面试题整理

1、怎么保证自己的类一定能调用到自己写的方法？（不知道有没有人会写一个分类，里面有和自己的类方法名字一模一样）

Category 并不会覆盖主类的同名方法，只是 Category 的方法排在主类方法的前面，OC 的消息发送机制是根据方法名在 method_list 中查找方法，找到第一个名字匹配的方法之后就不继续往下找了，每次调用的都是 method_list 中最前面的同名方法。
所以我们可以根据 selector 查找到这个类的所有同名 method，然后倒序调用，因为主类的同名方法在最后面。

2、isa指针里面储存了哪些信息？

现在的64位系统(arm64架构)中，苹果对isa做了优化，里面除了存储一个地址外还存储了很多其他信息。一个指针占8个字节，也就是64位，苹果只用了其中的33位来存储地址，其余31位用来存储其他信息。

- nonpointer：(isa的第0位)表示是否对 isa 指针开启指针优化。0：纯isa指针，1：优化过的isa。
- has_assoc：(isa的第1位)记录这个对象是否是关联对象。
- has_cxx_dtor：(isa的第2位)记录是否有c++的析构函数。
- shiftcls：(isa的第3-35位，共占33位)记录对象的地址值。
- magic：(isa的第36-41位，共占6位)用于在调试时分辨对象是否完成初始化。
- weakly_referenced：(isa的第42位)用于记录该对象是否被弱引用或曾经被弱引用过。
- deallocating：(isa的第43位)标志对象是否正在释放内存。
- has_sidetable_rc：(isa的第44位)用于标记是否有扩展的引用计数。当一个对象的引用计数比较少时，其引用计数就记录在isa中，当引用计数大于某个值时就会采用sideTable来协助存储引用计数。
- extra_rc：(isa的第45-63位，共占19位)，用来记录该对象的引用计数值-1。这里总共是19位，如果引用计数很大，19位存不下的话就会采用sideTable来协助存储。

3、为什么block要用copy修饰？

因为block在创建的时候，它的内存是分配在栈上的，而不是在堆上。栈区的特点是：对象随时有可能被销毁，一旦被销毁，在调用的时候，就会造成系统的崩溃。所以我们要使用copy把它拷贝到堆

上。

4、什么是离屏渲染以及离屏渲染的影响?

如果要在显示屏上显示内容，我们至少需要一块与屏幕像素数据量一样大的frame buffer（帧缓冲区），作为像素数据存储区域，然后由显示控制器把帧缓存区的数据显示到屏幕上。如果有时因为面临一些限制，比如说阴影，遮罩mask等，GPU无法把渲染结果直接写入frame buffer，而是先暂把中间的一个临时状态存在另外的内存区域，之后再写入frame buffer，这个过程被称之为离屏渲染。
离屏渲染的影响：需要额外开辟内存，增大了内存的消耗。需要多次切换上下文，从当前屏切换到离屏，从离屏切换到当前屏，上下文的切换是耗时的，可能会引起掉帧。

5、说说你对ro、rw和rwe的理解?

ro是在编译的时候生成的。当类在编译的时候，类的属性，方法，协议这些内容就存在ro这个结构体里面了，这是一块纯净的内存空间，不允许被修改。
rw是在运行的时候生成的，它会将ro的内容拷贝进去。实际访问类的方法、属性等内容的时候访问的rw中的内容。修改也是修改的里面的内容。
rwe的设计是为了节约内存。rwe只有在添加分类且分类和本类都为非懒加载类，胡总或者通过runtime动态修改类信息的时候才会存在。

6、苹果为什么要设计元类?

为了复用消息发送机制。我们调用方法的时候，编译器就会把方法转换为objc_msgSend这个函数，其实在底层就是通过objc_msgSend的第一个参数：消息的接收者的isa指针去找方法对应的实现，如果是实例对象，那通过isa就可以找到它的类对象，如果是类对象，那通过isa就可以找到它的元类对象。那如果没有元类的话，这个objc_msgSend起码还得加一个参数，来判断该方法是类方法还是对象方法，因为类方法和对象方法是可以重名的。而且还得判断，调用这到底是类对象还是实例对象。通过元类就可以巧妙的解决问题，让各类各司其职，实例对象就干存储属性值的事，类对象存储实例方法列表，元类对象存储类方法列表，这样就不用去判断对象类型和方法类型，从而提升消息发送的效率，并且不同类型的方法走的都是同一套流程。

7、原子属性能保障线程安全吗？为什么?

原子属性并不能够保障线程安全。通过objc源码，我们可以发现atomic只是对属性的getter/setter方法进行了加锁操作，这种安全仅仅能够保障属性的读\写安全。
举个例子：`self.count ++`。

这个self.count--其实既有setter操作又有getter操作，首先得getter到count的值，将count的值减1，然后通过setter去给count赋值。
所以这里就有可能出现一种情况，当我们读取完count的值得时候，count的值被另外一个线程给修改了，那么最终的 `self.count` 的值就和我们预期的不一样了。所以说原子属性并不能够保障线程安全。

8、如何手动关闭 KVO？如何手动触发 KVO？KVO的实现原理？

手动关闭KVO：在类里面实现下面这个方法，返回NO。

```
+(BOOL)automaticallyNotifiesObserversForKey:(NSString *)key {
    return NO;
}
```

手动触发KVO：KVO通知依赖于NSObject的两个方法， `willChangeValueForKey:` 和 `didChangevlueForKey:` 。在一个被观察属性发生改变之前调用 `willChangeValueForKey:` ，当改变发生后调用 `didChangevlueForKey:` 就可以实现手动触发。

```
-(void)setName:(NSString *)name {
    [self willChangeValueForKey:@"name"];
    _name = name;
    [self didChangeValueForKey:@"name"];
}
```

KVO的实现原理：在我们调用addObserve的时候，会动态生成一个以NSKVONotifying_开头的当前类的一个子类，对象的isa指针会指向这个类，系统会自动生成相应的setter方法，然后这个系统生成的子类会调用对应的setter方法，在这个setter方法的内部，其实就是调用 `willChangeValueForKey:` 和 `didChangeValueForKey:` 这俩个方法。

9、用一句话描述GCD发生的死锁现象？

dispatch_sync调用串行队列，但是串行队列的任务已经在当前线程执行了。

10、单例的弊端？

1、由于单利模式中没有抽象层，因此单例类的扩展有很大的困难。
2、单例类的职责过重，在一定程度上违背了“单一职责原则”。
3、单例一旦创建，会一直保存在内存当中，直到程序退出时才由系统自动释放这部分的内存，所以过多的单例会增大内存的消耗。

11、load()和initialize()的区别?

+load()方法：
当类被引用进项目的时候就会执行+load()方法(在main函数开始执行之前），与这个类是否被用到无关，每个类的load函数只会自动调用一次。由于load函数是系统自动加载的，因此不需要再调用[super load]，否则父类的load函数会多次执行。
+initialize方法：
+initialize()方法在类或者其子类的第一个方法被调用前调用。即使类文件被引用进项目，但是没有使用，+initialize()不会被调用。由于是系统自动调用，也不需要显式的调用父类的+initialize()方法，否则父类的+initialize()会被多次执行。假如这个类放到代码中，而这段代码并没有被执行，这个函数是不会被执行的。

12、简述APP main()函数执行前的启动流程?

在App开始启动后，系统首先加载可执行文件（自身App的所有.o文件的集合），然后加载动态链接库dyld，dyld是一个专门用来加载动态链接库的库。然后dyld会去找到可执行文件当中所依赖的动态库, 递归加载所有的依赖的动态库。
由于现在使用的都是虚拟内存（ASLR，地址空间布局随机化），可执行文件和动态链接库在虚拟内存中的加载地址每次启动都不固定，所以需要rebase/binding这2步来修复镜像中的资源指针，来指向正确的地址。rebase修复的是指向当前镜像内部的资源指针；而bind指向的是镜像外部的资源指针。
Objc 运行时的初始处理，包括 Objc 相关类的注册、category 注册、selector 唯一性检查，执行objc的+load函数，C++的构造函数等等。然后dyld会调起main()函数开始执行。

13、runtime 如何通过 selector 找到对应的 IMP 地址?

每一个类对象中都一个方法列表，方法列表中记录着方法的名称，方法实现，以及参数类型，其实 selector 本质 就是方法名称，通过这个方法名称就可以在方法列表中找到对应的方法实现。

14、简述你对self和super的理解?

self 是类的隐藏参数，指向当前调用方法的这个类的实例。super 本质是一个编译器标示符，和 self 指向的是同一个消息接受者。self 与 super 在调用方法时，在编译时有所不同，self为objc_msgSend，super为objc_msgSendSuper。objc_msgSend 会优先从当前类中找实现，objc_msgSendSuper 直接从父类中寻找实现。

15、简述一下dealloc的实现机制?

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. 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。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。

object_dispose()调用流程。

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

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。

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。