Cache 模块

1. Storage.swift

`TimeConstants` `struct` 定义了秒分时天
`StorageExpiration` `enum`类型，用来定义缓存过期时间，
`ExpirationExtending` `enum`类型，用来定义对缓存元素的延期。一般获取到缓存元素后，会对该元素的过期时间进行延期
`CacheCostCalcuable` `Protocol` 类型。又来定义元素所占的缓存大小
`DataTransformable` `Protocol`,用来定义对象转为Data，以及Data转为对象的接口定义。

2. MemoryStorage.swift

`MemoryStorage`定义为`enum`类型的原因是因为`This is a namespace for the memory storage types`,内部有`Backend<T: CacheCostCalcuable>`类型。内部缓存对象使用`NSCache`
```
let storage = NSCache<NSString, StorageObject<T>>()
```
缓存的配置信息由定义在`MemoryStorage` extension里定义的内部struct `Config`定义
```
extension MemoryStorage {
    var totalCostLimit: Int
    var countLimit: Int = .max
    var expiration: StorageExpiration = .second(300)
    var cleanInterval: TimeIOnterval = 120;
}
```
内存缓存默认300秒后过期，默认120秒清理一次过期缓存。
真正缓存在Cache的元素是有StorageObject包裹。由于NSCache无法知道具体缓存的keys.因此Backend内会有
`var keys = Set<String>()` 来记录缓存的Keys
缓存的的接口：
```
func store(value: T, forKey key: String, expiration: StorageExpiration? = nil) throws
```
首先会通过NSLock加锁如果expiration.isExpired 为true，会直接放弃缓存。将value封装为`StorageObject`,然后缓存
```
storage.setObject(object, forKey: key as NSString, cost: value.cacheCost)
keys.insert(key)
```
获取的接口：
```
```

```
  func value(forKey key: String, extendingExpiration: ExpirationExtendi
ng = .cacheTime) -> T?
 ```

 通过`objectForKey`获取到元素。如果`expired` 会返回nil。如果没有过期。则会延长过期时间。
 ```
 object.extendExpiration(extendingExpiration)
 ```
```

### 3. DiskStorage.swift

和`MemoryStorage.swift`类似。不同的是缓存的对象必须遵循`DataTransformable`协议

```
public class Backend<T: DataTransformable>
```

另外获取到磁盘的对象后，对缓存对象的过期时间更高操作放在一个穿行队列异步执行

```
do {
                let data = try Data(contentsOf: fileURL)
                let obj = try T.fromData(data)
                metaChangingQueue.async {
                    meta.extendExpiration(with: fileManager)
                }
                return obj
            } catch {
                throw KingfisherError.cacheError(reason: .cannotLoadDat
aFromDisk(url: fileURL, error: error))

            }
```

缓存的接口:

```
func store(value: T, forKey key: String,expiration: StorageExpiration?
= nil)
```

如果`expiration.expired`为true，则直接return，否则将 value转为Data，由key获取到fileURL，并且配置到file attribute。然后创建文件
```

```
let attributes:[FileAttributeKey : Any] = [
                // The last access date
                .creationDate: now.filterAttributeDate,
                // The estimated expiration date.
                .modificationDate: expiration.estimatedExpirationSinceN
ow.filterAttributeDate
            ]
                        config.fileManager.createFile(atPath: fileURL.p
ath, contents: data, attributes: attributes)
```

获取的接口:

```
func value(forKey key: String, referenceDate: Date, actuallyLoad: Bool)
 throws -> T?
```

由key获取到fileURL，然后封装为 FileMeta对象。这个对象封装了文件attribute的信息，如果对象过期，则丢弃，否则获取到Data。

4. CacheSerializer.swift

这个文件主要定义了各类image类型与data的互转接口，实际上的序列化操作在KingfisherWrapper的extension里

5. ImageCache.swift

缓存的主要管理类，包含了硬盘和内存缓存，和一个串行队列缓存的结果通过`ImageCacheResult`返回

```
public enum ImageCacheResult {
    case disk(Image)
    case memory(Image)
    case none
}
```

缓存的时候先缓存在memory中，后续会选择默认存在硬盘中，存硬盘的操作在异步串行队列中进行。
序列化，过期时间，回调的队列都在`KingfisherParsedOptionsInfo`这个类似小叮当的对象中。
当App进入到后台的，会异步开启一个backgroundTask

```
 var backgroundTask: UIBackgroundTaskIdentifier!
        backgroundTask = sharedApplication.beginBackgroundTask(expirati
onHandler: {
            endBackgroundTask(&backgroundTask!)
        })
        cleanExpiredDiskCache {
            endBackgroundTask(&backgroundTask)
        }
```

背后会移除过期的缓存，以及超出缓存限制的硬盘缓存。这里会先获取所有硬盘缓存对象的URL，compactMap为FileMeta对象，并且根据最后获取日期排序。然后遍历这些对象，对每个fileSize相加，后续对超出sizeLimit的Filemeta对象进行移除操作。

Networking

1.ImageDownloader.swift

这是主要的核心文件！
先说下Delegate文件，这个文件中Delegate类型主要是用来有效防止循环引用的。如果一个block里需要外层的self，可以将block封装成Delegate对象。Delegate自动弱持有了self，并在block的方法实现里将self返回。而不需要在外层手动weak self。下面有很好的实践

```
  let onCompleted = completionHandler.map {
                block -> Delegate<Result<ImageLoadingResult, Kingfisher
Error>, Void> in
                let delegate = Delegate<Result<ImageLoadingResult, King
fisherError>, Void>()
                delegate.delegate(on: self) { (self, callback) in
                    block(callback)
                }
                return delegate

            }
```

```
sessionTask.onTaskDone.delegate(on: self) { (self, done) in
                    let (result, callbacks) = done
}
```

每个请求都会生成一个callback对象

```
let callbak = SessionDataTask.TaskCallback(
                onCompleted: onCompleted, options: options
            )
```

但是相同url的请求只会生成一个`DownloadTask`。这个请求的callback保存在`sessionDelegate`

```
let downloadTask: DownloadTask
            if let existingTask = sessionDelegate.task(for: url) {
                downloadTask = sessionDelegate.append(existingTask, url
: url, callback: callbak)
            } else {
                let sessionDataTask = session.dataTask(with: request)
                sessionDataTask.priority = options.downloadPriority
                downloadTask = sessionDelegate.add(sessionDataTask, url
: url, callback: callbak)
            }
```

`sessionDelegate`将新生成`URLSessionDataTask`封装为为`DownloadTask`，并且以url为key保存在`sessionDelegate`的tasks里，sessionDelegate遵循了URLSessionDelegate协议，在协议方法里根据 url获取到相应的task。

这中间的callback，result，completionBlcok以及options的传输均封装在了`SessionDataTask.TaskCallback`中

```
struct TaskCallback {
        let onCompleted: Delegate<Result<ImageLoadingResult, Kingfisher
Error>, Void>?
        let options: KingfisherParsedOptionsInfo
    }
```

同一url请求公用一个`SessionDataTask`回调以数组的形式保存在task中

```
 private var callbackStore = [CancelToken: TaskCallback]()
```

Kingfisher

1. Kingfisher 内部的KingfisherCompatibleValue和KingfisherCompatible起的是一个namespace的作用。例如给String拓展一个方法，如果extension string的话，所有的String都可以string.method，但是通过如下方法，只能string.kf.method调用
```
extension String: KingfisherCompatibleValue { } extension
KingfisherWrapper where Base == String { var md5: String {} }
```
2. KingfisherOptionsInfo 这是一个比较重要的类型。框架内部的回调，参数传递，其实都借用了这个类。

重要文件

我看来比较有意思的几个文件，Delegate，Kingfisher， Downloader，SessionDelegate，KingfisherOptionsInfo。

总结

当Kingfisher从本地或者server加载图片的时候，首先会将源封装为Resource或者Provider。如果是server的图片，首先会判断是否在MemoryCache中，如果在并且没有过期，则直接返回，并且更新过期时间。如果没有在MemoryCache中，则判断是否在DiskCache中，如果有并且没有过期，则返回同时塞到MemoryCache中。如果缓存中都没有，则会生成Request，并且根据options里的modifier，决定是否modifier request。将complete closure封装为Delegate对象， 弱持有Downloader。将options和completion delegate。封装为SessionDataTask.TaskCallBack对象callback，sessionDelegate根据url判断是否已经有DownloadTask,如果有，则直接将callback添加到downloadTask的callbackStore中。如果没有DownloadTask，则由session根据request生成URLSessionDataTask，再由sessionDelegate根据URLSessionDataTask，和callback生成DownloadTask，sessionDelegate持有DownloadTask，以url为key。然后downloadTask.resume 开启下载。
URLSessionDelegate的回调在SessionDelegate中，在回调方法里，会根据url获取到对应的downloadTask。downloadTask持有options，可以进行progress，redirectHandler回调。网络下载图片成功后，会利用processor根据options对图片进行处理缩小，反转等处理。处理完毕会在KingfisherManager层收到回调，对图片进行缓存之后，返回给业务层。