是很OK的。

技术点

SDWebImage的主要任务就是图片的下载和缓存。为了支持这些操作，它主要使用了以下知识点：

1. dispatch_barrier_sync函数：该方法用于对操作设置屏幕，确保在执行完任务后才会执行后续操作。该方法常用于确保类的线程安全性操作。

2. NSMutableURLRequest：用于创建一个网络请求对象，我们可以根据需要来配置请求报头等信息。

3. NSOperation及NSOperationQueue：操作队列是Objective-C中一种高级的并发处理方法，现在它是基于GCD来实现的。相对于GCD来说，操作队列的优点是可以取消在任务处理队列中的任务，另外在管理操作间的依赖关系方面也容易一些。对SDWebImage中我们就看到了如何使用依赖将下载顺序设置成后进先出的顺序。

4. NSURLConnection：用于网络请求及响应处理。在iOS7.0后，苹果推出了一套新的网络请求接口，即NSURLSession类。

5. 开启一个后台任务。

6. NSCache类：一个类似于集合的容器。它存储key-value对，这一点类似于NSDictionary类。我们通常用使用缓存来临时存储短时间使用但创建昂贵的对象。重用这些对象可以优化性能，因为它们的值不需要重新计算。另外一方面，这些对象对于程序来说不是紧要的，在内存紧张时会被丢弃。

7. 清理缓存图片的策略：特别是最大缓存空间大小的设置。如果所有缓存文件的总大小超过这一大小，则会按照文件最后修改时间的逆序，以每次一半的递归来移除那些过早的文件，直到缓存的实际大小小于我们设置的最大使用空间。

8. 对图片的解压缩操作：这一操作可以查看SDWebImageDecoder.m中+decodedImageWithImage方法的实现。

9. 对GIF图片的处理

10. 对WebP图片的处理

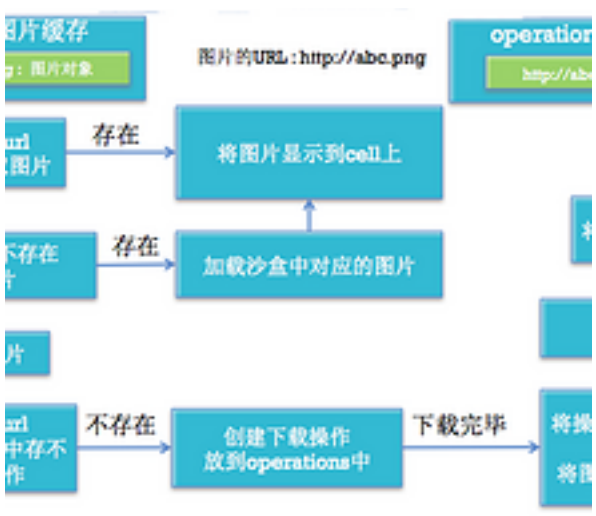
感兴趣的同学可以深入研究一下这些知识点。当然，这只是其中一部分，更多的知识还有待大家去发掘。

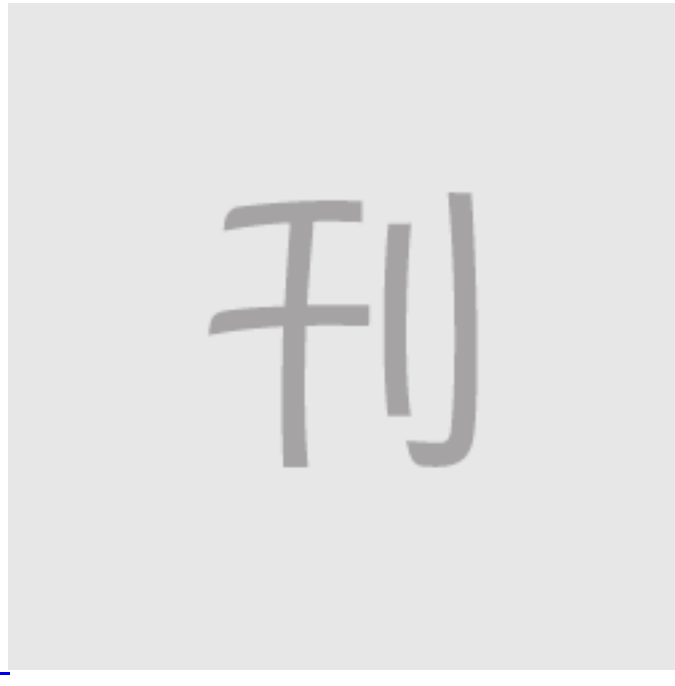
参考

- 1. iOS自动化版本号和编译号
- 2. 让MKMapView变得丰富多彩
- 3. iOS通用缓动动画CustomEasingAnimation
- 4. iOS runtime实战应用：成员变量和属性
- 5. iOS开发编码建议与编程经验（持续更新中）
- 6. 我是怎么踩过在 OSX 上录屏的坑的

相关推刊

- 刊主：小小一颗葱 

《iOS面试》 *16*
- 刊主：X_Liang 

《iOS第三方》 *5*
- 刊主：zongmumask 
《默认推刊》 *1*

我来评几句

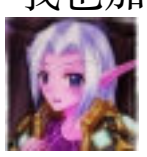
请输入评论内容...

登录后评论

已发表评论数(3)

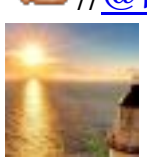
iOS程序犭袁02-08 11:41

我也加班[doge]//@_五虫禾: 哈哈.加班



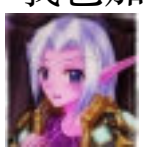
毁小慕02-08 10:55

 //@叶孤城___: sdwebimage源码解析， nice



RobinYongChao02-08 09:27

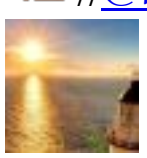
Good



我也加班[doge]//@_五虫禾: 哈哈.加班

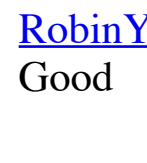
毁小慕02-08 10:55

//@叶孤城___: sdwebimage源码解析， nice



RobinYongChao02-08 09:27

Good



iOS程序犭袁02-08 11:41

相关站点

南峰子的技术博客

+订阅

热门文章

- 1.iOS自动化版本号和编译号

- 2. 让MKMapView变得丰富多彩
- 3. iOS通用缓动动画CustomEasingAnimation
- 4. iOS runtime实战应用：成员变量和属性
- 5. iOS开发编码建议与编程经验（持续更新中）

×

文章纠错

邮箱地址

错误类型 正文不准确

补充信息

提交

网站相关

- 关于我们
- 移动应用
- 建议反馈

关注我们

- 推酷网
- tuicool2012

友情链接

人人都是产品经理 PM256 移动信息化 行晓网 Code4App 智城外包网 虎嗅 IT耳朵 艾瑞网 创媒工场 经理人分享 市场部网 砍柴网 CocoaChina 北风网 云智慧 我赢职场 大数据时代 奇笛网 咕噜网 红联linux Win10之家 鸟哥笔记 爱游戏 投资潮 31会议网 极光推送 Teambition 硅谷网 leangoo 伙伴云表格 ZEALER中国 更多链接>>