小米、百度、bigo 、滴滴 、快手等面试后的一次阶段性总结

面试题出自小专栏的一篇文章 《小米、百度、bigo 、滴滴 、快手等面试后的一次阶段性总结》, 解答技术相关问题，若存在解答纰漏，欢迎提 issue 修正。

欢迎转载，转载请注明出处：pmst-swiftgg

调试好可运行的源码 objc-runtime，官网找 objc4

版本：V0.1

进度：50%

最后修订：2020/05/24

[TOC]

1. 小米

一面

1. 介绍有哪些设计原则，并让比较详细的说了其中开闭原则在项目中的应用。

2. 介绍设计模式，然后其中主要问了我抽象工厂和适配器两种模式。

3. 介绍 runloop 相关的知识和在实际开发中的使用情况

1. 卡顿监控，监听 source before 和 after waiting RunLoop 事件，设定超时信号量超时时间比如 50ms，若超时 3-5 次记为一次卡顿，可以借助 PLCrashreport 来 dump 出线程堆栈信息；
2. CPU 层面的异步渲染，YYAsyncLayer 监听 runloop before waiting 事件，对于 UI 操作都封装成了 YYTransaction 事物提交，此刻只是将对 UI 的操作事务都保存在数组中，等到 runloop 即将休眠的时候 flush 所有事务处理，即对 layer 进行 `setNeedsDisplay` 调用打上脏标记，然后系统会执行 displayLayer 方法（Note: YYAsyncLayer 是重写了 displayLayer 方法，此外还可以借助 delegate 实现），该方法中才是真正将绘制操作异步派发到子线程中去操作，也就是借助 Core Graphic 进行 context 绘制，然后得到 image，**回到主线程赋值 layer.contents = images** 保证线程安全。**这里为什么要借助 runloop，可能是为了精准的延后视图更新的时机吧。**

4. 要求详细的描述事件响应链

app 如何接收到触摸事件的

iOS Touch Event from the inside out

1 Touch Event 的生命周期

1.1 物理层面事件的生成

iPhone 采用电容触摸传感器，利用人体的电流感应工作，由一块四层复合玻璃屏的内表面和夹层各涂有一层导电层，最外层是一层矽土玻璃保护层。当我们手指触摸感应屏的时候，人体的电场让手指和触摸屏之间形成一个耦合电容，对高频电流来说电容是直接导体。于是手指从接触点吸走一个很小的电流，这个电流分从触摸屏的四脚上的电极流出，并且流经这四个电极的电流和手指到四个电极的距离成正比。控制器通过对这四个电流的比例做精确的计算，得出触摸点的距离。

更多文献：

- OLED 发光原理、面板结构及 OLED 关键技术深度解析
- 光学触摸屏原理
- 电容触摸屏原理
- 电阻式触摸屏原理(FLASH 演示版)
- iPhone 这十年在传感器上的演进

1.2 iOS 操作系统下封装和分发事件

iOS 操作系统看做是一个处理复杂逻辑的程序，不同进程之间彼此通信采用消息发送方式，即 IPC (Inter-Process Communication)。现在继续说上面电容触摸传感器产生的 Touch Event，它将交由 IOKit.framework 处理封装成 IOHIDEvent 对象；下一步很自然想到通过消息发送方式将事件传递出去，至于发送给谁，何时发送等一系列的判断逻辑又该交由谁处理呢？

答案是 **SpringBoard.app**，它接收到封装好的 **IOHIDEvent** 对象，经过逻辑判断后做进一步的调度分发。例如，它会判断前台是否运行有应用程序，有则将封装好的事件采用 mach port 机制传递给该应用的主线程。

Port 机制在 IPC 中的应用是 Mach 与其他传统内核的区别之一，在 Mach 中，用户进程调用内核交由 IPC 系统。与直接系统调用不同，用户进程首先向内核申请一个 port 的访问许可；然后利用 IPC 机制向这个 port 发送消息，本质还是系统调用，而处理是交由其他进程完成的。

1.3 IOHIDEvent -> UIEvent

应用程序主线程的 runloop 申请了一个 mach port 用于监听 `IOHIDEvent` 的 `Source1` 事件，回调方法是 `__IOHIDEventSystemClientQueueCallback()`，内部又进一步分发 `Source0` 事件，而 `Source0` 事件都是自定义的，非基于端口 port，包括触摸，滚动，selector 选择器事件，它的回调方法是 `__UIApplicationHandleEventQueue()`，将接收到的 `IOHIDEvent` 事件对象封装成我们熟悉的 `UIEvent` 事件；然后调用 `UIApplication` 实例对象的 `sendEvent:` 方法，将 `UIEvent` 传递给 `UIWindow` 做一些逻辑判断工作：比如触摸事件产生于哪些视图

上，有可能有多个，那又要确定哪个是最佳选项呢？ 等等一系列操作。这里先按下不表。

1.4 Hit-Testing 寻找最佳响应者

Source0 回调中将封装好的触摸事件 UIEvent（里面有多个 UITouch 即手势点击对象），传递给视图 UIWindow，其目的在于找到最佳响应者，这个过程称之为 Hit-Testing，字面上理解：hit 即触碰了屏幕某块区域，这个区域可能有多个视图叠加而成，那么这个触摸讲道理响应者有多个喽，那么“最佳”又该如何评判？这里要牢记几个规则：

1. 事件是自下而上传递，即 UIApplication -> UIWindow -> 子视图 -> ...-> 子视图中的子视图;
2. 后加的视图响应程度更高，即更靠近我们的视图;
3. 如果某个视图不想响应，则传递给比它响应程度稍低一级的视图，若能响应，你还得继续往下传递，若某个视图能响应了，但是没有子视图 它就是最佳响应者。
4. 寻找最佳响应者的过程中， UIEvent 中的 UITouch 会不断打上标签：比如 HitTest View 是哪个，superview 是哪个？关联了什么 Gesture Recognizer?

那么如何判定视图为响应者？由于 OC 中的类都继承自 NSObject ，因此默认判断逻辑已经在 hitTest:withEvent 方法中实现，它有两个作用： 1.询问当前视图是否能够响应事件 2.事件传递的桥梁。若当前视图无法响应事件，返回 nil 。代码如下：

```
- (UIView *)hitTest:(CGPoint)point withEvent:(UIEvent *)event{
  // 1. 前置条件要满足
  if (self.userInteractionEnabled == NO ||
  self.hidden == YES ||
  self.alpha <= 0.01) return nil;

  // 2. 判断点是否在视图内部 这是最起码的 note point 是在当前视图坐标系的点位置
    if ([self pointInside:point withEvent:event] == NO) return nil;

  // 3. 现在起码能确定当前视图能够是响应者 接下去询问子视图
    int count = (int)self.subviews.count;
    for (int i = count - 1; i >= 0; i--)
    {
      // 子视图
        UIView *childView = self.subviews[i];

    // 点需要先转换坐标系
        CGPoint childP = [self convertPoint:point toView:childView];
        // 子视图开始询问
        UIView *fitView = [childView hitTest:childP withEvent:event];
```

```
        if (fitView)
        {
      return fitView;
    }
    }

    return self;
}
```

1. 首先满足几个前置条件，可交互 userInteractionEnabled=YES；没有隐藏 self.hidden == NO；非透明 self.alpha <= 0.01 ———— 注意一旦不满足上述三个条件，当前视图及其子视图都不能作为响应者，Hit-Testing 判定也止步于此
2. 接着判断触摸点是否在视图内部 ———— 这个是最基本，无可厚非的判定规则
3. 此时已经能够说当前视图为响应者，但是不是**最佳**还不能下定论，因此需要进一步传递给子视图判定；注意 pointInside 也是默认实现的。

1.5 UIResponder Chain 响应链

Hit-Testing 过程中我们无法确定当前视图是否为“最佳”响应者，此时自然还不能处理事件。因此处理机制应该是找到所有响应者以及最佳响应者(**自下而上**)，由它们构成了一条响应链；接着将事件沿着响应链**自上而下**传递下去 ——最顶端自然是最佳响应者，事件除了被响应者消耗，还能被手势识别器或是 target-action 模式捕获并消耗。有时候，最佳响应者可能对处理 Event “毫无兴趣”，它们不会重写 touchBegan touchesMove..等四个方法；也不会添加任何手势；但如果是 control(控件) 比如 UIButton ，那么事件还是会被消耗掉的。

1.6 UITouch 、 UIEvent 、UIResponder

IOHIDEvent 前面说到是在 IOKit.framwork 中生成的然后经过一系列的分别才到达前台应用，然后应用主线程 runloop 处理 source1 回调中又进行 source0 事件分发，这里有个封装 UIEvent 的过程，那么 UITouch 呢？ 是不是也是那时候呢？换种思路：一个手指一次触摸屏幕 生成一个 UITouch 对象，内部应该开始进行识别了，因为可能是多个 Touch，并且触摸的先后顺序也不同，这样识别出来的 UIEvent 也不同。所以 UIEvent 对象中包含了触发该事件的触摸对象的集合，通过 allTouches 属性获取。

每个响应者都派生自 UIResponder 类，本身具有相应事件的能力，响应者默认实现 touchesBegin touchesMove touchesEnded touchesCancelled 四个方法。

事件在未截断的情况下沿着响应链传递给最佳响应者，伪代码如下：

```
0 - [AView touchesBegan:withEvent
1 - [UIWindow _sendTouchesForEvent]
2 - [UIWindow sendEvent]
```

```
3 - [UIApplication sendEvent]
4 __dispatchPreprocessEventFromEventQueue
5 __handleEventQueueInternal
6 _CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION_
7 _CFRunLOOPDoSource0
8 _CFRunLOOPDoSources0
9 _CFRunLoopRun
10 _CFRunLoopRunSpecific
11 GSEventRunModal
12 UIApplication
13 main
14 start

// UIApplication.m
- (void)sendEvent {
  [window sendEvent];
}

// UIWindow.m
- (void)sendEvent{
  [self _sendTouchesForEvent];
}

- (void)_sendTouchesForEvent{
  //find AView Because we know hitTest View
  [AView touchesBegan:withEvent];
}
```

二面

回文算法（算法）判断一个字符串是不是对称的字符串，比如 abba 或者 aba 这样的就是对称的。

```
bool isPalindrome(char *s){
  int len = strlen(s);
  int i = 0;
  int j = len-1;
  while(i < j){
    if(s[i] != s[j])return false;
  }
  retrun true;
}
```

leetcode 关于回文的相关算法题

block 的实现原理

1. block 的内部实现，结构体是什么样的

block 也是一个对象，主要分为 Imp 结构体 和 Desc 结构体，用 `clang -rewrite-objc` 命令将 oc 代码重写成 c++：

```
struct __block_impl {
  void *isa;
  int Flags;
  int Reserved;
  void *FuncPtr;
};

struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, i
nt flags=0) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
```

实际上 runtime 源码定义的 block 数据类型和上面还是有一定出路的，建议以下面的数据结构为准，另外强烈建议看下文中的《Block 原理探究代码篇》

```
typedef NS_OPTIONS(int,PTBlockFlags) {
    PTBlockFlagsHasCopyDisposeHelpers = (1 << 25),
    PTBlockFlagsHasSignature          = (1 << 30)
};
typedef struct PTBlock {
    __unused Class isa;
    PTBlockFlags flags;
    __unused int reserved;
    void (__unused *invoke)(struct PTBlock *block, ...);
    struct {
        unsigned long int reserved;
        unsigned long int size;
        // requires PTBlockFlagsHasCopyDisposeHelpers
        void (*copy)(void *dst, const void *src);
        void (*dispose)(const void *);
        // requires PTBlockFlagsHasSignature
        const char *signature;
        const char *layout;
    } *descriptor;
    // imported variables
     // Block 捕获的实例变量都在次
} *PTBlockRef;

typedef struct PTBlock_byref {
    void *isa;
```

```
    struct PTBlock_byref *forwarding;
    volatile int flags; // contains ref count
    unsigned int size;
    // 下面两个函数指针是不定的 要根据flags 来
//    void (*byref_keep)(struct PTBlock_byref *dst, struct PTBlock_
byref *src);
//    void (*byref_destroy)(struct PTBlock_byref *);
    // long shared[0];
} *PTBlock_byref_Ref;
```

2. block 是类吗，有哪些类型

   一般，block 有三种：_NSConcreteGlobalBlock、_NSConcreteStackBlock、_NSConcreteMallocBlock，根据 Block 对象创建时所处数据区不同而进行区别。

   1. 栈上 Block，引用了栈上变量，生命周期由系统控制的，一旦所属作用域结束，就被系统销毁了，ARC 下由于默认是 _strong 属性，所以打印的可能都是 _NSConcreteMallocBlock，**这里使用 unretained** 关键字去修饰下就行了，或者 MRC 下去验证
   2. 堆上 Block，使用 copy 或者 strong（ARC）下就从栈 Block 拷贝到堆上。
   3. 全局 Block，未引用任何栈上变量时就是全局 Block;

3. 一个 int 变量被 __block 修饰与否的区别？block 的变量截获

   值 copy 和指针 copy，__block 修饰的话允许在 block 内部修改变量，因为传入的是 int 变量的指针。

   外部变量有四种类型：自动变量、静态变量、静态全局变量、全局变量。

   全局变量和静态全局变量在 block 中是直接引用的，不需要通过结构去传入指针；

   函数/方法中的 static 静态变量是直接在 block 中保存了指针，如下测试代码：

```
int a = 1;
static int b = 2;

int main(int argc, const char * argv[]) {

    int c = 3;
    static int d = 4;
    NSMutableString *str = [[NSMutableString alloc]initWithString:@
"hello"];
    void (^blk)(void) = ^{
        a++;
        b++;
        d++;
```

```
        [str appendString:@"world"];
        NSLog(@"1---------- a = %d,b = %d,c = %d,d = %d,str = %@",
a,b,c,d,str);
    };

    a++;
    b++;
    c++;
    d++;
str = [[NSMutableString alloc]initWithString:@"haha"];
    NSLog(@"2---------- a = %d,b = %d,c = %d,d = %d,str = %@",a,b,
c,d,str);
    blk();

    return 0;
}
```

转成 c++ 代码：

```
struct __block_impl {
  void *isa;
  int Flags;
  int Reserved;
  void *FuncPtr;
};

int a = 1; // <------------------- NOTE
static int b = 2; // <------------------- NOTE
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  int *d;                          // <------------------- NOTE
  NSMutableString *str;               // <------------------- NOTE
  int c; // <------------------- NOTE
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, i
nt *_d, NSMutableString *_str, int _c, int flags=0) : d(_d), str(_s
tr), c(_c) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};

static void __main_block_func_0(struct __main_block_impl_0 *__cself)
 {
  int *d = __cself->d; // bound by copy
  NSMutableString *str = __cself->str; // bound by copy
  int c = __cself->c; // bound by copy
```

```
        a++;
        b++;
        (*d)++;
        ((void (*)(id, SEL, NSString *))(void *)objc_msgSend)((id)s
tr, sel_registerName("appendString:"), (NSString *)&__NSConstantStr
ingImpl__var_folders_7__3g67htjj4816xmx7ltbp2ntc0000gn_T_main_150b2
1_mi_1);
        NSLog((NSString *)&__NSConstantStringImpl__var_folders_7__3
g67htjj4816xmx7ltbp2ntc0000gn_T_main_150b21_mi_2,a,b,c,(*d),str);
    }
static void __main_block_copy_0(struct __main_block_impl_0*dst, str
uct __main_block_impl_0*src) {_Block_object_assign((void*)&dst->str,
 (void*)src->str, 3/*BLOCK_FIELD_IS_OBJECT*/);}

static void __main_block_dispose_0(struct __main_block_impl_0*src)
{_Block_object_dispose((void*)src->str, 3/*BLOCK_FIELD_IS_OBJECT*/
);}

static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
  void (*copy)(struct __main_block_impl_0*, struct __main_block_imp
l_0*);
  void (*dispose)(struct __main_block_impl_0*);
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0
), __main_block_copy_0, __main_block_dispose_0};

int main(int argc, const char * argv[]) {
    int c = 3;
    static int d = 4;
    NSMutableString *str = ((NSMutableString *(*)(id, SEL, NSString
 *))(void *)objc_msgSend)((id)((NSMutableString *(*)(id, SEL))(void
 *)objc_msgSend)((id)objc_getClass("NSMutableString"), sel_register
Name("alloc")), sel_registerName("initWithString:"), (NSString *)&_
_NSConstantStringImpl__var_folders_7__3g67htjj4816xmx7ltbp2ntc0000g
n_T_main_150b21_mi_0);
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)_
_main_block_func_0, &__main_block_desc_0_DATA, &d, str, c, 57042534
4));

    a++;
    b++;
    c++;
    d++;
    str = ((NSMutableString *(*)(id, SEL, NSString *))(void *)objc_
msgSend)((id)((NSMutableString *(*)(id, SEL))(void *)objc_msgSend)
((id)objc_getClass("NSMutableString"), sel_registerName("alloc")),
sel_registerName("initWithString:"), (NSString *)&__NSConstantStrin
```

```
gImpl__var_folders_7__3g67htjj4816xmx7ltbp2ntc0000gn_T_main_150b21_
mi_3);
    NSLog((NSString *)&__NSConstantStringImpl__var_folders_7__3g67h
tjj4816xmx7ltbp2ntc0000gn_T_main_150b21_mi_4,a,b,c,d,str);
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__b
lock_impl *)blk);

    return 0;
}
```

4. block 在修改 NSMutableArray，需不需要添加__block

   不需要，本身 block 内部就捕获了 NSMutableArray 指针，除非你要修改指针指向的对象，而这里明显只是修改内存数据，这个可以类比 NSMutableString。

5. 怎么进行内存管理的

   `static void *_Block_copy_internal(const void *arg, const int flags)` 和 `void _Block_release(void *arg)`

   推荐 iOS Block 原理探究以及循环引用的问题 一文。

6. block 可以用 strong 修饰吗

   ARC 貌似是可以的， strong 和 copy 的操作都是将栈上 block 拷贝到堆上。TODO：确认下。

7. 解决循环引用时为什么要用__strong、__weak 修饰

   __weak 就是为了避免 retainCycle，而 block 内部 __strong 则是在作用域 retain 持有当前对象做一些操作，结束后会释放掉它。

8. block 发生 copy 时机

   block 从栈上拷贝到堆上几种情况：

   - 调用 Block 的 copy 方法
   - 将 Block 作为函数返回值时
   - 将 Block 赋值给__strong 修饰的变量或 Block 类型成员变量时
   - 向 Cocoa 框架含有 usingBlock 的方法或者 GCD 的 API 传递 Block 参数时

9. Block 访问对象类型的 auto 变量时，在 ARC 和 MRC 下有什么区别

Block 原理探究代码篇

首先明确 Block 底层数据结构，之后所有的 demos 都基于此来学习知识点：

```
typedef NS_OPTIONS(int,PTBlockFlags) {
    PTBlockFlagsHasCopyDisposeHelpers = (1 << 25),
    PTBlockFlagsHasSignature          = (1 << 30)
};
typedef struct PTBlock {
    __unused Class isa;
    PTBlockFlags flags;
    __unused int reserved;
    void (__unused *invoke)(struct PTBlock *block, ...);
    struct {
        unsigned long int reserved;
        unsigned long int size;
        // requires PTBlockFlagsHasCopyDisposeHelpers
        void (*copy)(void *dst, const void *src);
        void (*dispose)(const void *);
        // requires PTBlockFlagsHasSignature
        const char *signature;
        const char *layout;
    } *descriptor;
    // imported variables
    // Block 捕获的实例变量都在次
} *PTBlockRef;

typedef struct PTBlock_byref {
    void *isa;
    struct PTBlock_byref *forwarding;
    volatile int flags; // contains ref count
    unsigned int size;
    // 下面两个函数指针是不定的 要根据flags 来
//    void (*byref_keep)(struct PTBlock_byref *dst, struct PTBlock_byre
f *src);
//    void (*byref_destroy)(struct PTBlock_byref *);
    // long shared[0];
} *PTBlock_byref_Ref;
```

1. 调用 block

```
void (^blk)(void) = ^{
  NSLog(@"hello world");
};
PTBlockRef block = (__bridge PTBlockRef)blk;
block->invoke(block);
```

2. block 函数签名

```
void (^blk)(int, short, NSString *) = ^(int a, short b, NSString *str){
  NSLog(@"a:%d b:%d str:%@",a,b,str);
};
PTBlockRef block = (__bridge PTBlockRef)blk;
if (block->flags & PTBlockFlagsHasSignature) {
  void *desc = block->descriptor;
```

```
  desc += 2 * sizeof(unsigned long int);
  if (block->flags & PTBlockFlagsHasCopyDisposeHelpers) {
    desc += 2 * sizeof(void *);
  }

  const char *signature = (*(const char **)desc);
  NSMethodSignature *sig = [NSMethodSignature signatureWithObjCTypes:si
gnature];
  NSLog(@"方法 signature:%s",signature);
}

// 打印内容如下:
// v24 @?0 i8 s12 @"NSString"16
// 其中 ? 是 An unknown type (among other things, this code is used for
 function pointers)
```

3. block 捕获栈上局部变量

捕获的变量都会按照顺序放置在 PTBlock 结构体后面，如此看来就是个变长结构体。

也就是说我们可以通过如下方式知道 block 捕获了哪些外部变量（全局变量除外）。

```
int a = 0x11223344;
int b = 0x55667788;
NSString *str = @"pmst";
void (^blk)(void) = ^{
  NSLog(@"a:%d b:%d str:%@",a,b, str);
};
PTBlockRef block = (__bridge PTBlockRef)blk;
void *pt = (void *)block + sizeof(struct PTBlock);
long long *ppt = pt;
NSString *str_ref = (__bridge id)((void *)(*ppt));
int *a_ref = pt + sizeof(NSString *);
int *b_ref = pt + sizeof(NSString *) + sizeof(int);

NSLog(@"a:0x%x b:0x%x str:%@",*a_ref, *b_ref, str_ref);
```

TODO：NSString layout 布局为何在第一位？

4. __block 变量（栈上）

```
__block int a = 0x99887766;
__unsafe_unretained void (^blk)(void) = ^{
  NSLog(@"__block a :%d",a);
};
NSLog(@"Block 类型 %@",[blk class]);
PTBlockRef block = (__bridge PTBlockRef)blk;
void *pt = (void *)block + sizeof(struct PTBlock);
long long *ppt = pt;
void *ref = (PTBlock_byref_Ref)(*ppt);
```

```
void *shared = ref + sizeof(struct PTBlock_byref);
int *a_ref = (int *)shared;
NSLog(@"a 指针：%p block a 指针:%p block a value:0x%x",&a, a_ref,*a_ref);
NSLog(@"PTBlock_byref 指针：%p",ref);
NSLog(@"PTBlock_byref forwarding 指针：%p",((PTBlock_byref_Ref)ref)->fo
rwarding);
/*
输出如下：
Block 类型 __NSStackBlock__
a 指针：0x7ffeefbff528 block a 指针:0x7ffeefbff528 block a value:0x99887
766
PTBlock_byref 指针：0x7ffeefbff510
PTBlock_byref forwarding 指针：0x7ffeefbff510
*/
```

可以看到 __block int a 已经变成了另外一个数据结构了，打印地址符合预期，此刻 block 以及其他的变量结构体都在栈上。

5. __block 变量，[block copy] 后的内存变化

```
__block int a = 0x99887766;
__unsafe_unretained void (^blk)(NSString *) = ^(NSString *flag){
  NSLog(@"[%@] 中 a 地址:%p",flag, &a);
};
NSLog(@"blk 类型 %@",[blk class]);
blk(@"origin block");
void (^copyblk)(NSString *) = [blk copy];
copyblk(@"copy block");
blk(@"origin block 二次调用");
/**
    输出如下：
blk 类型 __NSStackBlock__
[origin block] 中 a 地址:0x7ffeefbff528
copyblk 类型 __NSMallocBlock__
[copy block] 中 a 地址:0x102212468
[origin block 二次调用] 中 a 地址:0x102212468
*/
```

很明显对 blk 进行 copy 操作后，copyblk 已经“移驾”到堆上，随着拷贝的还有 __block 修饰的 a 变量（PTBlock_byref_Ref 类型）；

6. __block 变量中 forwarding 指针

```
__block int a = 0x99887766;
__unsafe_unretained void (^blk)(NSString *,id) = ^(NSString *flag, id b
blk){
  NSLog(@"[%@] a address:%p",flag, &a); // a 取值都是 ->forwarding->a 方
式
  PTBlockRef block = (__bridge PTBlockRef)bblk;
```

```
  void *pt = (void *)block + sizeof(struct PTBlock);
  long long *ppt = pt;
  void *ref = (PTBlock_byref_Ref)(*ppt);
  NSLog(@"[%@] PTBlock_byref_Ref 指针：%p",flag,ref);
  NSLog(@"[%@] PTBlock_byref_Ref forwarding 指针：%p",flag,((PTBlock_by
ref_Ref)ref)->forwarding);
  void *shared = ref + sizeof(struct PTBlock_byref);
  int *a_ref = (int *)shared;
  NSLog(@"[%@] a value : 0x%x a adress:%p", flag, *a_ref, a_ref);

};
NSLog(@"blk 类型 %@",[blk class]);
blk(@"origin block", blk);
void (^copyblk)(NSString *,id) = [blk copy];
NSLog(@"copyblk 类型 %@",[copyblk class]);
copyblk(@"copy block",copyblk);
blk(@"origin block after copy", blk);
/**
MRC 模式下输出：
blk 类型 __NSStackBlock__
[origin block] a address:0x7ffeefbff528
[origin block] PTBlock_byref_Ref 指针：0x7ffeefbff510
[origin block] PTBlock_byref_Ref forwarding 指针：0x7ffeefbff510
[origin block] a value : 0x99887766 a adress:0x7ffeefbff528
copyblk 类型 __NSMallocBlock__
[copy block] a address:0x1032041d8
[copy block] PTBlock_byref_Ref 指针：0x1032041c0
[copy block] PTBlock_byref_Ref forwarding 指针：0x1032041c0
[copy block] a value : 0x99887766 a adress:0x1032041d8
[origin block after copy] a address:0x1032041d8
[origin block after copy] PTBlock_byref_Ref 指针：0x7ffeefbff510
[origin block after copy] PTBlock_byref_Ref forwarding 指针：0x1032041c
0
[origin block after copy] a value : 0x99887766 a adress:0x7ffeefbff528

ARC 模式下输出（这个稍有出路）：
blk 类型 __NSStackBlock__
[origin block] a address:0x100604cc8
[origin block] PTBlock_byref_Ref 指针：0x100604cb0
[origin block] PTBlock_byref_Ref forwarding 指针：0x100604cb0
[origin block] a value : 0x99887766 a adress:0x100604cc8
copyblk 类型 __NSMallocBlock__
[copy block] a address:0x100604cc8
[copy block] PTBlock_byref_Ref 指针：0x100604cb0
[copy block] PTBlock_byref_Ref forwarding 指针：0x100604cb0
[copy block] a value : 0x99887766 a adress:0x100604cc8
*/
```

这里可以看到 forwarding 指针确实指向了结构体本身，随着 copy 行为确实进行了一次栈->堆的赋值——`block` 和 `__block` 变量。

建议用 lldb 命令去看内存布局。

比较详细的介绍 https 的过程。

1. 客户端携带 TLS 版本，加密算法、压缩算法和随机数字 C 发送给服务端；
2. 服务端接收后选择对应的 TLS 版本、加密算法和会话 ID，然后生成随机数字 S，以及下发证书给客户端，服务端的证书链，其中包含证书支持的域名、发行方和有效期等信息；
3. 客户端验证服务端下发的证书，AFNetworking 中就是证书挑战环节，验证完同时使用 C / S 随机数字用算法生成预主密钥（Pre Master Secret），同时也生成了会话密钥（对称加密的密钥）；接着客户端将预主密钥用服务端下发的证书中的公钥进行非对称加密，然后传递给服务端；
4. 服务端用私钥解密得到预主密钥，然后得到会话密钥，接下来就是双方进行对称加密的阶段了，这里涉及到了 Change Cipher Spec 消息；

推荐 Draveness 的 HTTPS 的 7 次握手以及 9 倍时延

进一步灵魂拷问：

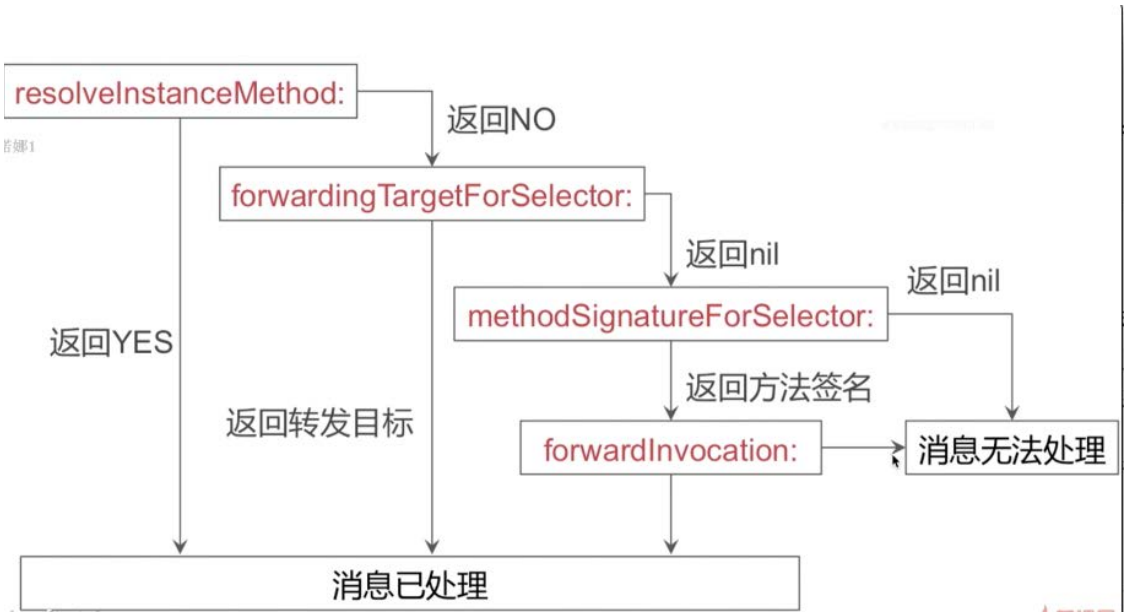
1. 会话密钥是如何生成的？其实还存在一个 Master Secret，这个是什么东东？
2. 非对称加密各个版本用的是什么算法；
3. TLS 1.2 TLS 1.3 是否熟悉 有什么不同？RTT 了解下
4. HTTPS 3.0 是否了解？ QUIC 协议为什么能保证安全性、可靠性？

过往开发中做过哪些优化向的工作，问的也比较详细。

如何检测项目中的卡顿问题（比如假死）

比较详细的介绍消息转发流程和事件响应链

OC 中的方法调用，编译后的代码最终都会转成 `objc_msgSend(id , SEL, ...)` 方法进行调用，这个方法第一个参数是一个消息接收者对象，runtime 通过这个对象的 isa 指针找到这个对象的类对象，从类对象中的 cache 中查找(**哈希查找，bucket 桶实现**)是否存在 SEL 对应的 IMP，若不存在，则会在 method_list 中查找（二分查找或者顺序查找），如果还是没找到，则会到 supper_class 中查找，仍然没找到的话，就会调用_objc_msgForward(id, SEL, ...)进行消息转发。



拓展：

1. Aspect 实现原理
2. 处理数组 字典等插入 nil 或者数组越界等问题，这里又涉及到了类簇问题，以及方法交换带来的坏处

GCD 的底层线程调度原理

介绍 hash 算法的原理

hash 算法原理详解

三面

一个二叉树逐层打印的算法题

参考文献：

- leetcode 二叉树打印专题
- hash 算法原理及应用漫谈【加图版】

如果现在做一个新的网络层框架，有哪些需要考量的点

2. 百度

1. 判断一个字符串是不是 ipv6 地址（要求尽全力的考虑所有异常的情况）

2. 介绍界面卡顿的优化有哪些可以优化的点。

从 CPU 和 GPU 两方面出发分析问题，谈几个优化点：

1. 缓存 Cell 高度，减少一些复杂的 layout，文字计算等，可进行缓存处理；

2. 异步渲染内容到图片，使用 SDWebImage + YYText 等；
3. 增量加载内容项；
4. 图片解码放到子线程处理，这里 FastImage 和 SDWebImage 都支持
5. 避免离屏渲染，可以开启 shouldRasterize 选项，但是会额外开辟 2 个屏幕内存，另外缓存一定时间后失效。

3. 介绍 UIResponder 的继承链。然后说事件响应链。

见上

3. Bigo

一面

(算法)找出一个页面中漏出部分面积最大的试图，重合的部分按照最上层的面积算漏出，会有时间复杂度的要求。

控件的点击事件和添加在上边的手势谁先响应，并说明原因

谈 CoreAnimation 和 CoreGraphic 的区别

说 @synchronized 锁的实现原理，并说明其中可能存在的问题。同时介绍了 iOS 开发中常见的锁。

底层维护了一个锁池，开销比较大，@synchronized 需要一个参数，猜测是信号量实现。

常见的锁，按类别来划分：自旋锁，`pthread_mutex_t` 互斥锁，递归锁，可抛错误的锁（通过设定 attribute 来实例化出来），条件锁，读写锁，信号量等等。

具体 iOS 用到的锁：

- @synchronized
- OSSpinLock
- os_unfair_lock
- NSLock
- `pthread_mutex_t` 互斥锁 递归锁 可抛错误的锁
- NSRecursiveLock
- dispatch_semaphore_t
- NSCodition
- NSCoditionLock
- dispatch_barrier_async / dispatch_barrier_sync
- atomic(property) set / get
- gcd dispatch_once

介绍编译的过程和原理

1. 预编译
2. 词法分析
3. 语法分析
4. 语义分析(静态分析)
5. 中间代码生成 IR
6. 中间代码优化
7. 汇编
8. 目标代码生成（.o 文件）
9. 链接
10. 可执行文件

谈对于 bitcode 的理解和作用。

详细的介绍了 Https 的过程。

见上解答。

二面

介绍属性常用修饰符，介绍 assign 和 weak 之间的区别。这块会延伸到内存管理相关，比如引用计数的方式。

常用修饰符：atomic 、nonatomic、assign、weak、strong、copy、retain、getter、setter、readonly 等等

assign 通常用于修饰基础数据类型，weak 修饰对象，引用计数都不加一，且后者会在对象释放后自动置为 nil。实现原理：

1.weak 的实现原理？SideTable 的结构是什么样的

```
NSObject *p = [[NSObject alloc] init];
__weak NSObject *p1 = p;
// ====> 底层是runtime 的 objc_initWeak
// xcrun -sdk iphoneos clang -arch arm64 -rewrite-objc -fobjc-arc -fobj
c-runtime=ios-13.2 main.m 得不到下面的代码，还是说命令参数不对。
NSObject objc_initWeak(&p, 对象指针);
```

通过 runtime 源码可以看到 objc_initWeak 实现：

```
id
objc_initWeakOrNil(id *location, id newObj)
{
    if (!newObj) {
        *location = nil;
        return nil;
```

```
    }

    return storeWeak<DontHaveOld, DoHaveNew, DontCrashIfDeallocating>
        (location, (objc_object*)newObj);
}
```

SideTable 结构体在 runtime 底层用于引用计数和弱引用关联表，其数据结构是这样：

```
struct SideTable {
    // 自旋锁
    spinlock_t slock;
    // 引用计数
    RefcountMap refcnts;
    // weak 引用
    weak_table_t weak_table;
}

struct weak_table_t {
    // 保存了所有指向指定对象的 weak 指针
    weak_entry_t *weak_entries;
    // 存储空间
    size_t    num_entries;
    // 参与判断引用计数辅助量
    uintptr_t mask;
    // hash key 最大偏移值
    uintptr_t max_hash_displacement;
};
```

根据对象的地址在缓存中取出对应的 SideTable 实例：

```
static SideTable *tableForPointer(const void *p)
```

或者如上面源码中 &SideTables()[newObj] 方式取表，**这里的 newObj 是实例对象用其指针作为 key 拿到 从全局的 SideTables 中拿到实例自身对应的那张 SideTable**。

```
static StripedMap<SideTable>& SideTables() {
    return *reinterpret_cast<StripedMap<SideTable>*>(SideTableBuf);
}
```

取出实例方法的实现中，使用了 C++ 标准转换运算符 **reinterpret_cast**，其表达方式为：

```
reinterpret_cast <new_type> (expression)
```

指向某个对象 A 的所有 weak 关键字修饰的引用都 append 到 weak_entry_t 结构体中的 referrers， 同时 weak_entry_t 的 referent 就是对象 A，之后在 dealloc

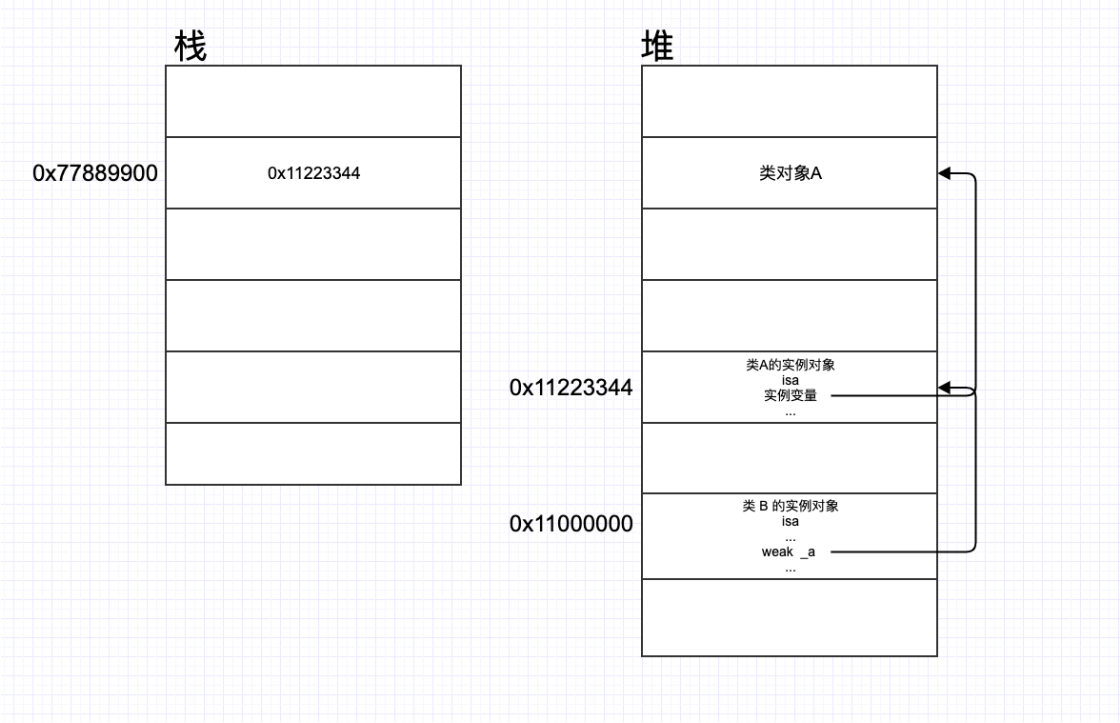
释放时遍历 weak_table 遍历时会判断 referent 是否为对象 A 取到 weak_entry_t，加入到该 SideTable 中 weak_table 中：

weak 底层实现其实不难，不同架构下 SideTable 表数量也是不同的（8 和 64）,所以用对象指针作为 Key，存在键值冲突的问题，因此在设计上也要解决该问题，源码有体现。

```
typedef objc_object ** weak_referrer_t;

struct weak_entry_t {
    DisguisedPtr<objc_object> referent;
    union {
        struct {
            weak_referrer_t *referrers;
            uintptr_t        out_of_line : 1;
            uintptr_t        num_refs : PTR_MINUS_1;
            uintptr_t        mask;
            uintptr_t        max_hash_displacement;
        };
        struct {
            // out_of_line=0 is LSB of one of these (don't care which)
            weak_referrer_t  inline_referrers[WEAK_INLINE_COUNT];
        };
 }
```

旧对象解除注册操作 weak_unregister_no_lock 和 新对象添加注册操作 weak_register_no_lock ，具体实现可前往 runtime 源码中查看或查看瓜的博文。



weak 关键字修饰的对象有两种情况：栈上和堆上。上图主要解释 id referent_id 和 id *referrer_id，

- 如果是栈上， referrer 值为 0x77889900，referent 值为 0x11223344
- 如果是堆上， referrer 值为 0x1100000+ offset（也就是 weak a 所在堆上的地址），referent 值为 0x11223344。

如此现在类 A 的实例对象有两个 weak 变量指向它，一个在堆上，一个在栈上。

```
void
weak_unregister_no_lock(weak_table_t *weak_table, id referent_id,
                        id *referrer_id)
{
    objc_object *referent = (objc_object *)referent_id;   //  0x11223344
    objc_object **referrer = (objc_object **)referrer_id; //  0x77889900

    weak_entry_t *entry;

    if (!referent) return;

    // 从 weak_table 中找到 referent 也就是上面类A 的实例对象
    if ((entry = weak_entry_for_referent(weak_table, referent))) {
        // 在 entry 结构体中的 referrers 数组中找到指针 referrer 所在位置
        // 将原本存储 referrer 值的位置置为 nil，相当于做了一个解绑操作
        // 因为 referrer 要和其他对象建立关系了
        remove_referrer(entry, referrer);
        bool empty = true;
        if (entry->out_of_line()  &&  entry->num_refs != 0) {
            empty = false;
        }
        else {
            for (size_t i = 0; i < WEAK_INLINE_COUNT; i++) {
                if (entry->inline_referrers[i]) {
                    empty = false;
                    break;
                }
            }
        }

        if (empty) {
            weak_entry_remove(weak_table, entry);
        }
    }

    // Do not set *referrer = nil. objc_storeWeak() requires that the
    // value not change.
}
```

weak 关键字修饰的属性或者变量为什么在对应类实例 dealloc 后会置为 nil，那是因为在类实例释放的时候，dealloc 会从全局的引用计数和 weak 计数表 sideTables 中，通过实例地址去找到属于自己的那张表，表中的 weak_table->weak_entries 存储了所有 entry 对象——其实就是所有指向这个实例对象的变量，`weak_entry_t` 中的 `referrers` 数组存储的就是变量或属性的内存地址，逐一置为 nil 即可。

聊对于 GCD 的理解，和 GCD 底层是如何进行线程调度的。聊 GCD 中常见方法的使用（group，信号量，barrier 等）

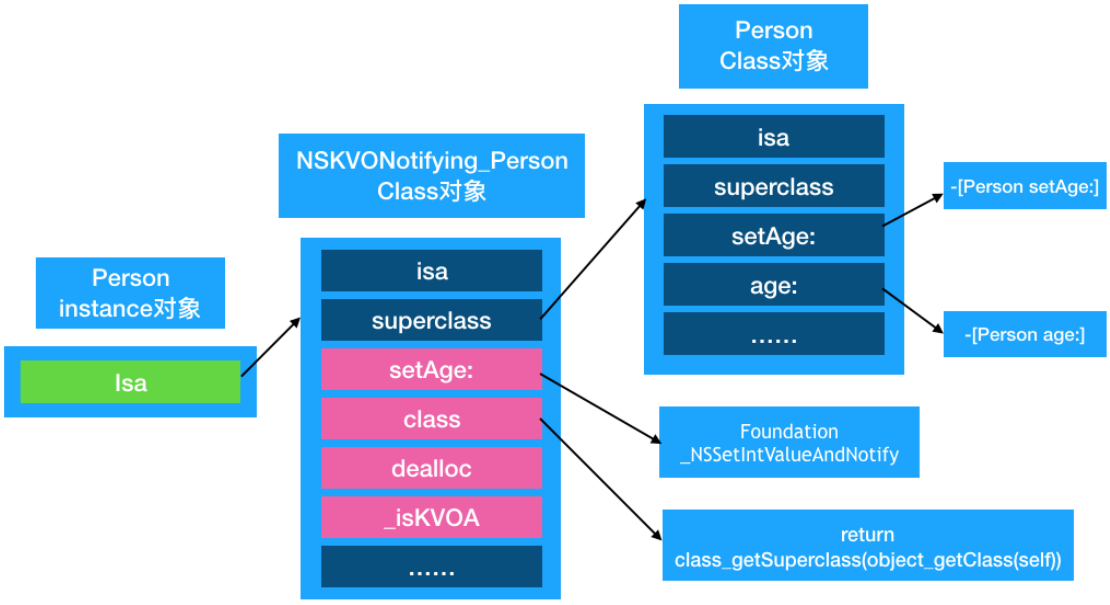
详细的介绍了 KVC 和 KVO 的原理。

KVO

同 `runloop` 一样，这也是标配的知识点了，同样列出几个典型问题

1. 实现原理

KVO 会为需要 observed 的对象动态创建一个子类，以 `NSKVONotifying_` 最为前缀，然后将对象的 isa 指针指向新的子类，同时重写 class 方法，返回原先类对象，这样外部就无感知了；其次重写所有要观察属性的 setter 方法，统一会走一个方法，然后内部是会调用 `willChangeValueForKey` 和 `didChangevlueForKey` 方法，在一个被观察属性发生改变之前，`willChangeValueForKey:`一定会被调用，这就 会记录旧的值。而当改变发生后，`didChangeValueForKey:`会被调用，继而 `observeValueForKey:ofObject:change:context:` 也会被调用。



图片出处 https://juejin.im/post/5adab70cf265da0b736d37a8

那么如何验证上面的说法呢？很简单，借助 runtime 即可，测试代码请点击 这里：

```
- (void)viewDidLoad {
    [super viewDidLoad];
    self.person = [[Person alloc] initWithName:@"pmst" age:18];
    self.teacher = [[Teacher alloc] initWithName:@"ppp" age:28];
    self.teacher.work = @"数学";
    self.teacher.numberOfStudent = 10;

    NSKeyValueObservingOptions options = NSKeyValueObservingOptionNew |
 NSKeyValueObservingOptionOld;

    RuntimeUtil *utils = [RuntimeUtil new];
    [utils logClassInfo:self.person.class];
    [self.person addObserver:self forKeyPath:@"age" options:options con
text:nil];
    [utils logClassInfo:object_getClass(self.person)];


    [utils logClassInfo:self.teacher.class];
    [self.teacher addObserver:self forKeyPath:@"age" options:options co
ntext:nil];
    [self.teacher addObserver:self forKeyPath:@"name" options:options c
ontext:nil];
    [self.teacher addObserver:self forKeyPath:@"work" options:options c
ontext:nil];
    [utils logClassInfo:object_getClass(self.teacher)];
}
```

这里 object_getClass() 方法实现也贴一下，如果直接使用 .class 那么因为被重写过，返回的还是原先对象的类对象，而直接用 runtime 方法的直接返回了 isa 指针。

```
Class object_getClass(id obj)
{
    if (obj) return obj->getIsa();
    else return Nil;
}
```

通过日志确实可以看到子类重写了对应属性的 setter 方法：

```
2020-03-25 23:11:00.607820+0800 02-25-KVO[28370:1005147] LOG:(NSKVONoti
fying_Teacher) INFO
2020-03-25 23:11:00.608190+0800 02-25-KVO[28370:1005147] ==== OUTPUT:NS
KVONotifying_Teacher properties ====
2020-03-25 23:11:00.608529+0800 02-25-KVO[28370:1005147] ==== OUTPUT:NS
KVONotifying_Teacher Method ====
2020-03-25 23:11:00.608876+0800 02-25-KVO[28370:1005147] method name:se
tWork:
2020-03-25 23:11:00.609219+0800 02-25-KVO[28370:1005147] method name:se
tName:
2020-03-25 23:11:00.646713+0800 02-25-KVO[28370:1005147] method name:se
```

```
tAge:
2020-03-25 23:11:00.646858+0800 02-25-KVO[28370:1005147] method name:cl
ass
2020-03-25 23:11:00.646971+0800 02-25-KVO[28370:1005147] method name:de
alloc
2020-03-25 23:11:00.647088+0800 02-25-KVO[28370:1005147] method name:_i
sKVOA
2020-03-25 23:11:00.647207+0800 02-25-KVO[28370:1005147] ==============
===========
```

疑惑点：看到有文章提出 KVO 之后，setXXX 方法转而调用 _NSSetBoolValueAndNotify、_NSSetCharValueAndNotify、_NSSetFloatValueAndNotify、_NSSetLongValueAndNotify 等方法，但是通过 runtime 打印 method 是存在的，猜测 SEL 是一样的，但是 IMP 被换掉了，关于源码的实现还未找到。TODO 下。

2. 如何手动关闭 kvo

KVO 和 KVC 相关接口太多，实际开发中直接查看接口文档即可。

```
+(BOOL)automaticallyNotifiesObserversForKey:(NSString *)key{
    if ([key isEqualToString:@"name"]) {
        return NO;
    }else{
        return [super automaticallyNotifiesObserversForKey:key];
    }
}

-(void)setName:(NSString *)name{

    if (_name!=name) {

        [self willChangeValueForKey:@"name"];
        _name=name;
        [self didChangeValueForKey:@"name"];
    }

}
```

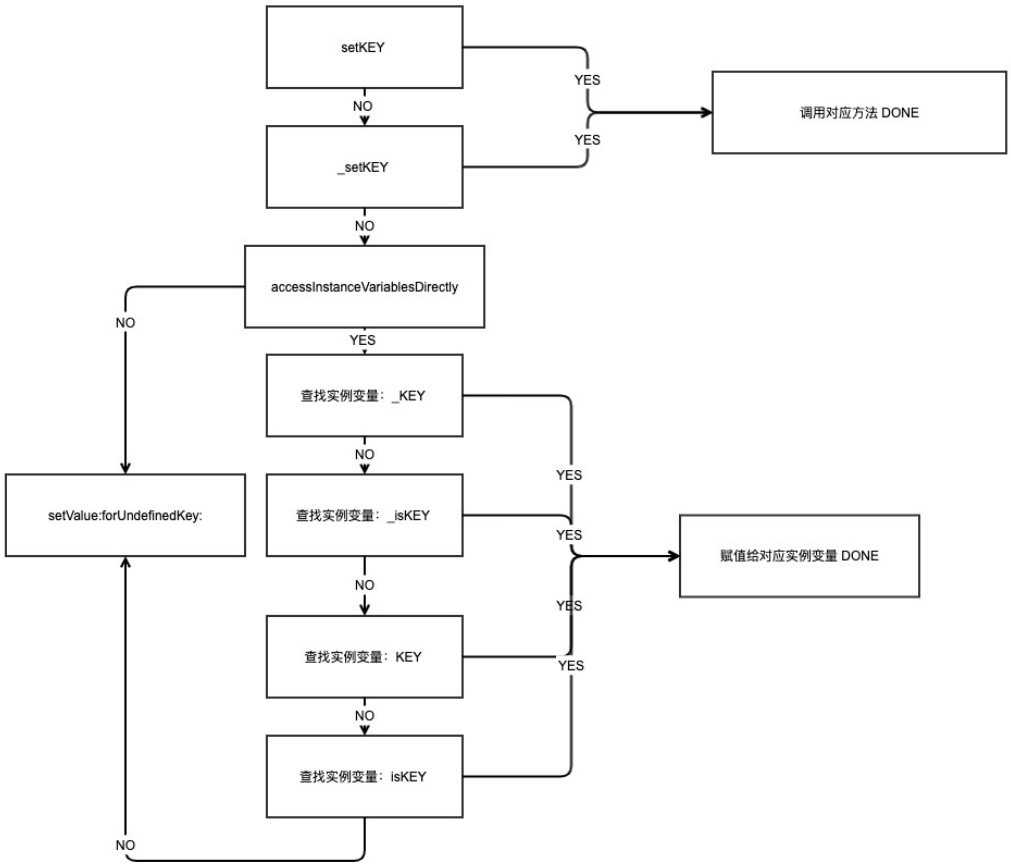
3. 通过 KVC 修改属性会触发 KVO 么

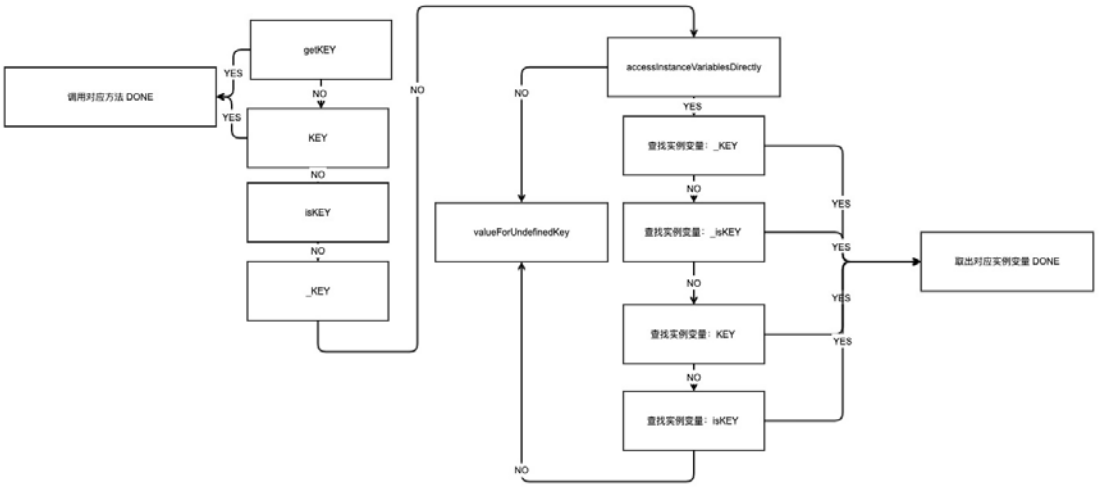
会触发 KVO 操作，KVC 时候会先查询对应的 getter 和 setter 方法，如果都没找到，调用

```
+ (BOOL)accessInstanceVariablesDirectly {
    return NO;
}
```

如果返回 YES，那么可以直接修改实例变量。

- KVC 调用 getter 流程：`getKEY`，`KEY`，`isKEY`，`_KEY`，接着是实例变量 `_KEY`,`_isKEY`，`KEY`，`isKEY`;
- KVC 调用 setter 流程：`setKEY` 和 `_setKEY`，实例变量顺序 `_KEY`,`_isKEY`，`KEY`，`isKEY`，没找到就调用 `setValue：forUndefinedKey:`





4. 哪些情况下使用 kvo 会崩溃，怎么防护崩溃

1. dealloc 没有移除 kvo 观察者，解决方案：创建一个中间对象，将其作为某个属性的观察者，然后 dealloc 的时候去做移除观察者，而调用者是持有中间对象的，调用者释放了，中间对象也释放了，dealloc 也就移除观察者了；
2. 多次重复移除同一个属性，移除了未注册的观察者
3. 被观察者提前被释放，被观察者在 dealloc 时仍然注册着 KVO，导致崩溃。 例如：被观察者是局部变量的情况（iOS 10 及之前会崩溃）比如 weak；
4. 添加了观察者，但未实现 `observeValueForKeyPath:ofObject:change:context:`方法，导致崩溃；
5. 添加或者移除时 `keypath == nil`，导致崩溃；

以下解决方案出自 iOS 开发：『Crash 防护系统』（二）KVO 防护 一文。

解决方案一：

FBKVOController 对 KVO 机制进行了额外的一层封装，框架不但可以自动帮我们移除观察者，还提供了 block 或者 selector 的方式供我们进行观察处理。不可否认的是，FBKVOController 为我们的开发提供了很大的便利性。但是相对而言，这种方式对项目代码的侵入性比较大，必须依靠编码规范来强制约束团队人员使用这种方式。

解决方案二：

1. 首先为 NSObject 建立一个分类，利用 Method Swizzling，实现自定义的 `BMP_addObserver:forKeyPath:options:context:`、`BMP_removeObserver:forKeyPath:`、`BMP_removeObserver:forKeyPath:context:`、`BMPKVO_dealloc` 方法，用来替换系统原生的添加移除观察者方法的实现。

2. 然后在观察者和被观察者之间建立一个 KVODelegate 对象，两者之间通过 KVODelegate 对象 建立联系。然后在添加和移除操作时，将 KVO 的相关信息例如 observer、keyPath、options、context 保存为 KVOInfo 对象，并添加到 KVODelegate 对象 中对应 的 关系哈希表 中，对应原有的添加观察者。 关系哈希表的数据结构：{keypath : [KVOInfo 对象 1, KVOInfo 对象 2, ...]}

3. 在添加和移除操作的时候，利用 KVODelegate 对象 做转发，把真正的观察者变为 KVODelegate 对象，而当被观察者的特定属性发生了改变，再由 KVODelegate 对象 分发到原有的观察者上。

4. **添加观察者时**：通过关系哈希表判断是否重复添加，只添加一次。
5. **移除观察者时**：通过关系哈希表是否已经进行过移除操作，避免多次移除。
6. **观察键值改变时**：同样通过关系哈希表判断，将改变操作分发到原有的观察者上。

解决方案三：

XXShield 实现方案和 BayMax 系统类似。也是利用一个 Proxy 对象用来做转发，真正的观察者是 Proxy，被观察者出现了通知信息，由 Proxy 做分发。不过不同点是 Proxy 里面保存的内容没有前者多。只保存了 _observed（被观察者） 和关系哈希表，这个关系哈希表中只维护了 keyPath 和 observer 的关系。

关系哈希表的数据结构：{keypath : [observer1, observer2 , ...](NSHashTable)} 。

XXShield 在 dealloc 中也做了类似将多余观察者移除掉的操作，是通过关系数据结构和 _observed ，然后调用原生移除观察者操作实现的。

5. kvo 的优缺点

优点：

1. 运用了设计模式：**观察者模式**
2. 支持**多个观察者观察同一属性**，或者**一个观察者监听不同属性**。
3. 开发人员不需要实现属性值变化了发送通知的方案，系统已经封装好了，大大减少开发工作量；
4. 能够对非我们创建的对象，即内部对象的状态改变作出响应，而且不需要改变内部对象（SDK 对象）的实现；
5. 能够提供观察的属性的最新值以及先前值；
6. 用 key paths 来观察属性，因此也可以观察嵌套对象；
7. 完成了对观察对象的抽象，因为不需要额外的代码来允许观察值能够被观察

缺点：

1. 观察的属性键值硬编码（字符串），编译器不会出现警告以及检查；
2. 由于允许对一个对象进行不同属性观察，所以在唯一回调方法中，会出现地狱式 `if-else if - else` 分支处理情况；

References：

- iOS 底层原理总结篇– 深入理解 KVC 实现机制
- iOS 开发：『Crash 防护系统』（二）KVO 防护
- ValiantCat / **XXShield**（第三方框架）
- JackLee18 / **JKCrashProtect**（第三方框架）
- [大白健康系统 – iOS APP 运行时 Crash 自动修复系统](

介绍消息转发过程

解答见上

介绍对于 Runloop 并介绍知道的应用场景。再具体场景中会有追问。

解答见上

介绍对于静态库和动态库的理解。

Q1: 静态库和动态库是什么？ A1: 可复用代码我们会封装成函数，多个这样的函数我们会整合到一个或多个源文件作为工具“库”，比如我们会把这些源文件放置到一个 Utility 文件夹，然后在一个头文件中，把允许外部调用的方法都声明在此处；进一步考虑，这些可复用函数依赖关系甚少，起码和业务无关，因此我们可以提前对这些源文件编译成二进制可执行文件，当工程需要用到的时候我们引入这个工具二进制文件即可，而工程会根据前面说到的头文件中声明的函数进行调用，最后整个工程在编译的时候，由于我们工具库已经编译成二进制文件了，因此无须编译，但是在 link 的时候需要将二进制文件和目标文件(比如 `.o`, `.obj`)link 到一起。

Q2: 二进制文件是什么？ A2: 所谓编译就是把源文件.c .m 等高级语言写的程序经过编译器一系列处理步骤（预处理->词法分析->语法分析->语义分析->中间语言 IR 生成->目标代码生成与优化（生成的目标代码是 asm 汇编）-> 转成机器语言，也就是 CPU 指令)），这么说来目标文件就是存储了 0，1 的数据块，关键看 CPU 如何解释这些 0，1，是指令呢还是数据。

Q3: 何为 Link? A3: 编译好的二进制文件最后会和其他目标文件 link 链接整合成一个二进制文件。简单来说，将一个个二进制文件按照一定顺序整合放置到一起，每个二进制文件都不知道自己会放置在那个地址，但是二进制内部的指标是确定的，因此我们会设置一个默认的地址 0x0000000，而内部都是相对地址，当 link 的时候我们会把预留的默认地址改成正确的地址。

Q4: 静态库和动态库的区别？ A4: 两者都是预先编译好的，在编译整个工程的时候，只对导入的静态库和动态库进行 link 操作；两者的不同时，静态库在编译的时候会 copy 一份到最后的目标二进制执行文件，而动态库则不会，而只存储指向动态库的引用。

Q5: 静态库和动态库在各平台上的类型是什么？ A5: window 下静态库`.lib`，动态库 `dll`；unix 和类 unix 系统下静态库 `.a`，动态库`.so`；mac 系统下静态库`.a` 和 `.framework`，动态库 `.dylib/.tbd`

Q6：iOS 的 framework 是什么呢？ A6: framework 其实是 iOS/mac 平台自有的一种格式，其对二进制库，头文件和资源进行了封装，便于分发和管理。系统的 framework 都是动态库，而自己开发的 framework 无论是动态还是静态，都是要复制到目标程序的，如此看来，貌似自己开发的都是“静态库”？？？苹果称之为 embedded framework。

Q7: armv7,armv7s,i386,x86_64 架构，为何常说静态库包含了多种架构？ A7: 上述都是指 CPU 的结构，支持的指令集。iPhone 的硬件配置几乎每年都有更新，其中 armv7 可简单理解为支持 iPhone5 之前的设备；而 armv7s 支持 iPhone5 之后的设备，arm64 是更新的设备；i386 和 x86_64 都是指 PC 的架构，前者是旧的 mac，32 位的，后者是 64 位。

Q8: 如何查看一个静态库包含哪些架构： A8: 使用 `lipo -info library` 命令，其他操作还有从已有库中移除/添加/瘦包/，更多使用可用 `man lipo` 查看。

在 webview 使用过程中存在的问题和解决方案。

三面

介绍了过往 RN 的使用经验和对于 Flutter 的理解。

- RN write once run anywhere，想法是美好的，但现实总是残酷，总是会遇到适配两端的情况，而且问题比较复杂，解决问题的时间成本非常高，而且必须要对 android/iOS 要熟悉了解，所以还是没有让前端开发与原生隔离；
- RN 中原生和 js 的交互都是由原生端主动发起的，不管是 js 调用原生还是原生调用 js。js 调用原生总是写消息到队列中，原生端定时（5ms）读取队列中的消息进行 flush 操作，所以一定存在通信上的损耗，不过一旦超时未处理 js 会主动调用原生端；
- RN 的热更新是绝对优势
- Flutter 渲染是借助 VSync 信号，但实际上还是通过 UIKit 层的接口来触发这个事件，似乎并没有借助所谓的系统接口

谈对于组件化的理解和市面上常见的组件化方案

- CTMediator
- Beehive
- MGJRouter

问了一些 APM 向上的问题。

谈个人对于项目架构选择的理解。自己如何进行架构的选择（主要对于 MVVM，MVC 等，后文有个人对于这一块的理解）

滴滴

一面

谈属性修饰符，如果 assign 修饰对象可能存在的问题和原因。

引用计数，野指针

比较的深入的聊了内存管理的内容，包含引用计数和 weak 修饰的对象的内存管理的过程。问的会比较深入。

见上

讲 runloop 的过往使用经验。

见上

介绍自己比较熟悉的三方库的实现原理

- AFNetworking
- SDWebimage
- Aspect

二面

对于锁的理解（自旋锁和互斥锁），以及 iOS 开发中常见的锁。同时要求介绍个人在开发过程中在哪些场景下用到过锁。

- 自旋锁就是忙等，轻量级，用户态
- 互斥锁即内核态，开销比较大

常见的锁见上解答

在实际开发中遇到过哪些多线程问题以及如何进行解决的。

为什么不能在异步线程中更新页面，介绍原因。

对于内存泄漏的了解，以及介绍知道的解决方案。

一个坦克从一个空间的起点到终点，中间在某些位置上有阻隔的情况下，判断从起点到终点是否有可行路径的算法题。

快手

一面

聊了 assign 修饰对象可能存在的问题

见上

介绍消息转发流程

见上

二面

比较详细的聊到的 block，深入的讲了其中的实现原理，并介绍不同变量的引用方式。

见上

介绍开发中常见的循环引用，并说明其中的原因和解决的方案和原理。

循环引用就是形成了 retainCycle

- self 持有 block，block 又持有 self
- NSTimer、CADisplayLink、RunLoop 三者

解决方案无非就是 weak 修饰，手动打破 retainCycle。

介绍 **Runloop** 并讲应用场景。

见上

二叉树翻转

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */


struct TreeNode* invertTree(struct TreeNode* root){
    if(root == NULL)return NULL;

    struct TreeNode *left = invertTree(root->left);
    struct TreeNode *right = invertTree(root->right);
    root->left = right;
    root->right = left;

    return root;
}
```

三面

一道多线程实际场景下的问题，要求远程写出实现方案的代码

聊对于 **MVVM**，**MVC** 和 **MVP** 的理解。

介绍过往项目中 **RN** 的使用经验和遇到的问题。

讲如何将一张内存极大的图片可以像地图一样的加载出来（只说实现思路）

聊对于组件化的理解，对于市面上的组件化方案的理解，优劣分析等。