面试题专栏

如何监控线上版本APP启动耗时（包含动态库的加载时间）？

如何优化 App 的启动耗时?

动态库和静态库的区别？为什么动态库会影响启动速度?

使用drawRect有什么影响?

单例的原理和弊端?

自动释放池的原理?

UITableView重用机制原理?

类簇的优缺点

类簇是Foundation框架中广泛使用的设计模式。**类簇在公共抽象超类下对多个私有的具体子类进行分组**。以这种方式对类进行分组简化了面向对象框架的公共可见体系结构，而不会降低其功能丰富度。**类簇是基于抽象工厂设计模式的**。

常见的类簇有 NSString、NSArray、NSDictionary等。 **以数组为例**：不管创建的是可变还是不可变的数组，在alloc之后得到的类都是 __NSPlaceholderArray。而当我们 init 一个不可变的空数组之后，得到的是 __NSArray0；如果有且只有一个元素，那就是 __NSSingleObjectArrayI；有多个元素的，叫做 __NSArrayI；init出来一个可变数组的话，都是 __NSArrayM。

优点：

- 可以将抽象基类背后的复杂细节隐藏起来。
- 程序员不会需要记住各种创建对象的具体类实现，简化了开发成本，提高了开发效率。
- 便于进行封装和组件化。
- 减少了 if-else 这样缺乏扩展性的代码。
- 增加新功能支持不影响其他代码。

缺点

- 已有的类簇非常不好扩展。

我们运用类簇的场景：

- 出现 bug 时，可以通过崩溃报告中的类簇关键字，快速定位 bug 位置。
- 在实现一些固定且并不需要经常修改的事物时，可以高效的选择类簇去实现。

例：

- 针对不同版本，不同机型往往需要不同的设置，这时可以选择使用类簇。
- app 的设置页面这种并不需要经常修改的页面，可以使用类簇去创建大量重复的布局代码。

NSOperation 与 GCD 的主要区别?

1. GCD 的核心是 C 语言写的系统服务，执行和操作简单高效，因此 NSOperation 底层也通过 GCD 实现，换个说法就是 NSOperation 是对 GCD 更高层次的抽象，这是他们之间最本质的区别。因此如果希望自定义任务，建议使用 NSOperation；
2. 依赖关系，NSOperation 可以设置两个 NSOperation 之间的依赖，第二个任务依赖于第一个任务完成执行，GCD 无法设置依赖关系，不过可以通过dispatch_barrier_async来实现这种效果；
3. KVO(键值对观察)，NSOperation 和容易判断 Operation 当前的状态(是否执行，是否取消)，对此 GCD 无法通过 KVO 进行判断；
4. 优先级，NSOperation 可以设置自身的优先级，但是优先级高的不一定先执行，GCD 只能设置队列的优先级，无法在执行的 block 设置优先级；
5. 继承，NSOperation 是一个抽象类，实际开发中常用的两个类是 NSInvocationOperation 和 NSBlockOperation，同样我们可以自定义 NSOperation，GCD 执行任务可以自由组装，没有继承那么高的代码复用度；
6. 效率，直接使用 GCD 效率确实会更高效，NSOperation 会多一点开销，但是通过 NSOperation 可以获得依赖，优先级，继承，键值对观察这些优势，相对于多的那么一点开销确实很划算，鱼和熊掌不可得兼，取舍在于开发者自己；

介绍下App启动的完整过程?

1. App启动过程

- 解析Info.plist

1. 加载相关信息，例如如闪屏
2. 沙箱建立、权限检查

- **Mach-O加载**

1. 如果是胖二进制文件，寻找合适当前CPU类别的部分
2. 加载所有依赖的Mach-O文件（递归调用Mach-O加载的方法）
3. 定位内部、外部指针引用，例如字符串、函数等
4. 执行声明为__attribute__((constructor))的C函数
5. 加载类扩展（Category）中的方法
6. C++静态对象加载、调用ObjC的 +load 函数

- **程序执行**

1. main函数
2. 执行UIApplicationMain函数
    a. 创建UIApplication对象
    b. 创建UIApplicationDelegate对象并复制
    c. 读取配置文件info.plist，设置程序启动的一些属性，(关于info.plist的内容可网上搜索下)
    d. 创建应用程序的Main Runloop循环

3. UIApplicationDelegate对象开始处理监听到的事件
    a. 程序启动成功之后，首先调用application:didFinishLaunchingWithOptions:方法，
    b. 如果info.plist文件中配置了启动storyboard文件名，则加载storyboard文件。
    c. 如果没有配置，则根据代码来创建UIWindow---->UIWindow的rootViewController-->显示

SDWebImage原理

SDWebImage

一个为UIImageView提供一个分类来支持远程服务器图片加载的库。

功能简介：

1. 一个添加了web图片加载和缓存管理的UIImageView分类
2. 一个异步图片下载器
3. 一个异步的内存加磁盘综合存储图片并且自动处理过期图片
4. 支持动态gif图
5. 支持webP格式的图片
6. 后台图片解压处理
7. 确保同样的图片url不会下载多次
8. 确保伪造的图片url不会重复尝试下载
9. 确保主线程不会阻塞

工作流程

1. 入口 setImageWithURL:placeholderImage:options: 会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。
2. 进入 SDWebImageManager-downloadWithURL:delegate:options:userInfo:，交给 SDImageCache 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:.
3. 先从内存图片缓存查找是否有图片，如果内存中已经有图片缓存，SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo: 到 SDWebImageManager。
4. SDWebImageManagerDelegate 回调 webImageManager:didFinishWithImage: 到 UIImageView+WebCache 等前端展示图片。
5. 如果内存缓存中没有，生成 NSInvocationOperation 添加到队列开始从硬盘查找图片是否已经缓存。
6. 根据 URLKey 在硬盘缓存目录下尝试读取图片文件。这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调 notifyDelegate:。

7. 如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中（如果空闲内存过小，会先清空内存缓存）。SDImageCacheDelegate 回调 imageCache:didFindImage:forKey:userInfo:。进而回调展示图片。
8. 如果从硬盘缓存目录读取不到图片，说明所有缓存都不存在该图片，需要下载图片，回调 imageCache:didNotFindImageForKey:userInfo:。
9. 共享或重新生成一个下载器 SDWebImageDownloader 开始下载图片。
10. 图片下载由 NSURLConnection 来做，实现相关 delegate 来判断图片下载中、下载完成和下载失败。
11. connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。
12. 图片解码处理在一个 NSOperationQueue 完成，不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，最好也在这里完成，效率会好很多。
13. 在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成，imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。
14. 通知所有的 downloadDelegates 下载完成，回调给需要的地方展示图片。将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。写文件到硬盘也在以单独 NSInvocationOperation 完成，避免拖慢主线程。
15. SDImageCache 在初始化的时候会注册一些消息通知，在内存警告或退到后台的时候清理内存图片缓存，应用结束的时候清理过期图片。
16. SDWI 也提供了 UIButton+WebCache 和 MKAnnotationView+WebCache，方便使用。
17. SDWebImagePrefetcher 可以预先下载图片，方便后续使用。

源码分析

主要用到的对象

一、图片下载

1、 SDWebImageDownloader

a. 单例，图片下载器，负责图片异步下载，并对图片加载做了优化处理
b. 图片的下载操作放在一个NSOperationQueue并发操作队列中，队列默认最大并发数是6
c. 每个图片对应一些回调（下载进度，完成回调等），回调信息会存在downloader的URLCallbacks（一个字典，key是url地址，value是图片下载回调数组）中，URLCallbacks可能被多个线程访问，所以downloader把下载任务放在一个barrierQueue中，并设置屏障保证同一时间只有一个线程访问URLCallbacks。，在创建回调URLCallbacks的block中创建了一个NSOperation并添加到NSOperationQueue中。
d. 每个图片下载都是一个operation类，创建后添加到一个队列中，SDWebimage定义了一个协议 SDWebImageOperation作为图片下载操作的基础协议，声明了一个cancel方法，用于取消

操作。

```
@protocol SDWebImageOperation <NSObject>
-(void)cancel;
@end
```

e. 对于图片的下载，SDWebImageDownloaderOperation完全依赖于NSURLConnection类，继承和实现了NSURLConnectionDataDelegate协议的方法

connection:didReceiveResponse:
connection:didReceiveData:
connectionDidFinishLoading:
connection:didFailWithError:
connection:willCacheResponse:
connectionShouldUseCredentialStorage:
–connection:willSendRequestForAuthenticationChalleng
–connection:didReceiveData:方法，接受数据，创建一个CGImageSourceRef对象，在首次获取数据时（图片width，height），图片下载完成之前，使用CGImageSourceRef对象创建一个图片对象，经过缩放、解压操作生成一个UIImage对象供回调使用，同时还有下载进度处理。
注：缩放：SDWebImageCompat中SDScaledImageForKey函数 解压：SDWebImageDecoder文件中decodedImageWithImage

2、SDWebImageDownloaderOption

a. 继承自NSOperation类，没有简单实现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
b. start方法中创建了下载使用的NSURLConnections对象，开启了图片的下载，并抛出一个下载开始的通知，
c. 小结：下载的核心是利用NSURLSession加载数据，每个图片的下载都有一个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。

3、SDWebImageDecoder（异步对图片进行解码）

二、缓存

减少网络流量，下载完图片后存储到本地，下载再获取同一张图片时，直接从本地获取，提升用户体验，能快速从本地获取呈现给用户。 SDWebImage提供了对图片进行了缓存，主要由SDImageCache完成。该类负责处理内存缓存以及一个可选的磁盘缓存，其中磁盘缓存的写操作是异步的，不会对UI造成影响。

1、内存缓存及磁盘缓存

a. 内存缓存的处理由NSCache对象实现，NSCache类似一个集合的容器，它存储key–value对，类似于nsdictionary类，我们通常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新能，同时这些对象对于程序来说不是紧要的，如果内存紧张就会自动释放。

b. 磁盘缓存的处理使用NSFileManager对象实现，图片存储的位置位于cache文件夹，另外SDImageCache还定义了一个串行队列来异步存储图片。

c. SDImageCache提供了大量方法来缓存、获取、移除及清空图片。对于图片的索引，我们通过一个key来索引，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值作为图片的文件名，对于一个远程下载的图片其url实作为这个key的最佳选择。

2、存储图片 先在内存中放置一份缓存，如果需要缓存到磁盘，将磁盘缓存操作作为一个task放到串行队列中处理，会先检查图片格式是jpeg还是png，将其转换为响应的图片数据，最后吧数据写入磁盘中（文件名是对key值做MD5后的串）

3、查询图片 内存和磁盘查询图片API：

– (UIImage *)imageFromMemoryCacheForKey:(NSString *)key;

– (UIImage *)imageFromDiskCacheForKey:(NSString *)key;

查看本地是否存在key指定的图片，使用一下API：

– (NSOperation *)queryDiskCacheForKey:(NSString *)key done:(SDWebImageQueryCompletedBlock)doneBlock;

4、移除图片 移除图片API：

– (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:(SDWebImageNoParamsBlock)completion;

5、清理图片（磁盘）

清空磁盘图片可以选择完全清空和部分清空，完全清空就是吧缓存文件夹删除。

– (void)clearDisk;

– (void)clearDiskOnCompletion:(SDWebImageNoParamsBlock)completion;

部分清理 会根据设置的一些参数移除部分文件，主要有两个指标：文件的缓存有效期（maxCacheAge：默认是1周）和最大缓存空间大小（maxCacheSize：如果所有文件大小大于最大值，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哪些过早的文件，知道缓存文件总大小小于最大值），具体代码参考

– (void)cleanDiskWithCompletionBlock；

6、小结 SDImageCache处理提供以上API，还提供了获取缓存大小，缓存中图片数量等API， 常用的接口和属性：

（1）–getSize ： 获得硬盘缓存的大小

（2）–getDiskCount： 获得硬盘缓存的图片数量

（3）–clearMemory ： 清理所有内存图片

（4）– removeImageForKey:(NSString *)key 系列的方法： 从内存、硬盘按要求指定清除图片

（5）maxMemoryCost ： 保存在存储器中像素的总和

（6）maxCacheSize ： 最大缓存大小 以字节为单位。默认没有设置，也就是为0，而清理磁盘缓存的先决条件为self.maxCacheSize > 0，所以0表示无限制。

（7）maxCacheAge： 在内存缓存保留的最长时间以秒为单位计算，默认是一周

三、SDWebImageManager

实际使用中并不直接使用SDWebImageDownloader和SDImageCache类对图片进行下载和存储，而是使用SDWebImageManager来管理。包括平常使用UIImageView+WebCache等控件的分类，都是使用SDWebImageManager来处理，该对象内部定义了一个图片下载器（SDWebImageDownloader）和图片缓存（SDImageCache）

```
@interface SDWebImageManager : NSObject
@property (weak, nonatomic) id <SDWebImageManagerDelegate> delegate;
@property (strong, nonatomic, readonly) SDImageCache *imageCache;
@property (strong, nonatomic, readonly) SDWebImageDownloader *imageDownloader;
...
@end
```

SDWebImageManager声明了一个delegate属性，其实是一个id对象，代理声明了两个方法

```
// 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 *)imageURL;
// 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 *)image withURL:(NSURL *)imageURL;
```

这两个方法会在SDWebImageManager的–downloadImageWithURL:options:progress:completed:方法中调用，而这个方法是SDWebImageManager类的核心所在（具体看源码）

SDWebImageManager的几个API：

（1）– (void)cancelAll ： 取消runningOperations中所有的操作，并全部删除

（2）– (BOOL)isRunning ： 检查是否有操作在运行，这里的操作指的是下载和缓存组成的组合操作

（3） – downloadImageWithURL:options:progress:completed: 核心方法

（4）– (BOOL)diskImageExistsForURL:(NSURL *)url ： 指定url的图片是否进行了磁盘缓存

四、视图扩展

- 在使用SDWebImage的时候，使用最多的是UIImageView+WebCache中的针对UIImageView的扩展，核心方法是sd_setImageWithURL:placeholderImage:options:progress:completed:， 其使用SDWebImageManager单例对象下载并缓存图片。
- 除了扩展UIImageView外，SDWebImage还扩展了UIView，UIButton，MKAnnotationView等视图类，具体可以参考源码，除了可以使用扩展的方法下载图片，同时也可以使用

SDWebImageManager下载图片。

- UIView+WebCacheOperation分类： 把当前view对应的图片操作对象存储起来（通过运行时设置属性），在基类中完成 存储的结构：一个loadOperationKey属性，value是一个字典（字典结构：key：UIImageViewAnimationImages或者UIImageViewImageLoad，value是 operation数组（动态图片）或者对象）
- UIButton+WebCache分类 会根据不同的按钮状态，下载的图片根据不同的状态进行设置 imageURLStorageKey:{state:url}

五、技术点

- 1.dispatch_barrier_sync函数，用于对操作设置顺序，确保在执行完任务后再确保后续操作。常用于确保线程安全性操作
- 2.NSMutableURLRequest：用于创建一个网络请求对象，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OC中一种告诫的并发处理方法，基于GCD实现，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容易一些，对SDWebImage中我们看到如何使用依赖将下载顺序设置成后进先出的顺序
- 4.NSURLSession：用于网络请求及相应处理
- 5.开启后台任务
- 6.NSCache类：一个类似于集合的容器，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.图片解压操作：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。

三次握手与四次挥手

在面试中，三次握手和四次挥手可以说是问的最频繁的一个知识点了，我相信大家也都看过很多关于三次握手与四次挥手的文章，今天的这篇文章，重点是围绕着面试，我们应该掌握哪些比较重要的点，哪些是比较被面试官给问到的，我觉得如果你能把我下面列举的一些点都记住、理解，我想就差不多了。

三次握手

当面试官问你为什么需要有三次握手、三次握手的作用、讲讲三次三次握手的时候，我想很多人会这样回答：

首先很多人会先讲下握手的过程：

1. **第一次握手：** 客户端给服务器发送一个 SYN 报文。
2. **第二次握手：** 服务器收到 SYN 报文之后，会应答一个 SYN+ACK 报文。
3. **第三次握手：** 客户端收到 SYN+ACK 报文之后，会回应一个 ACK 报文。
4. 服务器收到 ACK 报文之后，三次握手建立完成。

作用是为了确认双方的接收与发送能力是否正常。

这里我顺便解释一下为啥只有三次握手才能确认双方的接受与发送能力是否正常，而两次却不可以：

第一次握手： 客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。

第二次握手： 服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。不过此时服务器并不能确认客户端的接收能力是否正常。

第三次握手： 客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力正常，服务器自己的发送、接收能力也正常。

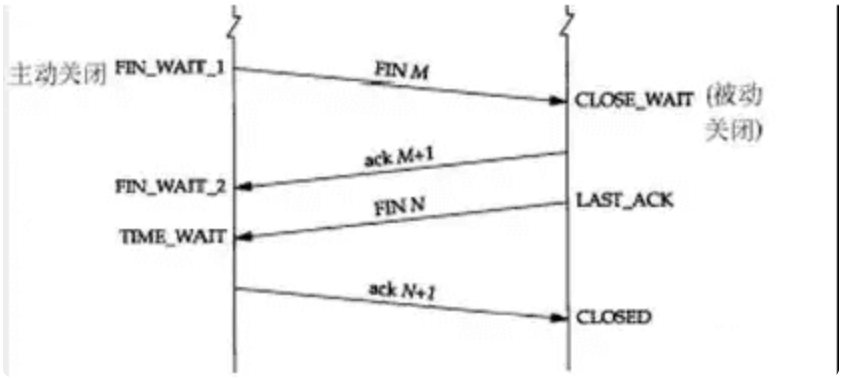
因此，需要三次握手才能确认双方的接收与发送能力是否正常。

这样回答其实也是可以的，但我觉得，这个过程的我们应该要描述的更详细一点，因为三次握手的过程中，双方是由很多状态的改变的，而这些状态，也是面试官可能会问的点。所以我觉得在回答三次握手的时候，我们应该要描述的详细一点，而且描述的详细一点意味着可以扯久一点。

加分的描述我觉得应该是这样：

刚开始客户端处于 closed 的状态，服务端处于 listen 状态。然后

1. **第一次握手：** 客户端给服务端发一个 SYN 报文，并指明客户端的初始化序列号ISN(c)。此时客户端处于 SYN_Send 状态。
2. **第二次握手：** 服务器收到客户端的 SYN 报文之后，会以自己的 SYN 报文作为应答，并且也是指定了自己的初始化序列号 ISN(s)，同时会把客户端的 ISN + 1 作为 ACK 的值，表示自己已经收到了客户端的 SYN，此时服务器处于 SYN_REVD 的状态。
3. **第三次握手：** 客户端收到 SYN 报文之后，会发送一个 ACK 报文，当然，也是一样把服务器的 ISN + 1 作为 ACK 的值，表示已经收到了服务端的 SYN 报文，此时客户端处于 establised 状态。
4. 服务器收到 ACK 报文之后，也处于 establised 状态，此时，双方以建立起了链接。



三次握手的作用

三次握手的作用也是有好多的，多记住几个，保证不亏。例如：

1. 确认双方的接受能力、发送能力是否正常。
2. 指定自己的初始化序列号，为后面的可靠传送做准备。
3. 如果是 https 协议的话，三次握手这个过程，还会进行数字证书的验证以及加密密钥的生成到。

单单这样还不足以应付三次握手，面试官可能还会问一些其他的问题，例如：

1、(ISN)是固定的吗

三次握手的一个重要功能是客户端和服务端交换ISN(Initial Sequence Number), 以便让对方知道接下来接收数据的时候如何按序列号组装数据。

如果ISN是固定的，攻击者很容易猜出后续的确认号，因此 ISN 是动态生成的。

2、什么是半连接队列

服务器第一次收到客户端的 SYN 之后，就会处于 SYN_RCVD 状态，此时双方还没有完全建立其连接，服务器会把此种状态下请求连接放在一个队列里，我们把这种队列称之为半连接队列。当然还有一个全连接队列，就是已经完成三次握手，建立起连接的就会放在全连接队列中。如果队列满了就有可能会出现丢包现象。

这里在补充一点关于SYN-ACK 重传次数的问题：服务器发送完SYN-ACK包，如果未收到客户确认包，服务器进行首次重传，等待一段时间仍未收到客户确认包，进行第二次重传，如果重传次数超 过系统规定的最大重传次数，系统将该连接信息从半连接队列中删除。注意，每次重传等待的时间不一定相同，一般会是指数增长，例如间隔时间为 1s, 2s, 4s, 8s, ….

3、三次握手过程中可以携带数据吗

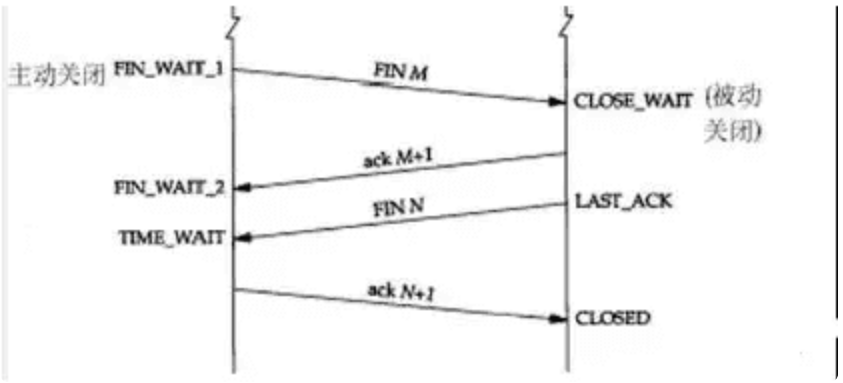
很多人可能会认为三次握手都不能携带数据，其实第三次握手的时候，是可以携带数据的。也就是说，第一次、第二次握手不可以携带数据，而第三次握手是可以携带数据的。

为什么这样呢？大家可以想一个问题，假如第一次握手可以携带数据的话，如果有人要恶意攻击服务器，那他每次都在第一次握手中的 SYN 报文中放入大量的数据，因为攻击者根本就不理服务器的接收、发送能力是否正常，然后疯狂着重复发 SYN 报文的话，这会让服务器花费很多时间、内存空间来接收这些报文。也就是说，第一次握手可以放数据的话，其中一个简单的原因就是会让服务器更加容易受到攻击了。
而对于第三次的话，此时客户端已经处于 established 状态，也就是说，对于客户端来说，他已经建立起连接了，并且也已经知道服务器的接收、发送能力是正常的了，所以能携带数据页没啥毛病。
关于三次握手的，https 的认证过程能知道一下最好，不过我就不说了，留着写 http 面试相关时的文章再说。

四次挥手

四次挥手也一样，千万不要对方一个 FIN 报文，我方一个 ACK 报文，再我方一个 FIN 报文，我方一个 ACK 报文。然后结束，最好是说的详细一点，例如想下面这样就差不多了，要把每个阶段的状态记好，我上次面试就被问了几个了，呵呵。我答错了，还以为自己答对了，当时还解释的头头是道，呵呵。
刚开始双方都处于 establised 状态，假如是客户端先发起关闭请求，则：

1. **第一次挥手：** 客户端发送一个 FIN 报文，报文中会指定一个序列号。此时客户端处于CLOSED_WAIT1状态。
2. **第二次握手：** 服务端收到 FIN 之后，会发送 ACK 报文，且把客户端的序列号值 + 1 作为 ACK 报文的序列号值，表明已经收到客户端的报文了，此时服务端处于CLOSE_WAIT2状态。
3. **第三次挥手：** 如果服务端也想断开连接了，和客户端的第一次挥手一样，发给 FIN 报文，且指定一个序列号。此时服务端处于 LAST_ACK 的状态。
4. **第四次挥手：** 客户端收到 FIN 之后，一样发送一个 ACK 报文作为应答，且把服务端的序列号值 + 1 作为自己 ACK 报文的序列号值，此时客户端处于 TIME_WAIT 状态。需要过一阵子以确保服务端收到自己的 ACK 报文之后才会进入 CLOSED 状态
5. 服务端收到 ACK 报文之后，就处于关闭连接了，处于 CLOSED 状态。



这里特别需要主要的就是TIME_WAIT这个状态了，这个是面试的高频考点，就是要理解，为什么客户端发送 ACK 之后不直接关闭，而是要等一阵子才关闭。这其中的原因就是，要确保服务器是否已经收到了我们的 ACK 报文，如果没有收到的话，服务器会重新发 FIN 报文给客户端，客户端再次收到 FIN 报文之后，就知道之前的 ACK 报文丢失了，然后再次发送 ACK 报文。

至于 TIME_WAIT 持续的时间至少是一个报文的来回时间。一般会设置一个计时，如果过了这个计时没有再次收到 FIN 报文，则代表对方成功就是 ACK 报文，此时处于 CLOSED 状态。

这里我给出每个状态所包含的含义，有兴趣的可以看看。

LISTEN – 侦听来自远方TCP端口的连接请求;

SYN-SENT –在发送连接请求后等待匹配的连接请求;

SYN-RECEIVED – 在收到和发送一个连接请求后等待对连接请求的确认;

ESTABLISHED – 代表一个打开的连接，数据可以传送给用户;

FIN-WAIT-1 – 等待远程TCP的连接中断请求，或先前的连接中断请求的确认;

FIN-WAIT-2 – 从远程TCP等待连接中断请求;

CLOSE-WAIT – 等待从本地用户发来的连接中断请求;

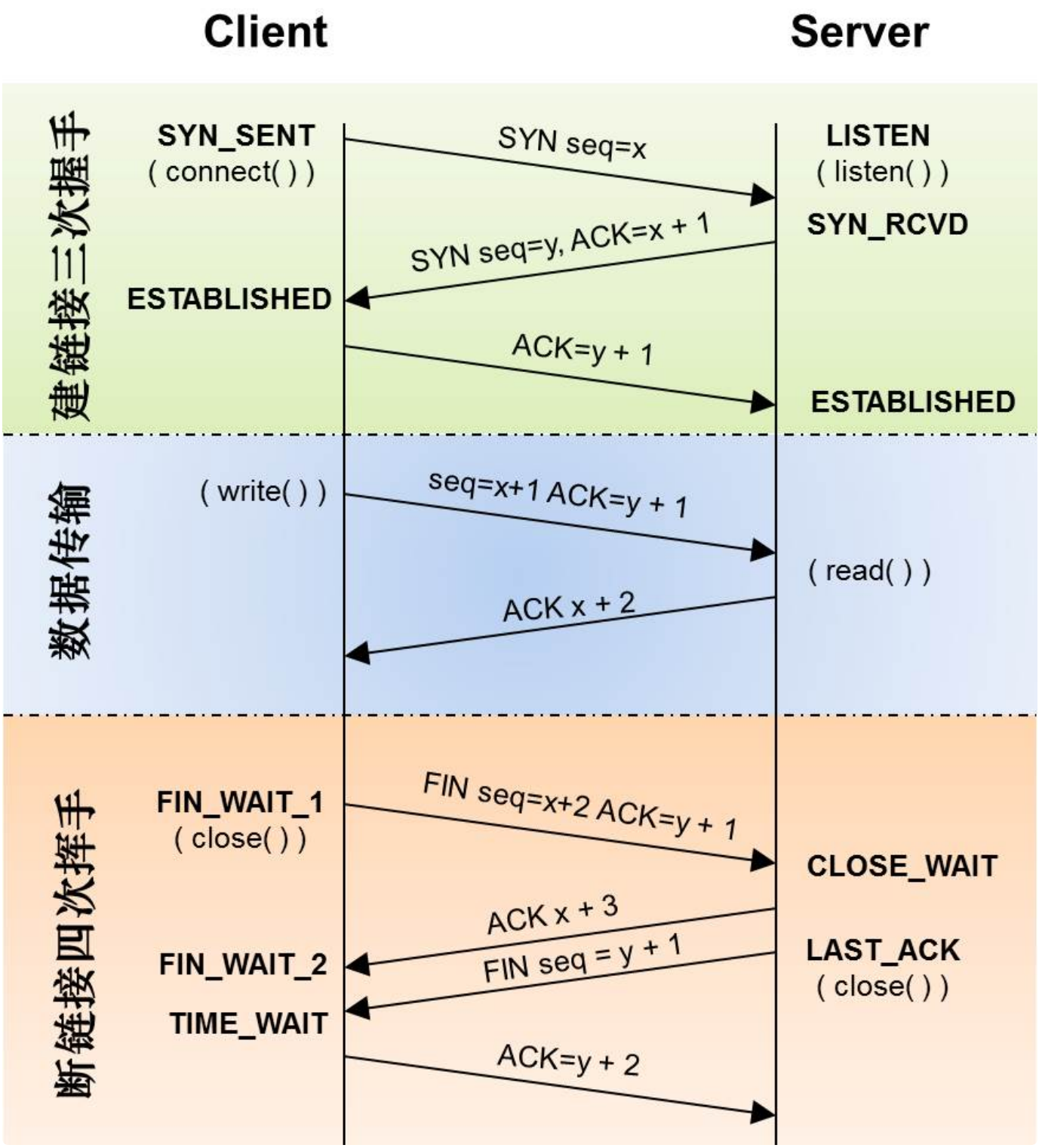
CLOSING –等待远程TCP对连接中断的确认;

LAST-ACK – 等待原来发向远程TCP的连接中断请求的确认;

TIME-WAIT –等待足够的时间以确保远程TCP接收到连接中断请求的确认;

CLOSED – 没有任何连接状态;

再放个三次握手与四次挥手的图：



怎么防止反编译?

1. 本地数据加密。

iOS应用防反编译加密技术之一：对NSUserDefaults，sqlite存储文件数据加密，保护帐号和关键信息

2. URL编码加密。

iOS应用防反编译加密技术之二：对程序中出现的URL进行编码加密，防止URL被静态分析

3. 网络传输数据加密。

iOS应用防反编译加密技术之三：对客户端传输数据提供加密方案，有效防止通过网络接口的拦截获取数据

4. 方法体，方法名高级混淆。

iOS应用防反编译加密技术之四：对应用程序的方法名和方法体进行混淆，保证源码被逆向后无法解析代码

5. 程序结构混排加密。

iOS应用防反编译加密技术之五：对应用程序逻辑结构进行打乱混排，保证源码可读性降到最低

为什么CTMediator方案优于基于Router的方案?

Router的缺点：

- 在组件化的实施过程中，注册URL并不是充分必要条件。组件是不需要向组件管理器注册URL的，注册了URL之后，会造成不必要的内存常驻。注册URL的目的其实是一个服务发现的过程，在iOS领域中，服务发现的方式是不需要通过主动注册的，使用runtime就可以了。另外，注册部分的代码的维护是一个相对麻烦的事情，每一次支持新调用时，都要去维护一次注册列表。如果有调用被弃用了，是经常会忘记删项目的。runtime由于不存在注册过程，那就也不会产生维护的操作，维护成本就降低了。 由于通过runtime做到了服务的自动发现，拓展调用接口的任务就仅在于各自的模块，任何一次新接口添加，新业务添加，都不必去主工程做操作，十分透明。
- 在iOS领域里，一定是组件化的中间件为openURL提供服务，而不是openURL方式为组件化提供服务。如果在给App实施组件化方案的过程中是基于openURL的方案的话，有一个致命缺陷：非常规对象(不能被字符串化到URL中的对象，例如UIImage)无法参与本地组件间调度。
- 在本地调用中使用URL的方式其实是不必要的，如果业务工程师在本地间调度时需要给出URL，那么就不可避免要提供params，在调用时要提供哪些params是业务工程师很容易懵逼的地方。
- 为了支持传递非常规参数，蘑菇街的方案采用了protocol，这个会侵入业务。由于业务中的某个对象需要被调用，因此必须要符合某个可被调用的protocol，然而这个protocol又不存在于当前业务领域，于是当前业务就不得不依赖public Protocol。这对于将来的业务迁移是有非常大的影响的。

CTMediator的优点：

- 调用时，区分了本地应用调用和远程应用调用。本地应用调用为远程应用调用提供服务。
- 组件仅通过Action暴露可调用接口，模块与模块之间的接口被固化在了Target–Action这一层，避免了实施组件化的改造过程中，对Business的侵入，同时也提高了组件化接口的可维护性。

- 方便传递各种类型的参数。

Block一定会造成强引用吗?

不一定，比如UIView的动画Block，UIView调用的是类方法，当前控制器不可能强引用一个类，所以循环无法形成，循环引用的原因是相互指引、相互持有，相互是关键，如果相互这一层关系达不到就没有所谓的循环引用。

断点续传如何实现的?

断点续传的理解可以分为两部分：一部分是断点，一部分是续传。断点的由来是在下载过程中，将一个下载文件分成了多个部分，同时进行多个部分一起的下载，当 某个时间点，任务被暂停了，此时下载暂停的位置就是断点了。续传就是当一个未完成的下载任务再次开始时，会从上次的断点继续传送。使用多线程断点续传下载的时候，将下载或上传任务（一个文件或一个压缩包）人为的划分为几个部分，每一个部分采用一个线程进行上传或下载，多个线程并发可以占用服务器端更多资源，从而加快下载速度。
在下载（或上传）过程中，如果网络故障、电量不足等原因导致下载中断，这就需要使用到断点续传功能。下次启动时，可以从记录位置（已经下载的部分）开始，继续下载以后未下载的部分，避免重复部分的下载。断点续传实质就是能记录上一次已下载完成的位置。
断点续传的过程

- 1.断点续传需要在下载过程中记录每条线程的下载进度；
- 2.每次下载开始之前先读取数据库，查询是否有未完成的记录，有就继续下载，没有则创建新记录插入数据库；
- 3.在每次向文件中写入数据之后，在数据库中更新下载进度；
- 4.下载完成之后删除数据库中下载记录。

JS没有有用过，原理是什么?

iOS 8.0之后使用WKWebView来实现JS交互，首先通过 userContentController 把需要观察的 JS 执行函数注册起来，然后通过一个协议方法，将所有注册过的 JS 函数执行的参数传递到此协议方法中。
注册需要观察的 JS 执行函数：
`[webView.configuration.userContentController addScriptMessageHandler:self name:@"jsFunc"];`
在 JS 中调用这个函数并传递参数数据：

window.webkit.messageHandlers.jsFunc.postMessage({name : "李四",age : 22});
OC 中遵守 WKScriptMessageHandler 协议：
– (void)userContentController:(WKUserContentController *)userContentController didReceiveScriptMessage:(WKScriptMessage *)message
这些具体的实现都基于JavaScriptCore框架，JavaScriptCore是一个苹果在iOS7引入的框架，该框架让 Objective-C 和 JavaScript 代码直接的交互变得更加的简单方便。

简述组件化实现过程

常用的组件化方案也不少，如URL跳转、Target-Action、服务注册、通知广播等，组件的存在方式是以每个pod库的形式存在的。那么我们组合组件的方法就是通过CocoaPods来添加安装各个组件，所以我们就需要制作CocoaPods远程私有库，然后将其发到公司的gitlab或GitHub，使工程能够Pod下载下来。

1. 创建主项目仓库，以及私有Pod源仓库；
2. 新建组件项目和远端私有仓库，并同步；
3. 新建组件X与外界联系媒介X_Category项目及远端仓库，并同步；
4. 主项目Podfile本地引用组件X，并使项目编译通过。

分类的底层是怎么实现的?

分类的底层结构：

```
struct category_t {
    const char *name;
    classref_t cls;
    struct method_list_t *instanceMethods; // 对象方法
    struct method_list_t *classMethods; // 类方法
    struct protocol_list_t *protocols; // 协议
    struct property_list_t *instanceProperties; // 属性
    // Fields below this point are not always present on disk.
    struct property_list_t *_classProperties;

    method_list_t *methodsForMeta(bool isMeta) {
        if (isMeta) return classMethods;
        else return instanceMethods;
    }

    property_list_t *propertiesForMeta(bool isMeta, struct header_info *hi);
};
```

从源码可以看出我们平时使用categroy的时候，对象方法，类方法，协议，和属性都可以找到对应的存储方式。并且我们发现分类结构体中是不存在成员变量的，因此分类中是不允许添加成员变量的。分类中添加的属性并不会帮助我们自动生成成员变量，只会生成get、set方法的声明，需要我们自己去实现。

在runtime中进行查找有没有分类，多个分类保存在category_list中，获取到分类列表之后，进行遍历，获取其中的方法、协议、属性等，分类的方法、属性、协议列表被放在了类对象中原本存储的方法、属性、协议列表前面。来保证分类方法优先调用，我们一般主观认为当分类重写本类的方法时，会覆盖本类的方法。其实经过底层的分析我们知道本质上并不是覆盖，而是优先调用。本类的方法依然在内存中。

总结：分类的实现原理是将category中的方法、属性、协议数据放在category_t结构体中，然后将结构体内的方法列表拷贝到类对象的方法列表中。

如何监控线上版本APP启动耗时（包含动态库的加载时间）？

最常用的方式就是通过 hook 关键函数的调用，然后计算来获得启动时长数据，iOS13.0 以后，在隐私–分析与改进–分析数据中有以log–power–xxx.session命名的日志文件，日志文件中提供了应用运行的一些基本数据信息，系统日志的基本格式如下：

其中，app_bundleid 表示启动应用的 bundleid，app_performance下的 launch 信息中就是关于启动时间的数据，count 表示当天启动该应用的次数，sessions 分别提供了每次启动的耗时（从点击图标到首屏渲染时的耗时）。基于上面的信息，我们可以获取到该 app_bundleid 对应的启动耗时数据，完整分析该日志文件，则可以获取到当日所有启动过的 APP 耗时数据。

创建一个自定义动态库（或直接使用已有的自定义动态库），在 +load 方法中进行埋点作为 APP 的启动时间，为了尽可能将其他动态库中的耗时统计到，我们可以将自定义的动态库放在所有动态库加载的第一位，只需要将其按照–framework“xxx”的格式写到第一位即可，

如何优化 App 的启动耗时?

iOS 的 App 启动时长大概可以这样计算：

t(App 总启动时间) = t1(main 调用之前的加载时间) + t2(main 调用之后的加载时间)。

t1 = 系统 dylib(动态链接库)和自身 App 可执行文件的加载。

t2 = main 方法执行之后到 AppDelegate 类中的 application:didFinishLaunchingWithOptions:方法执行结束前这段时间，主要是构建第一个界面，并完成渲染展示。

在 t1 阶段加快 App 启动的建议：

- 尽量使用静态库，减少动态库的使用，动态链接比较耗时。

- 如果要用动态库，尽量将多个 dylib 动态库合并成一个。
- 尽量避免对系统库使用 optional linking，如果 App 用到的系统库在你所有支持的系统版本上都有，就设置为 required，因为 optional 会有些额外的检查。
- 减少 Objective-C Class、Selector、Category 的数量。可以合并或者删减一些 OC 类。
- 删减一些无用的静态变量，删减没有被调用到或者已经废弃的方法。
- 将不必须在 +load 中做的事情尽量挪到+initialize中，+initialize 是在第一次初始化这个类之前被调用，+load 在加载类的时候就被调用。尽量将+load里的代码延后调用。
- 尽量不要用 C++ 虚函数，创建虚函数表有开销。
- 不要使用 __atribute__((constructor))将方法显式标记为初始化器，而是让初始化方法调用时才执行。比如使用 dispatch_once()，pthread_once()或 std::once()。
- 在初始化方法中不调用 dlopen()，dlopen()有性能和死锁的可能性。
- 在初始化方法中不创建线程。

在 t2 阶段加快 App 启动的建议：

- 尽量不要使用 xib/storyboard，而是用纯代码作为首页 UI。
- 如果要用 xib/storyboard，不要在 xib/storyboard 中存放太多的视图。
- 对application:didFinishLaunchingWithOptions:里的任务尽量延迟加载或懒加载。
- 不要在 NSUserDefaults 中存放太多的数据，NSUserDefaults 是一个 plist 文件，plist 文件被反序列化一次。
- 避免在启动时打印过多的 log。
- 少用 NSLog，因为每一次 NSLog 的调用都会创建一个新的 NSCalendar 实例。
- 每一段 SQLite 语句都是一个段被编译的程序，调用 sqlite3_prepare 将编译 SQLite 查询到字节码，使用 sqlite_bind_int 绑定参数到 SQLite 语句。
- 为了防止使用 GCD 创建过多的线程，解决方法是创建串行队列, 或者使用带有最大并发数限制的 NSOperationQueue。
- 线程安全：UIKit只能在主线程执行，除了 UIGraphics、UIBezierPath 之外，UIImage、CG、CA、Foundation 都不能从两个线程同时访问。
- 不要在主线程执行磁盘、网络、Lock 或者 dispatch_sync、发送消息给其他线程等操作。

动态库和静态库的区别？为什么动态库会影响启动速度？

静态库：链接时完整地拷贝至可执行文件中，被多次使用就有多份冗余拷贝。

动态库：链接时不复制，程序运行时由系统动态加载到内存，供程序调用，系统只加载一次，多个程序共用，节省内存。

影响启动速度是因为动态库在编译时并不会被拷贝到目标程序中，目标程序中只会存储指向动态库的引用。等到程序运行时，动态库才会被真正加载进来，所以会在程序启动时耗时。
减少启动过程中的动态库加载主要有以下两个方案：

1. 动态库转静态库；
2. 多个动态库进行合并。

使用drawRect有什么影响?

有touch event的时候之后，会重新调用drawRect进行强制绘制，如果有很多这样的按钮的话就会比较影响效率和性能。以下情况都会被调用：
1、如果在UIView初始化时没有设置rect大小，将直接导致drawRect不被自动调用。drawRect 掉用是在Controller->loadView, Controller->viewDidLoad 两方法之后掉用的.所以不用担心在 控制器中,这些View的drawRect就开始画了.这样可以在控制器中设置一些值给View(如果这些View draw的时候需要用到某些变量 值).
2、该方法在调用sizeToFit后被调用，所以可以先调用sizeToFit计算出size。然后系统自动调用drawRect:方法。
3、通过设置contentMode属性值为UIViewContentModeRedraw。那么将在每次设置或更改frame的时候自动调用drawRect:。
4、直接调用setNeedsDisplay，或者setNeedsDisplayInRect:触发drawRect:，但是有个前提条件是rect不能为0。

单例的原理和弊端?

原理：
dispatch_once主要是根据onceToken的值来决定怎么去执行代码：
当onceToken= 0时，线程执行dispatch_once的block中代码；
当onceToken= -1时，线程跳过dispatch_once的block中代码不执行；
当onceToken为其他值时，线程被阻塞，等待onceToken值改变；
某一线程要执行block中的代码时，首先需要改变onceToken的值，再去执行block中的代码。这里onceToken的值可能会变为一个非常大的数。
这样当其他线程再获取onceToken的值时，其他线程就会被阻塞。
当block线程执行完block之后。onceToken变为-1。其他线程不再阻塞，并且跳过block。
下次再调用时，block已经为-1。直接跳过block。
这样在首次调用时同步阻塞线程，生成单例之后，不再阻塞线程。

优点：

1：一个类只被实例化一次，提供了对唯一实例的受控访问。

2：节省系统资源

3：允许可变数目的实例。

缺点：

1：一个类只有一个对象，可能造成责任过重，在一定程度上违背了“单一职责原则”。

2：由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。

3：滥用单例将带来一些负面问题，如为了节省资源将数据库连接池对象设计为的单例类，可能会导致共享连接池对象的程序过多而出现连接池溢出；如果实例化的对象长时间不被利用，系统会认为是垃圾而被回收，这将导致对象状态的丢失。

自动释放池的原理?

1. 每一次运行循环开启时，会创建自动释放池；
2. 程序执行过程中的自动释放对象，出了作用域之后，会被添加到最近的自动释放池；
3. 运行循环结束前，会释放自动释放池。

自动释放池的主要底层数据结构是：__AtAutoreleasePool 和 AutoreleasePoolPage，调用了autorelease的对象最终由AutoreleasePoolPage对象来管理，所有的AutoreleasePoolPage对象通过双向链表的形式连接在一起。

objc4源码：NSObject.mm

```
#define I386_PGBYTES            4096
#define PAGE_SIZE               I386_PGBYTES
#define PAGE_MAX_SIZE           PAGE_SIZE
class AutoreleasePoolPage
{
    magic_t const magic;
    id *next; // 下一个存储autoreleaes对象的地址
    pthread_t const thread;
    AutoreleasePoolPage * const parent; // 父节点
    AutoreleasePoolPage *child; // 子节点
    uint32_t const depth;
    uint32_t hiwat;
    static size_t const SIZE = PAGE_MAX_SIZE;
};
```

页的数据结构是AutoreleasePoolPageData。每一个autoreleasepool对象只有一个哨兵，哨兵放在第一页中。每一个AutoreleasePoolPage对象占用4094字节内存，本身成员占用56字节，每一页的前56个字节存储页的AutoreleasePoolPageData结构体数据。第一页的第56往后8个字节存储哨兵，后面存储autorelease对象，总共可以存储504个。从第二页开始，每页可以存储505个对象。

objc_autoreleasepoolpush是一个查找child，递增next，创建新页的过程。objc_autoreleasepoolpop是一个查找parent，递减next，释放对象，销毁page的过程，遇到哨兵对象即停止。

UITableView重用机制原理?

查看UITableView头文件，会找到NSMutableArray *visiableCells，和NSMutableDictnery *reusableTableCells两个结构。visiableCells内保存当前显示的cells，reusableTableCells保存可重用的cells。

TableView显示之初，reusableTableCells为空，那么tableView dequeueReusableCellWithIdentifier:CellIdentifier返回nil。开始的cell都是通过 [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]来创建，而且cellForRowAtIndexPath只是调用最大显示cell数的次数。

比如：有100条数据，iPhone一屏最多显示10个cell。程序最开始显示TableView的情况是：

1. 用[[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:CellIdentifier]创建10次cell，并给cell指定同样的重用标识(当然，可以为不同显示类型的 cell指定不同的标识)。并且10个cell全部都加入到visiableCells数组，reusableTableCells为空。
2. 向下拖动tableView，当cell1完全移出屏幕，并且cell11(它也是alloc出来的，原因同上)完全显示出来的时候。cell11加入到 visiableCells，cell1移出visiableCells，cell1加入到reusableTableCells。
3. 接着向下拖动tableView，因为reusableTableCells中已经有值，所以，当需要显示新的 cell，cellForRowAtIndexPath再次被调用的时候，tableView dequeueReusableCellWithIdentifier:CellIdentifier，返回cell1。cell1加入到 visiableCells，cell1移出reusableTableCells；cell2移出visiableCells，cell2加入到 reusableTableCells。之后再需要显示的Cell就可以正常重用了。

所以整个过程并不难理解，但需要注意正是因为这样的原因：配置Cell的时候一定要注意，对取出的重用的cell做重新赋值，不要遗留老数据。