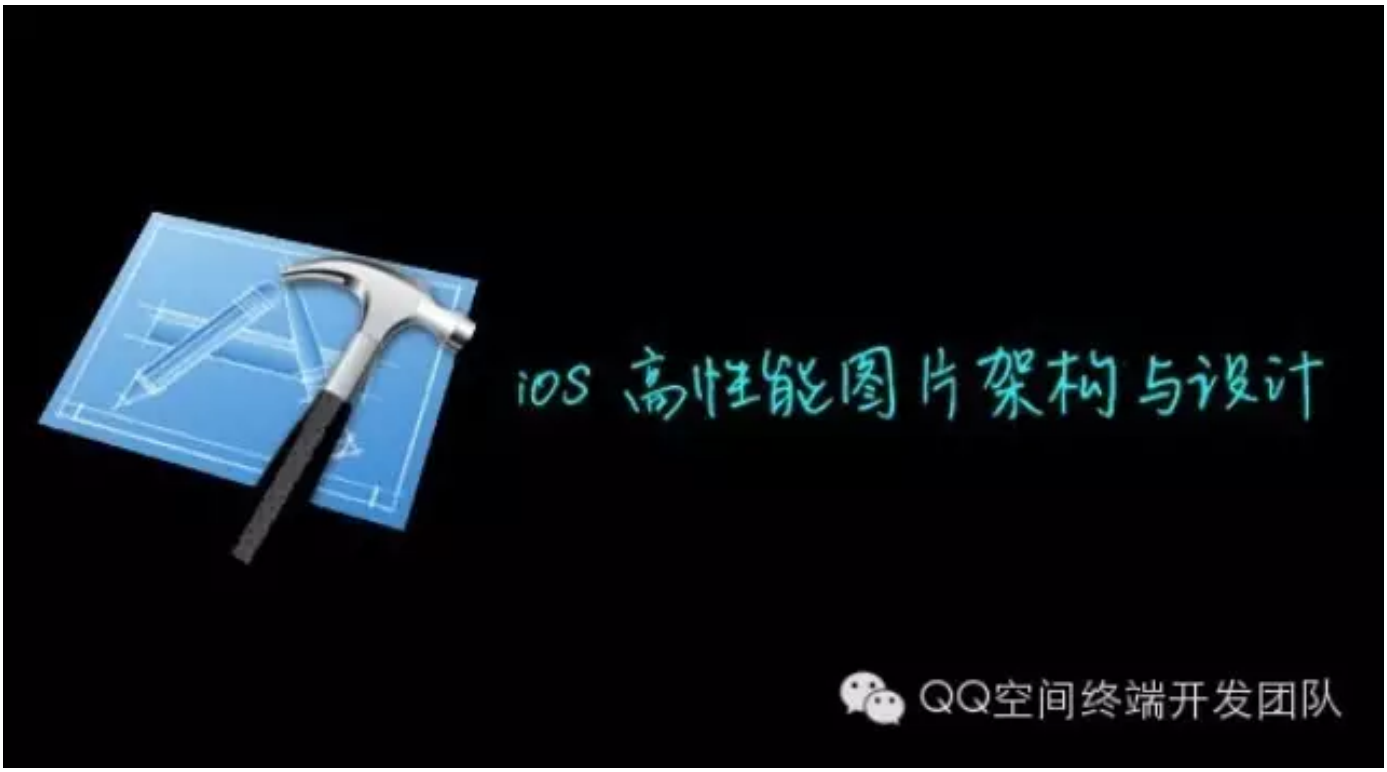
iOS高性能图片架构与设计

2015-10-15 柯灵杰 QQ空间开发团队



1 组件简介

一个优秀的图片组件应该具有这些特性：集并发控制，请求合并，下载，缓存，缓存自动淘汰，图片处理，动画的从数据源到图片显示的一站式解决方案。做到图片加载展示如丝般顺滑。

支持autolayout和代码布局，提供了对网络图片、系统相册图片、本地图片的加载与现实的支持。经过实际项目检验，性能优秀、运行稳定。

高度可定制性：可以和现有的任何下载组件和图片处理组件协同工作。

1.1 核心特性

1.1.1 框架的结构设计具有很强的兼容性和扩展性

使用了桥接模式的中间件设计具有很强的兼容性和拓展性。

现有的项目中往往具有成型的下载组件，相册图片加载组件等相关图片加载组件。如果图片组件也具有相同功能就显得多余且不利于统一管理，而如果没有这些功能，又难以方便的使用。

为了解决这个问题。本框架仅仅实现了表层的图片显示，中间层的请求调度，对于底层的数据加载和数据处理采用插件化的形式，可以很轻易的桥接其他组件，形成一站式的解决方案。

1.1.2 将图片进行预处理来降低内存的消耗和增加渲染速度

手机的内存是十分有限的，下载下来的图片大小往往比显示区域更大。这就造成了内存缓存资源的浪费，同时也降低了图片的渲染速度。在显示之前图片需要进行解码，缩放，显示这样的步骤。

为了优化内存，加快速度。框架提供了图片预处理的功能，根据实际的显示大小对图片进行解压、缩放处理，也就是后台线程预绘制。这样能降低图片占用的内存，并大幅提高图片的显示速度。

1.1.3 优秀的缓存淘汰算法，对于缓存命中有很大的提升

常规的图片缓存往往对图片数量进行限制，进行简单粗暴FIFO淘汰。但是本框架提供了高效的图片大小估算方式，并可选的限制缓存总大小或是缓存图片数量，根据图片的使用频率和时间进行智能化的淘汰。并对于批量突发图片对于缓存的污染有良好的防范能力。

1.2 主要特性

1.2.1 使用简单

只需要设置一个参数即可开始工作，其他参数都是可选的

1.2.2 高度可定制性，可以和任意下载组件和图片处理组件协同工作

插件化的设计使得组件可以轻易的加入到任何的项目中，和项目中已有的下载和图片处理组件协同工作。

1.2.3 支持autolayout、storyboard和代码布局

同时支持使用代码布局、autolayout布局、放入storyboard等多种操作方式，适应性强。

1.2.4 支持并发控制

提供对多个请求的可配置并发数量控制。并对I/O密集型和CPU密集型操作分别进行控制，提高程序的性能，减少调度的开销。

1.2.5 支持多种缩放模式

提供了对十几种常见缩放模式的支持，无需额外接入图片处理组件即可对图片进行初步的处理。

1.2.6 使用后台线程加载、绘制

核心逻辑运行运行在后台线程，实现异步的图片加载和处理，高效提高组件的运行效率。

1.2.7 高效的缓存

提供完善的缓存机制，大幅提高缓存的命中率，加快图片的显示速度。

1.2.8 支持预绘制，减少UI线程压力

根据实际的显示大小对图片进行解压、缩放处理，也就是后台线程预绘制。这样能降低图片占用的内存，并大幅提高图片的显示速度。

1.2.9 定制化进度条，失败、加载图片

可以设置图片的进度条，失败或加载状态显示的图片。

1.2.10 渐变显示动画

支持图片加载完成后的渐变显示动画，使图片的显示更加平滑。

2 基本结构

组件由QZImageView、QZImageManager、QZImageCache、QZImageLoader、QZImageProcessor五大部分组成它们分别负责图片显示，请求管理，缓存，数据加载，数据处理。

3 图片显示Qzimageview

QZImageView继承自UIView聚合UIImageView，实现对上层UIImageView的委托，对外提供操作的接口。在layoutSubviews时，对QZImageManager发起图片请求。收到QZImageManager传回的图片后显示在屏幕上。

4 请求管理Qzimagemanager

QZImageManager使用单例模式，对于所有QZImageView发来的请求进行统一处理。并根据请求进行下一步的操作。

当收到一个新的请求时。QZImageManager首先进行重复请求判断，对于多个不同QZImageView发来的相同的请求进行合并，加入TaskQueue中，在请求完成之后一同回调，防止重复请求。

然后，根据请求向QZImageCache查询内存缓存，如果缓存存在，则返回缓存图片，否则向QZImageLoader发起图片请求。

QZImageLoader返回图片之后，将图片传入QZImageProcessor中进行处理，处理完成之后传回给QZImageView进行显示。

5 内存缓存Qzimagecache

QZImageCache使用单例模式，由一个FIFO队列及一个LRU队列以及一个hashmap组成，使用Two Queues缓存淘汰算法。

5.1 LRU算法

当存在热点数据时，LRU的效率很好，但偶发性的、周期性的批量操作会导致LRU命中率急剧下降，缓存污染情况比较严重。

周期性的批量操作，会立即淘汰LRU队列中的大量数据，导致缓存命中率大幅度下降。而APP常规操作中，有大量偶发批量操作，比如：进入页面后立即返回，就是很典型的一种。

所以LRU算法并不是一个非常好的选择。

5.2 Two Queues算法

Two queues（2Q）算法是LRU的改进版，含有一个FIFO队列及一个LRU队列。

算法流程：

新访问的数据插入到FIFO队列；

如果数据在FIFO队列中一直没有被再次访问，则最终按照FIFO规则淘汰；

如果数据在FIFO队列中被再次访问，则将数据移到LRU队列头部；

如果数据在LRU队列再次被访问，则将数据移到LRU队列头部；

LRU队列淘汰末尾的数据。

在收到批量图片请求的时候，LRU队列依然能保持缓存清洁。

6 数据加载Qzimageloader

QZImageLoader使用单例模式和桥接模式。QZImageLoader本身并没有数据加载的功能，而是进行桥接，将其他有这样功能的组件连接起来。

在收到数据请求的时候，识别请求url的类型，将其分发到相应的数据源。比如收到了一个url为http://xxxxx.jpg的请求，发现DownloadSDK具有处理这样的请求的能力，于是将请求转发给DownloadSDK，并把请求回来的数据发回给QZImageManager。

因此，QZImageLoader具有高度的可定制性，可以桥接任意的数据加载、下载组件，实现对网络文件，本地文件，相册文件等数据的加载。

7 图片处理QZImageProcessor

QZImageProcessor使用单例模式和桥接模式。QZImageProcessor本身不进行图像处理，而是桥接任意的图像处理组件。

与QZImageLoader不同的是QZImageProcessor还实现了对多个图片操作的串行连接和并发控制。就像一个流水线一样的，将图片进行多种处理操作之后传回给QZImageManager，并在流水线的入口进行控制，防止堵塞。

因此，QZImageProcessor也具有高度的可定制性，可以轻松的与任意图片处理组件协作工作。

8 完整流程

设置Url或者大小改变，都会触发layoutSubviews。

1. QZImageView显示loadingImage并；
2. QZImageView向QZImageManager请求图片；
3. QZImageManager判断请求是否重复，决定是否合并请求；
4. QZImageManager向QZImageCache查询缓存；
5. QZImageView返回缓存，并提高该缓存在缓存队列中的优先级；
6. 如果没有缓存，QZImageManager向QZImageLoader请求数据；
7. QZImageLoader根据Url不同将请求分发给相应的QZImageDataSource加载数据；
8. QZImageLoader返回加载完成的数据给QZImageManager；
9. QZImageManager将数据发送给QZImageProcessor进行处理；
10. QZImageProcessor根据请求中带有的OperationList对图片进行处理；
11. QZImageProcessor返回处理后的图片给QZImageManager；
12. QZImageManager请求QZImageCache写入新的缓存；
13. QZImageManager返回缓存图片（有缓存时），或处理后的图片；
14. QZImageView显示图片。