内存管理面试题

一、在 Obj-C 中，如何检测内存泄漏？你知道哪些方式？

目前我知道的方式有以下几种

- Memory Leaks
- Alloctions
- Analyse
- Debug Memory Graph
- MLeaksFinder

泄露的内存主要有以下两种：

- Laek Memory 这种是忘记 Release 操作所泄露的内存。
- Abandon Memory 这种是循环引用，无法释放掉的内存。

上面所说的五种方式，其实前四种都比较麻烦，需要不断地调试运行，第五种是腾讯阅读团队出品，效果好一些

二、在 MRC 下如何重写属性的 Setter 和 Getter_.md

setter

```
-(void)setBrand:(NSString *)brand{
//如果实例变量指向的地址和参数指向的地址不同
    if (_brand != brand)
    {
        //将实例变量的引用计数减一
        [_brand release];
       //将参数变量的引用计数加一,并赋值给实例变量
        _brand = [brand retain];
    }
}
```

getter

```
-(NSString *)brand{
    //将实例变量的引用计数加1后,添加自动减1
    //作用,保证调用getter方法取值时可以取到值的同时在完全不需要使用后释放
    return [[_brand retain] autorelease];
}
```

重写 dealloc

```
//MRC下 手动释放内存 可重写dealloc但不要调用dealloc  会崩溃
-(void)dealloc{
    [_string release];
    //必须最后调用super dealloc
    [super  dealloc];
}
```

三、循环引用

循环引用的实质：多个对象相互之间有强引用，不能释放让系统回收。

如何解决循环引用？

1、避免产生循环引用，通常是将 strong 引用改为 weak 引用。
比如在修饰属性时用 weak
在 block 内调用对象方法时，使用其弱引用，这里可以使用两个宏

```
#define WS(weakSelf)           __weak __typeof(&*self)weakSelf = self; // 弱引用

#define ST(strongSelf)         __strong __typeof(&*self)strongSelf = weakSelf; //使用这个要先声明weakSelf
```

还可以使用__block 来修饰变量
在 MRC 下，__block 不会增加其引用计数，避免了循环引用
在 ARC 下，__block 修饰对象会被强引用，无法避免循环引用，需要手动解除。

2、在合适时机去手动断开循环引用。

通常我们使用第一种。

1、代理(delegate)循环引用属于相互循环引用

delegate 是 iOS 中开发中比较常遇到的循环引用，一般在声明 delegate 的时候都要使用弱引用 weak,或者 assign,当然怎么选择使用 assign 还是 weak，MRC 的话只能用 assign，在 ARC 的情况下最好使用 weak，因为 weak 修饰的变量在释放后自动指向 nil，防止野指针存在

2、NSTimer 循环引用属于相互循环使用

在控制器内，创建 NSTimer 作为其属性，由于定时器创建后也会强引用该控制器对象，那么该对象和定时器就相互循环引用了。
如何解决呢？
这里我们可以使用手动断开循环引用：
如果是不重复定时器，在回调方法里将定时器 invalidate 并置为 nil 即可。
如果是重复定时器，在合适的位置将其 invalidate 并置为 nil 即可

3、block 循环引用

一个简单的例子：

```
@property (copy, nonatomic) dispatch_block_t myBlock;
@property (copy, nonatomic) NSString *blockString;

- (void)testBlock {
    self.myBlock = ^() {
        NSLog(@"%@",self.blockString);
    };
}
```

由于 block 会对 block 中的对象进行持有操作,就相当于持有了其中的对象，而如果此时 block 中的对象又持有了该 block，则会造成循环引用。
解决方案就是使用__weak 修饰 self 即可

```
__weak typeof(self) weakSelf = self;

self.myBlock = ^() {
        NSLog(@"%@",weakSelf.blockString);
 };
```

- 并不是所有 block 都会造成循环引用。
  只有被强引用了的 block 才会产生循环引用
  而比如 dispatch_async(dispatch_get_main_queue(), ^{}),[UIView animateWithDuration:1 animations:^{}]这些系统方法等
  或者 block 并不是其属性而是临时变量,即栈 block

```
[self testWithBlock:^{
    NSLog(@"%@",self);
}];

- (void)testWithBlock:(dispatch_block_t)block {
    block();
}
```

还有一种场景，在 block 执行开始时 self 对象还未被释放，而执行过程中，self 被释放了，由于是用 weak 修饰的，那么 weakSelf 也被释放了，此时在 block 里访问 weakSelf 时，就可能会发生错误(向 nil 对象发消息并不会崩溃，但也没任何效果)。
对于这种场景，应该在 block 中对 对象使用__strong 修饰，使得在 block 期间对 对象持有，block 执行结束后，解除其持有。

```
__weak typeof(self) weakSelf = self;

self.myBlock = ^() {

        __strong __typeof(self) strongSelf = weakSelf;

        [strongSelf test];
 };
```

四、说一下什么是 悬垂指针？什么是 野指针?

悬垂指针

指针指向的内存已经被释放了，但是指针还存在，这就是一个 悬垂指针 或者说 迷途指针

野指针

没有进行初始化的指针，其实都是 野指针

五、说一下对 retain,copy,assign,weak,_Unsafe_Unretain 关键字的理解

Strong

Strong 修饰符表示指向并持有该对象，其修饰对象的引用计数会加 1。该对象只要引用计数不为 0 就不会被销毁。当然可以通过将变量强制赋值 nil 来进行销毁。

Weak

weak 修饰符指向但是并不持有该对象，引用计数也不会加 1。在 Runtime 中对该属性进行了相关操作，无需处理，可以自动销毁。weak 用来修饰对象，多用于避免循环引用的地方。weak 不可以修饰基本数据类型。

assign

assign 主要用于修饰基本数据类型，
例如 NSInteger，CGFloat，存储在栈中，内存不用程序员管理。assign 是可以修饰对象的，但是会出现问题。

copy

copy 关键字和 strong 类似，copy 多用于修饰有可变类型的不可变对象 NSString,NSArray,NSDictionary 上。

__unsafe_unretain

__unsafe_unretain 类似于 weak ，但是当对象被释放后，指针已然保存着之前的地址，被释放后的地址变为 僵尸对象，访问被释放的地址就会出问题，所以说他是不安全的。

__autoreleasing

将对象赋值给附有 __autoreleasing 修饰的变量等同于 ARC 无效时调用对象的 autorelease 方法,实质就是扔进了自动释放池。

五、是否了解 深拷贝 和 浅拷贝 的概念，集合类深拷贝如何实现

简而言之：

1、对不可变的非集合对象，copy 是指针拷贝，mutablecopy 是内容拷贝

2、对于可变的非集合对象，copy，mutablecopy 都是内容拷贝

3、对不可变的数组、字典、集合等集合类对象，copy 是指针拷贝，mutablecopy 是内容拷贝

4、对于可变的数组、字典、集合等集合类对象，copy，mutablecopy 都是内容拷贝

但是，对于集合对象的内容复制仅仅是对对象本身，但是对象的里面的元素还是指针复制。要想复制整个集合对象，就要用集合深复制的方法，有两种：

（1）使用 initWithArray:copyItems:方法，将第二个参数设置为 YES 即可

```
NSDictionary shallowCopyDict = [[NSDictionary alloc] initWithDictionary:someDictionary copyItems:YES];
```

（2）将集合对象进行归档（archive）然后解归档（unarchive）：

```
NSArray *trueDeepCopyArray = [NSKeyedUnarchiver unarchiveObjectWithData:[NSKeyedArchiver
archivedDataWithRootObject:oldArray]];
```

六、使用自动引用计数应遵循的原则

- 不能使用 retain、release、retainCount、autorelease。
- 不可以使用 NSAllocateObject、NSDeallocateObject。
- 必须遵守内存管理方法的命名规则。
- 不需要显示的调用 Dealloc。
- 使用 @autoreleasePool 来代替 NSAutoreleasePool。
- 不可以使用区域 NSZone。
- 对象性变量不可以作为 C 语言的结构体成员。
- 显示转换 id 和 void*。

七、能不能简述一下 Dealloc 的实现机制

Dealloc 的实现机制是内容管理部分的重点，把这个知识点弄明白，对于全方位的理解内存管理的只是很有必要。

1. Dealloc 调用流程

- 1.首先调用 _objc_rootDealloc()
- 2.接下来调用 rootDealloc()
- 3.这时候会判断是否可以被释放，判断的依据主要有 5 个，判断是否有以上五种情况

    - NONPointer_ISA
    - weakly_reference
    - has_assoc
    - has_cxx_dtor
    - has_sidetable_rc

- 4-1.如果有以上五中任意一种，将会调用 object_dispose()方法，做下一步的处理。
- 4-2.如果没有之前五种情况的任意一种，则可以执行释放操作，C 函数的 free()。
- 5.执行完毕。

2. object_dispose() 调用流程。

- 1.直接调用 objc_destructInstance()。
- 2.之后调用 C 函数的 free()。

3. objc_destructInstance() 调用流程

- 1.先判断 hasCxxDtor，如果有 C++ 的相关内容，要调用 object_cxxDestruct() ，销毁 C++ 相关的内容。
- 2.再判断 hasAssocitatedObjects，如果有的话，要调用 object_remove_associations()，销毁关联对象的一系列操作。
- 3.然后调用 clearDeallocating()。
- 4.执行完毕。

4. clearDeallocating() 调用流程。

- 1.先执行 sideTable_clearDellocating()。
- 2.再执行 weak_clear_no_lock,在这一步骤中，会将指向该对象的弱引用指针置为 nil。
- 3.接下来执行 table.refcnts.eraser()，从引用计数表中擦除该对象的引用计数。
- 4.至此为止，Dealloc 的执行流程结束。

八、内存中的 5 大区分别是什么？

- 栈区（stack）：由编译器自动分配释放 ，存放函数的参数值，局部变量的值等。其 操作方式类似于数据结构中的栈。
- 堆区（heap）：一般由程序员分配释放， 若程序员不释放，程序结束时可能由 OS 回收 。注意它与数据结构中的堆是两回事，分配方式倒是类似于链表。
- 全局区（静态区）（static）：全局变量和静态变量的存储是放在一块的，初始化的 全局变量和静态变量在一块区域， 未初始化的全局变量和未初始化的静态变量在相邻的另一块区域。 - 程序结束后由系统释放。
- 文字常量区：常量字符串就是放在这里的。 程序结束后由系统释放。
- 程序代码区：存放函数体的二进制代码。

九、内存管理默认的关键字是什么？

MRC

```
@property (atomic,readWrite,retain) UIView *view;
```

ARC

```
@property (atomic,readWrite,strong) UIView *view;
```

如果改为基本数据类型，那就是 assign。

十、内存管理方案

- taggedPointer ：存储小对象如 NSNumber。深入理解 Tagged Pointer
- NONPOINTER_ISA(非指针型的 isa):在 64 位架构下，isa 指针是占 64 比特位的，实际上只有 30 多位就已经够用了，为了提高利用率，剩余的比特位存储了内存管理的相关数据内容。
- 散列表：复杂的数据结构，包括了引用计数表和弱引用表
  通过 SideTables()结构来实现的，SideTables()结构下，有很多 SideTable 的数据结构。
  而 sideTable 当中包含了自旋锁，引用计数表，弱引用表。
  SideTables()实际上是一个哈希表，通过对象的地址来计算该对象的引用计数在哪个 sideTable 中。

自旋锁：

- 自旋锁是“忙等”的锁。
- 适用于轻量访问。

引用计数表和弱引用表实际是一个哈希表，来提高查找效率。

十一、内存布局

- 栈(stack):方法调用，局部变量等，是连续的，高地址往低地址扩展
- 堆(heap):通过 alloc 等分配的对象，是离散的，低地址往高地址扩展，需要我们手动控制
- 未初始化数据(bss):未初始化的全局变量等
- 已初始化数据(data):已初始化的全局变量等
- 代码段(text):程序代码

2、64bit 和 32bit 下 long 和 char 所占字节是不同的*

- char：1 字节（ASCII 2 = 256 个字符）
- char*（即指针变量）:4 个字节（32 位的寻址空间是 2,即 32 个 bit，也就是 4 个字节。同理 64 位编译器为 8 个字节）
- short int : 2 个字节 范围 -2～> 2 即 -32768～>32767
- int： 4 个字节 范围 -2147483648～>2147483647
- unsigned int : 4 个字节
- long: 4 个字节 范围 和 int 一样 64 位下 8 个字节，范围 -9223372036854775808～9223372036854775807
- long long: 8 个字节 范围-9223372036854775808～9223372036854775807
- unsigned long long: 8 个字节 最大值：1844674407370955161
- float: 4 个字节
- double: 8 个字节

3、static、const 和 sizeof 关键字

static 关键字

答：Static 的用途主要有两个，一是用于修饰存储类型使之成为静态存储类型，二是用于修饰链接属性使之成为内部链接属性。

- 1、静态存储类型：

在函数内定义的静态局部变量，该变量存在内存的静态区，所以即使该函数运行结束，静态变量的值不会被销毁，函数下次运行时能仍用到这个值。

在函数外定义的静态变量——静态全局变量，该变量的作用域只能在定义该变量的文件中，不能被其他文件通过 extern 引用。

- 2、内部链接属性

静态函数只能在声明它的源文件中使用。

const 关键字

- 1、声明常变量，使得指定的变量不能被修改。

```
const int a = 5;/*a的值一直为5，不能被改变*/

const int b; b = 10;/*b的值被赋值为10后，不能被改变*/

const int *ptr; /*ptr为指向整型常量的指针，ptr的值可以修改，但不能修改其所指向的值*/

int *const ptr;/*ptr为指向整型的常量指针，ptr的值不能修改，但可以修改其所指向的值*/

const int *const ptr;/*ptr为指向整型常量的常量指针，ptr及其指向的值都不能修改*/
```

- 2、修饰函数形参，使得形参在函数内不能被修改，表示输入参数。

如

```
int fun(const int a);或int fun(const char *str);
```

- 3、修饰函数返回值，使得函数的返回值不能被修改。

```
const char *getstr(void);使用：const *str= getstr();

const int getint(void);  使用：const int a =getint();
```

sizeof 关键字

sizeof 是在编译阶段处理，且不能被编译为机器码。sizeof 的结果等于对象或类型所占的内存字节数。sizeof 的返回值类型为 size_t。

- 变量：int a; sizeof(a)为 4；
- 指针：int *p; sizeof(p)为 4；
- 数组：int b[10]; sizeof(b)为数组的大小，4*10；int c[0]; sizeof(c)等于 0
- 结构体：struct (int a; char ch;)s1; sizeof(s1)为 8 与结构体字节对齐有关。
- 对结构体求 sizeof 时，有两个原则：

```
（1）展开后的结构体的第一个成员的偏移量应当是被展开的结构体中最大的成员的整数倍。

（2）结构体大小必须是所有成员大小的整数倍，这里所有成员计算的是展开后的成员，而不是将嵌套的结构体当做一个整体。
```

- 注意：不能对结构体中的位域成员使用 sizeof
- sizeof(void)等于 1
- sizeof(void *)等于 4

十二、讲一下 iOS 内存管理的理解

实际上是三种方案的结合

1. TaggedPointer（针对类似于 NSNumber 的小对象类型）
2. NONPOINTER_ISA（64 位系统下）

- 第一位的 0 或 1 代表是纯地址型 isa 指针，还是 NONPOINTER_ISA 指针。
- 第二位，代表是否有关联对象
- 第三位代表是否有 C++ 代码。
- 接下来 33 位代表指向的内存地址
- 接下来有 弱引用 的标记
- 接下来有是否 delloc 的标记....等等

3. 散列表（引用计数表、weak 表）

- SideTables 表在 非嵌入式的 64 位系统中，有 64 张 SideTable 表
- 每一张 SideTable 主要是由三部分组成。自旋锁、引用计数表、弱引用表。
- 全局的 引用计数 之所以不存在同一张表中，是为了避免资源竞争，解决效率的问题。
- 引用计数表 中引入了 分离锁的概念，将一张表分拆成多个部分，对他们分别加锁，可以实现并发操作，提升执行效率

十三、讲一下 @dynamic 关键字？

@dynamic 意味着编译器不会帮助我们自动合成 setter 和 getter 方法。我们需要手动实现、这里就涉及到 Runtime 的动态添加方法的知识点。

十四、简要说一下 @autoreleasePool 的数据结构？

简单说是双向链表，每张链表头尾相接，有 parent、child 指针

每创建一个池子，会在首部创建一个 哨兵 对象,作为标记

最外层池子的顶端会有一个 next 指针。当链表容量满了，就会在链表的顶端，并指向下一张表。

十五、访问 __weak 修饰的变量，是否已经被注册在了 @autoreleasePool 中？为什么？

答案是肯定的，__weak 修饰的变量属于弱引用，如果没有被注册到 @autoreleasePool 中，创建之后也就会随之销毁，为了延长它的生命周期，必须注册到 @autoreleasePool 中，以延缓释放。

十六、retain、release 的实现机制？

1.Retain 的实现机制。

```
SideTable& table = SideTables()[This];

size_t& refcntStorage = table.refcnts[This];

refcntStorage += SIZE_TABLE_RC_ONE;
```

2.Release 的实现机制。

```
SideTable& table = SideTables()[This];

size_t& refcntStorage = table.refcnts[This];

refcntStorage -= SIZE_TABLE_RC_ONE;
```

二者的实现机制类似，概括讲就是通过第一层 hash 算法，找到 指针变量 所对应的 sideTable。然后再通过一层 hash 算法，找到存储 引用计数 的 size_t，然后对其进行增减操作。retainCount 不是固定的 1，SIZE_TABLE_RC_ONE 是一个宏定义，实际上是一个值为 4 的偏移量。

十七、MRC（手动引用计数）和 ARC(自动引用计数)

1、MRC：alloc，retain，release，retainCount,autorelease,dealloc

2、ARC：

- ARC 是 LLVM 和 Runtime 协作的结果
- ARC 禁止手动调用 retain，release，retainCount,autorelease 关键字
- ARC 新增 weak，strong 关键字

3、引用计数管理：

- alloc: 经过一系列函数调用，最终调用了 calloc 函数，这里并没有设置引用计数为 1
- retain: 经过两次哈希查找，找到其对应引用计数值，然后将引用计数加 1(实际是加偏移量)
- release：和 retain 相反，经过两次哈希查找，找到其对应引用计数值，然后将引用计数减 1
- dealloc:

4、弱引用管理：

- 添加 weak 变量:通过哈希算法位置查找添加。如果查找对应位置中已经有了当前对象所对应的弱引用数组，就把新的弱引用变量添加到数组当中；如果没有，就创建一个弱引用数组，并将该弱引用变量添加到该数组中。
- 当一个被 weak 修饰的对象被释放后，weak 对象怎么处理的？
  清除 weak 变量，同时设置指向为 nil。当对象被 dealloc 释放后，在 dealloc 的内部实现中，会调用弱引用清除的相关函数，会根据当前对象指针查找弱引用表，找到当前对象所对应的弱引用数组，将数组中的所有弱引用指针都置为 nil。

5、自动释放池：

在当次 runloop 将要结束的时候调用 objc_autoreleasePoolPop，并 push 进来一个新的 AutoreleasePool

AutoreleasePoolPage 是以栈为结点通过双向链表的形式组合而成，是和线程一一对应的。
内部属性有 parent，child 对应前后两个结点，thread 对应线程 ，next 指针指向栈中下一个可填充的位置。

- AutoreleasePool 实现原理？

编译器会将 @autoreleasepool {} 改写为：

```
void * ctx = objc_autoreleasePoolPush;
    {}
objc_autoreleasePoolPop(ctx);
```

- objc_autoreleasePoolPush：
  把当前 next 位置置为 nil，即哨兵对象,然后 next 指针指向下一个可入栈位置，
  AutoreleasePool 的多层嵌套，即每次 objc_autoreleasePoolPush，实际上是不断地向栈中插入哨兵对象。
- objc_autoreleasePoolPop:
  根据传入的哨兵对象找到对应位置。
  给上次 push 操作之后添加的对象依次发送 release 消息。
  回退 next 指针到正确的位置。

十八、BAD_ACCESS 在什么情况下出现?

访问了已经被销毁的内存空间，就会报出这个错误。
根本原因是有 悬垂指针 没有被释放。

十九、autoReleasePool 什么时候释放?

App 启动后，苹果在主线程 RunLoop 里注册了两个 Observer，其回调都是 _wrapRunLoopWithAutoreleasePoolHandler()。

- 第一个 Observer 监视的事件是 Entry(即将进入 Loop)，其回调内会调用 _objc_autoreleasePoolPush() 创建自动释放池。其 order 是 -2147483647，优先级最高，保证创建释放池发生在其他所有回调之前。
- 第二个 Observer 监视了两个事件： BeforeWaiting(准备进入休眠) 时调用 _objc_autoreleasePoolPop() 和 _objc_autoreleasePoolPush() 释放旧的池并创建新池；Exit(即将退出 Loop) 时调用 _objc_autoreleasePoolPop() 来释放自动释放池。这个 Observer 的 order 是 2147483647，优先级最低，保证其释放池子发生在其他所有回调之后。

二十、ARC 自动内存管理的原则

- 自己生成的对象，自己持有
- 非自己生成的对象，自己可以持有
- 自己持有的对象不再需要时，需要对其进行释放
- 非自己持有的对象无法释放

二十一、ARC 在运行时做了哪些工作？

- 主要是指 weak 关键字。weak 修饰的变量能够在引用计数为 0 时被自动设置成 nil，显然是有运行时逻辑在工作的。
- 为了保证向后兼容性，ARC 在运行时检测到类函数中的 autorelease 后紧跟其后 retain，此时不直接调用对象的 autorelease 方法，而是改为调用 objc_autoreleaseReturnValue。objc_autoreleaseReturnValue 会检视当前方法返回之后即将要执行的那段代码，若那段代码要在返回对象上执行 retain 操作，则设置全局数据结构中的一个标志位，而不执行 autorelease 操作，与之相似，如果方法返回了一个自动释放的对象，而调用方法的代码要保留此对象，那么此时不直接执行 retain ，而是改为执行 objc_retainAoutoreleasedReturnValue 函数。此函数要检测刚才提到的标志位，若已经置位，则不执行 retain 操作，设置并检测标志位，要比调用 autorelease 和 retain 更快。

二十二、ARC 在编译时做了哪些工作

根据代码执行的上下文语境，在适当的位置插入 retain，release

二十三、ARC 的 retainCount 怎么存储的？

存在 64 张哈希表中，根据哈希算法去查找所在的位置，无需遍历，十分快捷

散列表（引用计数表、weak 表）
- SideTables 表在 非嵌入式的 64 位系统中，有 64 张 SideTable 表
- 每一张 SideTable 主要是由三部分组成。自旋锁、引用计数表、弱引用表。
- 全局的 引用计数 之所以不存在同一张表中，是为了避免资源竞争，解决效率的问题。
- 引用计数表 中引入了 分离锁的概念，将一张表分拆成多个部分，对他们分别加锁，可以实现并发操作，提升执行效率

引用计数表（哈希表）

通过指针的地址，查找到引用计数的地址，大大提升查找效率

通过 DisguisedPtr(objc_object) 函数存储，同时也通过这个函数查找，这样就避免了循环遍历。

二十四、__weak 属性修饰的变量，如何实现在变量没有强引用后自动置为 nil ？

用的弱引用 -weak 表。也是一张 哈希表。

被 weak 修饰的指针变量所指向的地址是 key ，所有指向这块内存地址的指针会被添加在一个数组里，这个数组是 Value。当内存地址销毁，数组里的所有对象被置为 nil。

二十五、__weak 和 _Unsafe_Unretain 的区别？

weak 修饰的指针变量，在指向的内存地址销毁后，会在 Runtime 的机制下，自动置为 nil。

_Unsafe_Unretain 不会置为 nil，容易出现 悬垂指针，发生崩溃。但是 _Unsafe_Unretain 比 __weak 效率高。