腾讯一面

BAT iOS面试题分享群：2466454 （吹水勿扰）

1.使用了第三方库, 有看他们是怎么实现的吗?
2.强连通量算法了解嘛?
3.遇到tableView卡顿嘛？会造成卡顿的原因大致有哪些?
4.M、V、C相互通讯规则你知道的有哪些?
5.NStimer准吗？谈谈你的看法？如果不准该怎样实现一个精确的NSTimer?

BAT面试题简书地址：http://www.jianshu.com/p/0e9e7486e1a7

1.使用了第三方库, 有看他们是怎么实现的吗?

例：SD、YY、AFN、MJ等!

<1>.SD为例：

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 SDWebImageManagerdownloadWithURL: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。
12.connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder
做图片解码处理。
13.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。
14.在主线程 notifyDelegateOnMainThreadWithInfo:
宣告解码完成，
imageDecoder:didFinishDecodingImage:userInfo:
回调给 SDWebImageDownloader。
15.imageDownloader:didFinishWithImage:
回调给 SDWebImageManager 告知图片下载完成。
16.通知所有的 downloadDelegates 下载完成，
回调给需要的地方展示图片。
17.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。

18.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。
19.SDWI 也提供了 UIButton+WebCache 和
MKAnnotationView+WebCache，方便使用。
20.SDWebImagePrefetcher 可以预先下载图片，
方便后续使用。

2.强连通分量了解嘛?

概念：

有向图强连通分量：在有向图G中，如果两个顶点vi,vj间（vi>vj）有一条从vi到vj的有向路径，同时还有一条从vj到vi的有向路径，则称两个顶点强连通(strongly connected)。如果有向图G的每两个顶点都强连通，称G是一个强连通图。有向图的极大强连通子图，称为强连通分量(strongly connected components)。

定义：

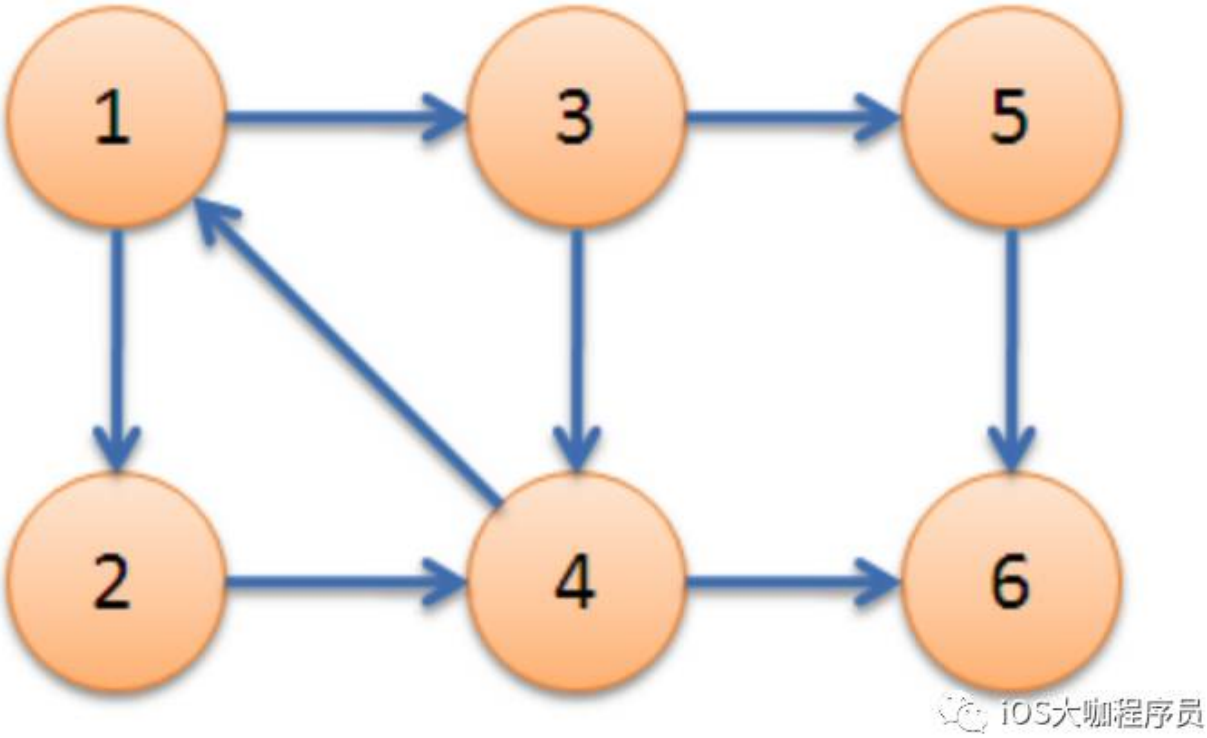
有向图强连通分量：

在有向图G中，如果两个顶点间至少存在一条路径，称两个顶点强连通（strongly connected）。

如果有向图G的每两个顶点都强连通，则称G是一个强连通图。

非强连通图有向图的极大强连通子图，成为强连通分量（strongly connected components）。

下图中，子图{1,2,3,4}为一个强连通分量，因为顶点1,2,3,4两两可达，{5}，{6}也分别是两个强连通分量。



直接根据定义，用双向遍历取交际的方法求强连通分量，时间复杂度为O（N^2+M）。更好的方法是Kosaraju算法或者Tarjan算法。

两者的时间复杂度都是O（N+M）。本文介绍的是Tarjan算法。

算法原理：（Tarjan）

Tarjan算法是基于对图深度优先搜索的算法，每个强连通分量为搜索树中的一颗子树。

搜索时，把当前搜索树中未处理的节点加入一个堆栈，回溯时可以盘对栈顶到栈中的节点是否为一个强连通分量。

定义DFN（u）为节点u搜索的次序编号（时间戳）。Low（u）为u或者u的子树能够追溯到的最早的栈中的节点的次序号。

由定义可以得出：

Low（u）= Min { DFN（u），Low（v）}（（u，v）为树枝边，u为v的父节点DFN（v），（u，v）为指向栈中节点的后向边（非横叉边））

当DFN（u）=Low（u）时，以u为根的搜索子树上所有节点是一个强连通分量。

代码实现：

[cpp]：

```
1. #include <stdio.h>
2. #include <string.h>
3. #include <vector>
4. #include <stack>
```

```
5.  using namespace std;
6.  #define MIN(a,b) ((a)<(b)?(a):(b))
7.  #define N 10005            // 题目中可能的最大点数
8.  stack<int>sta;              // 存储已遍历的结点
9.  vector<int>gra[N];          // 邻接表表示图
10. int dfn[N];                // 深度优先搜索访问次序
11. int low[N];                // 能追溯到的最早的次序
12. int InStack[N];            // 检查是否在栈中(2为在栈中，1为已访问，且不
    在栈中，0为不在)
13. vector<int> Component[N];  // 获得强连通分量结果
14. int InComponent[N];        // 记录每个点在第几号强连通分量里
15. int index,ComponentNumber; // 索引号，强连通分量个数
16. int n, m;                  // 点数，边数
17.
18. void init(void)
```

```
19. {
20.     memset(dfn, 0, sizeof(dfn));
21.     memset(low, 0, sizeof(low));
22.     memset(InStack, 0, sizeof(InStack));
23.     index = ComponentNumber = 0;
24.     for (int i = 1; i <= n; ++ i)
25.     {
26.         gra[i].clear();
27.         Component[i].clear();
28.     }
29.
30.     while(!sta.empty())
31.         sta.pop();
32. }
33.
34. void tarjan(int u)
```

```
35. {
36.     Instack[u] = 2;
37.     low[u] = dfn[u] = ++ index;
38.     sta.push(u);
39.
40.     for (int i = 0; i < gra[u].size(); ++ i)
41.     {
42.         int t = gra[u][i];
43.         if (dfn[t] == 0)
44.         {
45.             tarjan(t);
46.             low[u] = MIN(low[u], low[t]);
47.         }
48.         else if (InStack[t] == 2)
49.         {
50.             low[u] = MIN(low[u], dfn[t]);
```

```
51.         }
52.     }
53.
54.     if (low[u] == dfn[u])
55.     {
56.         ++ ComponentNumber;
57.         while (!sta.empty())
58.         {
59.             int j = sta.top();
60.             sta.pop();
61.             InStack[j] = 1;
62.             Component[ComponentNumber].push_back(j);
63.             InComponent[j]=ComponentNumber;
64.             if (j == u)
65.                 binputak;
66.         }
```

```
67.     }
68. }
69.
70. void input(void)
71. {
72.     for(int i=1;i<=m;i++)
73.     {
74.         int a,b;
75.         scanf("%d%d",&a,&b);
76.         gra[a].push_back(b);
77.     }
78. }
79.
80. void solve(void)
81. {
82.     for(int i=1;i<=n;i++)
```

```
83.         if(!dfn[i])
84.             tarjan(i);
85.     if(ComponentNumber>1)
86.         puts("No");
87.     else
88.         puts("Yes");
89. }
90.
91. int main()
92. {
93.     while(scanf("%d%d",&n,&m),n+m)
94.     {
95.         init();
96.         input();
97.         solve();
98.     }
```

```
99. }
```

3.遇到tableView卡顿嘛？会造成卡顿的原因大致有哪些?

可能造成tableView卡顿的原因有：

1.最常用的就是cell的重用， 注册重用标识符
如果不重用cell时，每当一个cell显示到屏幕上时，就会重新创建一个新的cell；

如果有很多数据的时候，就会堆积很多cell。

如果重用cell，为cell创建一个ID，每当需要显示cell 的时候，都会先去缓冲池中寻找可循环利用的cell，如果没有再重新创建cell

2.避免cell的重新布局

cell的布局填充等操作 比较耗时，一般创建时就布局好
如可以将cell单独放到一个自定义类，初始化时就布局好

3.提前计算并缓存cell的属性及内容

当我们创建cell的数据源方法时，编译器并不是先创建cell 再定cell的高度
而是先根据内容一次确定每一个cell的高度，高度确定后，再创建要显示的cell，滚动时，每当cell进入凭虚都会计算高度，提前估算高度告诉编译器，编译器知道高度后，紧接着就会创建cell，这时再调用高度的具体计算方法，这

样可以方式浪费时间去计算显示以外的cell

4.减少cell中控件的数量

尽量使cell得布局大致相同，不同风格的cell可以使用不用的重用标识符，初始化时添加控件，
不适用的可以先隐藏

5.不要使用ClearColor，无背景色，透明度也不要设置为0

渲染耗时比较长

6.使用局部更新

如果只是更新某组的话，使用reloadSection进行局部更新

7.加载网络数据，下载图片，使用异步加载，并缓存

8.少使用addView 给cell动态添加view

9.按需加载cell，cell滚动很快时，只加载范围内的cell

10.不要实现无用的代理方法，tableView只遵守两个协议

11.缓存行高：estimatedHeightForRow不能和HeightForRow里面的layoutIfNeed同时存在，这两者同时存在才会出现“窜动”的bug。所以我的建议是：只要是固定行高就写预估行高来减少行高调用次数提升性能。如果是动态行高就不要写预估方法了，用一个行高的缓存字典来减少代码的调用次数即可

12.不要做多余的绘制工作。
在实现drawRect:的时候，它的rect参数就是需要绘制的区域，这个区域之外的不需要进行绘制。
例如上例中，就可以用CGRectIntersectsRect、CGRectIntersection或CGRectContainsRect判断是否需要绘制image和text，然后再调用绘制方法。

13.预渲染图像。
当新的图像出现时，仍然会有短暂的停顿现象。解决的办法就是在bitmap context里先将其画一遍，导出成UIImage对象，然后再绘制到屏幕；

14.使用正确的数据结构来存储数据。

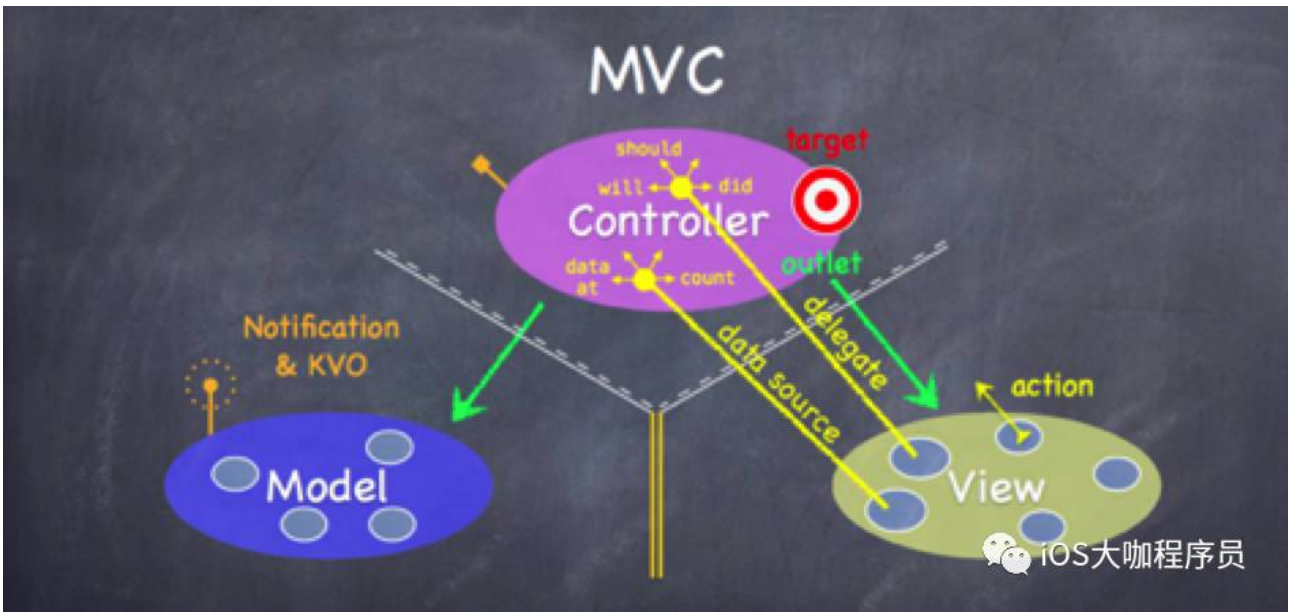
4.M、V、C相互通讯规则你知道的有哪些?

MVC 是一种设计思想，一种框架模式，是一种把应用中所有类组织起来的策略，它把你的程序分为三块，分别是：

M（Model）：实际上考虑的是“什么”问题，你的程序本质上是什么，独立于 UI 工作。是程序中用于处理应用程序逻辑的部分，通常负责存取数据。

C（Controller）：控制你 Model 如何呈现在屏幕上，当它需要数据的时候就告诉 Model，你帮我获取某某数据；当它需要 UI 展示和更新的时候就告诉 View，你帮我生成一个 UI 显示某某数据，是 Model 和 View 沟通的桥梁。
V（View）：Controller 的手下，是 Controller 要使用的类，用于构建视图，通常是根据 Model 来创建视图的。

要了解 MVC 如何工作，首先需要了解这三个模块间如何通信。
MVC通信规则



Controller to Model

可以直接单向通信。Controller 需要将 Model 呈现给用户，因此需要知道模型的一切，还需要有同 Model 完全通信的能力，并且能任意使用 Model 的公共 API。

Controller to View

可以直接单向通信。Controller 通过 View 来布局用户界面。

Model to View

永远不要直接通信。Model 是独立于 UI 的，并不需要和 View 直接通信，View 通过 Controller 获取 Model 数据。

View to Controller

View 不能对 Controller 知道的太多，因此要通过间接的方式通信。
Target action。首先 Controller 会给自己留一个 target，再把配套的 action 交给 View 作为联系方式。那么 View 接收到某些变化时，View 就会发送 action 给 target 从而达到通知的目的。这里 View 只需要发送 action，并不需要知道 Controller 如何去执行方法。

代理。有时候 View 没有足够的逻辑去判断用户操作是否符合规范，他会把判断这些问题的权力委托给其他对象，他只需获得答案就行了，并不会管是谁给的答案。

DataSoure。View 没有拥有他们所显示数据的权力，View 只能向 Controller 请求数据进行显示，Controller 则获取 Model 的数据整理排版后提供给 View。

Model 访问 Controller
同样的 Model 是独立于 UI 存在的，因此无法直接与 Controller 通信，但是当 Model 本身信息发生了改变的时候，会通过下面的方式进行间接通信。

Notification & KVO一种类似电台的方法，Model 信息改变时会广播消息给感兴趣的人，只要 Controller 接收到了这个广播的时候就会主动联系 Model，获取新的数据并提供给 View。
从上面的简单介绍中我们来简单概括一下 MVC 模式的优点。
1.低耦合性
2.有利于开发分工
3.有利于组件重用
4.可维护性

5.NStimer准吗？谈谈你的看法？如果不准该怎样实现一个精确的NSTimer?

1.不准

2.不准的原因如下：

1、NSTimer加在main runloop中，模式是NSDefaultRunLoopMode，main负责所有主线程事件，例如UI界面的操作，复杂的运算，这样在同一个runloop中timer就会产生阻塞。

2、模式的改变。主线程的 RunLoop 里有两个预置的 Mode：kCFRunLoopDefaultMode 和 UITrackingRunLoopMode。

当你创建一个 Timer 并加到 DefaultMode 时，Timer 会得到重复回调，但此时滑动一个ScrollView时，RunLoop 会将 mode 切换为 TrackingRunLoopMode，这时 Timer 就不会被回调，并且也不会影响到滑动操作。所以就会影响到NSTimer不准的情况。

PS:DefaultMode 是 App 平时所处的状态，rackingRunLoopMode 是追踪 ScrollView 滑动时的状态。

方法一：

1、在主线程中进行NSTimer操作，但是将NSTimer实例加到main runloop的特定mode（模式）中。避免被复杂运算操作或者UI界面刷新所干扰。

```
self.timer = [NSTimer timerWithTimeInterval:1 target:self
selector:@selector(showTime) userInfo:nil repeats:YES];

[[NSRunLoop currentRunLoop] addTimer:self.timer
forMode:NSRunLoopCommonModes];
```

2、在子线程中进行NSTimer的操作，再在主线程中修改UI界面显示操作结果；

```
- (void)timerMethod2 {

NSThread *thread = [[NSThread alloc] initWithTarget:self
```

```
selector:@selector(newThread) object:nil];

[thread start];

}

- (void)newThread

{

@autoreleasepool

{

[NSTimer scheduledTimerWithTimeInterval:1.0 target:self
selector:@selector(showTime) userInfo:nil repeats:YES];

[[NSRunLoop currentRunLoop] run];

}

}
```

总结：

一开始的时候系统就为我们将主线程的main runloop隐式的启动了。

在创建线程的时候，可以主动获取当前线程的runloop。每个子线程对应一个runloop

方法二：

使用示例

使用mach内核级的函数可以使用mach_absolute_time()获取到CPU的tickcount的计数值，可以通过”mach_timebase_info”函数获取到纳秒级的精确度 。然后使用mach_wait_until(uint64_t deadline)函数，直到指定的时间之后，就可以

执行指定任务了。

关于数据结构mach_timebase_info的定义如下：

struct mach_timebase_info {uint32_t numer;uint32_t denom;};

```
#include <mach/mach.h>
#include <mach/mach_time.h>

static const uint64_t NANOS_PER_USEC = 1000ULL;
static const uint64_t NANOS_PER_MILLISEC = 1000ULL *
NANOS_PER_USEC;
static const uint64_t NANOS_PER_SEC = 1000ULL *
NANOS_PER_MILLISEC;

static mach_timebase_info_data_t timebase_info;

static uint64_t nanos_to_abs(uint64_t nanos) {
    return nanos * timebase_info.denom / timebase_info.numer;
}

void example_mach_wait_until(int seconds)
{
    mach_timebase_info(&timebase_info);
    uint64_t time_to_wait = nanos_to_abs(seconds * NANOS_PER_SEC);
    uint64_t now = mach_absolute_time();
    mach_wait_until(now + time_to_wait);
}
```

方法三：直接使用GCD替代！

下次面试题，再见！