阿里-p6-一面
1.介绍下内存的几大区域?
2.你是如何组件化解耦的?
3.runtime如何通过selector找到对应的IMP地址
4.runloop内部实现逻辑?
5.你理解的多线程?
6.GCD执行原理?
7.怎么防止别人反编译你的app?
8.YYAsyncLayer如何异步绘制?
9.优化你是从哪几方面着手?

1.介绍下内存的几大区域?

1.栈区(stack) 由编译器自动分配并释放，存放函数的参数值，局部变量等。栈是系统数据结构，对应线程/进程是唯一的。优点是快速高效，缺点时有限制，数据不灵活。［先进后出］

栈空间分静态分配 和动态分配两种。

```
静态分配是编译器完成的，比如自动变量(auto)的分配。
动态分配由alloca函数完成。
栈的动态分配无需释放(是自动的)，也就没有释放函数。
为可移植的程序起见，栈的动态分配操作是不被鼓励的!
```


堆区(heap) 由程序员分配和释放，如果程序员不释放，程序结束时，可能会由操作系统回收 ，比如在ios 中 alloc 都是存放在堆中。
优点是灵活方便，数据适应面广泛，但是效率有一定降低。

堆是函数库内部数据结构，不一定唯一。
不同堆分配的内存无法互相操作。
堆空间的分配总是动态的

虽然程序结束时所有的数据空间都会被释放回系统，但是精确的申请内存，释放内存匹配是良好程序的基本要素。

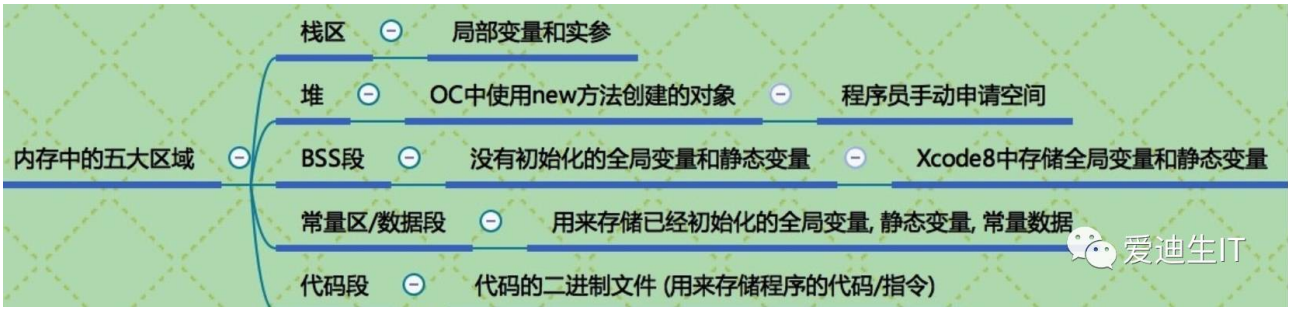
3.全局区(静态区) (static) 全局变量和静态变量的存储是放在一起的，初始化的全局变量和静态变量存放在一块区域，未初始化的全局变量和静态变量在相邻的另一块区域，程序结束后有系统释放。

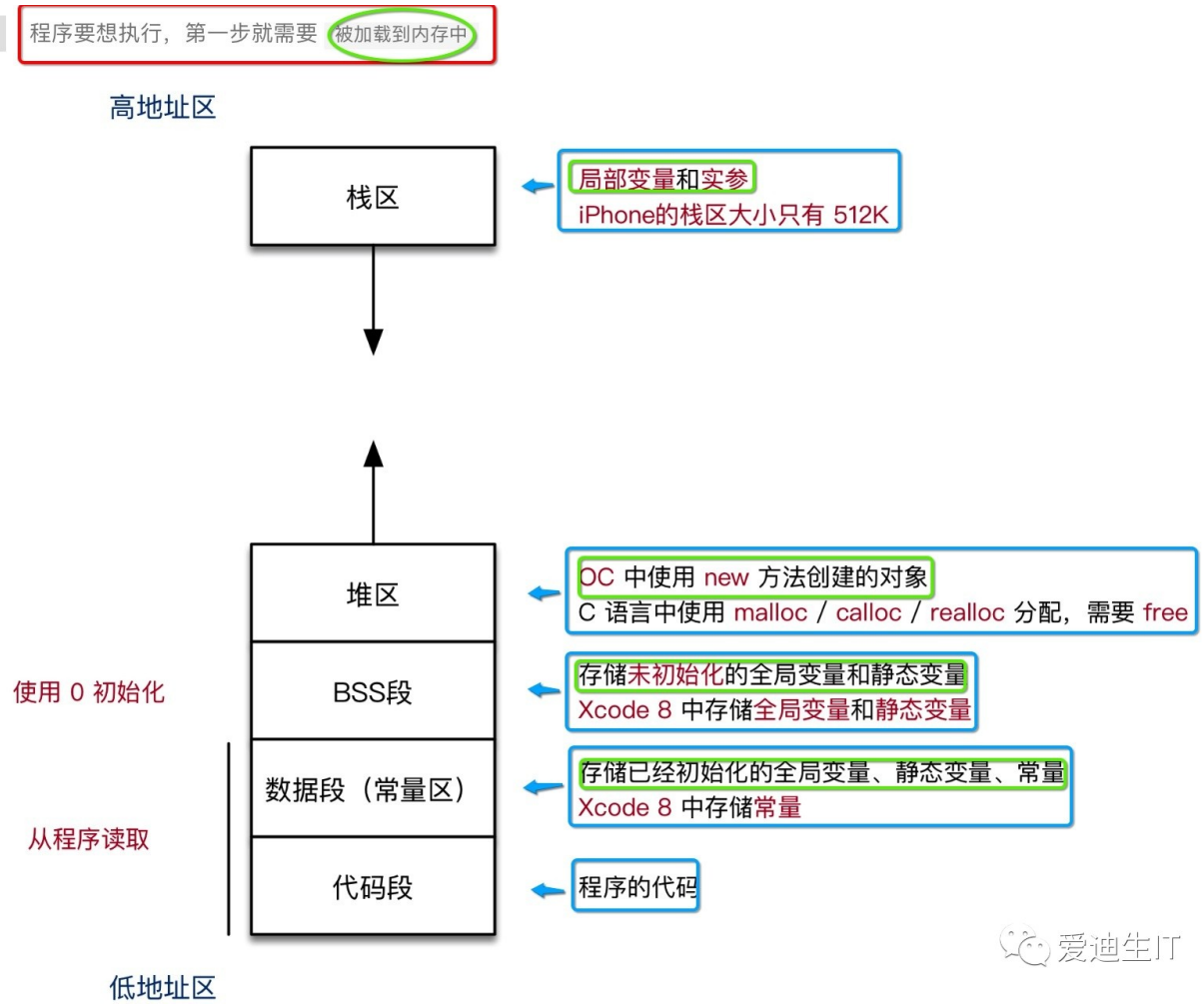
注意：全局区又可分为未初始化全局区：
.bss段和初始化全局区：data段。
举例：int a;未初始化的。int a = 10;已初始化的。

4.文字常量区 存放常量字符串，程序结束后由系统释放；

5.代码区 存放函数的二进制代码

大致如图：




程序要想执行，第一步就需要 被加载到内存中
高地址区
栈区
局部变量和实参
iPhone的栈区大小只有 512K
堆区
OC 中使用 new 方法创建的对象
C 语言中使用 malloc / calloc / realloc 分配，需要 free
使用 0 初始化
BSS段
存储未初始化的全局变量和静态变量
Xcode 8 中存储全局变量和静态变量
数据段（常量区）
存储已经初始化的全局变量、静态变量、常量
Xcode 8 中存储常量
从程序读取
代码段
程序的代码
爱迪生IT
低地址区

例子代码：

```
 int a = 10;  全局初始化区
 char *p;  全局未初始化区

main{
  int b; 栈区
  char s[] = "abc" 栈
  char *p1; 栈
  char *p2 = "123456";  123456\\\\0在常量区，p2在栈上。
  static int c =0;  全局（静态）初始化区

  w1 = (char *)malloc(10);
  w2 = (char *)malloc(20);
  分配得来得10和20字节的区域就在堆区。
}
```


可能被追问的问题一：

1.栈区 (stack [stæk]): 由编译器自动分配释放
局部变量是保存在栈区的
方法调用的实参也是保存在栈区的

2.堆区 (heap [hiːp]): 由程序员分配释放，若程序员不释放，会出现内存泄漏，赋值语句右侧使用 new 方法创建的对象，被创建对象的所有 成员变量！

3.BSS 段 : 程序结束后由系统释放

4.数据段 : 程序结束后由系统释放

5.代码段:程序结束后由系统释放
程序编译链接 后的二进制可执行代码

可能被追问的问题二：

比如申请后的系统是如何响应的?

1. 栈：存储每一个函数在执行的时候都会向操作系统索要资源，栈区就是函数运行时的内存，栈区中的变量由编译器负责分配和释放，内存随着函数的运行分配，随着函数的结束而释放，由系统自动完成。
   注意：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

2. 堆：
   1.首先应该知道操作系统有一个记录空闲内存地址的链表。
   2.当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。
   3 .由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中

可能被追问的问题三：

比如：申请大小的限制是怎样的?

1. 栈：栈是向低地址扩展的数据结构，是一块连续的内存的区域。是栈顶的地址和栈的最大容量是系统预先规定好的，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数 ），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

2. 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

栈：由系统自动分配，速度较快，不会产生内存碎片
堆：是由alloc分配的内存，速度比较慢，而且容易产生内存碎片，不过用起来最方便

打个比喻来说：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

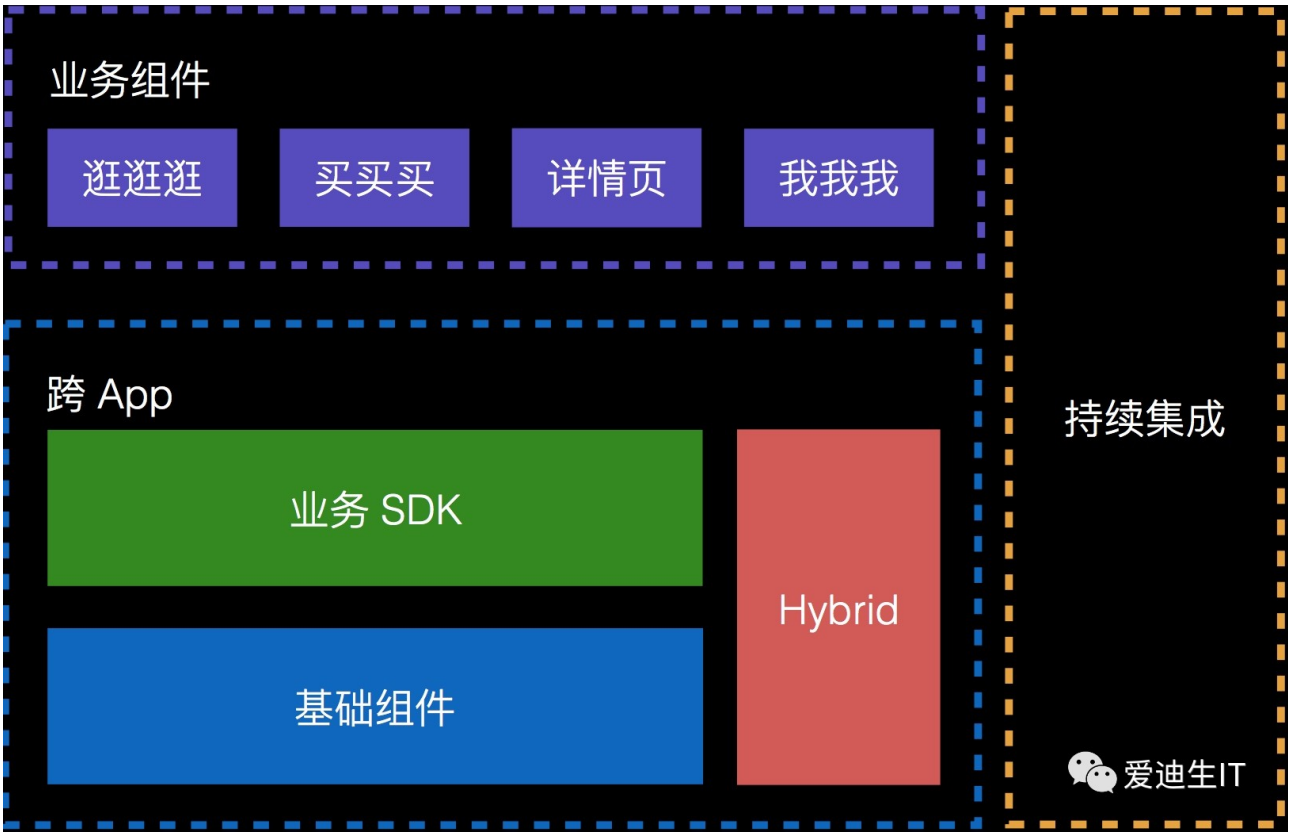
2.你是如何组件化解耦的?

实现代码的高内聚低耦合，方便多人多团队开发！

一般需要解耦的项目都会多多少少出现，一下几个情况：

1. 耦合比较严重（因为没有明确的约束，「组件」间引用的现象会比较多）
   2.容易出现冲突（尤其是使用 Xib，还有就是 Xcode Project，虽说有脚本可以改善）
   3.业务方的开发效率不够高（只关心自己的组件，却要编译整个项目，与其他不相干的代码糅合在一起）

先来看下，组件化之后的一个大概架构



「组件化」顾名思义就是把一个大的 App 拆成一个个小的组件，相互之间不直接引用。那如何做呢？

实现方式

组件间通信

以 iOS 为例，由于之前就是采用的 URL 跳转模式，理论上页面之间的跳转只需 open 一个 URL 即可。所以对于一个组件来说，只要定义「支持哪些 URL」即可，比如详情页，大概可以这么做的

```
[MGJRouter registerURLPattern:@"mgj://detail?id=:id" toHandler:^(NSDictionary *ro
uterParameters) {
    NSNumber *id = routerParameters[@"id"];
    // create view controller with id
    // push view controller
}];
```


首页只需调用 `[MGJRouter openURL:@"mgj://detail?id=404"]` 就可以打开相应的详情页。

那问题又来了，我怎么知道有哪些可用的 URL？为此，我们做了一个后台专门来管理。

ID	变量名	短链	描述	开发人员	需求方	创建时间	操作
4	MGJPAGE_ABOUT	mgj://about	关于蘑菇街	慧能	慧能	2015-12-17 19:28	约束 废弃
5	MGJPAGE_ADDRESS	mgj://address	管理收货地址	慧能	慧能	2015-12-17 19:28	约束 废弃
6	MGJPAGE_APPENDRATE	mgj://appendrate	追加评价	慧能	慧能	2015-12-17 19:28	约束 废弃
7	MGJPAGE_BEAUTY	mgj://beauty	美妆	慧能	慧能	2015-12-17 19:28	约束 废弃

然后可以把这些短链生成不同平台所需的文件，iOS 平台生成 .{h,m} 文件，Android 平台生成 .java 文件，并注入到项目中。这样开发人员只需在项目中打开该文件就知道所有的可用 URL 了。

目前还有一块没有做，就是参数这块，虽然描述了短链，但真想要生成完整的 URL，还需要知道如何传参数，这个正在开发中。

还有一种情况会稍微麻烦点，就是「组件A」要调用「组件B」的某个方法，比如在商品详情页要展示购物车的商品数量，就涉及到向购物车组件拿数据。

类似这种同步调用，iOS 之前采用了比较简单的方案，还是依托于 `MGJRouter`，不过添加了新的方法 `- (id)objectForURL:`，注册时也使用新的方法进行注册

```
[MGJRouter registerURLPattern:@"mgj://cart/ordercount" toObjectHandler:^id(NSDict
ionary *routerParamters){
    // do some calculation
    return @42;
}]
```


使用时 `NSNumber *orderCount = [MGJRouter objectForURL:@"mgj://cart/ordercount"]` 这样就拿到了购物车里的商品数。

稍微复杂但更具通用性的方法是使用「协议」 <-> 「类」绑定的方式，还是以购物车为例，购物车组件可以提供这么个 Protocol

```
@protocol MGJCart <NSObject>
+ (NSInteger)orderCount;
@end
```


可以看到通过协议可以直接指定返回的数据类型。然后在购物车组件内再新建个类实现这个协议，假设这个类名为`MGJCartImpl`，接着就可以把它与协议关联起来 `[ModuleManagerregisterClass:MGJCartImpl forProtocol:@protocol(MGJCart)]`，对于使用方来说，要拿到这个 `MGJCartImpl`，需要调用 `[ModuleManagerclassForProtocol:@protocol(MGJCart)]`。拿到之后再调用 `+(NSInteger)orderCount`就可以了。

那么，这个协议放在哪里比较合适呢？如果跟组件放在一起，使用时还是要先引入组件，如果有多个这样的组件就会比较麻烦了。所以我们把这些公共的协议统一放到了 `PublicProtocolDomain.h` 下，到时只依赖这一个文件就可以了。

Android 也是采用类似的方式。

组件生命周期管理

理想中的组件可以很方便地集成到主客中，并且有跟 AppDelegate 一致的回调方法。这也是 ModuleManager 做的事情。

先来看看现在的入口方法

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [MGJApp startApp];

    [[ModuleManager sharedInstance] loadModuleFromPlist:[[NSBundle mainBundle] pathForResource:@"modules" ofType:@"plist"]];
    NSArray *modules = [[ModuleManager sharedInstance] allModules];
    for (id<ModuleProtocol> module in modules) {
        if ([module respondsToSelector:_cmd]) {
            [module application:application didFinishLaunchingWithOptions:launchOptions];
        }
    }

    [self trackLaunchTime];
    return YES;
}
```


其中 [MGJApp startApp] 主要负责一些 SDK 的初始化。[self trackLaunchTime] 是我们打的一个点，用来监测从 main 方法开始到入口方法调用结束花了多长时间。其他的都由 ModuleManager 搞定，loadModuleFromPlist:pathForResource: 方法会读取 bundle 里的一个 plist 文件，这个文件的内容大概是这样的

Root	Array	(40 items)
Item 0	String	ComponentManager
Item 1	String	MiscModule
Item 2	String	UISkeletonModule
Item 3	String	WatchModule
Item 4	String	URLHandlerModule
Item 5	String	MGJIndexModule

每个 Module 都实现了 ModuleProtocol，其中有一个 -(BOOL)applicaiton:didFinishLaunchingWithOptions: 方法，如果实现了的话，就会被调用。

还有一个问题就是，系统的一些事件会有通知，比如 applicationDidBecomeActive 会有对应的 UIApplicationDidBecomeActiveNotification，组件如果要做响应的话，只需监听这个系统通知即可。但也有一些事件是没有通知的，比如 -application:didRegisterUserNotificationSettings:，这时组件如果也要做点事情，怎么办?

一个简单的解决方法是在 AppDelegate 的各个方法里，手动调一遍组件的对应的方法，如果有就执行。

```
- (void)application:(UIApplication *)application didRegisterForRemoteNotification
sWithDeviceToken:(NSData *)deviceToken
{
    NSArray *modules = [[ModuleManager sharedInstance] allModules];
    for (id<ModuleProtocol> module in modules) {
        if ([module respondsToSelector:_cmd]) {
            [module application:application didRegisterForRemoteNotificationsWith
DeviceToken:deviceToken];
        }
    }
}
```


壳工程

既然已经拆出去了，那拆出去的组件总得有个载体，这个载体就是壳工程，壳工程主要包含一些基础组件和业务SDK，这也是主工程包含的一些内容，所以如果在壳工程可以正常运行的话，到了主工程也没什么问题。不过这里存在版本同步问题，之后会说到。

遇到的问题

组件拆分

由于之前的代码都是在一个工程下的，所以要单独拿出来作为一个组件就会遇到不少问题。首先是组件的划分，当时在定义组件粒度时也花了些时间讨论，究竟是粒度粗点好，还是细

点好。粗点的话比较有利于拆分，细点的话灵活度比较高。最终还是选择粗一点的粒度，先拆出来再说。

假如要把详情页迁出来，就会发现它依赖了一些其他部分的代码，那最快的方式就是直接把代码拷过来，改个名使用。比较简单暴力。说起来比较简单，做的时候也是挺有挑战的，因为正常的业务并不会因为「组件化」而停止，所以开发同学们需要同时兼顾正常的业务和组件的拆分。

版本管理

我们的组件包括第三方库都是通过 Cocoapods 来管理的，其中组件使用了私有库。之所以选择 Cocoapods，一个是因为它比较方便，还有就是用户基数比较大，且社区也比较活跃（活跃到了会时不时地触发 Github 的 rate limit，导致长时间 clone 不下来… 见此），当然也有其他的管理方式，比如 submodule / subtree，在开发人员比较多的情况下，方便、灵活的方案容易占上风，虽然它也有自己的问题。主要有版本同步和更新/编译慢的问题。

假如基础组件做了个 API 接口升级，这个升级会对原有的接口做改动，自然就会升一个中位的版本号，比如原先是 1.6.19，那么现在就变成 1.7.0 了。而我们在 Podfile 里都是用 ~ 指定的，这样就会出现主工程的 pod 版本升上去了，但是壳工程没有同步到，然后群里就会各种反馈编译不过，而且这个编译不过的长尾有时能拖上两三天。

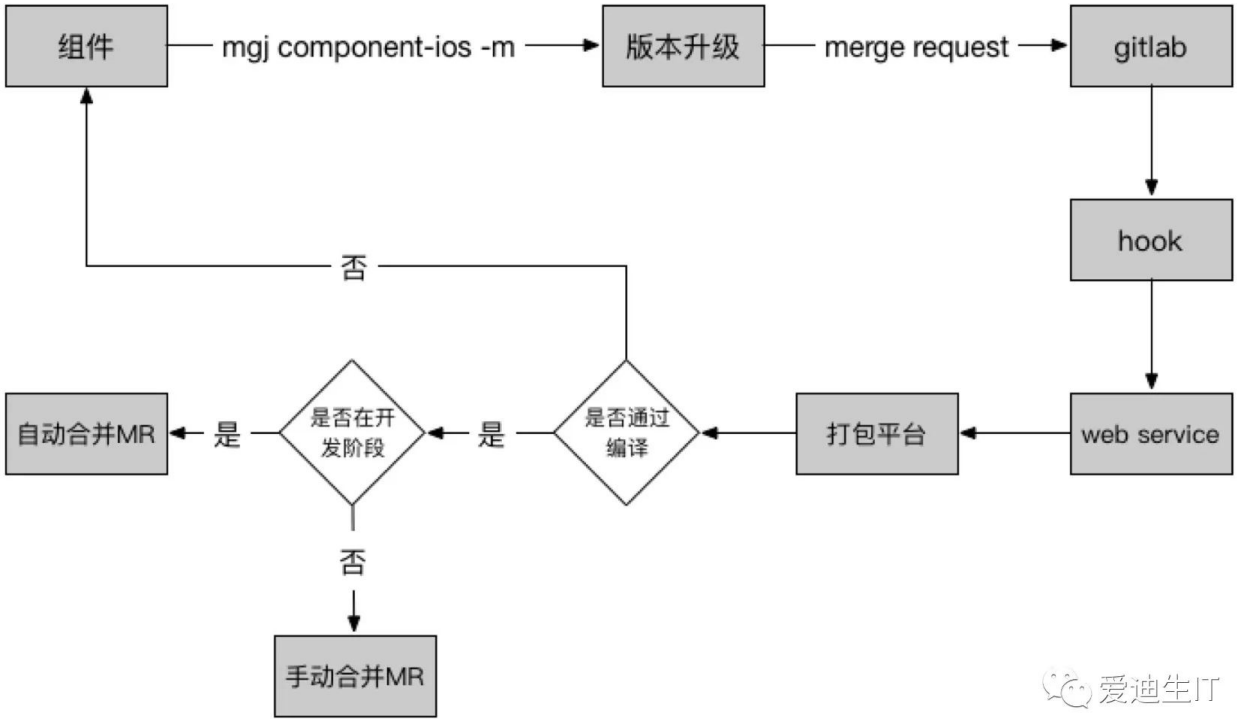
然后我们就想了个办法，如果不在壳工程里指定基础库的版本，只在主工程里指定呢，理论上应该可行，只要不出现某个基础库要同时维护多个版本的情况。但实践中发现，壳工程有时会莫名其妙地升不上去，在 podfile 里指定最新的版本又可以升上去，所以此路不通。

还有一个问题是 `pod update` 时间过长，经常会在 `Analyzing Dependency` 上卡 10 多分钟，非常影响效率。后来排查下来是跟组件的 Podspec 有关，配置了 subspec，且依赖比较多。

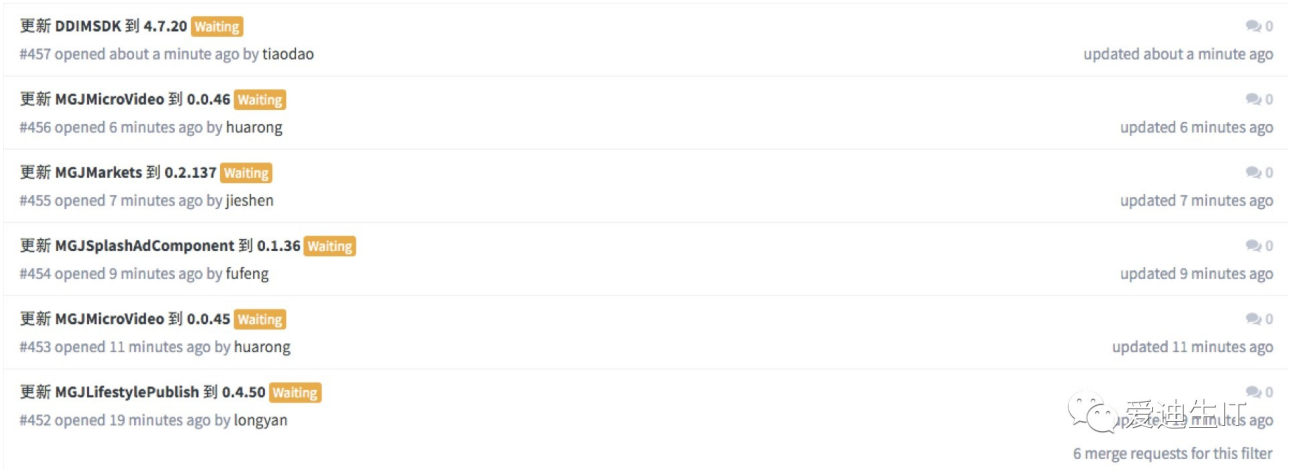
然后就是 pod update 之后的编译，由于是源码编译，所以这块的时间花费也不少，接下去会考虑 framework 的方式。

持续集成

在刚开始，持续集成还不是很完善，业务方升级组件，直接把 podspec 扔到 private repo 里就完事了。这样最简单，但也经常会带来编译通不过的问题。而且这种随意的版本升级也不太能保证质量。于是我们就搭建了一套持续集成系统，大概如此



每个组件升级之前都需要先通过编译，然后再决定是否升级。这套体系看起来不复杂，但在实施过程中经常会遇到后端的并发问题，导致业务方要么集成失败，要么要等不少时间。而且也没有一个地方可以呈现当前版本的组件版本信息。还有就是业务方对于这种命令行的升级方式接受度也不是很高。



基于此，在经过了几轮讨论之后，有了新版的持续集成平台，升级操作通过网页端来完成。

大致思路是，业务方如果要升级组件，假设现在的版本是 0.1.7，添加了一些 feature 之后，壳工程测试通过，想集成到主工程里看看效果，或者其他组件也想引用这个最新的，就可以在后台手动把版本升到 0.1.8-rc.1，这样的话，原先依赖 `~> 0.1.7` 的组件，不会升到 0.1.8，同时想要测试这个组件的话，只要手动把版本调到 0.1.8-rc.1 就可以了。这个过程不会触发 CI 的编译检查。

当测试通过后，就可以把尾部的 `-rc.n` 去掉，然后点击「集成」，就会走 CI 编译检查，通过的话，会在主工程的 podfile 里写上固定的版本号 0.1.8。也就是说，podfile 里所有的组件版本号都是固定的。

周边设施

基础组件及组件的文档 / Demo / 单元测试

无线基础的职能是为集团提供解决方案，只是在蘑菇街 App 里能 work 是远远不够的，所以就需要提供入口，知道有哪些可用组件，并且如何使用，就像这样（目前还未实现）

这就要求组件的负责人需要及时地更新 README / CHANGELOG / API，并且当发生 API 变更时，能够快速通知到使用方。

公共 UI 组件

组件化之后还有一个问题就是资源的重复性，以前在一个工程里的时候，资源都可以很方便地拿到，现在独立出去了，也不知道哪些是公用的，哪些是独有的，索性都放到自己的组件里，这样就会导致包变大。还有一个问题是每个组件可能是不同的产品经理在跟，而他们很可能只关注于自己关心的页面长什么样，而忽略了整体的样式。公共 UI 组件就是用来解决这些问题的，这些组件甚至可以跨 App 使用。（目前还未实现）

参考答案一：http://blog.csdn.net/GGGHub/article/details/52713642

参考答案二：http://limboy.me/tech/2016/03/10/mgj-components.html

3.runtime如何通过selector找到对应的IMP地址?

概述

类对象中有类方法和实例方法的列表，列表中记录着方法的名词、参数和实现，而selector本质就是方法名称，runtime通过这个方法名称就可以在列表中找到该方法对应的实现。
这里声明了一个指向struct objc_method_list指针的指针，可以包含类方法列表和实例方法列表

具体实现

在寻找IMP的地址时，runtime提供了两种方法

```
IMP class_getMethodImplementation(Class cls, SEL name);IMP method_getImplementation(Method m)
```

而根据官方描述，第一种方法可能会更快一些

@note \c class_getMethodImplementation may be faster than \c method_getImplementation(class_getInstanceMethod(cls, name)).

对于第一种方法而言，类方法和实例方法实际上都是通过调用class_getMethodImplementation()来寻找IMP地址的，不同之处在于传入的第一个参数不同

类方法（假设有一个类A）

```
class_getMethodImplementation(objc_getMetaClass("A"),@selector(methodName));
```

实例方法

```
class_getMethodImplementation([A class],@selector(methodName));
```

通过该传入的参数不同，找到不同的方法列表，方法列表中保存着下面方法的结构体，结构体中包含这方法的实现，selector本质就是方法的名称，通过该方法名称，即可在结构体中找到相应的实现。

```
struct objc_method {SEL method_namechar *method_typesIMP method_imp}
```

而对于第二种方法而言，传入的参数只有method，区分类方法和实例方法在于封装method的函数

类方法

```
Method class_getClassMethod(Class cls, SEL name
)
```

实例方法

```
Method class_getInstanceMethod(Class cls, SEL name
)
```

最后调用`IMP method_getImplementation(Method m)`获取IMP地址

实验

```
@implementation Test

- (instancetype)init
{
    self = [super init];
    if (self) {
        [self getIMPFromSelector:@selector(aaa)];
        [self getIMPFromSelector:@selector(test1)];
        [self getIMPFromSelector:@selector(test2)];
    }
    return self;
}

- (void)test1 {
    NSLog(@"test1");
}

+ (void)test2 {
    NSLog(@"test2");
}

- (void)getIMPFromSelector:(SEL)aSelector {

    // first method
    IMP instanceIMP1 = class_getMethodImplementation(objc_getClass("Test"), aSelector);
    IMP classIMP1 = class_getMethodImplementation(objc_getMetaClass("Test"), aSelector);

    // second method
    Method instanceMethod = class_getInstanceMethod(objc_getClass("Test"), aSelector);
    IMP instanceIMP2 = method_getImplementation(instanceMethod);

    Method classMethod1 = class_getClassMethod(objc_getClass("Test"), aSelector);
    IMP classIMP2 = method_getImplementation(classMethod1);

    Method classMethod2 = class_getClassMethod(objc_getMetaClass("Test"), aSelector);
    IMP classIMP3 = method_getImplementation(classMethod2);

    NSLog(@"instance1:%p instance2:%p class1:%p class2:%p class3:%p", instanceIMP1, instanceIMP2, classIMP1,
}

@end
```


这里有一个叫Test的类，在初始化方法里，调用了两次getIMPFromSelector:方法，第一个aaa方法是不存在的，test1和test2分别为实例方法和类方法

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Test *test1 = [[Test alloc] init];
    Test *test2 = [[Test alloc] init];
}
```


然后我同时实例化了两个Test的对象，打印信息如下

```
2016-09-15 11:16:11.092 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.092 GetIMPFromSelector[12399:659621] instance1:0x10fdc0720 instance2:0x10fdc0720 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x10fdc0750 class2:0x10fdc0750 class3:0x10fdc0750
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x10fdc0720 instance2:0x10fdc0720 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.094 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x10fdc0750 class2:0x10fdc0750 class3:0x10fdc0750
```


大家注意图中红色标注的地址出现了8次：0x1102db280，这个是在调用class_getMethodImplementation()方法时，无法找到对应实现时返回的相同的一个地址，无论该方法是在实例方法或类方法，无论是否对一个实例调用该方法，返回的地址都是相同的，但是每次运行该程序时返回的地址并不相同，而对于另一种方法，如果找不到对应的实现，则返回0，在图中我做了蓝色标记。

还有一点有趣的是class_getClassMethod()的第一个参数无论传入objc_getClass()还是objc_getMetaClass()，最终调用method_getImplementation()都可以成功的找到类方法的实现。而class_getInstanceMethod()的第一个参数如果传入objc_getMetaClass()，再调用method_getImplementation()时无法找到实例方法的实现却可以找到类方法的实现。

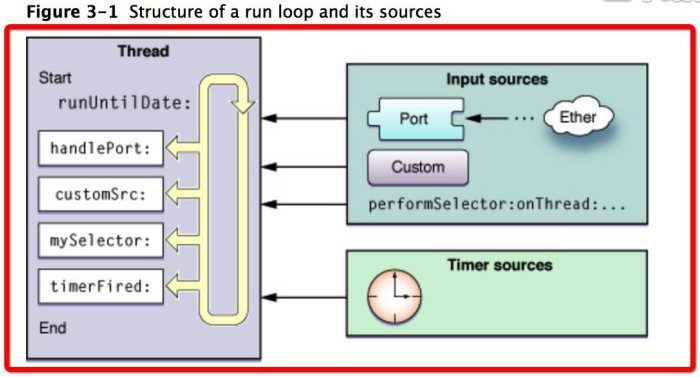
4.runloop内部实现逻辑?

A run loop receives events from two different types of sources. *Input sources* deliver asynchronous events, usually messages from another thread or from a different application. *Timer sources* deliver synchronous events, occurring at a scheduled time or repeating interval. Both types of source use an application-specific handler routine to process the event when it arrives.

Figure 3-1 shows the conceptual structure of a run loop and a variety of sources. The input sources deliver asynchronous events to the corresponding handlers and cause the `runUntilDate:` method (called on the thread's associated `NSRunLoop` object) to exit. Timer sources deliver events to their handler routines but do not cause the run loop to exit.

基本结构

Figure 3-1 Structure of a run loop and its sources



苹果在文档里的说明，RunLoop 内部的逻辑大致如下:

其内部代码整理如下：

可以看到，实际上 RunLoop 就是这样一个函数，其内部是一个 do-while 循环。当你调用 CFRunLoopRun() 时，线程就会一直停留在这个循环里；直到超时或被手动停止，该函数才会返回。

RunLoop 的底层实现

从上面代码可以看到，RunLoop 的核心是基于 mach port 的，其进入休眠时调用的函数是 mach_msg()。为了解释这个逻辑，下面稍微介绍一下 OSX/iOS 的系统架构。

苹果官方将整个系统大致划分为上述4个层次：

1. 应用层包括用户能接触到的图形应用，例如 Spotlight、Aqua、SpringBoard 等。
2. 应用框架层即开发人员接触到的 Cocoa 等框架。
3. 核心框架层包括各种核心框架、OpenGL 等内容。
4. Darwin 即操作系统的核心，包括系统内核、驱动、Shell 等内容，这一层是开源的，其所有源码都可以在 opensource.apple.com 里找到。

我们在深入看一下 Darwin 这个核心的架构：

其中，在硬件层上面的三个组成部分：Mach、BSD、IOKit(还包括一些上面没标注的内容)，共同组成了 XNU 内核。

XNU 内核的内环被称作 Mach，其作为一个微内核，仅提供了诸如处理器调度、IPC (进程间通信)等非常少量的基础服务。

BSD 层可以看作围绕 Mach 层的一个外环，其提供了诸如进程管理、文件系统和网络等功能。

IOKit 层是为设备驱动提供了一个面向对象(C++)的一个框架。

Mach 本身提供的 API 非常有限，而且苹果也不鼓励使用 Mach 的 API，但是这些API非常基础，如果没有这些API的话，其他任何工作都无法实施。在 Mach 中，所有的东西都是通过自己的对象实现的，进程、线程和虚拟内存都被称为"对象"。和其他架构不同， Mach 的对象间不能直接调用，只能通过消息传递的方式实现对象间的通信。"消息"是 Mach 中最基础的概念，消息在两个端口 (port) 之间传递，这就是 Mach 的 IPC (进程间通信) 的核心。

Mach 的消息定义是在头文件的，很简单：

```
typedef struct {
  mach_msg_header_t header;
  mach_msg_body_t body;
} mach_msg_base_t;

typedef struct {
  mach_msg_bits_t msgh_bits;
  mach_msg_size_t msgh_size;
  mach_port_t msgh_remote_port;
  mach_port_t msgh_local_port;
  mach_port_name_t msgh_voucher_port;
  mach_msg_id_t msgh_id;
} mach_msg_header_t;
```

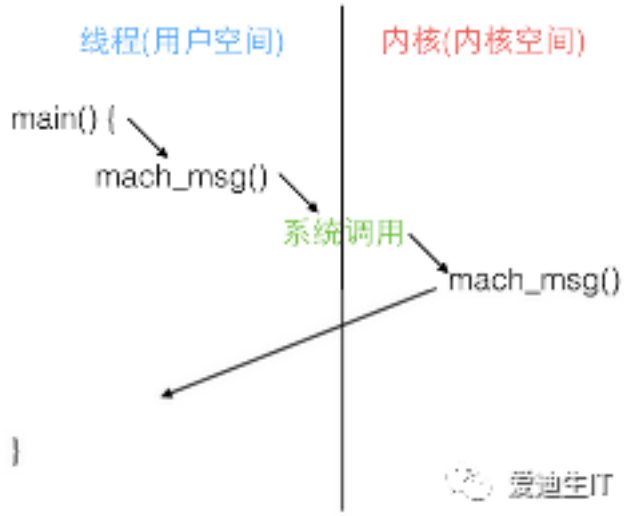
一条 Mach 消息实际上就是一个二进制数据包 (BLOB)，其头部定义了当前端口 local_port 和目标端口 remote_port，

发送和接受消息是通过同一个 API 进行的，其 option 标记了消息传递的方向：

```
mach_msg_return_t mach_msg(
mach_msg_header_t *msg,
mach_msg_option_t option,
mach_msg_size_t send_size,
mach_msg_size_t rcv_size,
mach_port_name_t rcv_name,
mach_msg_timeout_t timeout,
mach_port_name_t notify);
```

为了实现消息的发送和接收，mach_msg() 函数实际上是调用了一个 Mach 陷阱 (trap)，即函数mach_msg_trap()，陷阱这个概念在 Mach 中等同于系统调用。当你在用户态调用

mach_msg_trap() 时会触发陷阱机制，切换到内核态；内核态中内核实现的 mach_msg() 函数会完成实际的工作，如下图：



这些概念可以参考维基百科: System_call、Trap_(computing)。

RunLoop 的核心就是一个 mach_msg() (见上面代码的第7步)，RunLoop 调用这个函数去接收消息，如果没有别人发送 port 消息过来，内核会将线程置于等待状态。例如你在模拟器里跑起一个 iOS 的 App，然后在 App 静止时点击暂停，你会看到主线程调用栈是停留在 mach_msg_trap() 这个地方。

关于具体的如何利用 mach port 发送信息，可以看看 NSHipster 这一篇文章，或者这里的中文翻译 。

关于Mach的历史可以看看这篇很有趣的文章：Mac OS X 背后的故事（三）Mach 之父 Avie Tevanian。

苹果用 **RunLoop** 实现的功能

首先我们可以看一下 App 启动后 RunLoop 的状态：

可以看到，系统默认注册了5个Mode:

1. kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。

2. UITrackingRunLoopMode: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。

3. UIInitializationRunLoopMode: 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用。

4: GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到。

5: kCFRunLoopCommonModes: 这是一个占位的 Mode，没有实际作用。

你可以在这里看到更多的苹果内部的 Mode，但那些 Mode 在开发中就很难遇到了。

5.你理解的多线程?

1.可能会追问，每种多线程基于什么语言?

2.生命周期是如何管理?

3.你更倾向于哪种？追问至现在常用的两种你的看法是?

第一种：pthread

.特点：

1）一套通用的多线程API

2）适用于Unix\Linux\Windows等系统

3）跨平台\可移植

4）使用难度大

b.使用语言：c语言

c.使用频率：几乎不用

d.线程生命周期：由程序员进行管理

第二种：NSThread

a.特点：

1）使用更加面向对象

2）简单易用，可直接操作线程对象

b.使用语言：OC语言

c.使用频率：偶尔使用

d.线程生命周期：由程序员进行管理

第三种：GCD

a.特点：

1）旨在替代NSThread等线程技术

2）充分利用设备的多核（自动）

b.使用语言：C语言

c.使用频率：经常使用

d.线程生命周期：自动管理

第四种：NSOperation

a.特点：

1）基于GCD（底层是GCD）

2）比GCD多了一些更简单实用的功能

3）使用更加面向对象

b.使用语言：OC语言

c.使用频率：经常使用

d.线程生命周期：自动管理

多线程的原理

同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）

多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）

如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象

思考：如果线程非常非常多，会发生什么情况?

CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源

每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优点

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

多线程的缺点

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

你更倾向于哪一种?

倾向于GCD:

GCD技术是一个轻量的，底层实现隐藏的神奇技术，我们能够通过GCD和block轻松实现多线程编程，有时候，GCD相比其他系统提供的多线程方法更加有效，当然，有时候GCD不是最佳选择，另一个多线程编程的技术NSOprationQueue 让我们能够将后台线程以队列方式依序执行，并提供更多操作的入口，这和 GCD 的实现有些类似。

这种类似不是一个巧合，在早期，MacOX 与 iOS 的程序都普遍采用Operation Queue来进行编写后台线程代码，而之后出现的GCD技术大体是依照前者的原则来实现的，而随着GCD的普及，在iOS 4 与 MacOS X 10.6以后，Operation Queue的底层实现都是用GCD来实现的。

那这两者直接有什么区别呢?

1. GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2. 在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3. NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4. 我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5. 在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6. 我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。

总的来说，Operation queue 提供了更多你在编写多线程程序时需要的功能，并隐藏了许多线程调度，线程取消与线程优先级的复杂代码，为我们提供简单的API入口。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。但是我认为当我们的需求能够以更简单的底层代码完成的时候，简洁的GCD或许是个更好的选择，而Operation queue为我们提供能更多的选择。

倾向于：**NSOperation**

NSOperation相对于GCD：

1，NSOperation拥有更多的函数可用，具体查看api。NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

2，在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系。

3，NSOperationQueue支持KVO。可以监测operation是否正在执行（isExecuted）、是否结束（isFinished），是否取消（isCanceld）

4，GCD只支持FIFO的队列，而NSOperationQueue可以调整队列的执行顺序（通过调整权重）。NSOperationQueue可以方便的管理并发、NSOperation之间的优先级。

使用NSOperation的情况：各个操作之间有依赖关系、操作需要取消暂停、并发管理、控制操作之间优先级，限制同时能执行的线程数量.让线程在某时刻停止/继续等。

使用GCD的情况：一般的需求很简单的多线程操作，用GCD都可以了，简单高效。

从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。

当需求简单，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

5.你理解的多线程?

1.可能会追问，每种多线程基于什么语言?

2.生命周期是如何管理?

3.你更倾向于哪种？追问至现在常用的两种你的看法是?

第一种：pthread

.特点:

1）一套通用的多线程API

2）适用于Unix\Linux\Windows等系统

3）跨平台\可移植

4）使用难度大

b.使用语言：c语言

c.使用频率：几乎不用

d.线程生命周期：由程序员进行管理

第二种：NSThread

a.特点:

1）使用更加面向对象

2）简单易用，可直接操作线程对象

b.使用语言：OC语言

c.使用频率：偶尔使用

d.线程生命周期：由程序员进行管理

第三种：GCD

a.特点：

1）旨在替代NSThread等线程技术

2）充分利用设备的多核（自动）

b.使用语言：C语言

c.使用频率：经常使用

d.线程生命周期：自动管理

第四种：NSOperation

a.特点：

1）基于GCD（底层是GCD）

2）比GCD多了一些更简单实用的功能

3）使用更加面向对象

b.使用语言：OC语言

c.使用频率：经常使用

d.线程生命周期：自动管理

多线程的原理

同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）

多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）

如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象

思考：如果线程非常非常多，会发生什么情况?

CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源

每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优点

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

多线程的缺点

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

你更倾向于哪一种?

倾向于GCD:

GCD技术是一个轻量的，底层实现隐藏的神奇技术，我们能够通过GCD和block轻松实现多线程编程，有时候，GCD相比其他系统提供的多线程方法更加有效，当然，有时候GCD不是最佳选择，另一个多线程编程的技术NSOprationQueue 让我们能够将后台线程以队列方式依序执行，并提供更多操作的入口，这和 GCD 的实现有些类似。

这种类似不是一个巧合，在早期，MacOX 与 iOS 的程序都普遍采用Operation Queue来进行编写后台线程代码，而之后出现的GCD技术大体是依照前者的原

则来实现的，而随着GCD的普及，在iOS 4 与 MacOS X 10.6以后，Operation Queue的底层实现都是用GCD来实现的。

那这两者直接有什么区别呢?

1. GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2. 在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3. NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4. 我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5. 在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6. 我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。

总的来说，Operation queue 提供了更多你在编写多线程程序时需要的功能，并隐藏了许多线程调度，线程取消与线程优先级的复杂代码，为我们提供简单的API入口。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。但是我认为当我们的需求能够以更简单的底层代码完成的时候，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

倾向于：**NSOperation**

NSOperation相对于GCD：

1，NSOperation拥有更多的函数可用，具体查看api。NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

2，在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系。

3，NSOperationQueue支持KVO。可以监测operation是否正在执行（isExecuted）、是否结束（isFinished），是否取消（isCanceld）

4，GCD只支持FIFO的队列，而NSOperationQueue可以调整队列的执行顺序（通过调整权重）。NSOperationQueue可以方便的管理并发、NSOperation之间的优先级。

使用NSOperation的情况：各个操作之间有依赖关系、操作需要取消暂停、并发管理、控制操作之间优先级，限制同时能执行的线程数量.让线程在某时刻停止/继续等。

使用GCD的情况：一般的需求很简单的多线程操作，用GCD都可以了，简单高效。

从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。

当需求简单，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

6.GCD执行原理?

GCD有一个底层线程池，这个池中存放的是一个个的线程。之所以称为“池”，

很容易理解出这个“池”中的线程是可以重用的，当一段时间后这个线程没有被调用胡话，这个线程就会被销毁。注意：开多少条线程是由底层线程池决定的（线程建议控制再3~5条），池是系统自动来维护，不需要我们程序员来维护（看到这句话是不是很开心？）
而我们程序员需要关心的是什么呢？我们只关心的是向队列中添加任务，队列调度即可。

• 如果队列中存放的是同步任务，则任务出队后，底层线程池中会提供一条线程供这个任务执行，任务执行完毕后这条线程再回到线程池。这样队列中的任务反复调度，因为是同步的，所以当我们用currentThread打印的时候，就是同一条线程。

• 如果队列中存放的是异步的任务，（注意异步可以开线程），当任务出队后，底层线程池会提供一个线程供任务执行，因为是异步执行，队列中的任务不需等待当前任务执行完毕就可以调度下一个任务，这时底层线程池中会再次提供一个线程供第二个任务执行，执行完毕后再回到底层线程池中。

• 这样就对线程完成一个复用，而不需要每一个任务执行都开启新的线程，也就从而节约的系统的开销，提高了效率。在iOS7.0的时候，使用GCD系统通常只能开5~8条线程，iOS8.0以后，系统可以开启很多条线程，但是实在开发应用中，建议开启线程条数：3~5条最为合理。

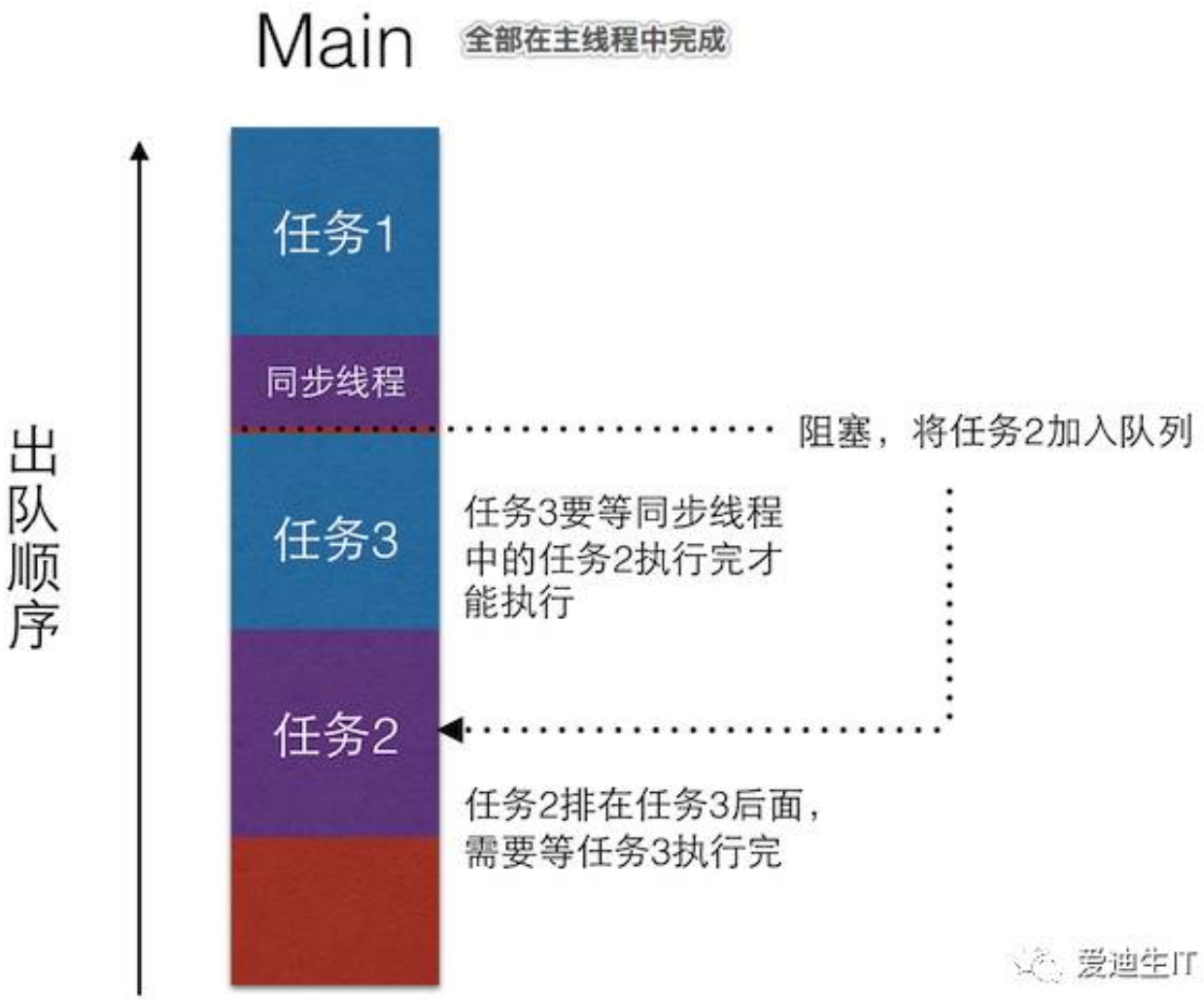
通过案例明白GCD的执行原理

案例一：

```
NSLog(@"1"); // 任务1

dispatch_sync(dispatch_get_main_queue(), ^{

    NSLog(@"2"); // 任务2
});

NSLog(@"3"); // 任务3

输出结果：
1
```

分析：
首先执行任务1，这是肯定没问题的，只是接下来，程序遇到了同步线程，那么它会进入等待，等待任务2执行完，然后执行任务3。但这是队列，有任务来，当然会将任务加到队尾，然后遵循FIFO原则执行任务。那么，现在任务2就会被加到最后，任务3排在了任务2前面，问题来了：

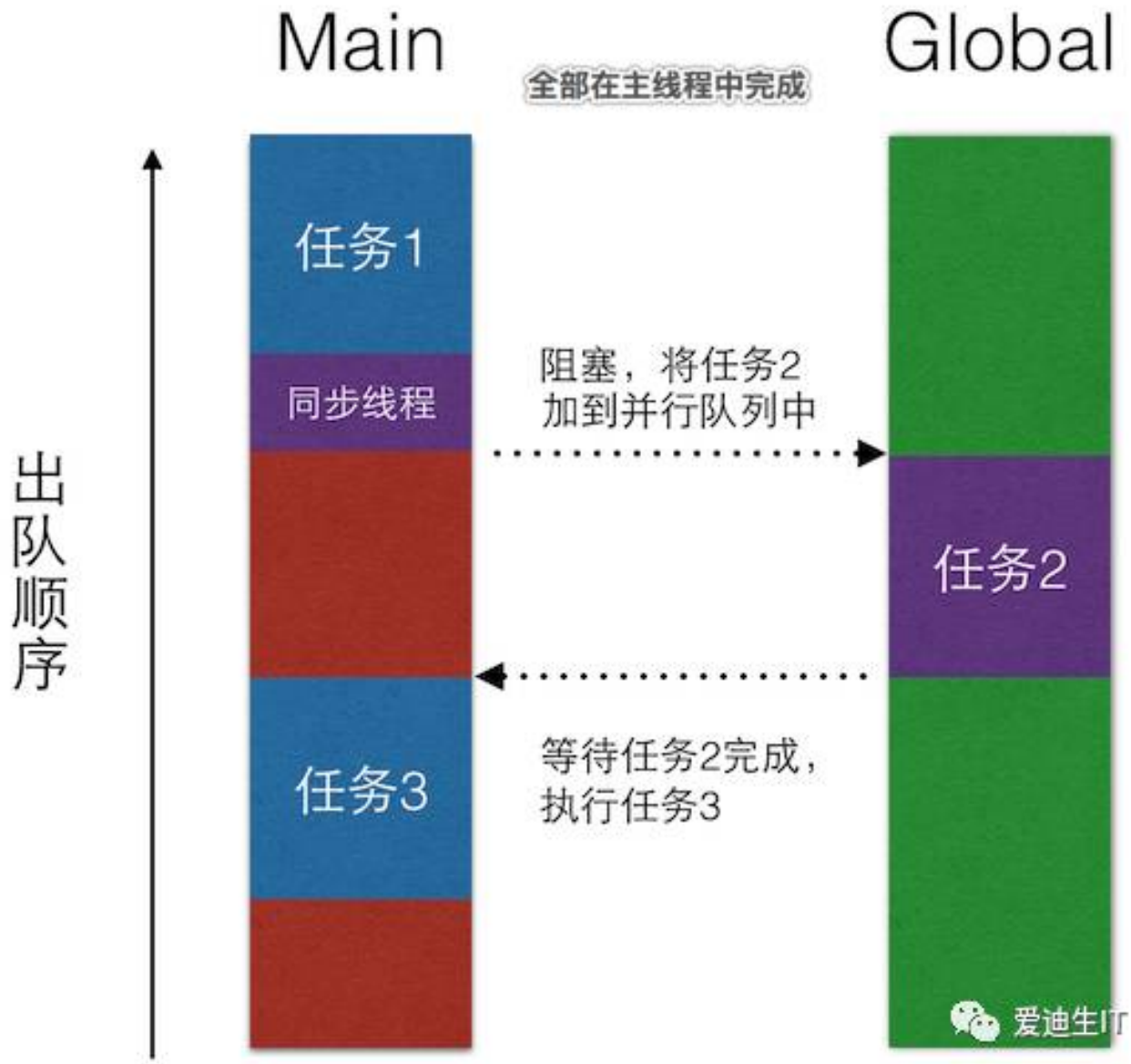
任务3要等任务2执行完才能执行，任务2又排在任务3后面，意味着任务2要在任务3执行完才能执行，所以他们进入了互相等待的局面。【既然这样，那干脆就卡在这里吧】这就是死锁。



案例二：

```
NSLog(@"1"); // 任务1

dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{

    NSLog(@"2"); // 任务2
});

NSLog(@"3"); // 任务3

输出结果：
1
2
3
```


分析：
首先执行任务1，接下来会遇到一个同步线程，程序会进入等待。等待任务2执行完成以后，才能继续执行任务3。从`dispatch_get_global_queue`可以看出，任务2被加入到了全局的并行队列中，当并行队列执行完任务2以后，返回到主队列，继续执行任务3。


Main
Global
全部在主线程中完成
任务1
同步线程
阻塞，将任务2
加到并行队列中
任务2
任务3
等待任务2完成，
执行任务3
出队顺序
爱迪生IT

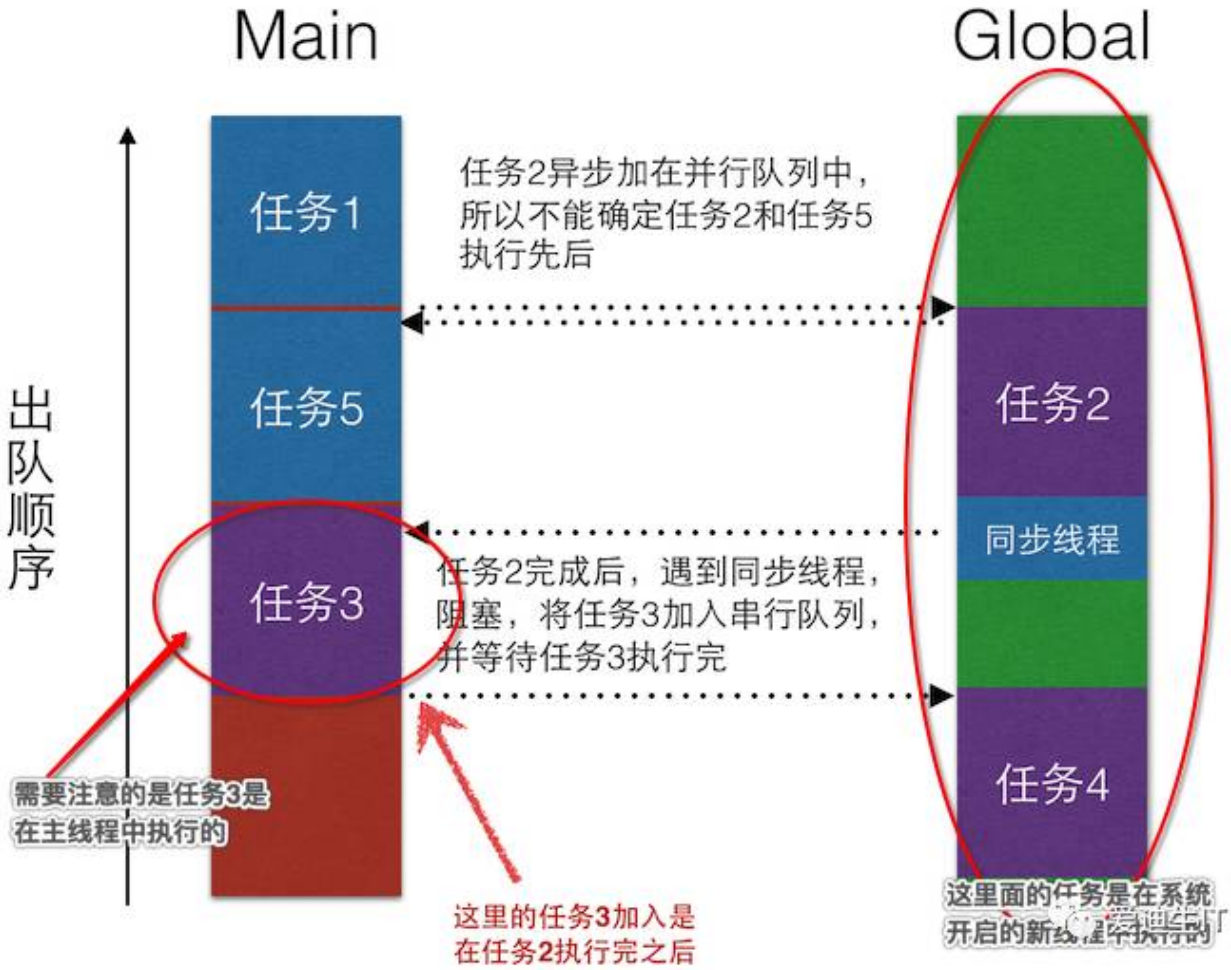
案例三：

```
dispatch_queue_t queue = dispatch_queue_create("com.demo.serialQueue", DISPATCH_Q
UEUE_SERIAL);

NSLog(@"1"); // 任务1

dispatch_async(queue, ^{

    NSLog(@"2"); // 任务2

    dispatch_sync(queue, ^{

        NSLog(@"3"); // 任务3

    });

    NSLog(@"4"); // 任务4

});

NSLog(@"5"); // 任务5(会在主线程中执行)

输出结果:
1
5
2
// 5和2的顺序不一定
```


案例四:

```
NSLog(@"1"); // 任务1

dispatch_async(dispatch_get_global_queue(0, 0), ^{

    NSLog(@"2"); // 任务2

    dispatch_sync(dispatch_get_main_queue(), ^{

        NSLog(@"3"); // 任务3

    });

    NSLog(@"4"); // 任务4

});

NSLog(@"5"); // 任务5

输出结果:
1
2
5
3
4
// 5和2的顺序不一定
```

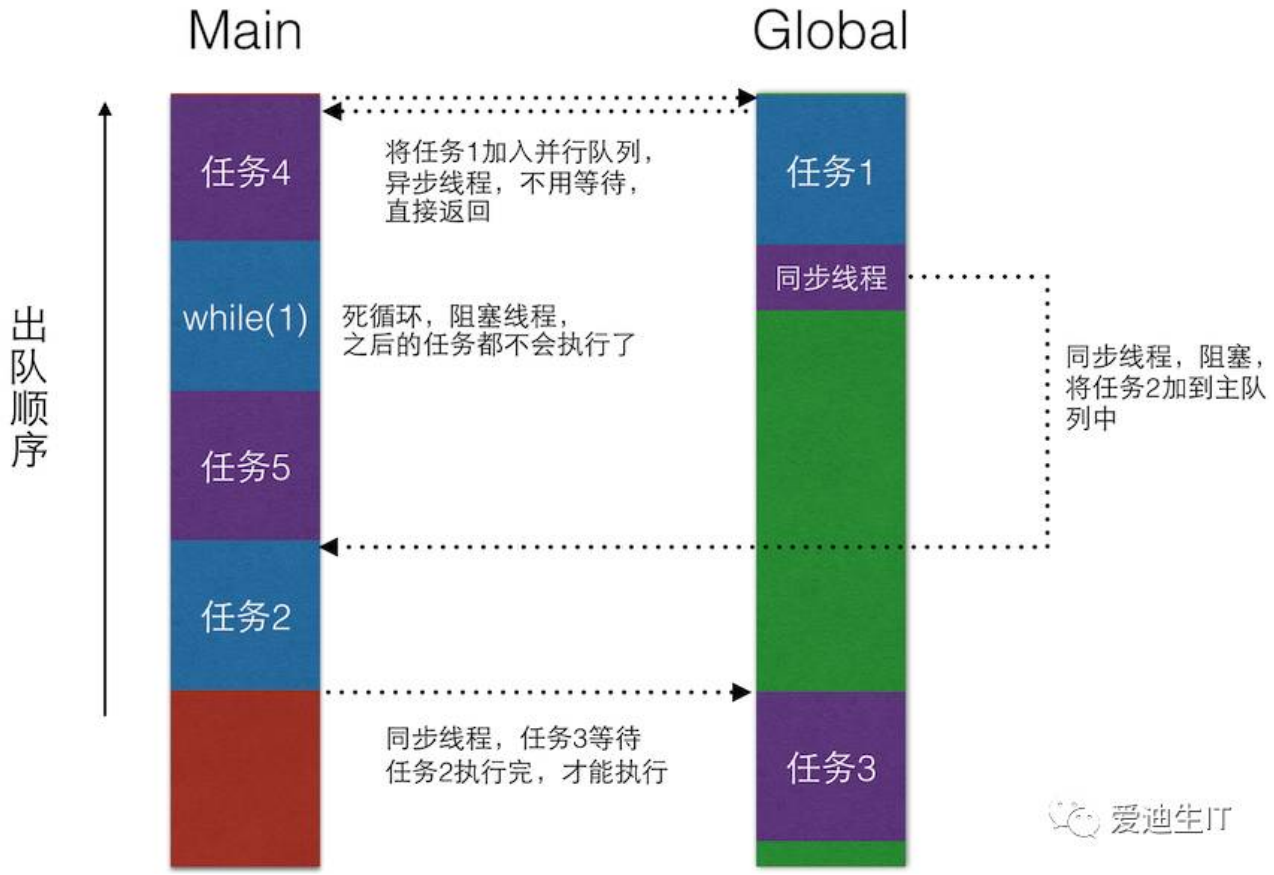

分析:
首先，将【任务1、异步线程、任务5】加入MainQueue中，异步线程中的任务是：【任务2、同步线程、任务4】。所以，先执行任务1，然后将异步线程中的任务加入到GlobalQueue中，因为异步线程，所以任务5不用等待，结果就是2和5的输出顺序不一定。然后再看异步线程中的任务执行顺序。任务2执行完以后，遇到同步线程。将同步线程中的任务加入到Main Queue中，这时加入的任务3在任务5的后面。当任务3执行完以后，没有了阻塞，程序继续执行任务4。


Main
Global
任务1
任务5
任务3
任务2异步加在并行队列中，
所以不能确定任务2和任务5
执行先后
出队顺序
任务2
同步线程
任务2完成后，遇到同步线程，
阻塞，将任务3加入串行队列，
并等待任务3执行完
任务4
需要注意的是任务3是
在主线程中执行的
这里的任务3加入是
在任务2执行完之后
这里面的任务是在系统

案例五：

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{

    NSLog(@"1"); // 任务1

    dispatch_sync(dispatch_get_main_queue(), ^{

        NSLog(@"2"); // 任务2

    });

    NSLog(@"3"); // 任务3

});

NSLog(@"4"); // 任务4

while (1) {

}

NSLog(@"5"); // 任务5

输出结果:
1
4
// 1和4的顺序不一定
```


分析:
和上面几个案例的分析类似，先来看看都有哪些任务加入了Main Queue:
【异步线程、任务4、死循环、任务5】。
在加入到Global Queue异步线程中的任务有:
【任务1、同步线程、任务3】。第一个就是异步线程，任务4不用等待，
所以结果任务1和任务4顺序不一定。任务4完成后，程序进入死循环，
Main Queue阻塞。但是加入到Global Queue的异步线程不受影响，
继续执行任务1后面的同步线程。同步线程中，将任务2加入到了主线程，
并且，任务3等待任务2完成以后才能执行。这时的主线程，已经被死循环阻塞了。
所以任务2无法执行，当然任务3也无法执行，在死循环后的任务5也不会执行。



7.怎么防止别人动态在你程序生成代码?

（这题是听错了面试官的意思）

面试官意思是怎么防止别人反编译你的app?

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

6.借助第三方APP加固，例如：网易云易盾

8.YYAsyncLayer如何异步绘制?

YYAsyncLayer是异步绘制与显示的工具。为了保证列表滚动流畅，将视图绘制、以及图片解码等任务放到后台线程，

YYKitDemo

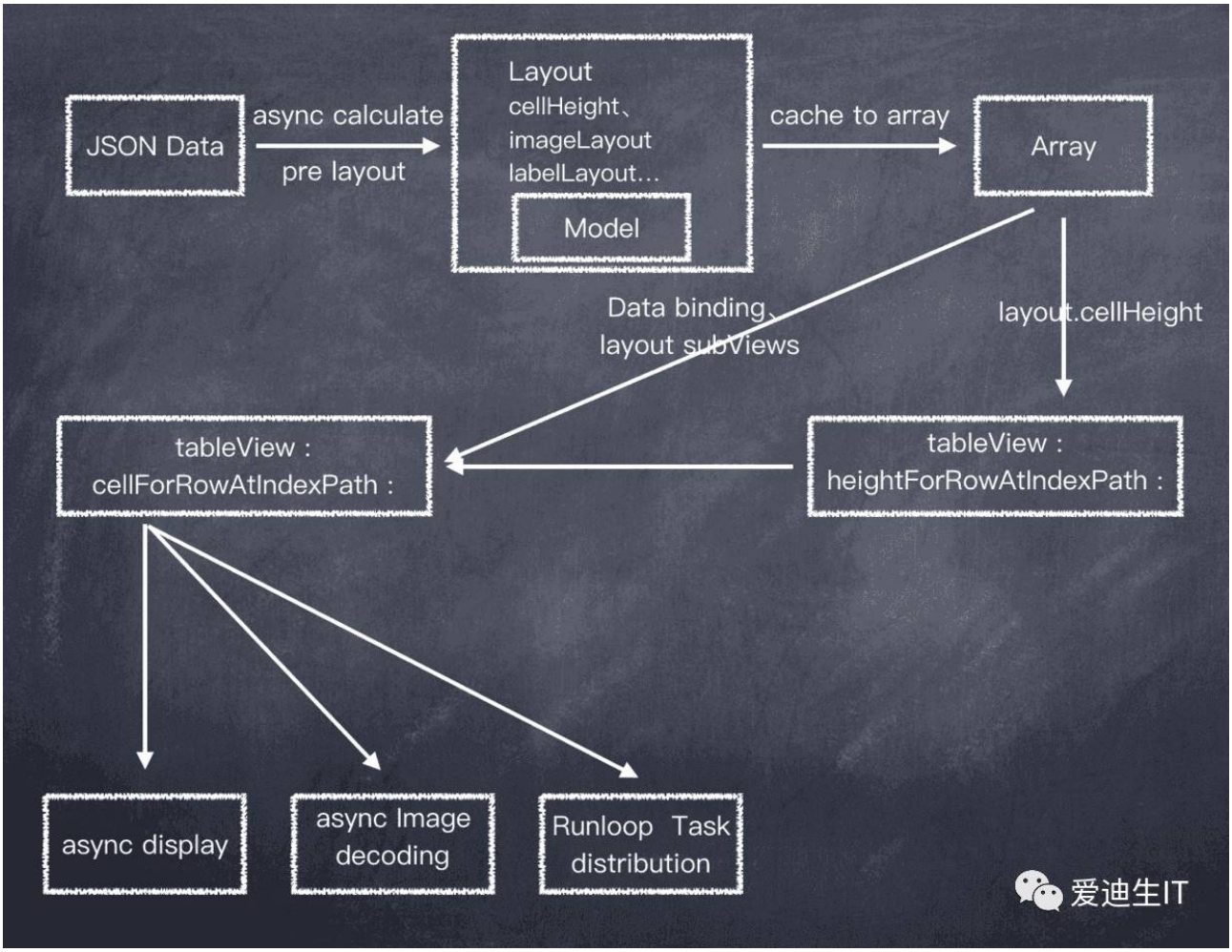
对于列表主要对两个代理方法的优化，一个与绘制显示有关，另一个与计算布局有关：

Objective-C

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
                                (NSIndexPath *)indexPath;

- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
                                (NSIndexPath *)indexPath;
```

常规逻辑可能觉得应该先调用`tableView : cellForRowAtIndexPath :`返回`UITableViewCell`对象，事实上调

用顺序是先返回UITableViewCell的高度，是因为UITableView继承自UIScrollView，滑动范围由属性contentSize来确定，UITableView的滑动范围需要通过每一行的UITableViewCell的高度计算确定，复杂cell如果在列表滚动过程中计算可能会造成一定程度的卡顿。
假设有20条数据，当前屏幕显示5条，tableView : heightForRowAtIndexPath :方法会先执行20次返回所有高度并计算出滑动范围，tableView : cellForRowAtIndexPath :执行 5 次返回当前屏幕显示的cell个数。



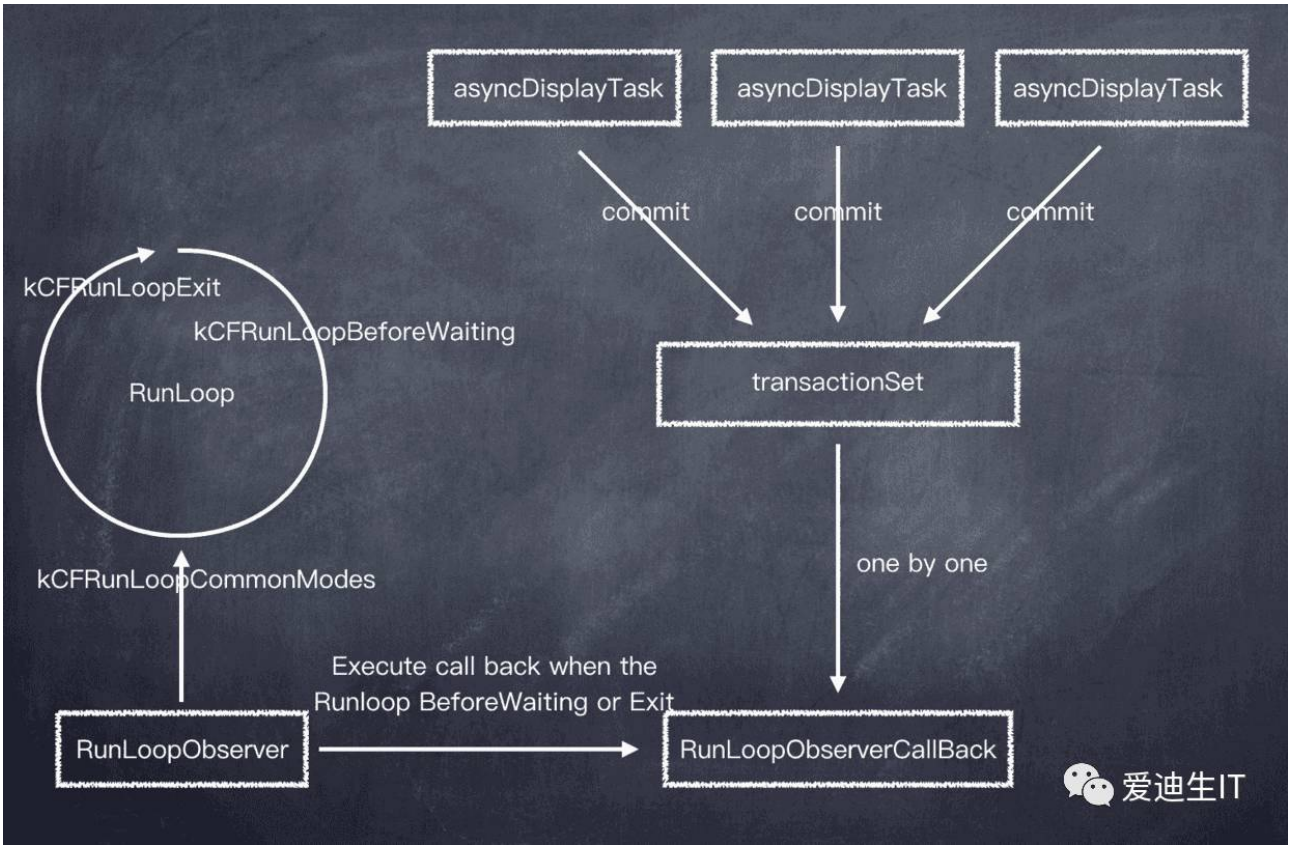
从图中简单看下流程，从网络请求返回JSON数据，将Cell的高度以及内部视图的布局封装为Layout对象，Cell显示之前在异步线程计算好所有布局对象，并存入数组，每次调用tableView: heightForRowAtIndexPath :只需要从数组中取出，可避免重复的布局计算。同时在调用tableView: cellForRowAtIndexPath :对Cell

内部视图异步绘制布局，以及图片的异步绘制解码，这里就要说到今天的主角YYAsyncLayer。

YYAsyncLayer

首先介绍里面几个类：

- YYAsyncLayer：继承自CALayer，绘制、创建绘制线程的部分都在这个类。
- YYTransaction：用于创建RunloopObserver监听MainRunloop的空闲时间，并将YYTranaction对象存放到集合中。
- YYSentinel：提供获取当前值的`value`（只读）属性，以及`-(int32_t)increase`自增加的方法返回一个新的`value`值，用于判断异步绘制任务是否被取消的工具。



AsyncDisplay.png

上图是整体异步绘制的实现思路，后面一步步说明。现在假设需要绘制Label，其实是继承自UIView，重写`+ (Class)layerClass`，在需要重新绘制的地方调用下面方法，比如`setter`，`layoutSubviews`。

Objective-C

```
+ (Class)layerClass {
return YYAsyncLayer.class;
}
- (void)setText:(NSString *)text {
_text = text.copy;
[[YYTransaction transactionWithTarget:self selector:@selector(contentsNeedUpdated)] commit];
}
- (void)layoutSubviews {
[super layoutSubviews];
[[YYTransaction transactionWithTarget:self selector:@selector(contentsNeedUpdated)] commit];
}
```

YYTransaction有`selector`、`target`的属性，`selector`其实就是`contentsNeedUpdated`方法，此时并不会立即在后台线程去更新显示，而是将YYTransaction对象本身提交保存在`transactionSet`的集合中，上图中所示。

Objective-C

```
+ (YYTransaction *)transactionWithTarget:(id)target selector:(SEL)selector{
    if (!target || !selector) return nil;
    YYTransaction *t = [YYTransaction new];
    t.target = target;
    t.selector = selector;
    return t;
}
- (void)commit {
    if (!_target || !_selector) return;
    YYTransactionSetup();
    [transactionSet addObject:self];
}

```

同时在YYTransaction.m中注册一个RunloopObserver，监听MainRunloop在`kCFRunLoopCommonModes`（包含`kCFRunLoopDefaultMode`、`UITrackingRunLoopMode`）下的`kCFRunLoopBeforeWaiting`和`kCFRunLoopExit`的状态，也就是说在一次Runloop空闲时去执行更新显示的操作。

`kCFRunLoopBeforeWaiting`：Runloop将要进入休眠。
`kCFRunLoopExit`：即将退出本次Runloop。

Objective-C

```
static void YYTransactionSetup() {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        transactionSet = [NSMutableSet new];
        CFRunLoopRef runloop = CFRunLoopGetMain();
        CFRunLoopObserverRef observer;
        observer = CFRunLoopObserverCreate(CFAllocatorGetDefault(),
                                           kCFRunLoopBeforeWaiting | kCFRunLoopExit,
                                           true,      // repeat
                                           0xFFFFFF,  // after CATransaction(2000000)
                                           YYRunLoopObserverCallBack, NULL);
        CFRunLoopAddObserver(runloop, observer, kCFRunLoopCommonModes);
        CFRelease(observer);
    });
}
```

下面是RunloopObserver的回调方法，从transactionSet取出transaction对象执行SEL的方法，分发到每一次Runloop执行，避免一次Runloop执行时间太长。

Objective-C

```
static void YYRunLoopObserverCallBack(CFRunLoopObserverRef observer,
CFRunLoopActivity activity, void *info) {

    if (transactionSet.count == 0) return;

    NSSet *currentSet = transactionSet;

    transactionSet = [NSMutableSet new];

    [currentSet enumerateObjectsUsingBlock:^(YYTransaction *transaction, BOOL
*stop) {

        #pragma clang diagnostic push

        #pragma clang diagnostic ignored "-Warc-performSelector-leaks"

        [transaction.target performSelector:transaction.selector];

        #pragma clang diagnostic pop

    }];

}
```

接下来是异步绘制，这里用了一个比较巧妙的方法处理，当使用GCD时提交大量并发任务到后台线程导致线程被锁住、休眠的情况，创建与程序当前激活CPU数量（`activeProcessorCount`）相同的串行队列，并限制`MAX_QUEUE_COUNT`，将队列存放在数组中。
YYAsyncLayer.m有一个方法`YYAsyncLayerGetDisplayQueue`来获取这个队列用于绘制（这部分YYKit中有独立的工具YYDispatchQueuePool）。创建队列中有一个参数是告诉队列执行任务的服务质量quality of service，在iOS8+之后相比之前系统有所不同。

- iOS8之前队列优先级：

```
DISPATCH_QUEUE_PRIORITY_HIGH 2 高优先级
DISPATCH_QUEUE_PRIORITY_DEFAULT 0 默认优先级
DISPATCH_QUEUE_PRIORITY_LOW (-2) 低优先级
DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN 后台优先级
```

- iOS8+之后：

QOS_CLASS_USER_INTERACTIVE 0x21,用户交互(希望尽快完成，不要放太耗时操作)
QOS_CLASS_USER_INITIATED 0x19,用户期望(不要放太耗时操作)
QOS_CLASS_DEFAULT 0x15,默认(用来重置对列使用的)
QOS_CLASS_UTILITY 0x11,实用工具(耗时操作，可以使用这个选项)
QOS_CLASS_BACKGROUND 0x09,后台
QOS_CLASS_UNSPECIFIED 0x00,未指定

Objective-C

```
/// Global display queue, used for content rendering.
static dispatch_queue_t YYAsyncLayerGetDisplayQueue() {
#ifdef YYDispatchQueuePool_h
return YYDispatchQueueGetForQOS(NSQualityOfServiceUserInitiated);
#else
#define MAX_QUEUE_COUNT 16


static int queueCount;
static dispatch_queue_t queues[MAX_QUEUE_COUNT];          //存放队列的数组
static dispatch_once_t onceToken;
static int32_t counter = 0;
dispatch_once(&onceToken, ^{
//程序激活的处理器数量
queueCount = (int)[NSProcessInfo processInfo].activeProcessorCount;
queueCount = queueCount  MAX_QUEUE_COUNT ? MAX_QUEUE_COUNT : queueCount);
if ([UIDevice currentDevice].systemVersion.floatValue >= 8.0) {
for (NSUInteger i = 0; i
```

接下来是关于绘制部分的代码，对外接口`YYAsyncLayerDelegate`代理中提供`- (YYAsyncLayerDisplayTask *)newAsyncDisplayTask`方法用于回调绘制的代码，以及是否异步绘制的BOOl类型属性`displaysAsynchronously`，同时重写CALayer的`display`方法来调

用绘制的方法`- (void)_displayAsync:(BOOL)async`。
这里有必要了解关于后台的绘制任务何时会被取消，下面两种情况需要取消，并调用了YYSentinel的`increase`方法，使`value`值增加（线程安全）：

- 在视图调用`setNeedsDisplay`时说明视图的内容需要被更新，将当前的绘制任务取消，需要重新显示。

- 以及视图被释放调用了`dealloc`方法。

在YYAsyncLayer.h中定义了`YYAsyncLayerDisplayTask`类，有三个block属性用于绘制的回调操作，从命名可以看出分别是将要绘制，正在绘制，以及绘制完成的回调，可以从block传入的参数`BOOL(^isCancelled)(void)`判断当前绘制是否被取消。

Objective-C

```
@property (nullable, nonatomic, copy) void (^willDisplay)(CALayer *layer);

@property (nullable, nonatomic, copy) void (^display)(CGContextRef context, CGSize size, BOOL(^isCancelled)(void));

@property (nullable, nonatomic, copy) void (^didDisplay)(CALayer *layer, BOOL finished);
```

下面是部分`- (void)_displayAsync:(BOOL)async`绘制的代码，主要是一些逻辑判断以及绘制函数，在异步执行之前通过`YYAsyncLayerGetDisplayQueue`创建的队列，这里通过`YYSentinel`判断当前的`value`是否等于之前的值，如果不相等，说明绘制任务被取消了，绘制过程会多次判断是否取消，如果是则return，保证被取消的任务能及时退出，如果绘制完毕则设置图片到`layer.contents`。

Objective-C

```
if (async) {  //异步
if (task.willDisplay) task.willDisplay(self);
YYSentinel *sentinel = _sentinel;
int32_t value = sentinel.value;
NSLog(@" --- %d ---", value);
//判断当前计数是否等于之前计数
BOOL (^isCancelled)() = ^BOOL() {
return value != sentinel.value;
};


CGSize size = self.bounds.size;
BOOL opaque = self.opaque;
CGFloat scale = self.contentsScale;
CGColorRef backgroundColor = (opaque && self.backgroundColor) ? CGColorRetain(self.backgroundColor) : NULL;
if (size.width
```

9.优化你是从哪几方面着手?

一、首页启动速度

启动过程中做的事情越少越好（尽可能将多个接口合并）

不在UI线程上作耗时的操作（数据的处理在子线程进行，处理完通知主线程刷新节目）

在合适的时机开始后台任务（例如在用户指引节目就可以开始准备加载的数据）

尽量减小包的大小

优化方法：

量化启动时间

启动速度模块化

辅助工具（友盟，听云，Flurry）

二、页面浏览速度

json的处理（iOS 自带的NSJSONSerialization，Jsonkit，SBJson）

数据的分页（后端数据多的话，就要分页返回，例如网易新闻，或者 微博记录）

数据压缩（大数据也可以压缩返回，减少流量，加快反应速度）

内容缓存（例如网易新闻的最新新闻列表都是要缓存到本地，从本地加载，可以缓存到内存，或者数据库，根据情况而定）

延时加载tab（比如app有5个tab，可以先加载第一个要显示的tab，其他的在显示时候加载，按需加载）

算法的优化（核心算法的优化，例如有些app 有个 联系人姓名用汉语拼音的首字母排序）

三、操作流畅度优化：

Tableview 优化（tableview cell的加载优化）

ViewController加载优化（不同view之间的跳转，可以提前准备好数据）

四、数据库的优化：

数据库设计上面的重构

查询语句的优化

分库分表（数据太多的时候，可以分不同的表或者库）

五、服务器端和客户端的交互优化：

客户端尽量减少请求

服务端尽量做多的逻辑处理

服务器端和客户端采取推拉结合的方式（可以利用一些同步机制）

通信协议的优化。（减少报文的大小）

电量使用优化（尽量不要使用后台运行）

六、非技术性能优化

产品设计的逻辑性（产品的设计一定要符合逻辑，或者逻辑尽量简单，否则会让程序员抓狂，有时候用了好大力气，才可以完成一个小小的逻辑设计问题）

界面交互的规范（每个模块的界面的交互尽量统一，符合操作习惯）

代码规范（这个可以隐形带来app 性能的提高，比如 用if else 还是switch ，或者是用！还是＝＝）

code review（坚持code Review 持续重构代码。减少代码的逻辑复杂度）

日常交流（经常分享一些代码，或者逻辑处理中的坑）

以上问题加参考答案，部分自己回答(群友回答)＋网上博客参考，回答的不好勿喷！

仅供学习使用！ 谢谢！

iOS群：2466454(吹水勿扰)

阿里-p6-一面
1.介绍下内存的几大区域?
2.你是如何组件化解耦的?
3.runtime如何通过selector找到对应的IMP地址
4.runloop内部实现逻辑?
5.你理解的多线程?
6.GCD执行原理?
7.怎么防止别人反编译你的app?
8.YYAsyncLayer如何异步绘制?
9.优化你是从哪几方面着手?

1.介绍下内存的几大区域?

1.栈区(stack) 由编译器自动分配并释放，存放函数的参数值，局部变量等。栈是系统数据结构，对应线程/进程是唯一的。优点是快速高效，缺点时有限制，数据不灵活。［先进后出］

栈空间分静态分配 和动态分配两种。

```
静态分配是编译器完成的，比如自动变量(auto)的分配。
动态分配由alloca函数完成。
栈的动态分配无需释放(是自动的)，也就没有释放函数。
为可移植的程序起见，栈的动态分配操作是不被鼓励的!
```


堆区(heap) 由程序员分配和释放，如果程序员不释放，程序结束时，可能会由操作系统回收 ，比如在ios 中 alloc 都是存放在堆中。
优点是灵活方便，数据适应面广泛，但是效率有一定降低。

```
堆是函数库内部数据结构，不一定唯一。
不同堆分配的内存无法互相操作。
堆空间的分配总是动态的
```


虽然程序结束时所有的数据空间都会被释放回系统，但是精确的申请内存，释放内存匹配是良好程序的基本要素。

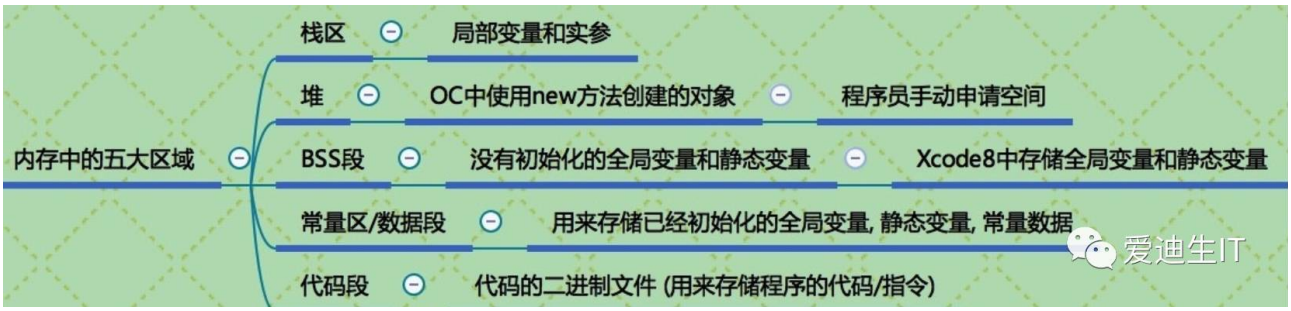
3.全局区(静态区) (static) 全局变量和静态变量的存储是放在一起的，初始化的全局变量和静态变量存放在一块区域，未初始化的全局变量和静态变量在相邻的另一块区域，程序结束后有系统释放。

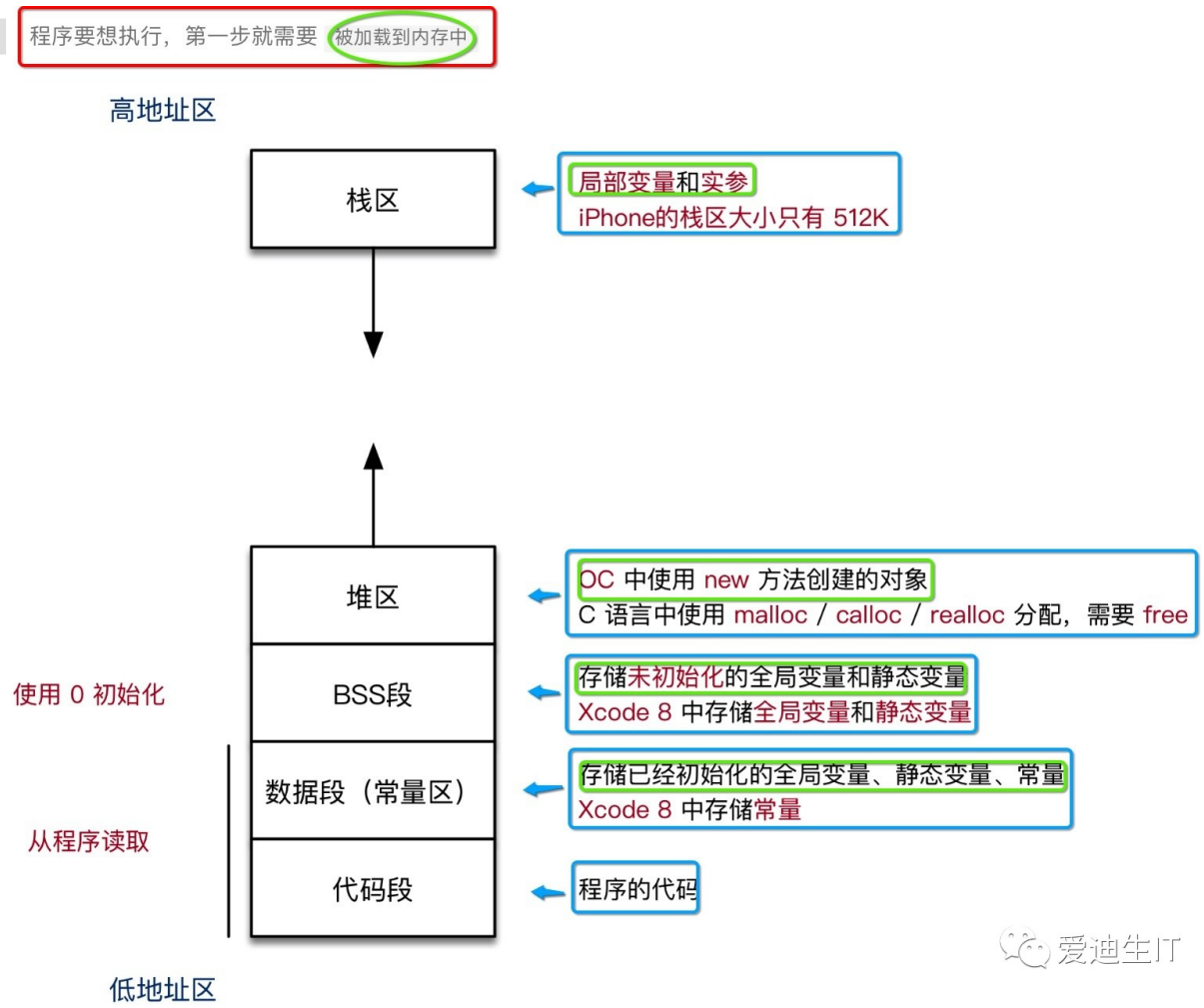
```
注意：全局区又可分为未初始化全局区：
      .bss段和初始化全局区：data段。
      举例：int a;未初始化的。int a = 10;已初始化的。
```


4.文字常量区 存放常量字符串，程序结束后由系统释放；

5.代码区 存放函数的二进制代码

大致如图：


内存中的五大区域
栈区
局部变量和实参
堆
OC中使用new方法创建的对象
程序员手动申请空间
BSS段
没有初始化的全局变量和静态变量
Xcode8中存储全局变量和静态变量
常量区/数据段
用来存储已经初始化的全局变量, 静态变量, 常量数据
代码段
代码的二进制文件 (用来存储程序的代码/指令)
爱迪生IT


程序要想执行，第一步就需要 被加载到内存中
高地址区
栈区
局部变量和实参
iPhone的栈区大小只有 512K
堆区
OC 中使用 new 方法创建的对象
C 语言中使用 malloc / calloc / realloc 分配，需要 free
使用 0 初始化
BSS段
存储未初始化的全局变量和静态变量
Xcode 8 中存储全局变量和静态变量
数据段（常量区）
存储已经初始化的全局变量、静态变量、常量
Xcode 8 中存储常量
从程序读取
代码段
程序的代码
爱迪生IT
低地址区

例子代码：

```
 int a = 10;  全局初始化区
 char *p;  全局未初始化区

main{
  int b; 栈区
  char s[] = "abc" 栈
  char *p1; 栈
  char *p2 = "123456";  123456\\\\0在常量区，p2在栈上。
  static int c =0;  全局（静态）初始化区

  w1 = (char *)malloc(10);
  w2 = (char *)malloc(20);
  分配得来得10和20字节的区域就在堆区。
}
```


可能被追问的问题一：

1.栈区 (stack [stæk]): 由编译器自动分配释放
局部变量是保存在栈区的
方法调用的实参也是保存在栈区的

2.堆区 (heap [hiːp]): 由程序员分配释放，若程序员不释放，会出现内存泄漏，赋值语句右侧使用 new 方法创建的对象，被创建对象的所有 成员变量！

3.BSS 段 : 程序结束后由系统释放

4.数据段 : 程序结束后由系统释放

5.代码段:程序结束后由系统释放
程序编译链接 后的二进制可执行代码

可能被追问的问题二：

比如申请后的系统是如何响应的?

1. 栈：存储每一个函数在执行的时候都会向操作系统索要资源，栈区就是函数运行时的内存，栈区中的变量由编译器负责分配和释放，内存随着函数的运行分配，随着函数的结束而释放，由系统自动完成。
   注意：只要栈的剩余空间大于所申请空间，系统将为程序提供内存，否则将报异常提示栈溢出。

2. 堆：
   1.首先应该知道操作系统有一个记录空闲内存地址的链表。
   2.当系统收到程序的申请时，会遍历该链表，寻找第一个空间大于所申请空间的堆结点，然后将该结点从空闲结点链表中删除，并将该结点的空间分配给程序。
   3 .由于找到的堆结点的大小不一定正好等于申请的大小，系统会自动的将多余的那部分重新放入空闲链表中

可能被追问的问题三：

比如：申请大小的限制是怎样的?

1. 栈：栈是向低地址扩展的数据结构，是一块连续的内存的区域。是栈顶的地址和栈的最大容量是系统预先规定好的，栈的大小是2M（也有的说是1M，总之是一个编译时就确定的常数 ），如果申请的空间超过栈的剩余空间时，将提示overflow。因此，能从栈获得的空间较小。

2. 堆：堆是向高地址扩展的数据结构，是不连续的内存区域。这是由于系统是用链表来存储的空闲内存地址的，自然是不连续的，而链表的遍历方向是由低地址向高地址。堆的大小受限于计算机系统中有效的虚拟内存。由此可见，堆获得的空间比较灵活，也比较大。

栈：由系统自动分配，速度较快，不会产生内存碎片
堆：是由alloc分配的内存，速度比较慢，而且容易产生内存碎片，不过用起来最方便

打个比喻来说：

使用栈就象我们去饭馆里吃饭，只管点菜（发出申请）、付钱、和吃（使用），吃饱了就走，不必理会切菜、洗菜等准备工作和洗碗、刷锅等扫尾工作，他的好处是快捷，但是自由度小。

使用堆就象是自己动手做喜欢吃的菜肴，比较麻烦，但是比较符合自己的口味，而且自由度大。

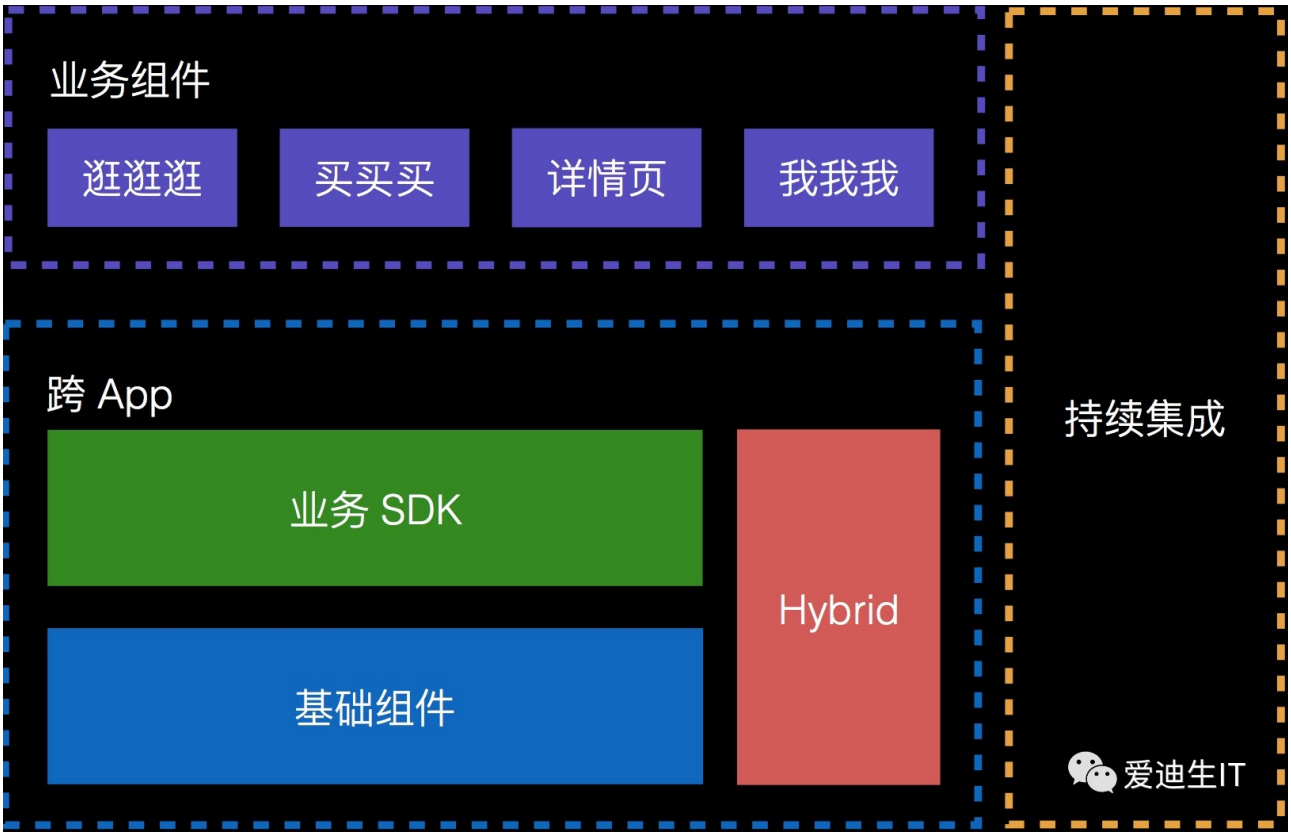
2.你是如何组件化解耦的?

实现代码的高内聚低耦合，方便多人多团队开发！

一般需要解耦的项目都会多多少少出现，一下几个情况：

1. 耦合比较严重（因为没有明确的约束，「组件」间引用的现象会比较多）
   2.容易出现冲突（尤其是使用 Xib，还有就是 Xcode Project，虽说有脚本可以改善）
   3.业务方的开发效率不够高（只关心自己的组件，却要编译整个项目，与其他不相干的代码糅合在一起）

先来看下，组件化之后的一个大概架构



「组件化」顾名思义就是把一个大的 App 拆成一个个小的组件，相互之间不直接引用。那如何做呢？

实现方式

组件间通信

以 iOS 为例，由于之前就是采用的 URL 跳转模式，理论上页面之间的跳转只需 open 一个 URL 即可。所以对于一个组件来说，只要定义「支持哪些 URL」即可，比如详情页，大概可以这么做的

```
[MGJRouter registerURLPattern:@"mgj://detail?id=:id" toHandler:^(NSDictionary *ro
uterParameters) {
    NSNumber *id = routerParameters[@"id"];
    // create view controller with id
    // push view controller
}];
```


首页只需调用 `[MGJRouter openURL:@"mgj://detail?id=404"]` 就可以打开相应的详情页。

那问题又来了，我怎么知道有哪些可用的 URL？为此，我们做了一个后台专门来管理。

ID	变量名	短链	描述	开发人员	需求方	创建时间	操作
4	MGJPAGE_ABOUT	mgj://about	关于蘑菇街	慧能	慧能	2015-12-17 19:28	约束 废弃
5	MGJPAGE_ADDRESS	mgj://address	管理收货地址	慧能	慧能	2015-12-17 19:28	约束 废弃
6	MGJPAGE_APPENDRATE	mgj://appendrate	追加评价	慧能	慧能	2015-12-17 19:28	约束 废弃
7	MGJPAGE_BEAUTY	mgj://beauty	美妆	慧能	慧能	2015-12-17 19:28	约束 废弃

然后可以把这些短链生成不同平台所需的文件，iOS 平台生成 .{h,m} 文件，Android 平台生成 .java 文件，并注入到项目中。这样开发人员只需在项目中打开该文件就知道所有的可用 URL 了。

目前还有一块没有做，就是参数这块，虽然描述了短链，但真想要生成完整的 URL，还需要知道如何传参数，这个正在开发中。

还有一种情况会稍微麻烦点，就是「组件A」要调用「组件B」的某个方法，比如在商品详情页要展示购物车的商品数量，就涉及到向购物车组件拿数据。

类似这种同步调用，iOS 之前采用了比较简单的方案，还是依托于 `MGJRouter`，不过添加了新的方法 `- (id)objectForURL:`，注册时也使用新的方法进行注册

```
[MGJRouter registerURLPattern:@"mgj://cart/ordercount" toObjectHandler:^id(NSDict
ionary *routerParamters){
    // do some calculation
    return @42;
}]
```


使用时 `NSNumber *orderCount = [MGJRouter objectForURL:@"mgj://cart/ordercount"]` 这样就拿到了购物车里的商品数。

稍微复杂但更具通用性的方法是使用「协议」 <-> 「类」绑定的方式，还是以购物车为例，购物车组件可以提供这么个 Protocol

```
@protocol MGJCart <NSObject>
+ (NSInteger)orderCount;
@end
```


可以看到通过协议可以直接指定返回的数据类型。然后在购物车组件内再新建个类实现这个协议，假设这个类名为`MGJCartImpl`，接着就可以把它与协议关联起来 `[ModuleManagerregisterClass:MGJCartImpl forProtocol:@protocol(MGJCart)]`，对于使用方来说，要拿到这个 `MGJCartImpl`，需要调用 `[ModuleManagerclassForProtocol:@protocol(MGJCart)]`。拿到之后再调用 `+(NSInteger)orderCount`就可以了。

那么，这个协议放在哪里比较合适呢？如果跟组件放在一起，使用时还是要先引入组件，如果有多个这样的组件就会比较麻烦了。所以我们把这些公共的协议统一放到了 `PublicProtocolDomain.h` 下，到时只依赖这一个文件就可以了。

Android 也是采用类似的方式。

组件生命周期管理

理想中的组件可以很方便地集成到主客中，并且有跟 AppDelegate 一致的回调方法。这也是 ModuleManager 做的事情。

先来看看现在的入口方法

```
- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)launchOptions
{
    [MGJApp startApp];

    [[ModuleManager sharedInstance] loadModuleFromPlist:[[NSBundle mainBundle] pathForResource:@"modules" ofType:@"plist"]];
    NSArray *modules = [[ModuleManager sharedInstance] allModules];
    for (id<ModuleProtocol> module in modules) {
        if ([module respondsToSelector:_cmd]) {
            [module application:application didFinishLaunchingWithOptions:launchOptions];
        }
    }

    [self trackLaunchTime];
    return YES;
}
```


其中 [MGJApp startApp] 主要负责一些 SDK 的初始化。[self trackLaunchTime] 是我们打的一个点，用来监测从 main 方法开始到入口方法调用结束花了多长时间。其他的都由 ModuleManager 搞定，loadModuleFromPlist:pathForResource: 方法会读取 bundle 里的一个 plist 文件，这个文件的内容大概是这样的

Root	Array	(40 items)
Item 0	String	ComponentManager
Item 1	String	MiscModule
Item 2	String	UISkeletonModule
Item 3	String	WatchModule
Item 4	String	URLHandlerModule
Item 5	String	MGJIndexModule

每个 Module 都实现了 ModuleProtocol，其中有一个 -(BOOL)applicaiton:didFinishLaunchingWithOptions: 方法，如果实现了的话，就会被调用。

还有一个问题就是，系统的一些事件会有通知，比如 applicationDidBecomeActive 会有对应的 UIApplicationDidBecomeActiveNotification，组件如果要做响应的话，只需监听这个系统通知即可。但也有一些事件是没有通知的，比如 -application:didRegisterUserNotificationSettings:，这时组件如果也要做点事情，怎么办?

一个简单的解决方法是在 AppDelegate 的各个方法里，手动调一遍组件的对应的方法，如果有就执行。

```
- (void)application:(UIApplication *)application didRegisterForRemoteNotification
sWithDeviceToken:(NSData *)deviceToken
{
    NSArray *modules = [[ModuleManager sharedInstance] allModules];
    for (id<ModuleProtocol> module in modules) {
        if ([module respondsToSelector:_cmd]) {
            [module application:application didRegisterForRemoteNotificationsWith
DeviceToken:deviceToken];
        }
    }
}
```


壳工程

既然已经拆出去了，那拆出去的组件总得有个载体，这个载体就是壳工程，壳工程主要包含一些基础组件和业务SDK，这也是主工程包含的一些内容，所以如果在壳工程可以正常运行的话，到了主工程也没什么问题。不过这里存在版本同步问题，之后会说到。

遇到的问题

组件拆分

由于之前的代码都是在一个工程下的，所以要单独拿出来作为一个组件就会遇到不少问题。首先是组件的划分，当时在定义组件粒度时也花了些时间讨论，究竟是粒度粗点好，还是细

点好。粗点的话比较有利于拆分，细点的话灵活度比较高。最终还是选择粗一点的粒度，先拆出来再说。

假如要把详情页迁出来，就会发现它依赖了一些其他部分的代码，那最快的方式就是直接把代码拷过来，改个名使用。比较简单暴力。说起来比较简单，做的时候也是挺有挑战的，因为正常的业务并不会因为「组件化」而停止，所以开发同学们需要同时兼顾正常的业务和组件的拆分。

版本管理

我们的组件包括第三方库都是通过 Cocoapods 来管理的，其中组件使用了私有库。之所以选择 Cocoapods，一个是因为它比较方便，还有就是用户基数比较大，且社区也比较活跃（活跃到了会时不时地触发 Github 的 rate limit，导致长时间 clone 不下来… 见此），当然也有其他的管理方式，比如 submodule / subtree，在开发人员比较多的情况下，方便、灵活的方案容易占上风，虽然它也有自己的问题。主要有版本同步和更新/编译慢的问题。

假如基础组件做了个 API 接口升级，这个升级会对原有的接口做改动，自然就会升一个中位的版本号，比如原先是 1.6.19，那么现在就变成 1.7.0 了。而我们在 Podfile 里都是用 ~ 指定的，这样就会出现主工程的 pod 版本升上去了，但是壳工程没有同步到，然后群里就会各种反馈编译不过，而且这个编译不过的长尾有时能拖上两三天。

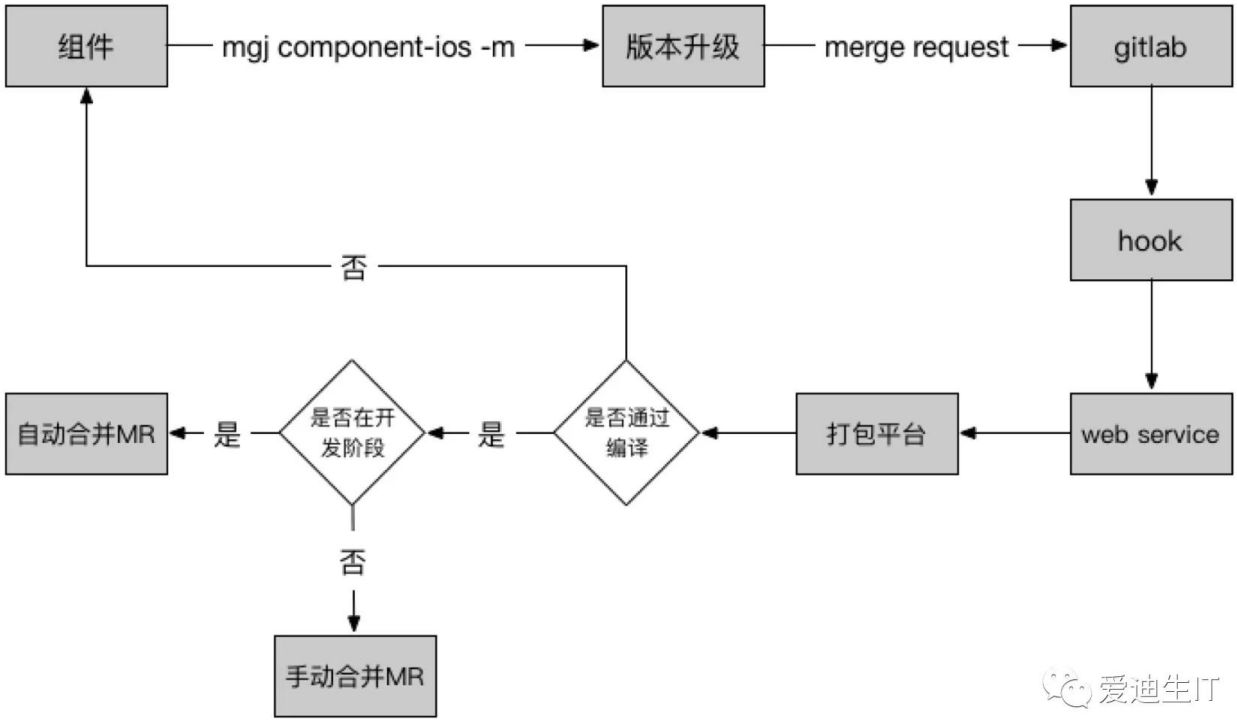
然后我们就想了个办法，如果不在壳工程里指定基础库的版本，只在主工程里指定呢，理论上应该可行，只要不出现某个基础库要同时维护多个版本的情况。但实践中发现，壳工程有时会莫名其妙地升不上去，在 podfile 里指定最新的版本又可以升上去，所以此路不通。

还有一个问题是 `pod update` 时间过长，经常会在 `Analyzing Dependency` 上卡 10 多分钟，非常影响效率。后来排查下来是跟组件的 Podspec 有关，配置了 subspec，且依赖比较多。

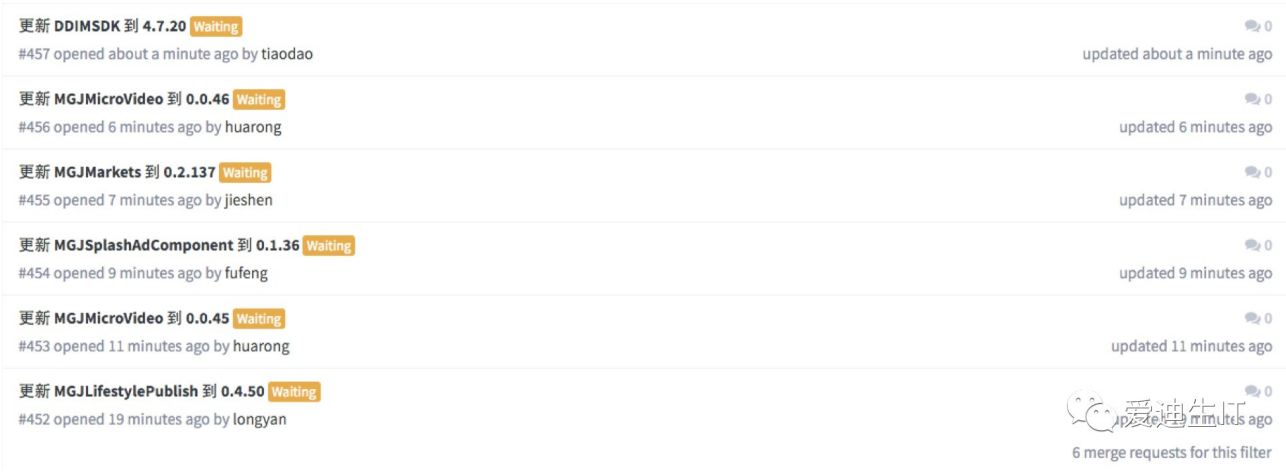
然后就是 pod update 之后的编译，由于是源码编译，所以这块的时间花费也不少，接下去会考虑 framework 的方式。

持续集成

在刚开始，持续集成还不是很完善，业务方升级组件，直接把 podspec 扔到 private repo 里就完事了。这样最简单，但也经常会带来编译通不过的问题。而且这种随意的版本升级也不太能保证质量。于是我们就搭建了一套持续集成系统，大概如此



每个组件升级之前都需要先通过编译，然后再决定是否升级。这套体系看起来不复杂，但在实施过程中经常会遇到后端的并发问题，导致业务方要么集成失败，要么要等不少时间。而且也没有一个地方可以呈现当前版本的组件版本信息。还有就是业务方对于这种命令行的升级方式接受度也不是很高。



基于此，在经过了几轮讨论之后，有了新版的持续集成平台，升级操作通过网页端来完成。

大致思路是，业务方如果要升级组件，假设现在的版本是 0.1.7，添加了一些 feature 之后，壳工程测试通过，想集成到主工程里看看效果，或者其他组件也想引用这个最新的，就可以在后台手动把版本升到 0.1.8-rc.1，这样的话，原先依赖 `~> 0.1.7` 的组件，不会升到 0.1.8，同时想要测试这个组件的话，只要手动把版本调到 0.1.8-rc.1 就可以了。这个过程不会触发 CI 的编译检查。

当测试通过后，就可以把尾部的 `-rc.n` 去掉，然后点击「集成」，就会走 CI 编译检查，通过的话，会在主工程的 podfile 里写上固定的版本号 0.1.8。也就是说，podfile 里所有的组件版本号都是固定的。

周边设施

基础组件及组件的文档 / Demo / 单元测试

无线基础的职能是为集团提供解决方案，只是在蘑菇街 App 里能 work 是远远不够的，所以就需要提供入口，知道有哪些可用组件，并且如何使用，就像这样（目前还未实现）

这就要求组件的负责人需要及时地更新 README / CHANGELOG / API，并且当发生 API 变更时，能够快速通知到使用方。

公共 UI 组件

组件化之后还有一个问题就是资源的重复性，以前在一个工程里的时候，资源都可以很方便地拿到，现在独立出去了，也不知道哪些是公用的，哪些是独有的，索性都放到自己的组件里，这样就会导致包变大。还有一个问题是每个组件可能是不同的产品经理在跟，而他们很可能只关注于自己关心的页面长什么样，而忽略了整体的样式。公共 UI 组件就是用来解决这些问题的，这些组件甚至可以跨 App 使用。（目前还未实现）

参考答案一：http://blog.csdn.net/GGGHub/article/details/52713642

参考答案二：http://limboy.me/tech/2016/03/10/mgj-components.html

3.runtime如何通过selector找到对应的IMP地址?

概述

类对象中有类方法和实例方法的列表，列表中记录着方法的名词、参数和实现，而selector本质就是方法名称，runtime通过这个方法名称就可以在列表中找到该方法对应的实现。
这里声明了一个指向struct objc_method_list指针的指针，可以包含类方法列表和实例方法列表

具体实现

在寻找IMP的地址时，runtime提供了两种方法

```
IMP class_getMethodImplementation(Class cls, SEL name);IMP method_getImplementation(Method m)
```

而根据官方描述，第一种方法可能会更快一些

@note \c class_getMethodImplementation may be faster than \c method_getImplementation(class_getInstanceMethod(cls, name)).

对于第一种方法而言，类方法和实例方法实际上都是通过调用class_getMethodImplementation()来寻找IMP地址的，不同之处在于传入的第一个参数不同

类方法（假设有一个类A）

```
class_getMethodImplementation(objc_getMetaClass("A"),@selector(methodName));
```

实例方法

```
class_getMethodImplementation([A class],@selector(methodName));
```

通过该传入的参数不同，找到不同的方法列表，方法列表中保存着下面方法的结构体，结构体中包含这方法的实现，selector本质就是方法的名称，通过该方法名称，即可在结构体中找到相应的实现。

```
struct objc_method {SEL method_namechar *method_typesIMP method_imp}
```

而对于第二种方法而言，传入的参数只有method，区分类方法和实例方法在于封装method的函数

类方法

```
Method class_getClassMethod(Class cls, SEL name
)
```

实例方法

```
Method class_getInstanceMethod(Class cls, SEL name
)
```

最后调用`IMP method_getImplementation(Method m)`获取IMP地址

实验

```
@implementation Test

- (instancetype)init
{
    self = [super init];
    if (self) {
        [self getIMPFromSelector:@selector(aaa)];
        [self getIMPFromSelector:@selector(test1)];
        [self getIMPFromSelector:@selector(test2)];
    }
    return self;
}

- (void)test1 {
    NSLog(@"test1");
}

+ (void)test2 {
    NSLog(@"test2");
}

- (void)getIMPFromSelector:(SEL)aSelector {

    // first method
    IMP instanceIMP1 = class_getMethodImplementation(objc_getClass("Test"), aSelector);
    IMP classIMP1 = class_getMethodImplementation(objc_getMetaClass("Test"), aSelector);

    // second method
    Method instanceMethod = class_getInstanceMethod(objc_getClass("Test"), aSelector);
    IMP instanceIMP2 = method_getImplementation(instanceMethod);

    Method classMethod1 = class_getClassMethod(objc_getClass("Test"), aSelector);
    IMP classIMP2 = method_getImplementation(classMethod1);

    Method classMethod2 = class_getClassMethod(objc_getMetaClass("Test"), aSelector);
    IMP classIMP3 = method_getImplementation(classMethod2);

    NSLog(@"instance1:%p instance2:%p class1:%p class2:%p class3:%p", instanceIMP1, instanceIMP2, classIMP1,
}

@end
```


这里有一个叫Test的类，在初始化方法里，调用了两次getIMPFromSelector:方法，第一个aaa方法是不存在的，test1和test2分别为实例方法和类方法

```
- (void)viewDidLoad {
    [super viewDidLoad];
    Test *test1 = [[Test alloc] init];
    Test *test2 = [[Test alloc] init];
}
```


然后我同时实例化了两个Test的对象，打印信息如下

```
2016-09-15 11:16:11.092 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.092 GetIMPFromSelector[12399:659621] instance1:0x10fdc0720 instance2:0x10fdc0720 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x10fdc0750 class2:0x10fdc0750 class3:0x10fdc0750
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.093 GetIMPFromSelector[12399:659621] instance1:0x10fdc0720 instance2:0x10fdc0720 class1:0x1102db280 class2:0x0 class3:0x0
2016-09-15 11:16:11.094 GetIMPFromSelector[12399:659621] instance1:0x1102db280 instance2:0x0 class1:0x10fdc0750 class2:0x10fdc0750 class3:0x10fdc0750
```


大家注意图中红色标注的地址出现了8次：0x1102db280，这个是在调用class_getMethodImplementation()方法时，无法找到对应实现时返回的相同的一个地址，无论该方法是在实例方法或类方法，无论是否对一个实例调用该方法，返回的地址都是相同的，但是每次运行该程序时返回的地址并不相同，而对于另一种方法，如果找不到对应的实现，则返回0，在图中我做了蓝色标记。

还有一点有趣的是class_getClassMethod()的第一个参数无论传入objc_getClass()还是objc_getMetaClass()，最终调用method_getImplementation()都可以成功的找到类方法的实现。而class_getInstanceMethod()的第一个参数如果传入objc_getMetaClass()，再调用method_getImplementation()时无法找到实例方法的实现却可以找到类方法的实现。

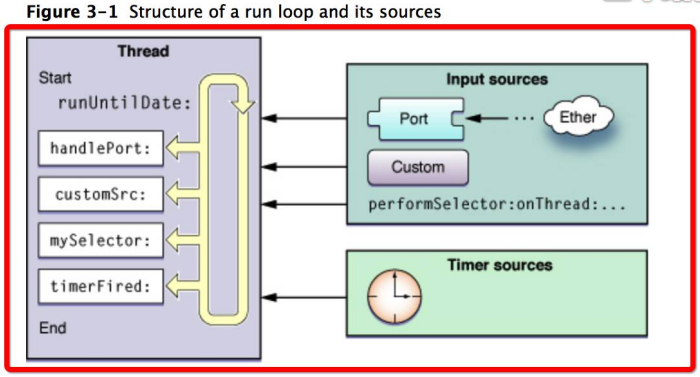
4.runloop内部实现逻辑?

A run loop receives events from two different types of sources. *Input sources* deliver asynchronous events, usually messages from another thread or from a different application. *Timer sources* deliver synchronous events, occurring at a scheduled time or repeating interval. Both types of source use an application-specific handler routine to process the event when it arrives.

Figure 3-1 shows the conceptual structure of a run loop and a variety of sources. The input sources deliver asynchronous events to the corresponding handlers and cause the `runUntilDate:` method (called on the thread's associated `NSRunLoop` object) to exit. Timer sources deliver events to their handler routines but do not cause the run loop to exit.

基本结构

Figure 3-1 Structure of a run loop and its sources



苹果在文档里的说明，RunLoop 内部的逻辑大致如下:

其内部代码整理如下：

可以看到，实际上 RunLoop 就是这样一个函数，其内部是一个 do-while 循环。当你调用 CFRunLoopRun() 时，线程就会一直停留在这个循环里；直到超时或被手动停止，该函数才会返回。

RunLoop 的底层实现

从上面代码可以看到，RunLoop 的核心是基于 mach port 的，其进入休眠时调用的函数是 mach_msg()。为了解释这个逻辑，下面稍微介绍一下 OSX/iOS 的系统架构。

苹果官方将整个系统大致划分为上述4个层次：

1. 应用层包括用户能接触到的图形应用，例如 Spotlight、Aqua、SpringBoard 等。
2. 应用框架层即开发人员接触到的 Cocoa 等框架。
3. 核心框架层包括各种核心框架、OpenGL 等内容。
4. Darwin 即操作系统的核心，包括系统内核、驱动、Shell 等内容，这一层是开源的，其所有源码都可以在 opensource.apple.com 里找到。

我们在深入看一下 Darwin 这个核心的架构：

其中，在硬件层上面的三个组成部分：Mach、BSD、IOKit(还包括一些上面没标注的内容)，共同组成了 XNU 内核。

XNU 内核的内环被称作 Mach，其作为一个微内核，仅提供了诸如处理器调度、IPC (进程间通信)等非常少量的基础服务。

BSD 层可以看作围绕 Mach 层的一个外环，其提供了诸如进程管理、文件系统和网络等功能。

IOKit 层是为设备驱动提供了一个面向对象(C++)的一个框架。

Mach 本身提供的 API 非常有限，而且苹果也不鼓励使用 Mach 的 API，但是这些API非常基础，如果没有这些API的话，其他任何工作都无法实施。在 Mach 中，所有的东西都是通过自己的对象实现的，进程、线程和虚拟内存都被称为"对象"。和其他架构不同， Mach 的对象间不能直接调用，只能通过消息传递的方式实现对象间的通信。"消息"是 Mach 中最基础的概念，消息在两个端口 (port) 之间传递，这就是 Mach 的 IPC (进程间通信) 的核心。

Mach 的消息定义是在头文件的，很简单：

```
typedef struct {
  mach_msg_header_t header;
  mach_msg_body_t body;
} mach_msg_base_t;

typedef struct {
  mach_msg_bits_t msgh_bits;
  mach_msg_size_t msgh_size;
  mach_port_t msgh_remote_port;
  mach_port_t msgh_local_port;
  mach_port_name_t msgh_voucher_port;
  mach_msg_id_t msgh_id;
} mach_msg_header_t;
```

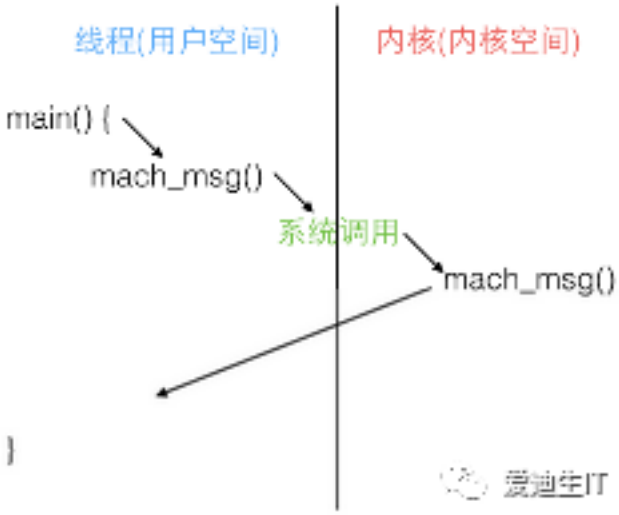
一条 Mach 消息实际上就是一个二进制数据包 (BLOB)，其头部定义了当前端口 local_port 和目标端口 remote_port，

发送和接受消息是通过同一个 API 进行的，其 option 标记了消息传递的方向：

```
mach_msg_return_t mach_msg(
mach_msg_header_t *msg,
mach_msg_option_t option,
mach_msg_size_t send_size,
mach_msg_size_t rcv_size,
mach_port_name_t rcv_name,
mach_msg_timeout_t timeout,
mach_port_name_t notify);
```

为了实现消息的发送和接收，mach_msg() 函数实际上是调用了一个 Mach 陷阱 (trap)，即函数mach_msg_trap()，陷阱这个概念在 Mach 中等同于系统调用。当你在用户态调用

mach_msg_trap() 时会触发陷阱机制，切换到内核态；内核态中内核实现的 mach_msg() 函数会完成实际的工作，如下图：



这些概念可以参考维基百科: System_call、Trap_(computing)。

RunLoop 的核心就是一个 mach_msg() (见上面代码的第7步)，RunLoop 调用这个函数去接收消息，如果没有别人发送 port 消息过来，内核会将线程置于等待状态。例如你在模拟器里跑起一个 iOS 的 App，然后在 App 静止时点击暂停，你会看到主线程调用栈是停留在 mach_msg_trap() 这个地方。

关于具体的如何利用 mach port 发送信息，可以看看 NSHipster 这一篇文章，或者这里的中文翻译 。

关于Mach的历史可以看看这篇很有趣的文章：Mac OS X 背后的故事（三）Mach 之父 Avie Tevanian。

苹果用 **RunLoop** 实现的功能

首先我们可以看一下 App 启动后 RunLoop 的状态：

可以看到，系统默认注册了5个Mode:

1. kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。

2. UITrackingRunLoopMode: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。

3. UIInitializationRunLoopMode: 在刚启动 App 时第进入的第一个 Mode，启动完成后就不再使用。

4: GSEventReceiveRunLoopMode: 接受系统事件的内部 Mode，通常用不到。

5: kCFRunLoopCommonModes: 这是一个占位的 Mode，没有实际作用。

你可以在这里看到更多的苹果内部的 Mode，但那些 Mode 在开发中就很难遇到了。

5.你理解的多线程?

1.可能会追问，每种多线程基于什么语言?

2.生命周期是如何管理?

3.你更倾向于哪种？追问至现在常用的两种你的看法是?

第一种：pthread

.特点：

1）一套通用的多线程API

2）适用于Unix\Linux\Windows等系统

3）跨平台\可移植

4）使用难度大

b.使用语言：c语言

c.使用频率：几乎不用

d.线程生命周期：由程序员进行管理

第二种：NSThread

a.特点：

1）使用更加面向对象

2）简单易用，可直接操作线程对象

b.使用语言：OC语言

c.使用频率：偶尔使用

d.线程生命周期：由程序员进行管理

第三种：GCD

a.特点：

1）旨在替代NSThread等线程技术

2）充分利用设备的多核（自动）

b.使用语言：C语言

c.使用频率：经常使用

d.线程生命周期：自动管理

第四种：NSOperation

a.特点：

1）基于GCD（底层是GCD）

2）比GCD多了一些更简单实用的功能

3）使用更加面向对象

b.使用语言：OC语言

c.使用频率：经常使用

d.线程生命周期：自动管理

多线程的原理

同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）

多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）

如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象

思考：如果线程非常非常多，会发生什么情况?

CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源

每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优点

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

多线程的缺点

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

你更倾向于哪一种?

倾向于GCD:

GCD技术是一个轻量的，底层实现隐藏的神奇技术，我们能够通过GCD和block轻松实现多线程编程，有时候，GCD相比其他系统提供的多线程方法更加有效，当然，有时候GCD不是最佳选择，另一个多线程编程的技术NSOprationQueue 让我们能够将后台线程以队列方式依序执行，并提供更多操作的入口，这和 GCD 的实现有些类似。

这种类似不是一个巧合，在早期，MacOX 与 iOS 的程序都普遍采用Operation Queue来进行编写后台线程代码，而之后出现的GCD技术大体是依照前者的原则来实现的，而随着GCD的普及，在iOS 4 与 MacOS X 10.6以后，Operation Queue的底层实现都是用GCD来实现的。

那这两者直接有什么区别呢?

1. GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2. 在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3. NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4. 我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5. 在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6. 我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。

总的来说，Operation queue 提供了更多你在编写多线程程序时需要的功能，并隐藏了许多线程调度，线程取消与线程优先级的复杂代码，为我们提供简单的API入口。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。但是我认为当我们的需求能够以更简单的底层代码完成的时候，简洁的GCD或许是个更好的选择，而Operation queue为我们提供能更多的选择。

倾向于：**NSOperation**

NSOperation相对于GCD：

1，NSOperation拥有更多的函数可用，具体查看api。NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

2，在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系。

3，NSOperationQueue支持KVO。可以监测operation是否正在执行（isExecuted）、是否结束（isFinished），是否取消（isCanceld）

4，GCD只支持FIFO的队列，而NSOperationQueue可以调整队列的执行顺序（通过调整权重）。NSOperationQueue可以方便的管理并发、NSOperation之间的优先级。

使用NSOperation的情况：各个操作之间有依赖关系、操作需要取消暂停、并发管理、控制操作之间优先级，限制同时能执行的线程数量.让线程在某时刻停止/继续等。

使用GCD的情况：一般的需求很简单的多线程操作，用GCD都可以了，简单高效。

从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。

当需求简单，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

5.你理解的多线程?

1.可能会追问，每种多线程基于什么语言?

2.生命周期是如何管理?

3.你更倾向于哪种？追问至现在常用的两种你的看法是?

第一种：pthread

.特点:

1）一套通用的多线程API

2）适用于Unix\Linux\Windows等系统

3）跨平台\可移植

4）使用难度大

b.使用语言：c语言

c.使用频率：几乎不用

d.线程生命周期：由程序员进行管理

第二种：NSThread

a.特点:

1）使用更加面向对象

2）简单易用，可直接操作线程对象

b.使用语言：OC语言

c.使用频率：偶尔使用

d.线程生命周期：由程序员进行管理

第三种：GCD

a.特点：

1）旨在替代NSThread等线程技术

2）充分利用设备的多核（自动）

b.使用语言：C语言

c.使用频率：经常使用

d.线程生命周期：自动管理

第四种：NSOperation

a.特点：

1）基于GCD（底层是GCD）

2）比GCD多了一些更简单实用的功能

3）使用更加面向对象

b.使用语言：OC语言

c.使用频率：经常使用

d.线程生命周期：自动管理

多线程的原理

同一时间，CPU只能处理1条线程，只有1条线程在工作（执行）

多线程并发（同时）执行，其实是CPU快速地在多条线程之间调度（切换）

如果CPU调度线程的时间足够快，就造成了多线程并发执行的假象

思考：如果线程非常非常多，会发生什么情况?

CPU会在N多线程之间调度，CPU会累死，消耗大量的CPU资源

每条线程被调度执行的频次会降低（线程的执行效率降低）

多线程的优点

能适当提高程序的执行效率

能适当提高资源利用率（CPU、内存利用率）

多线程的缺点

开启线程需要占用一定的内存空间（默认情况下，主线程占用1M，子线程占用512KB），如果开启大量的线程，会占用大量的内存空间，降低程序的性能

线程越多，CPU在调度线程上的开销就越大

程序设计更加复杂：比如线程之间的通信、多线程的数据共享

你更倾向于哪一种?

倾向于GCD:

GCD技术是一个轻量的，底层实现隐藏的神奇技术，我们能够通过GCD和block轻松实现多线程编程，有时候，GCD相比其他系统提供的多线程方法更加有效，当然，有时候GCD不是最佳选择，另一个多线程编程的技术NSOprationQueue 让我们能够将后台线程以队列方式依序执行，并提供更多操作的入口，这和 GCD 的实现有些类似。

这种类似不是一个巧合，在早期，MacOX 与 iOS 的程序都普遍采用Operation Queue来进行编写后台线程代码，而之后出现的GCD技术大体是依照前者的原

则来实现的，而随着GCD的普及，在iOS 4 与 MacOS X 10.6以后，Operation Queue的底层实现都是用GCD来实现的。

那这两者直接有什么区别呢?

1. GCD是底层的C语言构成的API，而NSOperationQueue及相关对象是Objc的对象。在GCD中，在队列中执行的是由block构成的任务，这是一个轻量级的数据结构；而Operation作为一个对象，为我们提供了更多的选择；

2. 在NSOperationQueue中，我们可以随时取消已经设定要准备执行的任务(当然，已经开始的任务就无法阻止了)，而GCD没法停止已经加入queue的block(其实是有的，但需要许多复杂的代码)；

3. NSOperation能够方便地设置依赖关系，我们可以让一个Operation依赖于另一个Operation，这样的话尽管两个Operation处于同一个并行队列中，但前者会直到后者执行完毕后再执行；

4. 我们能将KVO应用在NSOperation中，可以监听一个Operation是否完成或取消，这样子能比GCD更加有效地掌控我们执行的后台任务；

5. 在NSOperation中，我们能够设置NSOperation的priority优先级，能够使同一个并行队列中的任务区分先后地执行，而在GCD中，我们只能区分不同任务队列的优先级，如果要区分block任务的优先级，也需要大量的复杂代码；

6. 我们能够对NSOperation进行继承，在这之上添加成员变量与成员方法，提高整个代码的复用度，这比简单地将block任务排入执行队列更有自由度，能够在其之上添加更多自定制的功能。

总的来说，Operation queue 提供了更多你在编写多线程程序时需要的功能，并隐藏了许多线程调度，线程取消与线程优先级的复杂代码，为我们提供简单的API入口。从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。但是我认为当我们的需求能够以更简单的底层代码完成的时候，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

倾向于：**NSOperation**

NSOperation相对于GCD：

1，NSOperation拥有更多的函数可用，具体查看api。NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

2，在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系。

3，NSOperationQueue支持KVO。可以监测operation是否正在执行（isExecuted）、是否结束（isFinished），是否取消（isCanceld）

4，GCD只支持FIFO的队列，而NSOperationQueue可以调整队列的执行顺序（通过调整权重）。NSOperationQueue可以方便的管理并发、NSOperation之间的优先级。

使用NSOperation的情况：各个操作之间有依赖关系、操作需要取消暂停、并发管理、控制操作之间优先级，限制同时能执行的线程数量.让线程在某时刻停止/继续等。

使用GCD的情况：一般的需求很简单的多线程操作，用GCD都可以了，简单高效。

从编程原则来说，一般我们需要尽可能的使用高等级、封装完美的API，在必须时才使用底层API。

当需求简单，简洁的GCD或许是个更好的选择，而Operation queue 为我们提供能更多的选择。

6.GCD执行原理?

GCD有一个底层线程池，这个池中存放的是一个个的线程。之所以称为“池”，

很容易理解出这个“池”中的线程是可以重用的，当一段时间后这个线程没有被调用胡话，这个线程就会被销毁。注意：开多少条线程是由底层线程池决定的（线程建议控制再3~5条），池是系统自动来维护，不需要我们程序员来维护（看到这句话是不是很开心？）
而我们程序员需要关心的是什么呢？我们只关心的是向队列中添加任务，队列调度即可。

• 如果队列中存放的是同步任务，则任务出队后，底层线程池中会提供一条线程供这个任务执行，任务执行完毕后这条线程再回到线程池。这样队列中的任务反复调度，因为是同步的，所以当我们用currentThread打印的时候，就是同一条线程。

• 如果队列中存放的是异步的任务，（注意异步可以开线程），当任务出队后，底层线程池会提供一个线程供任务执行，因为是异步执行，队列中的任务不需等待当前任务执行完毕就可以调度下一个任务，这时底层线程池中会再次提供一个线程供第二个任务执行，执行完毕后再回到底层线程池中。

• 这样就对线程完成一个复用，而不需要每一个任务执行都开启新的线程，也就从而节约的系统的开销，提高了效率。在iOS7.0的时候，使用GCD系统通常只能开5~8条线程，iOS8.0以后，系统可以开启很多条线程，但是实在开发应用中，建议开启线程条数：3~5条最为合理。

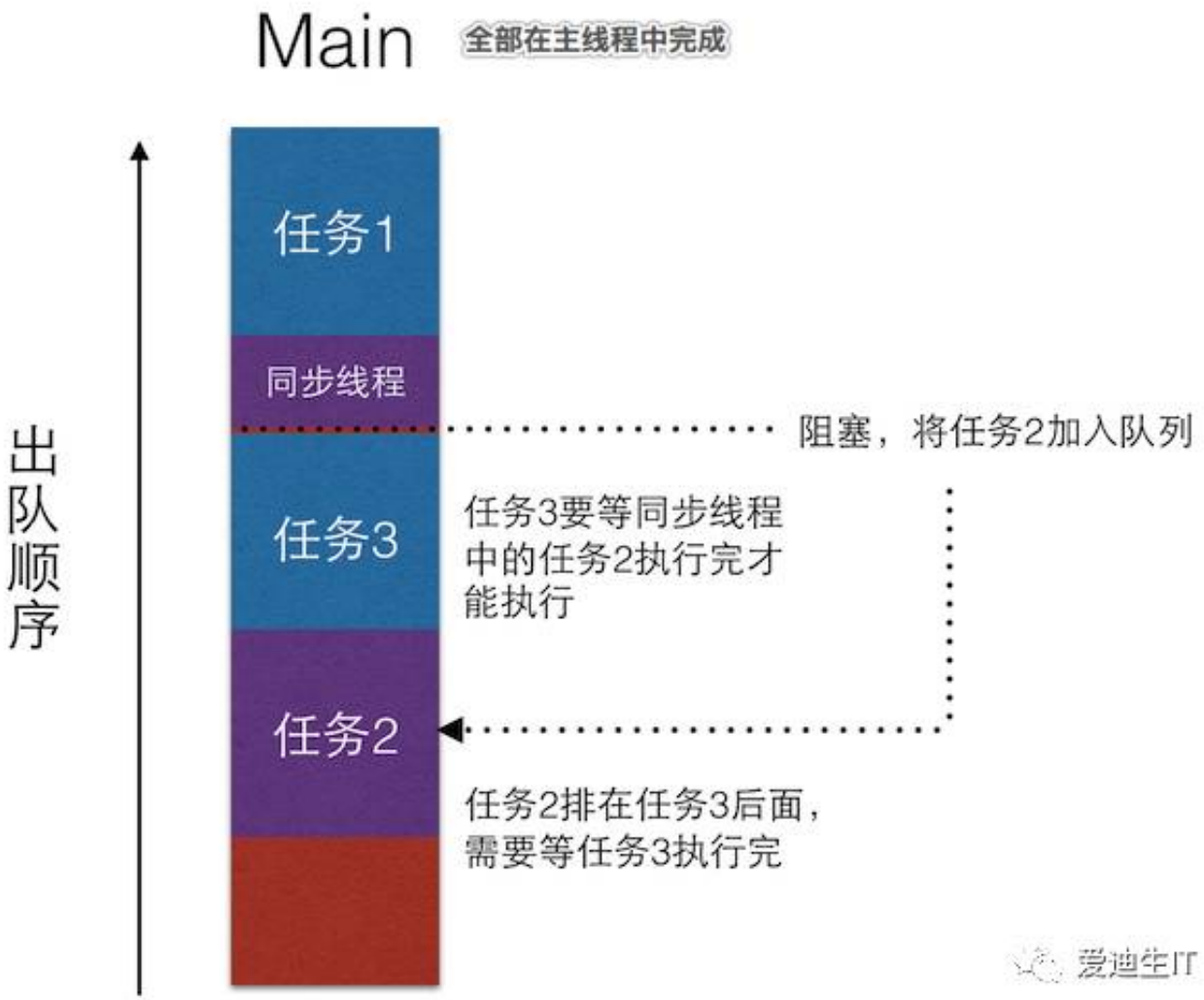
通过案例明白GCD的执行原理

案例一：

```
NSLog(@"1"); // 任务1

dispatch_sync(dispatch_get_main_queue(), ^{

    NSLog(@"2"); // 任务2
});

NSLog(@"3"); // 任务3

输出结果：
1
```

分析：
首先执行任务1，这是肯定没问题的，只是接下来，程序遇到了同步线程，那么它会进入等待，等待任务2执行完，然后执行任务3。但这是队列，有任务来，当然会将任务加到队尾，然后遵循FIFO原则执行任务。那么，现在任务2就会被加到最后，任务3排在了任务2前面，问题来了：

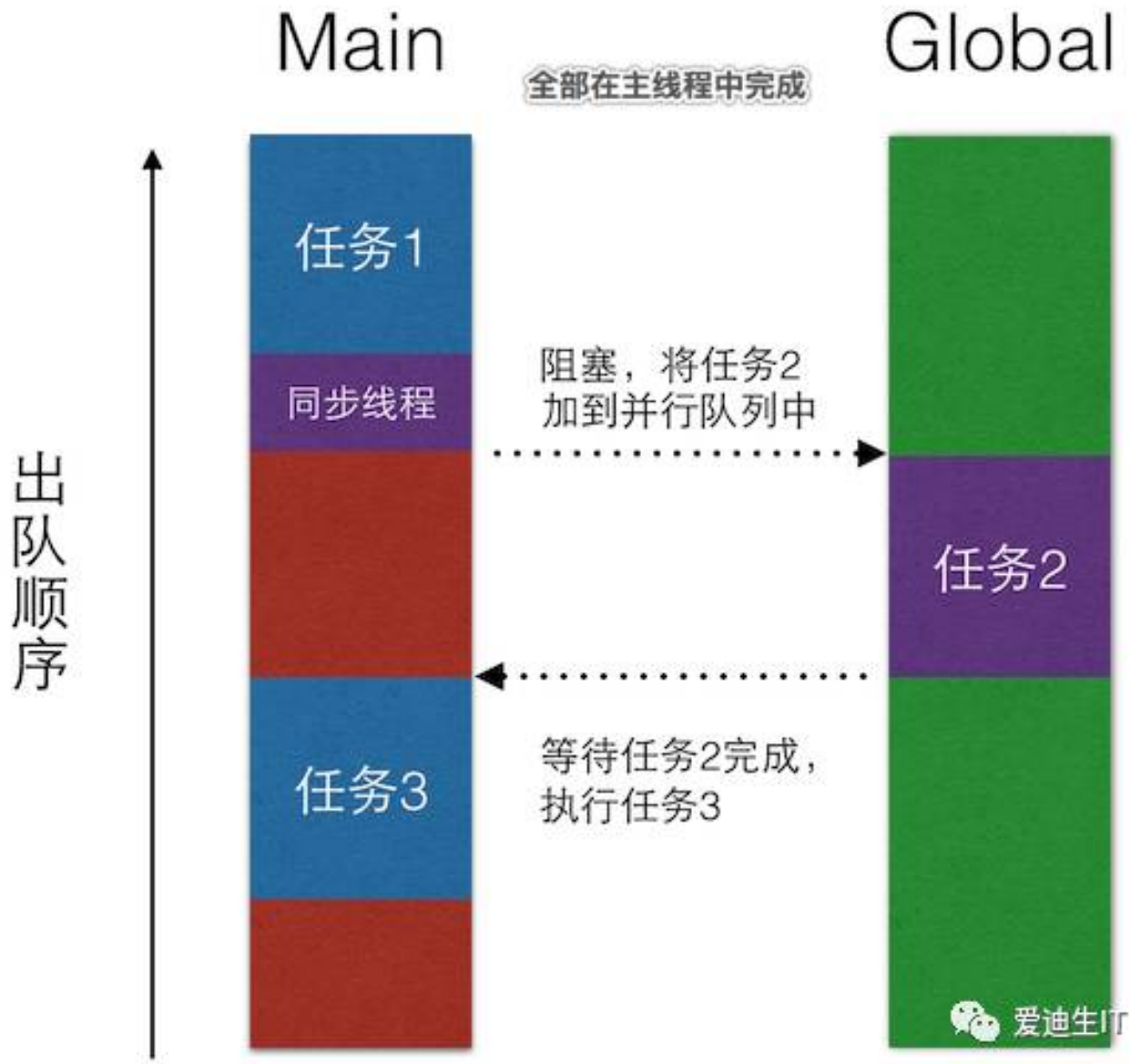
任务3要等任务2执行完才能执行，任务2又排在任务3后面，意味着任务2要在任务3执行完才能执行，所以他们进入了互相等待的局面。【既然这样，那干脆就卡在这里吧】这就是死锁。



案例二：

```
NSLog(@"1"); // 任务1

dispatch_sync(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0), ^{

    NSLog(@"2"); // 任务2
});

NSLog(@"3"); // 任务3

输出结果：
1
2
3
```


分析：
首先执行任务1，接下来会遇到一个同步线程，程序会进入等待。等待任务2执行完成以后，才能继续执行任务3。从`dispatch_get_global_queue`可以看出，任务2被加入到了全局的并行队列中，当并行队列执行完任务2以后，返回到主队列，继续执行任务3。


Main
Global
全部在主线程中完成
任务1
同步线程
阻塞，将任务2
加到并行队列中
任务2
任务3
等待任务2完成，
执行任务3
出队顺序
爱迪生IT

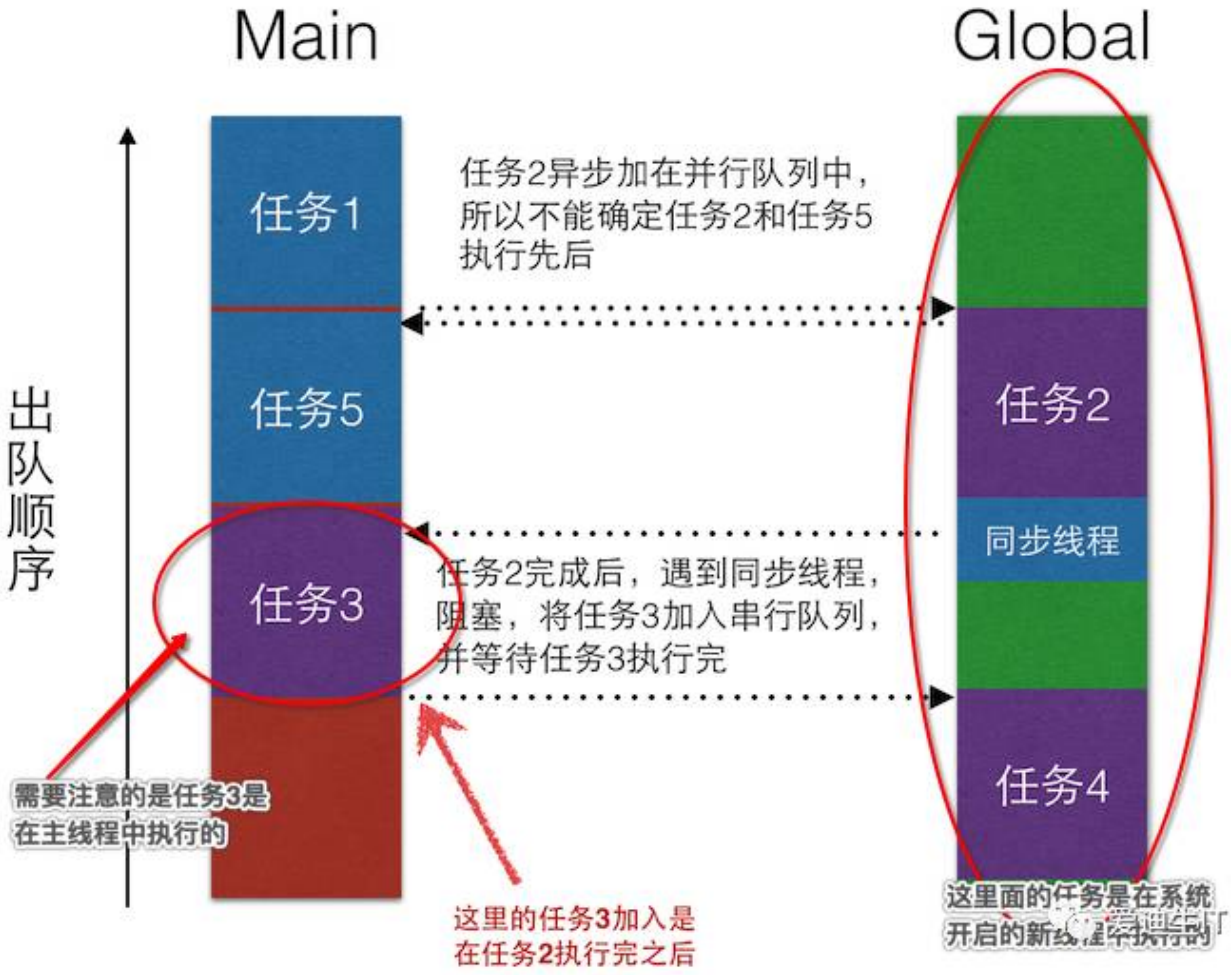
案例三：

```
dispatch_queue_t queue = dispatch_queue_create("com.demo.serialQueue", DISPATCH_Q
UEUE_SERIAL);

NSLog(@"1"); // 任务1

dispatch_async(queue, ^{

    NSLog(@"2"); // 任务2

    dispatch_sync(queue, ^{

        NSLog(@"3"); // 任务3

    });

    NSLog(@"4"); // 任务4

});

NSLog(@"5"); // 任务5(会在主线程中执行)

输出结果:
1
5
2
// 5和2的顺序不一定
```


案例四:

```
NSLog(@"1"); // 任务1

dispatch_async(dispatch_get_global_queue(0, 0), ^{

    NSLog(@"2"); // 任务2

    dispatch_sync(dispatch_get_main_queue(), ^{

        NSLog(@"3"); // 任务3

    });

    NSLog(@"4"); // 任务4

});

NSLog(@"5"); // 任务5

输出结果:
1
2
5
3
4
// 5和2的顺序不一定
```

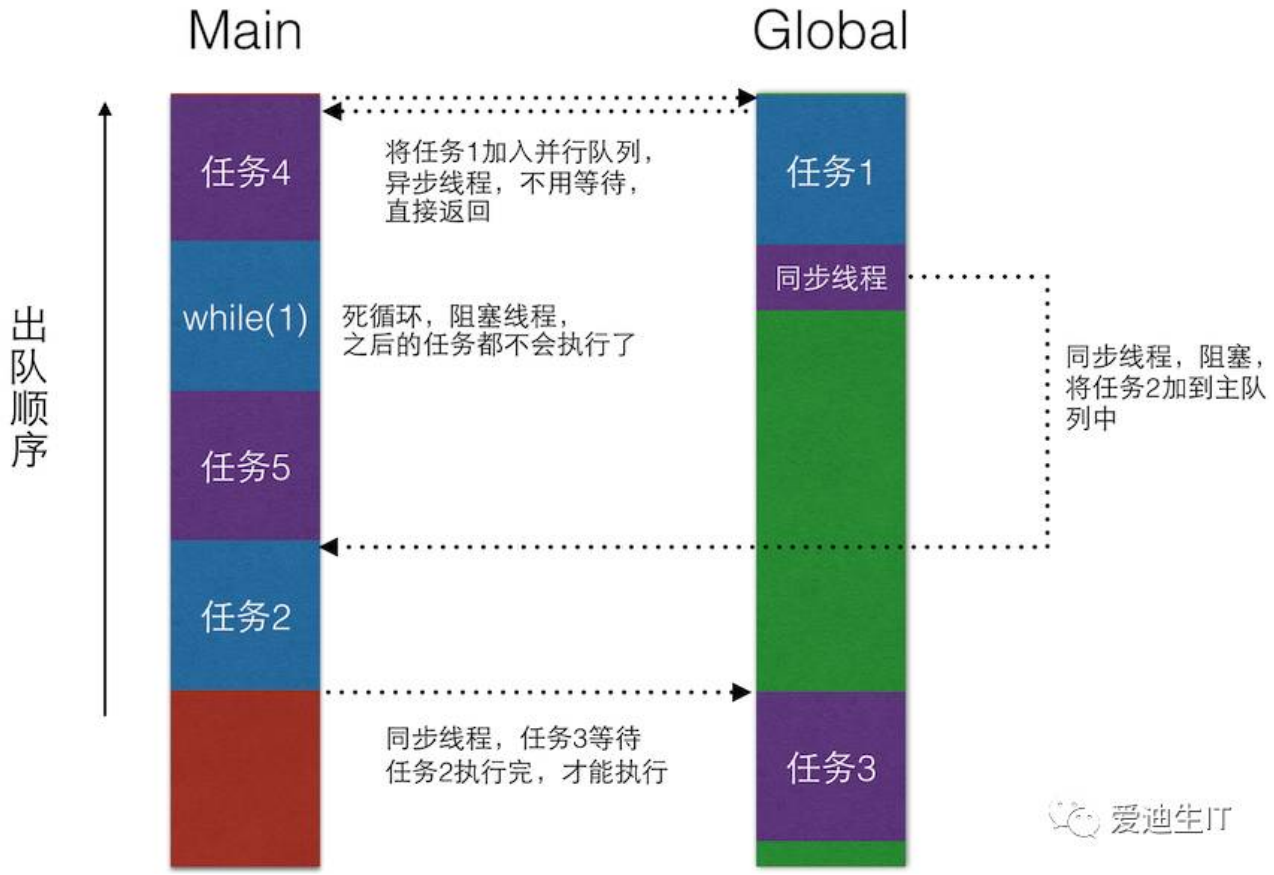

分析:
首先，将【任务1、异步线程、任务5】加入MainQueue中，异步线程中的任务是：【任务2、同步线程、任务4】。所以，先执行任务1，然后将异步线程中的任务加入到GlobalQueue中，因为异步线程，所以任务5不用等待，结果就是2和5的输出顺序不一定。然后再看异步线程中的任务执行顺序。任务2执行完以后，遇到同步线程。将同步线程中的任务加入到Main Queue中，这时加入的任务3在任务5的后面。当任务3执行完以后，没有了阻塞，程序继续执行任务4。


Main
Global
任务1
任务5
任务3
任务2异步加在并行队列中，
所以不能确定任务2和任务5
执行先后
任务2
同步线程
任务2完成后，遇到同步线程，
阻塞，将任务3加入串行队列，
并等待任务3执行完
任务4
出队顺序
需要注意的是任务3是
在主线程中执行的
这里的任务3加入是
在任务2执行完之后
这里面的任务是在系统

案例五：

```
dispatch_async(dispatch_get_global_queue(0, 0), ^{

    NSLog(@"1"); // 任务1

    dispatch_sync(dispatch_get_main_queue(), ^{

        NSLog(@"2"); // 任务2

    });

    NSLog(@"3"); // 任务3

});

NSLog(@"4"); // 任务4

while (1) {

}

NSLog(@"5"); // 任务5

输出结果:
1
4
// 1和4的顺序不一定
```


分析:
和上面几个案例的分析类似，先来看看都有哪些任务加入了Main Queue:
【异步线程、任务4、死循环、任务5】。
在加入到Global Queue异步线程中的任务有:
【任务1、同步线程、任务3】。第一个就是异步线程，任务4不用等待，
所以结果任务1和任务4顺序不一定。任务4完成后，程序进入死循环，
Main Queue阻塞。但是加入到Global Queue的异步线程不受影响，
继续执行任务1后面的同步线程。同步线程中，将任务2加入到了主线程，
并且，任务3等待任务2完成以后才能执行。这时的主线程，已经被死循环阻塞了。
所以任务2无法执行，当然任务3也无法执行，在死循环后的任务5也不会执行。



7.怎么防止别人动态在你程序生成代码?

（这题是听错了面试官的意思）

面试官意思是怎么防止别人反编译你的app?

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

6.借助第三方APP加固，例如：网易云易盾

8.YYAsyncLayer如何异步绘制?

YYAsyncLayer是异步绘制与显示的工具。为了保证列表滚动流畅，将视图绘制、以及图片解码等任务放到后台线程，

YYKitDemo

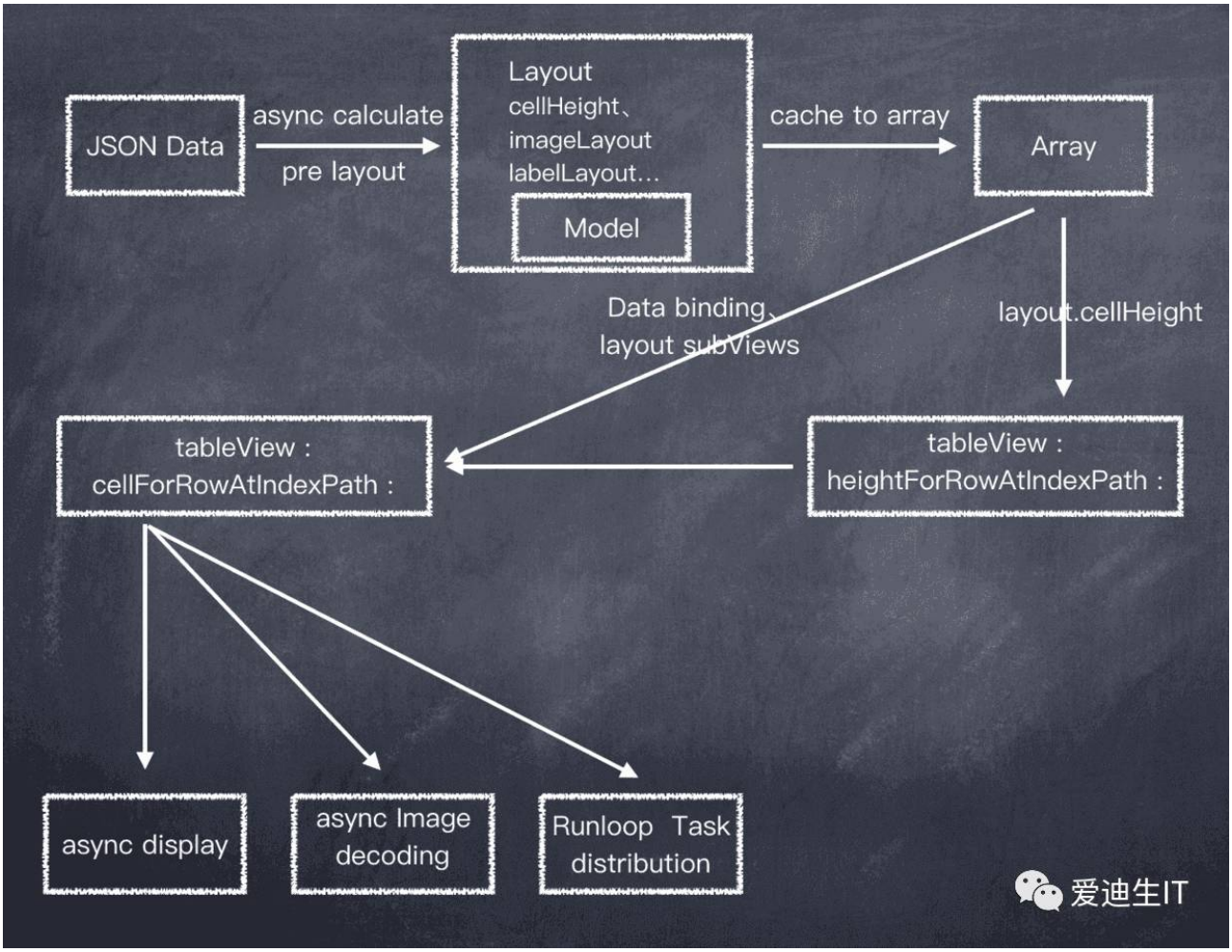
对于列表主要对两个代理方法的优化，一个与绘制显示有关，另一个与计算布局有关：

Objective-C

```
- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:
                                (NSIndexPath *)indexPath;

- (CGFloat)tableView:(UITableView *)tableView heightForRowAtIndexPath:
                                (NSIndexPath *)indexPath;
```

常规逻辑可能觉得应该先调用`tableView : cellForRowAtIndexPath :`返回`UITableViewCell`对象，事实上调

用顺序是先返回UITableViewCell的高度，是因为UITableView继承自UIScrollView，滑动范围由属性contentSize来确定，UITableView的滑动范围需要通过每一行的UITableViewCell的高度计算确定，复杂cell如果在列表滚动过程中计算可能会造成一定程度的卡顿。
假设有20条数据，当前屏幕显示5条，tableView : heightForRowAtIndexPath :方法会先执行20次返回所有高度并计算出滑动范围，tableView : cellForRowAtIndexPath :执行 5 次返回当前屏幕显示的cell个数。



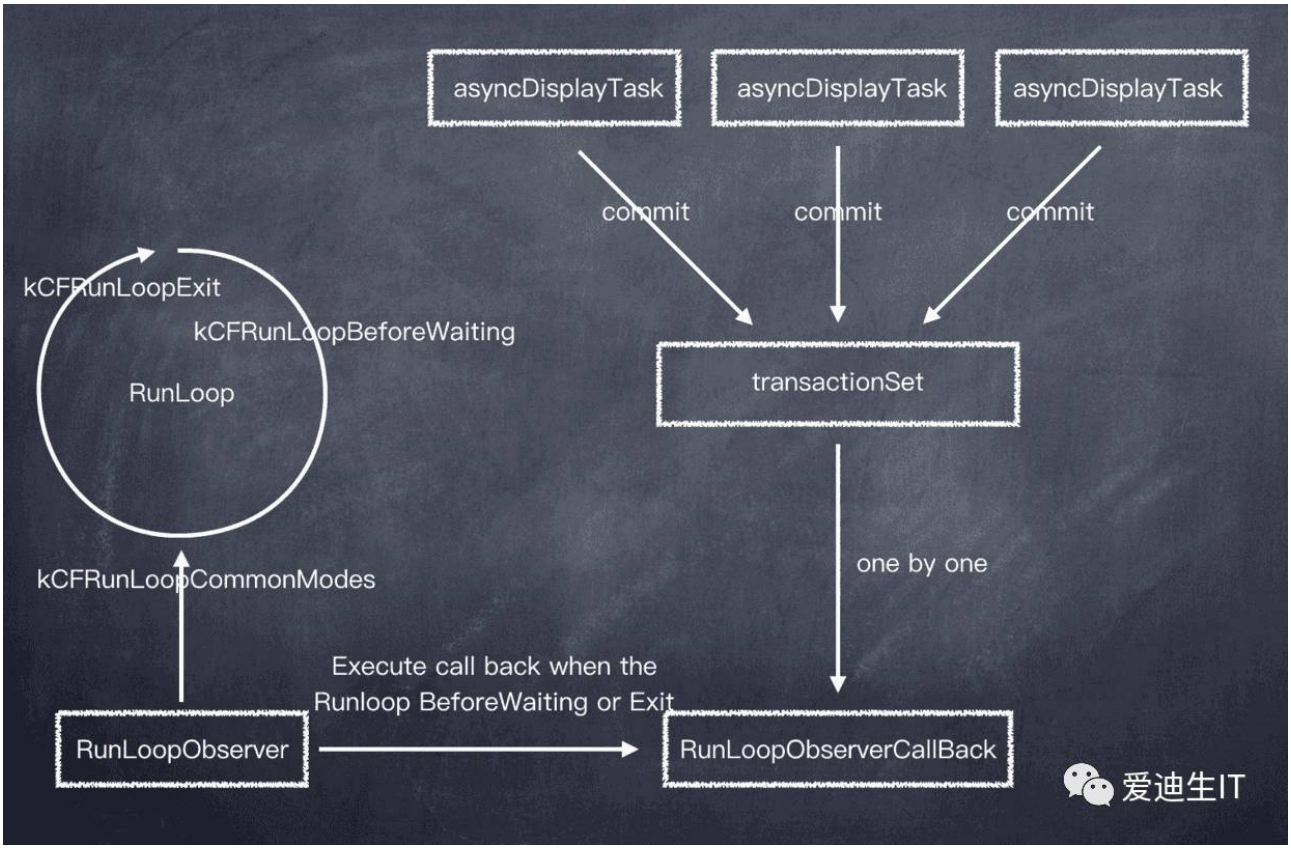
从图中简单看下流程，从网络请求返回JSON数据，将Cell的高度以及内部视图的布局封装为Layout对象，Cell显示之前在异步线程计算好所有布局对象，并存入数组，每次调用tableView: heightForRowAtIndexPath :只需要从数组中取出，可避免重复的布局计算。同时在调用tableView: cellForRowAtIndexPath :对Cell

内部视图异步绘制布局，以及图片的异步绘制解码，这里就要说到今天的主角YYAsyncLayer。

YYAsyncLayer

首先介绍里面几个类：

- YYAsyncLayer：继承自CALayer，绘制、创建绘制线程的部分都在这个类。
- YYTransaction：用于创建RunloopObserver监听MainRunloop的空闲时间，并将YYTranaction对象存放到集合中。
- YYSentinel：提供获取当前值的`value`（只读）属性，以及`-(int32_t)increase`自增加的方法返回一个新的`value`值，用于判断异步绘制任务是否被取消的工具。



AsyncDisplay.png

上图是整体异步绘制的实现思路，后面一步步说明。现在假设需要绘制Label，其实是继承自UIView，重写`+ (Class)layerClass`，在需要重新绘制的地方调用下面方法，比如`setter`，`layoutSubviews`。

Objective-C

```
+ (Class)layerClass {
return YYAsyncLayer.class;
}
- (void)setText:(NSString *)text {
_text = text.copy;
[[YYTransaction transactionWithTarget:self selector:@selector(contentsNeedUpdated)] commit];
}
- (void)layoutSubviews {
[super layoutSubviews];
[[YYTransaction transactionWithTarget:self selector:@selector(contentsNeedUpdated)] commit];
}
```

YYTransaction有`selector`、`target`的属性，`selector`其实就是`contentsNeedUpdated`方法，此时并不会立即在后台线程去更新显示，而是将YYTransaction对象本身提交保存在`transactionSet`的集合中，上图中所示。

Objective-C

```
+ (YYTransaction *)transactionWithTarget:(id)target selector:(SEL)selector{
    if (!target || !selector) return nil;
    YYTransaction *t = [YYTransaction new];
    t.target = target;
    t.selector = selector;
    return t;
}
- (void)commit {
    if (!_target || !_selector) return;
    YYTransactionSetup();
    [transactionSet addObject:self];
}

```

同时在YYTransaction.m中注册一个RunloopObserver，监听MainRunloop在`kCFRunLoopCommonModes`（包含`kCFRunLoopDefaultMode`、`UITrackingRunLoopMode`）下的`kCFRunLoopBeforeWaiting`和`kCFRunLoopExit`的状态，也就是说在一次Runloop空闲时去执行更新显示的操作。

`kCFRunLoopBeforeWaiting`：Runloop将要进入休眠。
`kCFRunLoopExit`：即将退出本次Runloop。

Objective-C

```
static void YYTransactionSetup() {
    static dispatch_once_t onceToken;
    dispatch_once(&onceToken, ^{
        transactionSet = [NSMutableSet new];
        CFRunLoopRef runloop = CFRunLoopGetMain();
        CFRunLoopObserverRef observer;
        observer = CFRunLoopObserverCreate(CFAllocatorGetDefault(),
                                           kCFRunLoopBeforeWaiting | kCFRunLoopExit,
                                           true,      // repeat
                                           0xFFFFFF,  // after CATransaction(2000000)
                                           YYRunLoopObserverCallBack, NULL);
        CFRunLoopAddObserver(runloop, observer, kCFRunLoopCommonModes);
        CFRelease(observer);
    });
}
```

下面是RunloopObserver的回调方法，从transactionSet取出transaction对象执行SEL的方法，分发到每一次Runloop执行，避免一次Runloop执行时间太长。

Objective-C

```
static void YYRunLoopObserverCallBack(CFRunLoopObserverRef observer,
CFRunLoopActivity activity, void *info) {

    if (transactionSet.count == 0) return;

    NSSet *currentSet = transactionSet;

    transactionSet = [NSMutableSet new];

    [currentSet enumerateObjectsUsingBlock:^(YYTransaction *transaction, BOOL
*stop) {

        #pragma clang diagnostic push

        #pragma clang diagnostic ignored "-Warc-performSelector-leaks"

        [transaction.target performSelector:transaction.selector];

        #pragma clang diagnostic pop

    }];

}
```

接下来是异步绘制，这里用了一个比较巧妙的方法处理，当使用GCD时提交大量并发任务到后台线程导致线程被锁住、休眠的情况，创建与程序当前激活CPU数量（`activeProcessorCount`）相同的串行队列，并限制`MAX_QUEUE_COUNT`，将队列存放在数组中。
YYAsyncLayer.m有一个方法`YYAsyncLayerGetDisplayQueue`来获取这个队列用于绘制（这部分YYKit中有独立的工具YYDispatchQueuePool）。创建队列中有一个参数是告诉队列执行任务的服务质量quality of service，在iOS8+之后相比之前系统有所不同。

- iOS8之前队列优先级：

```
DISPATCH_QUEUE_PRIORITY_HIGH 2 高优先级
DISPATCH_QUEUE_PRIORITY_DEFAULT 0 默认优先级
DISPATCH_QUEUE_PRIORITY_LOW (-2) 低优先级
DISPATCH_QUEUE_PRIORITY_BACKGROUND INT16_MIN 后台优先级
```

- iOS8+之后：

QOS_CLASS_USER_INTERACTIVE 0x21,用户交互(希望尽快完成，不要放太耗时操作)
QOS_CLASS_USER_INITIATED 0x19,用户期望(不要放太耗时操作)
QOS_CLASS_DEFAULT 0x15,默认(用来重置对列使用的)
QOS_CLASS_UTILITY 0x11,实用工具(耗时操作，可以使用这个选项)
QOS_CLASS_BACKGROUND 0x09,后台
QOS_CLASS_UNSPECIFIED 0x00,未指定

Objective-C

```
/// Global display queue, used for content rendering.
static dispatch_queue_t YYAsyncLayerGetDisplayQueue() {
#ifdef YYDispatchQueuePool_h
return YYDispatchQueueGetForQOS(NSQualityOfServiceUserInitiated);
#else
#define MAX_QUEUE_COUNT 16


static int queueCount;
static dispatch_queue_t queues[MAX_QUEUE_COUNT];          //存放队列的数组
static dispatch_once_t onceToken;
static int32_t counter = 0;
dispatch_once(&onceToken, ^{
//程序激活的处理器数量
queueCount = (int)[NSProcessInfo processInfo].activeProcessorCount;
queueCount = queueCount  MAX_QUEUE_COUNT ? MAX_QUEUE_COUNT : queueCount);
if ([UIDevice currentDevice].systemVersion.floatValue >= 8.0) {
for (NSUInteger i = 0; i
```

接下来是关于绘制部分的代码，对外接口`YYAsyncLayerDelegate`代理中提供`- (YYAsyncLayerDisplayTask *)newAsyncDisplayTask`方法用于回调绘制的代码，以及是否异步绘制的BOOl类型属性`displaysAsynchronously`，同时重写CALayer的`display`方法来调

用绘制的方法`- (void)_displayAsync:(BOOL)async`。
这里有必要了解关于后台的绘制任务何时会被取消，下面两种情况需要取消，并调用了YYSentinel的`increase`方法，使`value`值增加（线程安全）：

- 在视图调用`setNeedsDisplay`时说明视图的内容需要被更新，将当前的绘制任务取消，需要重新显示。

- 以及视图被释放调用了`dealloc`方法。

在YYAsyncLayer.h中定义了`YYAsyncLayerDisplayTask`类，有三个block属性用于绘制的回调操作，从命名可以看出分别是将要绘制，正在绘制，以及绘制完成的回调，可以从block传入的参数`BOOL(^isCancelled)(void)`判断当前绘制是否被取消。

Objective-C

```
@property (nullable, nonatomic, copy) void (^willDisplay)(CALayer *layer);

@property (nullable, nonatomic, copy) void (^display)(CGContextRef context, CGSize size, BOOL(^isCancelled)(void));

@property (nullable, nonatomic, copy) void (^didDisplay)(CALayer *layer, BOOL finished);
```

下面是部分`- (void)_displayAsync:(BOOL)async`绘制的代码，主要是一些逻辑判断以及绘制函数，在异步执行之前通过`YYAsyncLayerGetDisplayQueue`创建的队列，这里通过`YYSentinel`判断当前的`value`是否等于之前的值，如果不相等，说明绘制任务被取消了，绘制过程会多次判断是否取消，如果是则return，保证被取消的任务能及时退出，如果绘制完毕则设置图片到`layer.contents`。

Objective-C

```
if (async) {  //异步
if (task.willDisplay) task.willDisplay(self);
YYSentinel *sentinel = _sentinel;
int32_t value = sentinel.value;
NSLog(@" --- %d ---", value);
//判断当前计数是否等于之前计数
BOOL (^isCancelled)() = ^BOOL() {
return value != sentinel.value;
};


CGSize size = self.bounds.size;
BOOL opaque = self.opaque;
CGFloat scale = self.contentsScale;
CGColorRef backgroundColor = (opaque && self.backgroundColor) ? CGColorRetain(self.backgroundColor) : NULL;
if (size.width
```

9.优化你是从哪几方面着手?

一、首页启动速度

启动过程中做的事情越少越好（尽可能将多个接口合并）

不在UI线程上作耗时的操作（数据的处理在子线程进行，处理完通知主线程刷新节目）

在合适的时机开始后台任务（例如在用户指引节目就可以开始准备加载的数据）

尽量减小包的大小

优化方法：

量化启动时间

启动速度模块化

辅助工具（友盟，听云，Flurry）

二、页面浏览速度

json的处理（iOS 自带的NSJSONSerialization，Jsonkit，SBJson）

数据的分页（后端数据多的话，就要分页返回，例如网易新闻，或者 微博记录）

数据压缩（大数据也可以压缩返回，减少流量，加快反应速度）

内容缓存（例如网易新闻的最新新闻列表都是要缓存到本地，从本地加载，可以缓存到内存，或者数据库，根据情况而定）

延时加载tab（比如app有5个tab，可以先加载第一个要显示的tab，其他的在显示时候加载，按需加载）

算法的优化（核心算法的优化，例如有些app 有个 联系人姓名用汉语拼音的首字母排序）

三、操作流畅度优化：

Tableview 优化（tableview cell的加载优化）

ViewController加载优化（不同view之间的跳转，可以提前准备好数据）

四、数据库的优化：

数据库设计上面的重构

查询语句的优化

分库分表（数据太多的时候，可以分不同的表或者库）

五、服务器端和客户端的交互优化：

客户端尽量减少请求

服务端尽量做多的逻辑处理

服务器端和客户端采取推拉结合的方式（可以利用一些同步机制）

通信协议的优化。（减少报文的大小）

电量使用优化（尽量不要使用后台运行）

六、非技术性能优化

产品设计的逻辑性（产品的设计一定要符合逻辑，或者逻辑尽量简单，否则会让程序员抓狂，有时候用了好大力气，才可以完成一个小小的逻辑设计问题）

界面交互的规范（每个模块的界面的交互尽量统一，符合操作习惯）

代码规范（这个可以隐形带来app 性能的提高，比如 用if else 还是switch ，或者是用！还是＝＝）

code review（坚持code Review 持续重构代码。减少代码的逻辑复杂度）

日常交流（经常分享一些代码，或者逻辑处理中的坑）

以上问题加参考答案，部分自己回答(群友回答)＋网上博客参考，回答的不好勿喷！

仅供学习使用！ 谢谢！

公众号二维码，扫码关注最新iOS动态！