**一、Runtime**

**一个objc对象的isa的指针指向什么？有什么作用？**

指向他的类对象,从而可以找到对象上的方法

详解：下图很好的描述了对象，类，元类之间的关系:

图中实线是 super\_class指针，虚线是isa指针。

1.Root class (class)其实就是NSObject，NSObject是没有超类的，所以Root class(class)的superclass指向nil。  
2.每个Class都有一个isa指针指向唯一的Meta class  
3.Root class(meta)的superclass指向Root class(class)，也就是NSObject，形成一个回路。  
4.每个Meta class的isa指针都指向Root class (meta)。

**一个 NSObject 对象占用多少内存空间？**

受限于内存分配的机制，一个 NSObject对象都会分配 16byte 的内存空间。

但是实际上在 64位 下，只使用了 8byte;  
在32位下，只使用了 4byte

一个 NSObject 实例对象成员变量所占的大小，实际上是 8 字节

#import <Objc/Runtime>

Class\_getInstanceSize([NSObject Class])

本质是

size\_t class\_getInstanceSize(Class cls)

{

if (!cls) return 0;

return cls->alignedInstanceSize();

}

获取 Obj-C 指针所指向的内存的大小，实际上是16 字节

#import <malloc/malloc.h>

malloc\_size((\_\_bridge const void \*)obj);

对象在分配内存空间时，会进行内存对齐，所以在 iOS 中，分配内存空间都是 16字节 的倍数。

**说一下对 class\_rw\_t 的理解？**

rw代表可读可写。

ObjC 类中的属性、方法还有遵循的协议等信息都保存在 class\_rw\_t 中：

// 可读可写

struct class\_rw\_t {

// Be warned that Symbolication knows the layout of this structure.

uint32\_t flags;

uint32\_t version;

const class\_ro\_t \*ro; // 指向只读的结构体,存放类初始信息

/\*

这三个都是二位数组，是可读可写的，包含了类的初始内容、分类的内容。

methods中，存储 method\_list\_t ----> method\_t

二维数组，method\_list\_t --> method\_t

这三个二位数组中的数据有一部分是从class\_ro\_t中合并过来的。

\*/

method\_array\_t methods; // 方法列表（类对象存放对象方法，元类对象存放类方法）

property\_array\_t properties; // 属性列表

protocol\_array\_t protocols; //协议列表

Class firstSubclass;

Class nextSiblingClass;

//...

}

**说一下对 class\_ro\_t 的理解？**

存储了当前类在编译期就已经确定的属性、方法以及遵循的协议。

struct class\_ro\_t {

uint32\_t flags;

uint32\_t instanceStart;

uint32\_t instanceSize;

uint32\_t reserved;

const uint8\_t \* ivarLayout;

const char \* name;

method\_list\_t \* baseMethodList;

protocol\_list\_t \* baseProtocols;

const ivar\_list\_t \* ivars;

const uint8\_t \* weakIvarLayout;

property\_list\_t \*baseProperties;

};

baseMethodList，baseProtocols，ivars，baseProperties三个都是以为数组。

**说一下对 isa 指针的理解， 对象的isa 指针指向哪里？isa 指针有哪两种类型？**

isa 等价于 is kind of

* 实例对象 isa 指向类对象
* 类对象指 isa 向元类对象
* 元类对象的 isa 指向元类的基类

isa 有两种类型

* 纯指针，指向内存地址
* NON\_POINTER\_ISA，除了内存地址，还存有一些其他信息

# isa源码分析

在Runtime源码查看isa\_t是共用体。简化结构如下：

union isa\_t

{

Class cls;

uintptr\_t bits;

# if \_\_arm64\_\_ // arm64架构

# define ISA\_MASK 0x0000000ffffffff8ULL //用来取出33位内存地址使用（&）操作

# define ISA\_MAGIC\_MASK 0x000003f000000001ULL

# define ISA\_MAGIC\_VALUE 0x000001a000000001ULL

struct {

uintptr\_t nonpointer : 1; //0：代表普通指针，1：表示优化过的，可以存储更多信息。

uintptr\_t has\_assoc : 1; //是否设置过关联对象。如果没设置过，释放会更快

uintptr\_t has\_cxx\_dtor : 1; //是否有C++的析构函数

uintptr\_t shiftcls : 33; // MACH\_VM\_MAX\_ADDRESS 0x1000000000 内存地址值

uintptr\_t magic : 6; //用于在调试时分辨对象是否未完成初始化

uintptr\_t weakly\_referenced : 1; //是否有被弱引用指向过

uintptr\_t deallocating : 1; //是否正在释放

uintptr\_t has\_sidetable\_rc : 1; //引用计数器是否过大无法存储在ISA中。如果为1，那么引用计数会存储在一个叫做SideTable的类的属性中

uintptr\_t extra\_rc : 19; //里面存储的值是引用计数器减1

# define RC\_ONE (1ULL<<45)

# define RC\_HALF (1ULL<<18)

};

# elif \_\_x86\_64\_\_ // arm86架构,模拟器是arm86

# define ISA\_MASK 0x00007ffffffffff8ULL

# define ISA\_MAGIC\_MASK 0x001f800000000001ULL

# define ISA\_MAGIC\_VALUE 0x001d800000000001ULL

struct {

uintptr\_t nonpointer : 1;

uintptr\_t has\_assoc : 1;

uintptr\_t has\_cxx\_dtor : 1;

uintptr\_t shiftcls : 44; // MACH\_VM\_MAX\_ADDRESS 0x7fffffe00000

uintptr\_t magic : 6;

uintptr\_t weakly\_referenced : 1;

uintptr\_t deallocating : 1;

uintptr\_t has\_sidetable\_rc : 1;

uintptr\_t extra\_rc : 8;

# define RC\_ONE (1ULL<<56)

# define RC\_HALF (1ULL<<7)

};

# else

# error unknown architecture for packed isa

# endif

}

## 说一下 Runtime 的方法缓存？存储的形式、数据结构以及查找的过程？

cache\_t增量扩展的哈希表结构。哈希表内部存储的 bucket\_t。

bucket\_t 中存储的是 SEL 和 IMP的键值对。

* 如果是有序方法列表，采用二分查找
* 如果是无序方法列表，直接遍历查找

### cache\_t结构体

// 缓存曾经调用过的方法，提高查找速率

struct cache\_t {

struct bucket\_t \*\_buckets; // 散列表

mask\_t \_mask; //散列表的长度 - 1

mask\_t \_occupied; // 已经缓存的方法数量，散列表的长度使大于已经缓存的数量的。

//...

}

struct bucket\_t {

cache\_key\_t \_key; //SEL作为Key @selector()

IMP \_imp; // 函数的内存地址

//...

}

散列表查找过程，在objc-cache.mm文件中

// 查询散列表，k

bucket\_t \* cache\_t::find(cache\_key\_t k, id receiver)

{

assert(k != 0); // 断言

bucket\_t \*b = buckets(); // 获取散列表

mask\_t m = mask(); // 散列表长度 - 1

mask\_t begin = cache\_hash(k, m); // & 操作

mask\_t i = begin; // 索引值

do {

if (b[i].key() == 0 || b[i].key() == k) {

return &b[i];

}

} while ((i = cache\_next(i, m)) != begin);

// i 的值最大等于mask,最小等于0。

// hack

Class cls = (Class)((uintptr\_t)this - offsetof(objc\_class, cache));

cache\_t::bad\_cache(receiver, (SEL)k, cls);

}

上面是查询散列表函数，其中cache\_hash(k, m)是静态内联方法，将传入的key和mask进行&操作返回uint32\_t索引值。do-while循环查找过程，当发生冲突cache\_next方法将索引值减1。

**使用runtime Associate方法关联的对象，需要在主对象dealloc的时候释放么？**

无论在MRC下还是ARC下均不需要，被关联的对象在生命周期内要比对象本身释放的晚很多，它们会在被 NSObject -dealloc 调用的object\_dispose()方法中释放。

**详解：**

1、调用 -release ：引用计数变为零

对象正在被销毁，生命周期即将结束.

不能再有新的 \_\_weak 弱引用，否则将指向 nil.

调用 [self dealloc]

2、 父类调用 -dealloc

继承关系中最直接继承的父类再调用 -dealloc

如果是 MRC 代码 则会手动释放实例变量们（iVars）

继承关系中每一层的父类 都再调用 -dealloc

>3、NSObject 调 -dealloc

只做一件事：调用 Objective-C runtime 中object\_dispose() 方法

>4. 调用 object\_dispose()

为 C++ 的实例变量们（iVars）调用 destructors

为 ARC 状态下的 实例变量们（iVars） 调用 -release

解除所有使用 runtime Associate方法关联的对象

解除所有 \_\_weak 引用

调用 free()

实例对象的数据结构？

具体可以参看 Runtime 源代码，在文件 objc-private.h 的第 127-232 行。

struct objc\_object {

isa\_t isa;

//...

}

本质上 objc\_object 的私有属性只有一个 isa 指针。指向 类对象 的内存地址。

**什么是method swizzling（俗称黑魔法)**

简单说就是进行方法交换

在Objective-C中调用一个方法，其实是向一个对象发送消息，查找消息的唯一依据是selector的名字。利用Objective-C的动态特性，可以实现在运行时偷换selector对应的方法实现，达到给方法挂钩的目的。  
每个类都有一个方法列表，存放着方法的名字和方法实现的映射关系，selector的本质其实就是方法名，IMP有点类似函数指针，指向具体的Method实现，通过selector就可以找到对应的IMP。  
换方法的几种实现方式

* 利用 method\_exchangeImplementations 交换两个方法的实现
* 利用 class\_replaceMethod替换方法的实现
* 利用 method\_setImplementation 来直接设置某个方法的IMP

**什么时候会报unrecognized selector的异常？**

objc在向一个对象发送消息时，runtime库会根据对象的isa指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果，在最顶层的父类中依然找不到相应的方法时，会进入消息转发阶段，如果消息三次转发流程仍未实现，则程序在运行时会挂掉并抛出异常unrecognized selector sent to XXX 。

如何给 Category 添加属性？关联对象以什么形式进行存储？

查看的是 关联对象 的知识点。

详细的说一下 关联对象。

关联对象 以哈希表的格式，存储在一个全局的单例中。

@interface NSObject (Extension)

@property (nonatomic,copy ) NSString \*name;

@end

@implementation NSObject (Extension)

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

objc\_setAssociatedObject(self, @selector(name), name, OBJC\_ASSOCIATION\_COPY\_NONATOMIC);

}

- (NSString \*)name {

return objc\_getAssociatedObject(self,@selector(name));

}

@end

**能否向编译后得到的类中增加实例变量？能否向运行时创建的类中添加实例变量？为什么？**

不能向编译后得到的类中增加实例变量；

能向运行时创建的类中添加实例变量；

1.因为编译后的类已经注册在 runtime 中,类结构体中的 objc\_ivar\_list 实例变量的链表和 instance\_size 实例变量的内存大小已经确定，同时runtime会调用 class\_setvarlayout 或 class\_setWeaklvarLayout 来处理strong weak 引用.所以不能向存在的类中添加实例变量。  
2.运行时创建的类是可以添加实例变量，调用class\_addIvar函数. 但是的在调用 objc\_allocateClassPair 之后，objc\_registerClassPair 之前,原因同上.

类对象的数据结构？

具体可以参看 Runtime 源代码。

类对象就是 objc\_class。

struct objc\_class : objc\_object {

// Class ISA;

Class superclass; //父类指针

cache\_t cache; // formerly cache pointer and vtable 方法缓存

class\_data\_bits\_t bits; // class\_rw\_t \* plus custom rr/alloc flags 用于获取地址

class\_rw\_t \*data() {

return bits.data(); // &FAST\_DATA\_MASK 获取地址值

}

它的结构相对丰富一些。继承自objc\_object结构体，所以包含isa指针

* isa：指向元类
* superClass: 指向父类
* Cache: 方法的缓存列表
* data: 顾名思义，就是数据。是一个被封装好的 class\_rw\_t 。
* **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.

**runtime如何实现weak变量的自动置nil？知道SideTable吗？**

runtime 对注册的类会进行布局，对于 weak 修饰的对象会放入一个 hash 表中。 用 weak 指向的对象内存地址作为key，当此对象的引用计数为0的时候会 dealloc，假如 weak 指向的对象内存地址是a，那么就会以a为键， 在这个 weak表中搜索，找到所有以a为键的 weak 对象，从而设置为 nil。

**更细一点的回答：**

1.初始化时：runtime会调用objc\_initWeak函数，初始化一个新的weak指针指向对象的地址。  
2.添加引用时：objc\_initWeak函数会调用objc\_storeWeak() 函数， objc\_storeWeak()的作用是更新指针指向，创建对应的弱引用表。  
3.释放时,调用clearDeallocating函数。clearDeallocating函数首先根据对象地址获取所有weak指针地址的数组，然后遍历这个数组把其中的数据设为nil，最后把这个entry从weak表中删除，最后清理对象的记录。

SideTable结构体是负责管理类的引用计数表和weak表，

详解：参考自《Objective-C高级编程》一书  
**1.初始化时：runtime会调用objc\_initWeak函数，初始化一个新的weak指针指向对象的地址。**

{

NSObject \*obj = [[NSObject alloc] init];

id \_\_weak obj1 = obj;

}

当我们初始化一个weak变量时，runtime会调用 NSObject.mm 中的objc\_initWeak函数。

// 编译器的模拟代码

id obj1;

objc\_initWeak(&obj1, obj);

/\*obj引用计数变为0，变量作用域结束\*/

objc\_destroyWeak(&obj1);

通过objc\_initWeak函数初始化“附有weak修饰符的变量（obj1）”，在变量作用域结束时通过objc\_destoryWeak函数释放该变量（obj1）。

**2.添加引用时：objc\_initWeak函数会调用objc\_storeWeak() 函数， objc\_storeWeak()的作用是更新指针指向，创建对应的弱引用表。**

objc\_initWeak函数将“附有weak修饰符的变量（obj1）”初始化为0（nil）后，会将“赋值对象”（obj）作为参数，调用objc\_storeWeak函数。

obj1 = 0；

obj\_storeWeak(&obj1, obj);

**也就是说：**

weak 修饰的指针默认值是 nil （在Objective-C中向nil发送消息是安全的）

然后obj\_destroyWeak函数将0（nil）作为参数，调用objc\_storeWeak函数。

objc\_storeWeak(&obj1, 0);

前面的源代码与下列源代码相同。

// 编译器的模拟代码

id obj1;

obj1 = 0;

objc\_storeWeak(&obj1, obj);

/\* ... obj的引用计数变为0，被置nil ... \*/

objc\_storeWeak(&obj1, 0);

objc\_storeWeak函数把第二个参数的赋值对象（obj）的内存地址作为键值，将第一个参数\_\_weak修饰的属性变量（obj1）的内存地址注册到 weak 表中。如果第二个参数（obj）为0（nil），那么把变量（obj1）的地址从weak表中删除。

由于一个对象可同时赋值给多个附有\_\_weak修饰符的变量中，所以对于一个键值，可注册多个变量的地址。

可以把objc\_storeWeak(&a, b)理解为：objc\_storeWeak(value, key)，并且当key变nil，将value置nil。在b非nil时，a和b指向同一个内存地址，在b变nil时，a变nil。此时向a发送消息不会崩溃：在Objective-C中向nil发送消息是安全的。

**3.释放时,调用clearDeallocating函数。clearDeallocating函数首先根据对象地址获取所有weak指针地址的数组，然后遍历这个数组把其中的数据设为nil，最后把这个entry从weak表中删除，最后清理对象的记录。**

当weak引用指向的对象被释放时，又是如何去处理weak指针的呢？当释放对象时，其基本流程如下：

1.调用objc\_release  
2.因为对象的引用计数为0，所以执行dealloc  
3.在dealloc中，调用了\_objc\_rootDealloc函数  
4.在\_objc\_rootDealloc中，调用了object\_dispose函数  
5.调用objc\_destructInstance  
6.最后调用objc\_clear\_deallocating

对象被释放时调用的objc\_clear\_deallocating函数:

1.从weak表中获取废弃对象的地址为键值的记录  
2.将包含在记录中的所有附有 weak修饰符变量的地址，赋值为nil  
3.将weak表中该记录删除  
4.从引用计数表中删除废弃对象的地址为键值的记录

**总结:**

其实Weak表是一个hash（哈希）表，Key是weak所指对象的地址，Value是weak指针的地址（这个地址的值是所指对象指针的地址）数组。

**objc中向一个nil对象发送消息将会发生什么？**

如果向一个nil对象发送消息，首先在寻找对象的isa指针时就是0地址返回了，所以不会出现任何错误。也不会崩溃。

详解：

如果一个方法返回值是一个对象，那么发送给nil的消息将返回0(nil)；

如果方法返回值为指针类型，其指针大小为小于或者等于sizeof(void\*) ，float，double，long double 或者long long的整型标量，发送给nil的消息将返回0；

如果方法返回值为结构体,发送给nil的消息将返回0。结构体中各个字段的值将都是0；

如果方法的返回值不是上述提到的几种情况，那么发送给nil的消息的返回值将是未定义的。

**objc在向一个对象发送消息时，发生了什么？**

objc在向一个对象发送消息时，runtime会根据对象的isa指针找到该对象实际所属的类，然后在该类中的方法列表以及其父类方法列表中寻找方法运行，如果一直到根类还没找到，转向拦截调用，走消息转发机制，一旦找到 ，就去执行它的实现IMP 。

**isKindOfClass 与 isMemberOfClass**

下面代码输出什么？

@interface Sark : NSObject

@end

@implementation Sark

@end

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

@autoreleasepool {

BOOL res1 = [(id)[NSObject class] isKindOfClass:[NSObject class]];

BOOL res2 = [(id)[NSObject class] isMemberOfClass:[NSObject class]];

BOOL res3 = [(id)[Sark class] isKindOfClass:[Sark class]];

BOOL res4 = [(id)[Sark class] isMemberOfClass:[Sark class]];

NSLog(@"%d %d %d %d", res1, res2, res3, res4);

}

return 0;

}

答案：1000

**详解：**

在isKindOfClass中有一个循环，先判断class是否等于meta class，不等就继续循环判断是否等于meta class的super class，不等再继续取super class，如此循环下去。

[NSObject class]执行完之后调用isKindOfClass，第一次判断先判断NSObject和 NSObject的meta class是否相等，之前讲到meta class的时候放了一张很详细的图，从图上我们也可以看出，NSObject的meta class与本身不等。接着第二次循环判断NSObject与meta class的superclass是否相等。还是从那张图上面我们可以看到：Root class(meta) 的superclass就是 Root  
class(class)，也就是NSObject本身。所以第二次循环相等，于是第一行res1输出应该为YES。

同理，[Sark class]执行完之后调用isKindOfClass，第一次for循环，Sark的Meta Class与[Sark class]不等，第二次for循环，Sark Meta Class的super class 指向的是 NSObject Meta Class， 和Sark Class不相等。第三次for循环，NSObject Meta Class的super class指向的是NSObject Class，和 Sark Class 不相等。第四次循环，NSObject Class 的super class 指向 nil， 和 Sark Class不相等。第四次循环之后，退出循环，所以第三行的res3输出为NO。

isMemberOfClass的源码实现是拿到自己的isa指针和自己比较，是否相等。

第二行isa 指向 NSObject 的 Meta Class，所以和 NSObject Class不相等。第四行，isa指向Sark的Meta Class，和Sark Class也不等，所以第二行res2和第四行res4都输出NO。

## Category 在编译过后，是在什么时机与原有的类合并到一起的？

1. 程序启动后，通过编译之后，Runtime 会进行初始化，调用 \_objc\_init。
2. 然后会 map\_images。
3. 接下来调用 map\_images\_nolock。
4. 再然后就是 read\_images，这个方法会读取所有的类的相关信息。
5. 最后是调用 reMethodizeClass:，这个方法是重新方法化的意思。
6. 在 reMethodizeClass: 方法内部会调用 **attachCategories:** ，这个方法会传入 Class 和 Category ，会将方法列表，协议列表等与原有的类合并。最后加入到 **class\_rw\_t** 结构体中。

## Category 有哪些用途？

* 给系统类添加方法、属性（需要关联对象）。
* 对某个类大量的方法，可以实现按照不同的名称归类。

## Category 的实现原理？

被添加在了 class\_rw\_t 的对应结构里。

Category 实际上是 Category\_t 的结构体，在运行时，新添加的方法，都被以倒序插入到原有方法列表的最前面，所以不同的Category，添加了同一个方法，执行的实际上是最后一个。

拿方法列表举例，实际上是一个二维的数组。

Category 如果翻看源码的话就会知道实际上是一个 \_catrgory\_t 的结构体。

--  
例如我们在程序中写了一个 Nsobject+Tools 的分类，那么被编译为 C++ 之后，实际上是：

static struct \_catrgory\_t \_OBJC\_$\_CATEGORY\_NSObject\_$\_Tools \_\_attribute\_\_ ((used,section),("\_\_DATA,\_\_objc\_\_const"))

{

// name

// class

// instance method list

// class method list

// protocol list

// properties

}

Category 在刚刚编译完的时候，和原来的类是分开的，只有在程序运行起来后，通过 Runtime ，Category 和原来的类才会合并到一起。

mememove，memcpy：这俩方法是位移、复制，简单理解就是原有的方法移动到最后，根根新开辟的控件，把前面的位置留给分类，然后分类中的方法，按照倒序依次插入，可以得出的结论就就是，越晚参与编译的分类，里面的方法才是生效的那个。

**\_objc\_msgForward函数是做什么的，直接调用它将会发生什么？**

\_objc\_msgForward是 IMP 类型，用于消息转发的：当向一个对象发送一条消息，但它并没有实现的时候，\_objc\_msgForward会尝试做消息转发。

详解：\_objc\_msgForward在进行消息转发的过程中会涉及以下这几个方法：

1. List itemresolveInstanceMethod:方法 (或resolveClassMethod:)。
2. List itemforwardingTargetForSelector:方法
3. List itemmethodSignatureForSelector:方法
4. List itemforwardInvocation:方法
5. List itemdoesNotRecognizeSelector: 方法

**[self class] 与 [super class]**

下面的代码输出什么？

@implementation Son : Father

- (id)init

{

self = [super init];

if (self) {

NSLog(@"%@", NSStringFromClass([self class]));

NSLog(@"%@", NSStringFromClass([super class]));

}

return self;

}

@end

NSStringFromClass([self class]) = Son  
NSStringFromClass([super class]) = Son

详解：这个题目主要是考察关于 Objective-C 中对 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 是指向的同一个消息接受者。不同点在于：super 会告诉编译器，当调用方法时，去调用父类的方法，而不是本类中的方法。

当使用 self 调用方法时，会从当前类的方法列表中开始找，如果没有，就从父类中再找；而当使用 super 时，则从父类的方法列表中开始找。然后调用父类的这个方法。

在调用[super class]的时候，runtime会去调用objc\_msgSendSuper方法，而不是objc\_msgSend；

OBJC\_EXPORT void objc\_msgSendSuper(void /\* struct objc\_super \*super, SEL op, ... \*/ )

/// Specifies the superclass of an instance.

struct objc\_super {

/// Specifies an instance of a class.

\_\_unsafe\_unretained id receiver;

/// Specifies the particular superclass of the instance to message.

#if !defined(\_\_cplusplus) && !\_\_OBJC2\_\_

/\* For compatibility with old objc-runtime.h header \*/

\_\_unsafe\_unretained Class class;

#else

\_\_unsafe\_unretained Class super\_class;

#endif

/\* super\_class is the first class to search \*/

};

在objc\_msgSendSuper方法中，第一个参数是一个objc\_super的结构体，这个结构体里面有两个变量，一个是接收消息的receiver，一个是当前类的父类super\_class。

objc\_msgSendSuper的工作原理应该是这样的:  
从objc\_super结构体指向的superClass父类的方法列表开始查找selector，找到后以objc->receiver去调用父类的这个selector。注意，最后的调用者是objc->receiver，而不是super\_class！

那么objc\_msgSendSuper最后就转变成:

// 注意这里是从父类开始msgSend，而不是从本类开始

objc\_msgSend(objc\_super->receiver, @selector(class))

/// Specifies an instance of a class. 这是类的一个实例

\_\_unsafe\_unretained id receiver;

// 由于是实例调用，所以是减号方法

- (Class)class {

return object\_getClass(self);

}

由于找到了父类NSObject里面的class方法的IMP，又因为传入的入参objc\_super->receiver = self。self就是son，调用class，所以父类的方法class执行IMP之后，输出还是son，最后输出两个都一样，都是输出son。

### 题目一：下面的代码输出什么？

@implementation Son : Father

- (id)init {

self = [super init];

if (self) {

NSLog(@"%@", NSStringFromClass([self class]));

NSLog(@"%@", NSStringFromClass([super class]));

}

return self;

}

@end

**结果：** Son / Son

**分析：**

对于上面的答案，第一个的结果应该是我们的预期结果，但是第二个结果却让我们很费解了。

那我们利用前面文章讲过的知识点来分析一下整个的流程。

因为，Son 及 Father 都没有实现 -(Class)calss 方法，所以这里所有的调用最终都会找到基类 NSObject 中，并且在其中找到 -(Class)calss 方法。那我们需要了解的就是在 NSObject 中这个方法的实现了。

在 NSObject.mm 中可以找到 -(Class)class 的实现：

- (Class)class {

return object\_getClass(self);

}

在 objc\_class.mm 中找到 object\_getClass 的实现：

Class object\_getClass(id obj)

{

if (obj) return obj->getIsa();

else return Nil;

}

ps：上面的方法定义可以去[官方OpenSource](https://links.jianshu.com/go?to=https%3A%2F%2Flink.juejin.im%3Ftarget%3Dhttps%253A%252F%252Fopensource.apple.com%252Fsource%252Fobjc4%252F)中下载源码哦。

可以看到，最终这个方法返回的是，调用这个方法的 objc 的 isa 指针。那我们只需要知道在题干中的代码里面最终是谁在调用 -(Class)class 方法就可以找到答案了。

接下来，我们利用 **clang -rewrite-objc** 命令，将题干的代码转化为如下代码：

NSLog((NSString \*)&\_\_NSConstantStringImpl\_\_var\_folders\_8k\_cgm28r0d0bz94xnnrr606rf40000gn\_T\_Car\_3f2069\_mi\_0, NSStringFromClass(((Class (\*)(id, SEL))(void \*)objc\_msgSend)((id)self, sel\_registerName("class"))));

NSLog((NSString \*)&\_\_NSConstantStringImpl\_\_var\_folders\_8k\_cgm28r0d0bz94xnnrr606rf40000gn\_T\_Car\_3f2069\_mi\_1, NSStringFromClass(((Class (\*)(\_\_rw\_objc\_super \*, SEL))(void \*)objc\_msgSendSuper)((\_\_rw\_objc\_super){(id)self, (id)class\_getSuperclass(objc\_getClass("Car"))}, sel\_registerName("class"))));

从上方可以得出，调用 [Father class] 的时候，本质是在调用

objc\_msgSendSuper(struct objc\_super \*super, SEL op, ...)

struct objc\_super 的定义如下：

struct objc\_super {

/// Specifies an instance of a class.

\_\_unsafe\_unretained \_Nonnull id receiver;

/// Specifies the particular superclass of the instance to message.

#if !defined(\_\_cplusplus) && !\_\_OBJC2\_\_

/\* For compatibility with old objc-runtime.h header \*/

\_\_unsafe\_unretained \_Nonnull Class class;

#else

\_\_unsafe\_unretained \_Nonnull Class super\_class;

#endif

/\* super\_class is the first class to search \*/

};

从定义可以得知：当利用 super 调用方法时，只要编译器看到super这个标志，就会让当前对象去调用父类方法，本质还是当前对象在调用，是去父类找实现，super 仅仅是一个编译指示器。但是消息的接收者 receiver 依然是self。最终在 NSObject 获取 isa 指针的时候，获取到的依旧是 self 的 isa，所以，我们得到的结果是：Son。

**扩展一下：** 看看下方的代码会输出什么？

@interface Father : NSObject

@end

@implementation Father

- (Class)class {

return [Father class];

}

@end

---

@interface Son : Father

@end

@implementation Son

- (id)init {

self = [super init];

if (self) {

NSLog(@"%@", NSStringFromClass([self class]));

NSLog(@"%@", NSStringFromClass([super class]));

}

return self;

}

@end

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

Son \*foo = [[Son alloc]init];

return 0;

}

---输出：---

Father

Father

### 题目二：以下的代码会输出什么结果？

@interface Sark : NSObject

@end

@implementation Sark

@end

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

@autoreleasepool {

// insert code here...

NSLog(@"%@", [NSObject class]);

NSLog(@"%@", [Sark class]);

BOOL res1 = [(id)[NSObject class] isKindOfClass:[NSObject class]];

BOOL res2 = [(id)[NSObject class] isMemberOfClass:[NSObject class]];

BOOL res3 = [(id)[Sark class] isKindOfClass:[Sark class]];

BOOL res4 = [(id)[Sark class] isMemberOfClass:[Sark class]];

NSLog(@"%d--%d--%d--%d", res1, res2, res3, res4);

}

return 0;

}

**结果：** 1--0--0--0

**分析：**

首先，我们先去查看一下题干中两个方法的源码：

- (BOOL)isMemberOfClass:(Class)cls {

return [self class] == cls;

}

- (BOOL)isKindOfClass:(Class)cls {

for (Class tcls = [self class]; tcls; tcls = tcls->superclass) {

if (tcls == cls) return YES;

}

return NO;

}

可以得知：

* isKindOfClass 的执行过程是拿到自己的 isa 指针和自己比较，若不等则继续取 isa 指针所指的 super class 进行比较。如此循环。
* isMemberOfClass 是拿到自己的 isa 指针和自己比较，是否相等。

1. [NSObject class] 执行完之后调用 isKindOfClass，第一次判断先判断 NSObject 和 NSObject 的 meta class 是否相等，之前讲到 meta class 的时候放了一张很详细的图，从图上我们也可以看出，NSObject 的 meta class 与本身不等。接着第二次循环判断 NSObject 与meta class 的 superclass 是否相等。还是从那张图上面我们可以看到：Root class(meta) 的 superclass 就是 Root class(class)，也就是 NSObject 本身。所以第二次循环相等，于是第一行 res1 输出应该为YES。
2. isa 指向 NSObject 的 Meta Class，所以和 NSObject Class不相等。
3. [Sark class] 执行完之后调用 isKindOfClass，第一次 for 循环，Sark 的 Meta Class 与 [Sark class] 不等，第二次 for 循环，Sark Meta Class 的 super class 指向的是 NSObject Meta Class， 和 Sark Class 不相等。第三次 for 循环，NSObject Meta Class 的 super class 指向的是 NSObject Class，和 Sark Class 不相等。第四次循环，NSObject Class 的super class 指向 nil， 和 Sark Class 不相等。第四次循环之后，退出循环，所以第三行的 res3 输出为 NO。
4. isa 指向 Sark 的 Meta Class，和 Sark Class 也不等。

#### 二、Runloop

## 为什么 NSTimer 有时候不好使？

因为创建的 NSTimer 默认是被加入到了 defaultMode，所以当 Runloop 的 Mode 变化时，当前的 NSTimer 就不会工作了。

## AFNetworking 中如何运用 Runloop?

AFURLConnectionOperation 这个类是基于 NSURLConnection 构建的，其希望能在后台线程接收 Delegate 回调。为此 AFNetworking 单独创建了一个线程，并在这个线程中启动了一个 RunLoop：

+ (void)networkRequestThreadEntryPoint:(id)\_\_unused object {

@autoreleasepool {

[[NSThread currentThread] setName:@"AFNetworking"];

NSRunLoop \*runLoop = [NSRunLoop currentRunLoop];

[runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];

[runLoop run];

}

}

+ (NSThread \*)networkRequestThread {

static NSThread \*\_networkRequestThread = nil;

static dispatch\_once\_t oncePredicate;

dispatch\_once(&oncePredicate, ^{

\_networkRequestThread = [[NSThread alloc] initWithTarget:self selector:@selector(networkRequestThreadEntryPoint:) object:nil];

[\_networkRequestThread start];

});

return \_networkRequestThread;

}

RunLoop 启动前内部必须要有至少一个 Timer/Observer/Source，所以 AFNetworking 在 [runLoop run] 之前先创建了一个新的 NSMachPort 添加进去了。通常情况下，调用者需要持有这个 NSMachPort (mach\_port) 并在外部线程通过这个 port 发送消息到 loop 内；但此处添加 port 只是为了让 RunLoop 不至于退出，并没有用于实际的发送消息。

- (void)start {

[self.lock lock];

if ([self isCancelled]) {

[self performSelector:@selector(cancelConnection) onThread:[[self class] networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes allObjects]];

} else if ([self isReady]) {

self.state = AFOperationExecutingState;

[self performSelector:@selector(operationDidStart) onThread:[[self class] networkRequestThread] withObject:nil waitUntilDone:NO modes:[self.runLoopModes allObjects]];

}

[self.lock unlock];

}

当需要这个后台线程执行任务时，AFNetworking 通过调用 [NSObject performSelector:onThread:..] 将这个任务扔到了后台线程的 RunLoop 中。

### autoreleasePool 在何时被释放？

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。

在主线程执行的代码，通常是写在诸如事件回调、Timer回调内的。这些回调会被 RunLoop 创建好的 AutoreleasePool 环绕着，所以不会出现内存泄漏，开发者也不必显示创建 Pool 了。

## PerformSelector 的实现原理？

当调用 NSObject 的 performSelecter:afterDelay: 后，实际上其内部会创建一个 Timer 并添加到当前线程的 RunLoop 中。所以如果当前线程没有 RunLoop，则这个方法会失效。

当调用 performSelector:onThread: 时，实际上其会创建一个 Timer 加到对应的线程去，同样的，如果对应线程没有 RunLoop 该方法也会失效。

## PerformSelector:afterDelay:这个方法在子线程中是否起作用？为什么？怎么解决？

不起作用，子线程默认没有 Runloop，也就没有 Timer。

解决的办法是可以使用 GCD 来实现：Dispatch\_after�

**RunLoop的Mode**

关于Mode首先要知道一个RunLoop 对象中可能包含多个Mode，且每次调用 RunLoop 的主函数时，只能指定其中一个 Mode(CurrentMode)。切换 Mode，需要重新指定一个 Mode 。主要是为了分隔开不同的 Source、Timer、Observer，让它们之间互不影响。

当RunLoop运行在Mode1上时，是无法接受处理Mode2或Mode3上的Source、Timer、Observer事件的

总共是有五种CFRunLoopMode:

* kCFRunLoopDefaultMode：默认模式，主线程是在这个运行模式下运行
* UITrackingRunLoopMode：跟踪用户交互事件（用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他Mode影响）
* 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
* GSEventReceiveRunLoopMode：接受系统内部事件，通常用不到
* kCFRunLoopCommonModes：伪模式，不是一种真正的运行模式，是同步Source/Timer/Observer到多个Mode中的一种解决方案

**RunLoop的实现机制**

对于RunLoop而言最核心的事情就是保证线程在没有消息的时候休眠，在有消息时唤醒，以提高程序性能。RunLoop这个机制是依靠系统内核来完成的（苹果操作系统核心组件Darwin中的Mach）。

RunLoop通过mach\_msg()函数接收、发送消息。它的本质是调用函数mach\_msg\_trap()，相当于是一个系统调用，会触发内核状态切换。在用户态调用 mach\_msg\_trap()时会切换到内核态；内核态中内核实现的mach\_msg()函数会完成实际的工作。  
即基于port的source1，监听端口，端口有消息就会触发回调；而source0，要手动标记为待处理和手动唤醒RunLoop

[Mach消息发送机制](https://www.jianshu.com/p/a764aad31847)  
大致逻辑为：  
1、通知观察者 RunLoop 即将启动。  
2、通知观察者即将要处理Timer事件。  
3、通知观察者即将要处理source0事件。  
4、处理source0事件。  
5、如果基于端口的源(Source1)准备好并处于等待状态，进入步骤9。  
6、通知观察者线程即将进入休眠状态。  
7、将线程置于休眠状态，由用户态切换到内核态，直到下面的任一事件发生才唤醒线程。

* 一个基于 port 的Source1 的事件(图里应该是source0)。
* 一个 Timer 到时间了。
* RunLoop 自身的超时时间到了。
* 被其他调用者手动唤醒。

8、通知观察者线程将被唤醒。  
9、处理唤醒时收到的事件。

* 如果用户定义的定时器启动，处理定时器事件并重启RunLoop。进入步骤2。
* 如果输入源启动，传递相应的消息。
* 如果RunLoop被显示唤醒而且时间还没超时，重启RunLoop。进入步骤2

10、通知观察者RunLoop结束。

#### RunLoop和线程

* 线程和RunLoop是一一对应的,其映射关系是保存在一个全局的 Dictionary 里
* 自己创建的线程默认是没有开启RunLoop的

**1、怎么创建一个常驻线程？**

1、为当前线程开启一个RunLoop（第一次调用 [NSRunLoop currentRunLoop]方法时实际是会先去创建一个RunLoop）  
2、向当前RunLoop中添加一个Port/Source等维持RunLoop的事件循环（如果RunLoop的mode中一个item都没有，RunLoop会退出）  
3、启动该RunLoop

@autoreleasepool {

NSRunLoop \*runLoop = [NSRunLoop currentRunLoop];

[[NSRunLoop currentRunLoop] addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];

[runLoop run];

}

**2、输出下边代码的执行顺序**

NSLog(@"1");

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

NSLog(@"2");

[self performSelector:@selector(test) withObject:nil afterDelay:10];

NSLog(@"3");

});

NSLog(@"4");

- (void)test

{

NSLog(@"5");

}

答案是1423，test方法并不会执行。  
原因是如果是带afterDelay的延时函数，会在内部创建一个 NSTimer，然后添加到当前线程的RunLoop中。也就是如果当前线程没有开启RunLoop，该方法会失效。  
那么我们改成:

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

NSLog(@"2");

[[NSRunLoop currentRunLoop] run];

[self performSelector:@selector(test) withObject:nil afterDelay:10];

NSLog(@"3");

});

然而test方法依然不执行。  
原因是如果RunLoop的mode中一个item都没有，RunLoop会退出。即在调用RunLoop的run方法后，由于其mode中没有添加任何item去维持RunLoop的时间循环，RunLoop随即还是会退出。  
所以我们自己启动RunLoop，一定要在添加item后

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

NSLog(@"2");

[self performSelector:@selector(test) withObject:nil afterDelay:10];

[[NSRunLoop currentRunLoop] run];

NSLog(@"3");

});

**3、怎样保证子线程数据回来更新UI的时候不打断用户的滑动操作？**

当我们在子请求数据的同时滑动浏览当前页面，如果数据请求成功要切回主线程更新UI，那么就会影响当前正在滑动的体验。  
我们就可以将更新UI事件放在主线程的NSDefaultRunLoopMode上执行即可，这样就会等用户不再滑动页面，主线程RunLoop由UITrackingRunLoopMode切换到NSDefaultRunLoopMode时再去更新UI

[self performSelectorOnMainThread:@selector(reloadData) withObject:nil waitUntilDone:NO modes:@[N

#### RunLoop的数据结构

NSRunLoop(Foundation)是CFRunLoop(CoreFoundation)的封装，提供了面向对象的API  
RunLoop 相关的主要涉及五个类：

CFRunLoop：RunLoop对象  
CFRunLoopMode：运行模式  
CFRunLoopSource：输入源/事件源  
CFRunLoopTimer：定时源  
CFRunLoopObserver：观察者

**1、CFRunLoop**

由pthread(线程对象，说明RunLoop和线程是一一对应的)、currentMode(当前所处的运行模式)、modes(多个运行模式的集合)、commonModes(模式名称字符串集合)、commonModelItems(Observer,Timer,Source集合)构成

**2、CFRunLoopMode**

由name、source0、source1、observers、timers构成

**3、CFRunLoopSource**

分为source0和source1两种

* source0:  
  即非基于port的，也就是用户触发的事件。需要手动唤醒线程，将当前线程从内核态切换到用户态
* source1:  
  基于port的，包含一个 mach\_port 和一个回调，可监听系统端口和通过内核和其他线程发送的消息，能主动唤醒RunLoop，接收分发系统事件。  
  具备唤醒线程的能力

**4、CFRunLoopTimer**

基于时间的触发器，基本上说的就是NSTimer。在预设的时间点唤醒RunLoop执行回调。因为它是基于RunLoop的，因此它不是实时的（就是NSTimer 是不准确的。 因为RunLoop只负责分发源的消息。如果线程当前正在处理繁重的任务，就有可能导致Timer本次延时，或者少执行一次）。

**5、CFRunLoopObserver**

监听以下时间点:CFRunLoopActivity

* kCFRunLoopEntry   
  RunLoop准备启动
* kCFRunLoopBeforeTimers   
  RunLoop将要处理一些Timer相关事件
* kCFRunLoopBeforeSources   
  RunLoop将要处理一些Source事件
* kCFRunLoopBeforeWaiting   
  RunLoop将要进行休眠状态,即将由用户态切换到内核态
* kCFRunLoopAfterWaiting  
  RunLoop被唤醒，即从内核态切换到用户态后
* kCFRunLoopExit  
  RunLoop退出
* kCFRunLoopAllActivities  
  监听所有状态

**6、各数据结构之间的联系**

线程和RunLoop一一对应， RunLoop和Mode是一对多的，Mode和source、timer、observer也是一对多的

#### RunLoop概念

RunLoop是通过内部维护的事件循环(Event Loop)来对事件/消息进行管理的一个对象。

1、没有消息处理时，休眠已避免资源占用，由用户态切换到内核态(CPU-内核态和用户态)  
2、有消息需要处理时，立刻被唤醒，由内核态切换到用户态

**为什么main函数不会退出？**

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

@autoreleasepool {

return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));

}

}

UIApplicationMain内部默认开启了主线程的RunLoop，并执行了一段无限循环的代码（不是简单的for循环或while循环）

//无限循环代码模式(伪代码)

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

BOOL running = YES;

do {

// 执行各种任务，处理各种事件

// ......

} while (running);

return 0;

}

UIApplicationMain函数一直没有返回，而是不断地接收处理消息以及等待休眠，所以运行程序之后会保持持续运行状态。

#### RunLoop与NSTimer

一个比较常见的问题：滑动tableView时，定时器还会生效吗？  
默认情况下RunLoop运行在kCFRunLoopDefaultMode下，而当滑动tableView时，RunLoop切换到UITrackingRunLoopMode，而Timer是在kCFRunLoopDefaultMode下的，就无法接受处理Timer的事件。  
怎么去解决这个问题呢？把Timer添加到UITrackingRunLoopMode上并不能解决问题，因为这样在默认情况下就无法接受定时器事件了。  
所以我们需要把Timer同时添加到UITrackingRunLoopMode和kCFRunLoopDefaultMode上。  
那么如何把timer同时添加到多个mode上呢？就要用到NSRunLoopCommonModes了

[[NSRunLoop currentRunLoop] addTimer:timer forMode:NSRunLoopCommonModes];

Timer就被添加到多个mode上，这样即使RunLoop由kCFRunLoopDefaultMode切换到UITrackingRunLoopMode下，也不会影响接收Timer事件

## 讲一下 Observer ？

typedef CF\_OPTIONS(CFOptionFlags, CFRunLoopActivity) {

kCFRunLoopEntry = (1UL << 0), // 即将进入Loop

kCFRunLoopBeforeTimers = (1UL << 1), // 即将处理 Timer

kCFRunLoopBeforeSources = (1UL << 2), // 即将处理 Source

kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠

kCFRunLoopAfterWaiting = (1UL << 6), // 刚从休眠中唤醒

kCFRunLoopExit = (1UL << 7), // 即将退出Loop

};

## 解释一下 NSTimer。

NSTimer 其实就是 CFRunLoopTimerRef，他们之间是 toll-free bridged 的。一个 NSTimer 注册到 RunLoop 后，RunLoop 会为其重复的时间点注册好事件。例如 10:00, 10:10, 10:20 这几个时间点。RunLoop 为了节省资源，并不会在非常准确的时间点回调这个Timer。Timer 有个属性叫做 Tolerance (宽容度)，标示了当时间点到后，容许有多少最大误差。

如果某个时间点被错过了，例如执行了一个很长的任务，则那个时间点的回调也会跳过去，不会延后执行。就比如等公交，如果 10:10 时我忙着玩手机错过了那个点的公交，那我只能等 10:20 这一趟了。

CADisplayLink 是一个和屏幕刷新率一致的定时器（但实际实现原理更复杂，和 NSTimer 并不一样，其内部实际是操作了一个 Source）。如果在两次屏幕刷新之间执行了一个长任务，那其中就会有一帧被跳过去（和 NSTimer 相似），造成界面卡顿的感觉。在快速滑动 TableView 时，即使一帧的卡顿也会让用户有所察觉。Facebook 开源的 AsyncDisplayLink 就是为了解决界面卡顿的问题，其内部也用到了 RunLoop。

## 解释一下 事件响应 的过程？

苹果注册了一个 Source1 (基于 mach port 的) 用来接收系统事件，其回调函数为 \_\_IOHIDEventSystemClientQueueCallback()。

当一个硬件事件(触摸/锁屏/摇晃等)发生后，首先由 IOKit.framework 生成一个 IOHIDEvent 事件并由 SpringBoard 接收。这个过程的详细情况可以参考这里。SpringBoard 只接收按键(锁屏/静音等)，触摸，加速，接近传感器等几种 Event，随后用 mach port 转发给需要的 App 进程。随后苹果注册的那个 Source1 就会触发回调，并调用 \_UIApplicationHandleEventQueue() 进行应用内部的分发。

\_UIApplicationHandleEventQueue() 会把 IOHIDEvent 处理并包装成 UIEvent 进行处理或分发，其中包括识别 UIGesture/处理屏幕旋转/发送给 UIWindow 等。通常事件比如 UIButton 点击、touchesBegin/Move/End/Cancel 事件都是在这个回调中完成的。

## 解释一下 手势识别 的过程？

当上面的 \_UIApplicationHandleEventQueue()识别了一个手势时，其首先会调用 Cancel 将当前的 touchesBegin/Move/End 系列回调打断。随后系统将对应的 UIGestureRecognizer 标记为待处理。

苹果注册了一个 Observer 监测 BeforeWaiting (Loop即将进入休眠) 事件，这个 Observer 的回调函数是 \_UIGestureRecognizerUpdateObserver()，其内部会获取所有刚被标记为待处理的 GestureRecognizer，并执行GestureRecognizer 的回调。

当有 UIGestureRecognizer 的变化(创建/销毁/状态改变)时，这个回调都会进行相应处理。

## 什么是异步绘制？

异步绘制，就是可以在子线程把需要绘制的图形，提前在子线程处理好。将准备好的图像数据直接返给主线程使用，这样可以降低主线程的压力。

###### 异步绘制的过程

要通过系统的 [view.delegate displayLayer:] 这个入口来实现异步绘制。

* 代理负责生成对应的 Bitmap
* 设置该 Bitmap 为 layer.contents 属性的值。

## 利用 runloop 解释一下页面的渲染的过程？

当我们调用 [UIView setNeedsDisplay] 时，这时会调用当前 View.layer 的 [view.layer setNeedsDisplay]方法。

这等于给当前的 layer 打上了一个脏标记，而此时并没有直接进行绘制工作。而是会到当前的 Runloop 即将休眠，也就是 beforeWaiting 时才会进行绘制工作。

紧接着会调用 [CALayer display]，进入到真正绘制的工作。CALayer 层会判断自己的 delegate 有没有实现异步绘制的代理方法 displayer:，这个代理方法是异步绘制的入口，如果没有实现这个方法，那么会继续进行系统绘制的流程，然后绘制结束。

CALayer 内部会创建一个 Backing Store，用来获取图形上下文。接下来会判断这个 layer 是否有 delegate。

如果有的话，会调用 [layer.delegate drawLayer:inContext:]，并且会返回给我们 [UIView DrawRect:] 的回调，让我们在系统绘制的基础之上再做一些事情。

如果没有 delegate，那么会调用 [CALayer drawInContext:]。

以上两个分支，最终 CALayer 都会将位图提交到 Backing Store，最后提交给 GPU。

至此绘制的过程结束。

#### 三、Objective\_C语言特性

**KVC(Key-value coding)**

-(id)valueForKey:(NSString \*)key;

-(void)setValue:(id)value forKey:(NSString \*)key;

KVC就是指iOS的开发中，可以允许开发者通过Key名直接访问对象的属性，或者给对象的属性赋值。而不需要调用明确的存取方法。这样就可以在运行时动态地访问和修改对象的属性。而不是在编译时确定，这也是iOS开发中的黑魔法之一。很多高级的iOS开发技巧都是基于KVC实现的

**当调用setValue：属性值forKey：@”name“的代码时，，底层的执行机制如下：**

* 程序优先调用set<Key>:属性值方法，代码通过setter方法完成设置。注意，这里的<key>是指成员变量名，首字母大小写要符合KVC的命名规则，下同
* 如果没有找到setName：方法，KVC机制会检查+ (BOOL)accessInstanceVariablesDirectly方法有没有返回YES，默认该方法会返回YES，如果你重写了该方法让其返回NO的话，那么在这一步KVC会执行setValue：forUndefinedKey：方法，不过一般开发者不会这么做。所以KVC机制会搜索该类里面有没有名为<key>的成员变量，无论该变量是在类接口处定义，还是在类实现处定义，也无论用了什么样的访问修饰符，只在存在以<key>命名的变量，KVC都可以对该成员变量赋值。
* 如果该类即没有set<key>：方法，也没有\_<key>成员变量，KVC机制会搜索\_is<Key>的成员变量。
* 和上面一样，如果该类即没有set<Key>：方法，也没有\_<key>和\_is<Key>成员变量，KVC机制再会继续搜索<key>和is<Key>的成员变量。再给它们赋值。
* 如果上面列出的方法或者成员变量都不存在，系统将会执行该对象的setValue：forUndefinedKey：方法，默认是抛出异常。

即如果没有找到Set<Key>方法的话，会按照\_key，\_iskey，key，iskey的顺序搜索成员并进行赋值操作。

如果开发者想让这个类禁用KVC，那么重写+ (BOOL)accessInstanceVariablesDirectly方法让其返回NO即可，这样的话如果KVC没有找到set<Key>:属性名时，会直接用setValue：forUndefinedKey：方法。

**当调用valueForKey：@”name“的代码时，KVC对key的搜索方式不同于setValue：属性值 forKey：@”name“，其搜索方式如下：**

* 首先按get<Key>,<key>,is<Key>的顺序方法查找getter方法，找到的话会直接调用。如果是BOOL或者Int等值类型， 会将其包装成一个NSNumber对象。
* 如果上面的getter没有找到，KVC则会查找countOf<Key>,objectIn<Key>AtIndex或<Key>AtIndexes格式的方法。如果countOf<Key>方法和另外两个方法中的一个被找到，那么就会返回一个可以响应NSArray所有方法的代理集合(它是NSKeyValueArray，是NSArray的子类)，调用这个代理集合的方法，或者说给这个代理集合发送属于NSArray的方法，就会以countOf<Key>,objectIn<Key>AtIndex或<Key>AtIndexes这几个方法组合的形式调用。还有一个可选的get<Key>:range:方法。所以你想重新定义KVC的一些功能，你可以添加这些方法，需要注意的是你的方法名要符合KVC的标准命名方法，包括方法签名。
* 如果上面的方法没有找到，那么会同时查找countOf<Key>，enumeratorOf<Key>,memberOf<Key>格式的方法。如果这三个方法都找到，那么就返回一个可以响应NSSet所的方法的代理集合，和上面一样，给这个代理集合发NSSet的消息，就会以countOf<Key>，enumeratorOf<Key>,memberOf<Key>组合的形式调用。
* 如果还没有找到，再检查类方法+

(BOOL)accessInstanceVariablesDirectly,如果返回YES(默认行为)，那么和先前的设值一样，会按\_<key>,\_is<Key>,<key>,is<Key>的顺序搜索成员变量名，这里不推荐这么做，因为这样直接访问实例变量破坏了封装性，使代码更脆弱。如果重写了类方法+ (BOOL)accessInstanceVariablesDirectly返回NO的话，那么会直接调用valueForUndefinedKey:方法，默认是抛出异常。

**KVO (Key-value observing)**

KVO是观察者模式的另一实现。  
使用了isa混写(isa-swizzling)来实现KVO

使用setter方法改变值KVO会生效，使用setValue:forKey即KVC改变值KVO也会生效，因为KVC会去调用setter方法

- (void)setValue:(id)value

{

[self willChangeValueForKey:@"key"];

[super setValue:value];

[self didChangeValueForKey:@"key"];

}

* 那么通过直接赋值成员变量会触发KVO吗？  
  不会，因为不会调用setter方法，需要加上  
  willChangeValueForKey和didChangeValueForKey方法来手动触发才行

**一、分类**

* 1.分类的作用？  
  声明私有方法，分解体积大的类文件，把framework的私有方法公开
* 2.分类的特点  
  运行时决议，可以为系统类添加分类 。  
  说得详细些，在运行时时期，将 Category 中的实例方法列表、协议列表、属性列表添加到主类中后（所以Category中的方法在方法列表中的位置是在主类的同名方法之前的），然后会递归调用所有类的 load 方法，这一切都是在main函数之前执行的。
* 3.分类可以添加哪些内容？  
  实例方法，类方法，协议，属性（添加getter和setter方法，并没有实例变量，添加实例变量需要用关联对象）
* 4.如果工程里有两个分类A和B，两个分类中有一个同名的方法，哪个方法最终生效？  
  取决于分类的编译顺序，最后编译的那个分类的同名方法最终生效，而之前的都会被覆盖掉(这里并不是真正的覆盖，因为其余方法仍然存在，只是访问不到，因为在动态添加类的方法的时候是倒序遍历方法列表的，而最后编译的分类的方法会放在方法列表前面，访问的时候就会先被访问到，同理如果声明了一个和原类方法同名的方法，也会覆盖掉原类的方法)。
* 5.如果声明了两个同名的分类会怎样？  
  会报错，所以第三方的分类，一般都带有命名前缀
* 6.分类能添加成员变量吗？  
  不能。只能通过关联对象(objc\_setAssociatedObject)来模拟实现成员变量，但其实质是关联内容，所有对象的关联内容都放在同一个全局容器哈希表中:AssociationsHashMap,由AssociationsManager统一管理。

**二、扩展**

* 1.一般用扩展做什么？  
  声明私有属性，声明方法（没什么意义），声明私有成员变量
* 2.扩展的特点  
  编译时决议，只能以声明的形式存在，多数情况下寄生在宿主类的.m中，不能为系统类添加扩展。

**三、代理（Delegate）**

代理是一种设计模式，以@protocol形式体现，一般是一对一传递。  
一般以weak关键词以规避循环引用。

**属性关键字**

1.读写权限：readonly,readwrite(默认)  
2.原子性: atomic(默认)，nonatomic。atomic读写线程安全，但效率低，而且不是绝对的安全，比如如果修饰的是数组，那么对数组的读写是安全的，但如果是操作数组进行添加移除其中对象的还，就不保证安全了。  
3.引用计数：

* retain/strong
* assign：修饰基本数据类型，修饰对象类型时，不改变其引用计数，会产生悬垂指针，修饰的对象在被释放后，assign指针仍然指向原对象内存地址，如果使用assign指针继续访问原对象的话，就可能会导致内存泄漏或程序异常
* weak：不改变被修饰对象的引用计数，所指对象在被释放后，weak指针会自动置为nil
* copy：分为深拷贝和浅拷贝  
  浅拷贝：对内存地址的复制，让目标对象指针和原对象指向同一片内存空间会增加引用计数  
  深拷贝：对对象内容的复制，开辟新的内存空间

可变对象的copy和mutableCopy都是深拷贝  
不可变对象的copy是浅拷贝，mutableCopy是深拷贝  
copy方法返回的都是不可变对象

* @property (nonatomic, copy) NSMutableArray \* array;这样写有什么影响？  
  因为copy方法返回的都是不可变对象，所以array对象实际上是不可变的，如果对其进行可变操作如添加移除对象，则会造成程序crash

**通知（NSNotification）**

使用观察者模式来实现的用于跨层传递信息的机制。传递方式是一对多的。

* 如果实现通知机制？

1.应用服务提供商从服务器端把要发送的消息和设备令牌（device token）发送给苹果的消息推送服务器APNs。  
2.APNs根据设备令牌在已注册的设备（iPhone、iPad、iTouch、mac等）查找对应的设备，将消息发送给相应的设备。  
3.客户端设备接将接收到的消息传递给相应的应用程序，应用程序根据用户设置弹出通知消息。

#### 四、Animation

## 请说一下对 CALayer 的认识。

layer 层是涂层绘制、渲染、以及动画的完成者，它无法直接的处理触摸事件（也可以捕捉事件）

layer 包含的方面非常多，常见的属性有 Frame、Bounds、Position、AnchorPoint、Contents 等等。

想详细了解 CALayer 以及动画的 ，可以看看这本书 - [Core-Animation](https://links.jianshu.com/go?to=https%3A%2F%2Flegacy.gitbook.com%2Fbook%2Fzsisme%2Fios-%2Fdetails)

CALayer 的 Contents 有几下几个主要的属性：

###### ContentsRect

单位制（0 - 1），限制显示的范围区域

###### ContentGravity

类似于 ContentMode，不过不是枚举值，而是字符串

###### ContentsScale

决定了物理显示屏是 几@X屏

###### ContentsCenter

跟拉伸有关的属性

#### 五、Block

###### Block变量截获

**1、局部变量截获 是值截获。 比如:**

NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

return n\*num;

};

num = 1;

NSLog(@"%zd",block(2));

这里的输出是6而不是2，原因就是对局部变量num的截获是值截获。  
同样，在block里如果修改变量num，也是无效的，甚至编译器会报错。

NSMutableArray \* arr = [NSMutableArray arrayWithObjects:@"1",@"2", nil];

void(^block)(void) = ^{

NSLog(@"%@",arr);//局部变量

[arr addObject:@"4"];

};

[arr addObject:@"3"];

arr = nil;

block();

打印为1，2，3  
局部对象变量也是一样，截获的是值，而不是指针，在外部将其置为nil，对block没有影响，而该对象调用方法会影响

**2、局部静态变量截获 是指针截获。**

static NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

return n\*num;

};

num = 1;

NSLog(@"%zd",block(2));

输出为2，意味着num = 1这里的修改num值是有效的，即是指针截获。  
同样，在block里去修改变量m，也是有效的。

**3、全局变量，静态全局变量截获：不截获,直接取值。**

我们同样用clang编译看下结果。

static NSInteger num3 = 300;

NSInteger num4 = 3000;

- (void)blockTest

{

NSInteger num = 30;

static NSInteger num2 = 3;

\_\_block NSInteger num5 = 30000;

void(^block)(void) = ^{

NSLog(@"%zd",num);//局部变量

NSLog(@"%zd",num2);//静态变量

NSLog(@"%zd",num3);//全局变量

NSLog(@"%zd",num4);//全局静态变量

NSLog(@"%zd",num5);//\_\_block修饰变量

};

block();

}

编译后

struct \_\_WYTest\_\_blockTest\_block\_impl\_0 {

struct \_\_block\_impl impl;

struct \_\_WYTest\_\_blockTest\_block\_desc\_0\* Desc;

NSInteger num;//局部变量

NSInteger \*num2;//静态变量

\_\_Block\_byref\_num5\_0 \*num5; // by ref//\_\_block修饰变量

\_\_WYTest\_\_blockTest\_block\_impl\_0(void \*fp, struct \_\_WYTest\_\_blockTest\_block\_desc\_0 \*desc, NSInteger \_num, NSInteger \*\_num2, \_\_Block\_byref\_num5\_0 \*\_num5, int flags=0) : num(\_num), num2(\_num2), num5(\_num5->\_\_forwarding) {

impl.isa = &\_NSConcreteStackBlock;

impl.Flags = flags;

impl.FuncPtr = fp;

Desc = desc;

}

};

（ impl.isa = &\_NSConcreteStackBlock;这里注意到这一句，即说明该block是栈block）  
可以看到局部变量被编译成值形式，而静态变量被编成指针形式，全局变量并未截获。而\_\_block修饰的变量也是以指针形式截获的，并且生成了一个新的结构体对象：

struct \_\_Block\_byref\_num5\_0 {

void \*\_\_isa;

\_\_Block\_byref\_num5\_0 \*\_\_forwarding;

int \_\_flags;

int \_\_size;

NSInteger num5;

};

该对象有个属性：num5，即我们用\_\_block修饰的变量。  
这里\_\_forwarding是指向自身的(栈block)。  
一般情况下，如果我们要对block截获的局部变量进行赋值操作需添加\_\_block  
修饰符，而对全局变量，静态变量是不需要添加\_\_block修饰符的。  
另外，block里访问self或成员变量都会去截获self。

###### 什么是Block？

Block是将函数及其执行上下文封装起来的对象。

比如：

NSInteger num = 3;

NSInteger(^block)(NSInteger) = ^NSInteger(NSInteger n){

return n\*num;

};

block(2);

通过clang -rewrite-objc WYTest.m命令编译该.m文件，发现该block被编译成这个形式:

NSInteger num = 3;

NSInteger(\*block)(NSInteger) = ((NSInteger (\*)(NSInteger))&\_\_WYTest\_\_blockTest\_block\_impl\_0((void \*)\_\_WYTest\_\_blockTest\_block\_func\_0, &\_\_WYTest\_\_blockTest\_block\_desc\_0\_DATA, num));

((NSInteger (\*)(\_\_block\_impl \*, NSInteger))((\_\_block\_impl \*)block)->FuncPtr)((\_\_block\_impl \*)block, 2);

其中WYTest是文件名，blockTest是方法名，这些可以忽略。  
其中\_\_WYTest\_\_blockTest\_block\_impl\_0结构体为

struct \_\_WYTest\_\_blockTest\_block\_impl\_0 {

struct \_\_block\_impl impl;

struct \_\_WYTest\_\_blockTest\_block\_desc\_0\* Desc;

NSInteger num;

\_\_WYTest\_\_blockTest\_block\_impl\_0(void \*fp, struct \_\_WYTest\_\_blockTest\_block\_desc\_0 \*desc, NSInteger \_num, int flags=0) : num(\_num) {

impl.isa = &\_NSConcreteStackBlock;

impl.Flags = flags;

impl.FuncPtr = fp;

Desc = desc;

}

};

\_\_block\_impl结构体为

struct \_\_block\_impl {

void \*isa;//isa指针，所以说Block是对象

int Flags;

int Reserved;

void \*FuncPtr;//函数指针

};

block内部有isa指针，所以说其本质也是OC对象  
block内部则为:

static NSInteger \_\_WYTest\_\_blockTest\_block\_func\_0(struct \_\_WYTest\_\_blockTest\_block\_impl\_0 \*\_\_cself, NSInteger n) {

NSInteger num = \_\_cself->num; // bound by copy

return n\*num;

}

所以说 Block是将函数及其执行上下文封装起来的对象  
既然block内部封装了函数，那么它同样也有参数和返回值。

#### 七、多线程

###### 一、 进程：

* 1.进程是一个具有一定独立功能的程序关于某次数据集合的一次运行活动，它是操作系统分配资源的基本单元.
* 2.进程是指在系统中正在运行的一个应用程序，就是一段程序的执行过程,我们可以理解为手机上的一个app.
* 3.每个进程之间是独立的，每个进程均运行在其专用且受保护的内存空间内，拥有独立运行所需的全部资源

###### 二、 线程

* 1.程序执行流的最小单元，线程是进程中的一个实体.
* 2.一个进程要想执行任务,必须至少有一条线程.应用程序启动的时候，系统会默认开启一条线程,也就是主线程

###### 三、 进程和线程的关系

* 1.线程是进程的执行单元，进程的所有任务都在线程中执行
* 2.线程是 CPU 分配资源和调度的最小单位
* 3.一个程序可以对应多个进程(多进程),一个进程中可有多个线程,但至少要有一条线程
* 4.同一个进程内的线程共享进程资源

###### 多进程

打开mac的活动监视器，可以看到很多个进程同时运行

* 进程是程序在计算机上的一次执行活动。当你运行一个程序，你就启动了一个进程。显然，程序是死的(静态的)，进程是活的(动态的)。
* 进程可以分为系统进程和用户进程。凡是用于完成操作系统的各种功能的进程就是系统进程，它们就是处于运行状态下的操作系统本身;所有由用户启动的进程都是用户进程。进程是操作系统进行资源分配的单位。
* 进程又被细化为线程，也就是一个进程下有多个能独立运行的更小的单位。在同一个时间里，同一个计算机系统中如果允许两个或两个以上的进程处于运行状态，这便是多进程。

###### 多线程

* 1.同一时间，CPU只能处理1条线程，只有1条线程在执行。多线程并发执行，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
* 2.如果线程非常非常多，CPU会在N多线程之间调度，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）
* 3.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）
* 4.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

###### 任务

就是执行操作的意思，也就是在线程中执行的那段代码。在 GCD 中是放在 block 中的。执行任务有两种方式：同步执行（sync）和异步执行（async）

* **同步(Sync)**：同步添加任务到指定的队列中，在添加的任务执行结束之前，会一直等待，直到队列里面的任务完成之后再继续执行，即会阻塞线程。只能在当前线程中执行任务(是当前线程，不一定是主线程)，不具备开启新线程的能力。
* **异步(Async)**：线程会立即返回，无需等待就会继续执行下面的任务，不阻塞当前线程。可以在新的线程中执行任务，具备开启新线程的能力(并不一定开启新线程)。如果不是添加到主队列上，异步会在子线程中执行任务

###### 队列

队列（Dispatch Queue）：这里的队列指执行任务的等待队列，即用来存放任务的队列。队列是一种特殊的线性表，采用 FIFO（先进先出）的原则，即新任务总是被插入到队列的末尾，而读取任务的时候总是从队列的头部开始读取。每读取一个任务，则从队列中释放一个任务  
在 GCD 中有两种队列：串行队列和并发队列。两者都符合 FIFO（先进先出）的原则。两者的主要区别是：执行顺序不同，以及开启线程数不同。

* 串行队列（Serial Dispatch Queue）：  
  同一时间内，队列中只能执行一个任务，只有当前的任务执行完成之后，才能执行下一个任务。（只开启一个线程，一个任务执行完毕后，再执行下一个任务）。主队列是主线程上的一个串行队列,是系统自动为我们创建的
* 并发队列（Concurrent Dispatch Queue）：  
  同时允许多个任务并发执行。（可以开启多个线程，并且同时执行任务）。并发队列的并发功能只有在异步（dispatch\_async）函数下才有效

###### iOS中的多线程

**主要有三种：NSThread、NSoperationQueue、GCD**

**1. NSThread：轻量级别的多线程技术**

是我们自己手动开辟的子线程，如果使用的是初始化方式就需要我们自己启动，如果使用的是构造器方式它就会自动启动。只要是我们手动开辟的线程，都需要我们自己管理该线程，不只是启动，还有该线程使用完毕后的资源回收

NSThread \*thread = [[NSThread alloc] initWithTarget:self selector:@selector(testThread:) object:@"我是参数"];

// 当使用初始化方法出来的主线程需要start启动

[thread start];

// 可以为开辟的子线程起名字

thread.name = @"NSThread线程";

// 调整Thread的权限 线程权限的范围值为0 ~ 1 。越大权限越高，先执行的概率就会越高，由于是概率，所以并不能很准确的的实现我们想要的执行顺序，默认值是0.5

thread.threadPriority = 1;

// 取消当前已经启动的线程

[thread cancel];

// 通过遍历构造器开辟子线程

[NSThread detachNewThreadSelector:@selector(testThread:) toTarget:self withObject:@"构造器方式"];

* performSelector...只要是NSObject的子类或者对象都可以通过调用方法进入子线程和主线程，其实这些方法所开辟的子线程也是NSThread的另一种体现方式。  
  在编译阶段并不会去检查方法是否有效存在，如果不存在只会给出警告

//在当前线程。延迟1s执行。响应了OC语言的动态性:延迟到运行时才绑定方法

[self performSelector:@selector(aaa) withObject:nil afterDelay:1];

// 回到主线程。waitUntilDone:是否将该回调方法执行完在执行后面的代码，如果为YES:就必须等回调方法执行完成之后才能执行后面的代码，说白了就是阻塞当前的线程；如果是NO：就是不等回调方法结束，不会阻塞当前线程

[self performSelectorOnMainThread:@selector(aaa) withObject:nil waitUntilDone:YES];

//开辟子线程

[self performSelectorInBackground:@selector(aaa) withObject:nil];

//在指定线程执行

[self performSelector:@selector(aaa) onThread:[NSThread currentThread] withObject:nil waitUntilDone:YES]

需要注意的是：如果是带afterDelay的延时函数，会在内部创建一个 NSTimer，然后添加到当前线程的Runloop中。也就是如果当前线程没有开启runloop，该方法会失效。在子线程中，需要启动runloop(注意调用顺序)

[self performSelector:@selector(aaa) withObject:nil afterDelay:1];

[[NSRunLoop currentRunLoop] run];

而performSelector:withObject:只是一个单纯的消息发送，和时间没有一点关系。所以不需要添加到子线程的Runloop中也能执行

**2、GCD 对比 NSOprationQueue**

我们要明确NSOperationQueue与GCD之间的关系  
GCD是面向底层的C语言的API，NSOpertaionQueue用GCD构建封装的，是GCD的高级抽象。

1、GCD执行效率更高，而且由于队列中执行的是由block构成的任务，这是一个轻量级的数据结构，写起来更方便  
2、GCD只支持FIFO的队列，而NSOperationQueue可以通过设置最大并发数，设置优先级，添加依赖关系等调整执行顺序  
3、NSOperationQueue甚至可以跨队列设置依赖关系，但是GCD只能通过设置串行队列，或者在队列内添加barrier(dispatch\_barrier\_async)任务，才能控制执行顺序,较为复杂  
4、NSOperationQueue因为面向对象，所以支持KVO，可以监测operation是否正在执行（isExecuted）、是否结束（isFinished）、是否取消（isCanceld）

* 实际项目开发中，很多时候只是会用到异步操作，不会有特别复杂的线程关系管理，所以苹果推崇的且优化完善、运行快速的GCD是首选
* 如果考虑异步操作之间的事务性，顺序行，依赖关系，比如多线程并发下载，GCD需要自己写更多的代码来实现，而NSOperationQueue已经内建了这些支持
* 不论是GCD还是NSOperationQueue，我们接触的都是任务和队列，都没有直接接触到线程，事实上线程管理也的确不需要我们操心，系统对于线程的创建，调度管理和释放都做得很好。而NSThread需要我们自己去管理线程的生命周期，还要考虑线程同步、加锁问题，造成一些性能上的开销

###### GCD---队列

iOS中，有GCD、NSOperation、NSThread等几种多线程技术方案。

而GCD共有三种队列类型：  
main queue：通过dispatch\_get\_main\_queue()获得，这是一个与主线程相关的串行队列。

global queue：全局队列是并发队列，由整个进程共享。存在着高、中、低三种优先级的全局队列。调用dispath\_get\_global\_queue并传入优先级来访问队列。

自定义队列：通过函数dispatch\_queue\_create创建的队列。

###### 死锁

死锁就是队列引起的循环等待

**1、一个比较常见的死锁例子:主队列同步**

- (void)viewDidLoad {

[super viewDidLoad];

dispatch\_sync(dispatch\_get\_main\_queue(), ^{

NSLog(@"deallock");

});

// Do any additional setup after loading the view, typically from a nib.

}

在主线程中运用主队列同步，也就是把任务放到了主线程的队列中。  
同步对于任务是立刻执行的，那么当把任务放进主队列时，它就会立马执行,只有执行完这个任务，viewDidLoad才会继续向下执行。  
而viewDidLoad和任务都是在主队列上的，由于队列的先进先出原则，任务又需等待viewDidLoad执行完毕后才能继续执行，viewDidLoad和这个任务就形成了相互循环等待，就造成了死锁。  
想避免这种死锁，可以将同步改成异步dispatch\_async,或者将dispatch\_get\_main\_queue换成其他串行或并行队列，都可以解决。

**2、同样，下边的代码也会造成死锁：**

dispatch\_queue\_t serialQueue = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_SERIAL);

dispatch\_async(serialQueue, ^{

dispatch\_sync(serialQueue, ^{

NSLog(@"deadlock");

});

});

外面的函数无论是同步还是异步都会造成死锁。  
这是因为里面的任务和外面的任务都在同一个serialQueue队列内，又是同步，这就和上边主队列同步的例子一样造成了死锁  
解决方法也和上边一样，将里面的同步改成异步dispatch\_async,或者将serialQueue换成其他串行或并行队列，都可以解决

dispatch\_queue\_t serialQueue = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_SERIAL);

dispatch\_queue\_t serialQueue2 = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_SERIAL);

dispatch\_async(serialQueue, ^{

dispatch\_sync(serialQueue2, ^{

NSLog(@"deadlock");

});

});

这样是不会死锁的,并且serialQueue和serialQueue2是在同一个线程中的。

###### GCD任务执行顺序

1、串行队列先异步后同步

dispatch\_queue\_t serialQueue = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_SERIAL);

NSLog(@"1");

dispatch\_async(serialQueue, ^{

NSLog(@"2");

});

NSLog(@"3");

dispatch\_sync(serialQueue, ^{

NSLog(@"4");

});

NSLog(@"5");

打印顺序是13245  
原因是:  
首先先打印1  
接下来将任务2其添加至串行队列上，由于任务2是异步，不会阻塞线程，继续向下执行，打印3  
然后是任务4,将任务4添加至串行队列上，因为任务4和任务2在同一串行队列，根据队列先进先出原则，任务4必须等任务2执行后才能执行，又因为任务4是同步任务，会阻塞线程，只有执行完任务4才能继续向下执行打印5  
所以最终顺序就是13245。  
这里的任务4在主线程中执行，而任务2在子线程中执行。  
如果任务4是添加到另一个串行队列或者并行队列，则任务2和任务4无序执行(可以添加多个任务看效果)

**2、performSelector**

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

[self performSelector:@selector(test:) withObject:nil afterDelay:0];

});

这里的test方法是不会去执行的，原因在于

- (void)performSelector:(SEL)aSelector withObject:(nullable id)anArgument afterDelay:(NSTimeInterval)delay;

这个方法要创建提交任务到runloop上的，而gcd底层创建的线程是默认没有开启对应runloop的，所有这个方法就会失效。  
而如果将dispatch\_get\_global\_queue改成主队列，由于主队列所在的主线程是默认开启了runloop的，就会去执行(将dispatch\_async改成同步，因为同步是在当前线程执行，那么如果当前线程是主线程，test方法也是会去执行的)。

###### dispatch\_barrier\_async

**1、问：怎么用GCD实现多读单写？**

多读单写的意思就是：可以多个读者同时读取数据，而在读的时候，不能取写入数据。并且，在写的过程中，不能有其他写者去写。即读者之间是并发的，写者与读者或其他写者是互斥的。

这里的写处理就是通过栅栏的形式去写。  
就可以用dispatch\_barrier\_sync(栅栏函数)去实现

**2、dispatch\_barrier\_sync的用法：**

dispatch\_queue\_t concurrentQueue = dispatch\_queue\_create("test", DISPATCH\_QUEUE\_CONCURRENT);

for (NSInteger i = 0; i < 10; i++) {

dispatch\_sync(concurrentQueue, ^{

NSLog(@"%zd",i);

});

}

dispatch\_barrier\_sync(concurrentQueue, ^{

NSLog(@"barrier");

});

for (NSInteger i = 10; i < 20; i++) {

dispatch\_sync(concurrentQueue, ^{

NSLog(@"%zd",i);

});

}

这里的dispatch\_barrier\_sync上的队列要和需要阻塞的任务在同一队列上，否则是无效的。  
从打印上看，任务0-9和任务任务10-19因为是异步并发的原因，彼此是无序的。而由于栅栏函数的存在，导致顺序必然是先执行任务0-9，再执行栅栏函数，再去执行任务10-19。

* dispatch\_barrier\_sync: Submits a barrier block object for execution and waits until that block completes.(提交一个栅栏函数在执行中,它会等待栅栏函数执行完)
* dispatch\_barrier\_async: Submits a barrier block for asynchronous execution and returns immediately.(提交一个栅栏函数在异步执行中,它会立马返回)  
  而dispatch\_barrier\_sync和dispatch\_barrier\_async的区别也就在于会不会阻塞当前线程  
  比如，上述代码如果在dispatch\_barrier\_async后随便加一条打印，则会先去执行该打印，再去执行任务0-9和栅栏函数；而如果是dispatch\_barrier\_sync，则会在任务0-9和栅栏函数后去执行这条打印。

**3、则可以这样设计多读单写：**

- (id)readDataForKey:(NSString \*)key

{

\_\_block id result;

dispatch\_sync(\_concurrentQueue, ^{

result = [self valueForKey:key];

});

return result;

}

- (void)writeData:(id)data forKey:(NSString \*)key

{

dispatch\_barrier\_async(\_concurrentQueue, ^{

[self setValue:data forKey:key];

});

}

###### dispatch\_group\_async

场景：在n个耗时并发任务都完成后，再去执行接下来的任务。比如，在n个网络请求完成后去刷新UI页面。

dispatch\_queue\_t concurrentQueue = dispatch\_queue\_create("test1", DISPATCH\_QUEUE\_CONCURRENT);

dispatch\_group\_t group = dispatch\_group\_create();

for (NSInteger i = 0; i < 10; i++) {

dispatch\_group\_async(group, concurrentQueue, ^{

sleep(1);

NSLog(@"%zd:网络请求",i);

});

}

dispatch\_group\_notify(group, dispatch\_get\_main\_queue(), ^{

NSLog(@"刷新页面");

});

###### Dispatch Semaphore

GCD 中的信号量是指 Dispatch Semaphore，是持有计数的信号。

Dispatch Semaphore 提供了三个函数

**1.dispatch\_semaphore\_create：创建一个Semaphore并初始化信号的总量**  
**2.dispatch\_semaphore\_signal：发送一个信号，让信号总量加1**  
**3.dispatch\_semaphore\_wait：可以使总信号量减1，当信号总量为0时就会一直等待（阻塞所在线程），否则就可以正常执行。**

Dispatch Semaphore 在实际开发中主要用于：

* 保持线程同步，将异步执行任务转换为同步执行任务
* 保证线程安全，为线程加锁

**1、保持线程同步：**

dispatch\_semaphore\_t semaphore = dispatch\_semaphore\_create(0);

\_\_block NSInteger number = 0;

dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{

number = 100;

dispatch\_semaphore\_signal(semaphore);

});

dispatch\_semaphore\_wait(semaphore, DISPATCH\_TIME\_FOREVER);

NSLog(@"semaphore---end,number = %zd",number);

dispatch\_semaphore\_wait加锁阻塞了当前线程，dispatch\_semaphore\_signal解锁后当前线程继续执行

**2、保证线程安全，为线程加锁：**

在线程安全中可以将dispatch\_semaphore\_wait看作加锁，而dispatch\_semaphore\_signal看作解锁  
首先创建全局变量

\_semaphore = dispatch\_semaphore\_create(1);

注意到这里的初始化信号量是1。

- (void)asyncTask

{

dispatch\_semaphore\_wait(\_semaphore, DISPATCH\_TIME\_FOREVER);

count++;

sleep(1);

NSLog(@"执行任务:%zd",count);

dispatch\_semaphore\_signal(\_semaphore);

}

异步并发调用asyncTask

for (NSInteger i = 0; i < 100; i++) {

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

[self asyncTask];

});

}

然后发现打印是从任务1顺序执行到100，没有发生两个任务同时执行的情况。

原因如下:  
在子线程中并发执行asyncTask，那么第一个添加到并发队列里的，会将信号量减1，此时信号量等于0，可以执行接下来的任务。而并发队列中其他任务，由于此时信号量不等于0，必须等当前正在执行的任务执行完毕后调用dispatch\_semaphore\_signal将信号量加1，才可以继续执行接下来的任务，以此类推，从而达到线程加锁的目的。

###### 延时函数(dispatch\_after)

dispatch\_after能让我们添加进队列的任务延时执行，该函数并不是在指定时间后执行处理，而只是在指定时间追加处理到dispatch\_queue

//第一个参数是time，第二个参数是dispatch\_queue，第三个参数是要执行的block

dispatch\_after(dispatch\_time(DISPATCH\_TIME\_NOW, (int64\_t)(2 \* NSEC\_PER\_SEC)), dispatch\_get\_main\_queue(), ^{

NSLog(@"dispatch\_after");

});

由于其内部使用的是dispatch\_time\_t管理时间，而不是NSTimer。  
所以如果在子线程中调用，相比performSelector:afterDelay,不用关心runloop是否开启

###### 使用dispatch\_once实现单例

+ (instancetype)shareInstance {

static dispatch\_once\_t onceToken;

static id instance = nil;

dispatch\_once(&onceToken, ^{

instance = [[self alloc] init];

});

return instance;

}

###### NSOperationQueue的优点

NSOperation、NSOperationQueue 是苹果提供给我们的一套多线程解决方案。实际上 NSOperation、NSOperationQueue 是基于 GCD 更高一层的封装，完全面向对象。但是比 GCD 更简单易用、代码可读性也更高。

1、可以添加任务依赖，方便控制执行顺序

2、可以设定操作执行的优先级

3、任务执行状态控制:isReady,isExecuting,isFinished,isCancelled

如果只是重写NSOperation的main方法，由底层控制变更任务执行及完成状态，以及任务退出  
如果重写了NSOperation的start方法，自行控制任务状态  
系统通过KVO的方式移除isFinished==YES的NSOperation

4、可以设置最大并发量

###### NSOperation和NSOperationQueue

* **操作（Operation）：**

执行操作的意思，换句话说就是你在线程中执行的那段代码。  
在 GCD 中是放在 block 中的。在 NSOperation 中，使用 NSOperation 子类 NSInvocationOperation、NSBlockOperation，或者自定义子类来封装操作。

* **操作队列（Operation Queues）：**

这里的队列指操作队列，即用来存放操作的队列。不同于 GCD 中的调度队列 FIFO（先进先出）的原则。NSOperationQueue 对于添加到队列中的操作，首先进入准备就绪的状态（就绪状态取决于操作之间的依赖关系），然后进入就绪状态的操作的开始执行顺序（非结束执行顺序）由操作之间相对的优先级决定（优先级是操作对象自身的属性）。

操作队列通过设置最大并发操作数（maxConcurrentOperationCount）来控制并发、串行。

NSOperationQueue 为我们提供了两种不同类型的队列：主队列和自定义队列。主队列运行在主线程之上，而自定义队列在后台执行。

###### NSThread+runloop实现常驻线程

NSThread在实际开发中比较常用到的场景就是去实现常驻线程。

* 由于每次开辟子线程都会消耗cpu，在需要频繁使用子线程的情况下，频繁开辟子线程会消耗大量的cpu，而且创建线程都是任务执行完成之后也就释放了，不能再次利用，那么如何创建一个线程可以让它可以再次工作呢？也就是创建一个常驻线程。

首先常驻线程既然是常驻，那么我们可以用GCD实现一个单例来保存NSThread

+ (NSThread \*)shareThread {

static NSThread \*shareThread = nil;

static dispatch\_once\_t oncePredicate;

dispatch\_once(&oncePredicate, ^{

shareThread = [[NSThread alloc] initWithTarget:self selector:@selector(threadTest) object:nil];

[shareThread setName:@"threadTest"];

[shareThread start];

});

return shareThread;

}

这样创建的thread就不会销毁了吗？

[self performSelector:@selector(test) onThread:[ViewController shareThread] withObject:nil waitUntilDone:NO];

- (void)test

{

NSLog(@"test:%@", [NSThread currentThread]);

}

并没有打印，说明test方法没有被调用。  
那么可以用runloop来让线程常驻

+ (NSThread \*)shareThread {

static NSThread \*shareThread = nil;

static dispatch\_once\_t oncePredicate;

dispatch\_once(&oncePredicate, ^{

shareThread = [[NSThread alloc] initWithTarget:self selector:@selector(threadTest2) object:nil];

[shareThread setName:@"threadTest"];

[shareThread start];

});

return shareThread;

}

+ (void)threadTest

{

@autoreleasepool {

NSRunLoop \*runLoop = [NSRunLoop currentRunLoop];

[runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];

[runLoop run];

}

}

这时候再去调用performSelector就有打印了。

###### 自旋锁与互斥锁

**自旋锁：**

是一种用于保护多线程共享资源的锁，与一般互斥锁（mutex）不同之处在于当自旋锁尝试获取锁时以忙等待（busy waiting）的形式不断地循环检查锁是否可用。当上一个线程的任务没有执行完毕的时候（被锁住），那么下一个线程会一直等待（不会睡眠），当上一个线程的任务执行完毕，下一个线程会立即执行。  
在多CPU的环境中，对持有锁较短的程序来说，使用自旋锁代替一般的互斥锁往往能够提高程序的性能。

**互斥锁：**

当上一个线程的任务没有执行完毕的时候（被锁住），那么下一个线程会进入睡眠状态等待任务执行完毕，当上一个线程的任务执行完毕，下一个线程会自动唤醒然后执行任务。

**总结：**

自旋锁会忙等: 所谓忙等，即在访问被锁资源时，调用者线程不会休眠，而是不停循环在那里，直到被锁资源释放锁。  
　　互斥锁会休眠: 所谓休眠，即在访问被锁资源时，调用者线程会休眠，此时cpu可以调度其他线程工作。直到被锁资源释放锁。此时会唤醒休眠线程。  
　　  
**优缺点：**

自旋锁的优点在于，因为自旋锁不会引起调用者睡眠，所以不会进行线程调度，CPU时间片轮转等耗时操作。所有如果能在很短的时间内获得锁，自旋锁的效率远高于互斥锁。

缺点在于，自旋锁一直占用CPU，他在未获得锁的情况下，一直运行－－自旋，所以占用着CPU，如果不能在很短的时 间内获得锁，这无疑会使CPU效率降低。自旋锁不能实现递归调用。  
　　  
**自旋锁：atomic、OSSpinLock、dispatch\_semaphore\_t**  
**互斥锁：pthread\_mutex、@ synchronized、NSLock、NSConditionLock 、NSCondition、NSRecursiveLock**

#### 八、内存管理

## 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，感兴趣的可以看一下这两篇文章：

* [MLeaksFinder：精准 iOS 内存泄露检测工具](https://links.jianshu.com/go?to=http%3A%2F%2Fwereadteam.github.io%2F2016%2F02%2F22%2FMLeaksFinder%2F)
* [MLeaksFinder 新特性](https://links.jianshu.com/go?to=http%3A%2F%2Fwereadteam.github.io%2F2016%2F07%2F20%2FMLeaksFinder2%2F)

## 在 MRC 下如何重写属性的 Setter 和 Getter?

###### 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];

}

###### 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];

};

## 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

## 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上 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。

# iOS 内存管理面试题（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 archive

## 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\*。

## 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。

#### 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。

## 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。

###### 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。

###### 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。

###### 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

## 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

## 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。

## 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。

## 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。

## 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。

###### 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。

## 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。

## 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。

在主线程执行的代码，通常是写在诸如事件回调、Timer回调内的。这些回调会被 RunLoop 创建好的 AutoreleasePool 环绕着，所以不会出现内存泄漏，开发者也不必显示创建 Pool 了。

## 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

## 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。

## 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

## 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。

## \_\_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。

## \_\_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。

#### 九、设计模式

## 如何设计一个时长统计框架？

### 记录器

* 页面式记录器
* 流式记录器
* 自定义式

### 记录管理者

* 内存记录缓存
* 磁盘存储
* 上传器

### 如何降低数据的丢失率？

* 定期写入磁盘
* 每当达到某个值的时候，就写入磁盘

### 记录上传的时机

* 前后台切换的时候可以上传
* 从无网到有网切换的时候可以上传

### 上传时机的选择

* 立即上传
* 定时上传
* 延时上传

## 如何设计一个图片缓存框架？

可以模仿 SDWebImage 来实现。

#### 构成

* Manager
* 内存缓存
* 磁盘缓存
* 网络下载
* Code Manager
  + 图片解码
  + 图片解压缩

图片的存储是以图片的单向 hash 值为 Key

### 内存设计需要考虑的问题

##### 存储的 Size

因为内存的空间有限，我们针对不同尺寸的图片，给出不同的方案

* 10K 以下的50个
* 100Kb 以下的20个
* 100kb 以上的10个

##### 淘汰的策略

内存的淘汰策略 采取 LRU（最近最少使用算法）

###### 触发淘汰策略的时机有三种

* 1.定期检查（不建议，耗性能）
* 2.提高检查触发频率（一定要注意开销）
  + 1.前后台切换的时候
  + 2.每次读写的时候

### 磁盘设计需要考虑的问题

* 存储方式
* 大小限制（有固定的大小）
* 移除策略（可以设置为7天或者15天）

### 网络设计需要考虑的问题

* 图片请求的最大并发量
* 请求超时策略
* 请求优先级

### 图片解码

应用 策略模式，针对 jpg、png、gif 等不同的图片格式进行解码

##### 图片解码的时机

* 在 子线程 图片刚下载完时
* 在 子线程 刚从磁盘读取完时

避免在主线程解压缩、解码，避免卡顿

## 编程中的六大设计原则？

##### 1.单一职责原则

通俗地讲就是一个类只做一件事

* CALayer：动画和视图的显示。
* UIView：只负责事件传递、事件响应。

##### 2.开闭原则

对修改关闭，对扩展开放。  
要考虑到后续的扩展性，而不是在原有的基础上来回修改

##### 3.接口隔离原则

使用多个专门的协议、而不是一个庞大臃肿的协议

* UITableviewDelegate
* UITableViewDataSource

##### 4.依赖倒置原则

抽象不应该依赖于具体实现、具体实现可以依赖于抽象。  
调用接口感觉不到内部是如何操作的

##### 5.里氏替换原则

父类可以被子类无缝替换，且原有的功能不受任何影响

例如 KVO

##### 6.迪米特法则

一个对象应当对其他对象尽可能少的了解，实现高聚合、低耦合

#### 十、数据安全及加密

## RSA非对称加密

对称加密[算法]在加密和解密时使用的是同一个秘钥；而[非对称加密算法]需要两个[密钥]来进行加密和解密，这两个秘钥是[公开密钥]（public key，简称公钥）和私有密钥（private key，简称私钥）。

### RSA加密

与对称加密[算法]不同，[非对称加密算法]需要两个[密钥]：[公开密钥]（publickey）和私有密钥（privatekey）。公开密钥与私有密钥是一对，如果用公开密钥对数据进行加密，只有用对应的私有密钥才能解密；如果用私有密钥对数据进行加密，那么只有用对应的公开密钥才能解密。因为加密和解密使用的是两个不同的[密钥]，所以这种算法叫作[非对称加密算法]。

#### RSA加密原理

RSA是常用的加密模式，其加密原理可用以下的例子进行简要的论述。

随机取两个质数

P = 61;

q = 53;

N = P \* Q = 3233;

// E是1-n之间的一个随机的质数

E = 17;

// D是通过一系列数学运算得出的一个数字，

// 运算方法后续会附上阮一峰老师的两篇文章链接

// (N,D)(N,E)要满足可以互相解值运算

// 假如(N,D)是公钥，（N,E）是私钥

// 满足私钥加密，公钥解密或者反过来公钥加密，私钥解密。

// 也要满足只知道（N，D）就想知道（N,E），那就要把N这个大的整数进行因数分解。

// 因数分解只能使用暴力穷举，N越大，相应的也就越安全

// 当 N 大到1024位或者2048位时，以目前的技术破解几乎不可能，所以很安全

**简述 SSL 加密的过程用了哪些加密方法，为何这么作？**

SSL 加密的过程之前有些过，此处不再赘述。

SSL 加密，在过程中实际使用了 对称加密 和 非对称加密 的结合。

主要的考虑是先使用 非对称加密 进行连接，这样做是为了避免中间人攻击秘钥被劫持，但是 非对称加密 的效率比较低。所以一旦建立了安全的连接之后，就可以使用轻量的 对称加密。

#### 十一、数据结构与算法

#### 集合结构 线性结构 树形结构 图形结构

* 1.1、集合结构 说白了就是一个集合，就是一个圆圈中有很多个元素，元素与元素之间没有任何关系 这个很简单
* 1.2、线性结构 说白了就是一个条线上站着很多个人。 这条线不一定是直的。也可以是弯的。也可以是值的 相当于一条线被分成了好几段的样子 （发挥你的想象力）。 线性结构是一对一的关系
* 1.3、树形结构 说白了 做开发的肯定或多或少的知道xml 解析 树形结构跟他非常类似。也可以想象成一个金字塔。树形结构是一对多的关系
* 1.4、图形结构 这个就比较复杂了。他呢 无穷。无边 无向（没有方向）图形机构 你可以理解为多对多 类似于我们人的交集关系

#### 数据结构的存储

数据结构的存储一般常用的有两种 顺序存储结构 和 链式存储结构

* 2.1 顺序存储结构

发挥想象力啊。 举个列子。数组。1-2-3-4-5-6-7-8-9-10。这个就是一个顺序存储结构 ，存储是按顺序的 举例说明啊。 栈。做开发的都熟悉。栈是先进后出 ，后进先出的形式 对不对 ？！他的你可以这样理解

hello world 在栈里面从栈底到栈顶的逻辑依次为 h-e-l-l-o-w-o-r-l-d 这就是顺序存储 再比如 队列 ，队列是先进先出的对吧，从头到尾 h-e-l-l-o-w-o-r-l-d 就是这样排对的

* 2.2 链式存储结构

再次发挥想象力 这个稍微复杂一点 这个图片我一直弄好 ，回头找美工问问，再贴上 例如 还是一个数组

1-2-3-4-5-6-7-8-9-10 链式存储就不一样了 1(地址)-2(地址)-7(地址)-4(地址)-5(地址)-9(地址)-8(地址)-3(地址)-6(地址)-10(地址)。每个数字后面跟着一个地址 而且存储形式不再是顺序 ，也就说顺序乱了，1（地址） 1后面跟着的这个地址指向的是2，2后面的地址指向的是3，3后面的地址指向是谁你应该清楚了吧。他执行的时候是 1(地址)-2(地址)-3(地址)-4(地址)-5(地址)-6(地址)-7(地址)-8(地址)-9(地址)-10(地址)，但是存储的时候就是完全随机的。明白了？！

#### 单向链表\双向链表\循环链表

还是举例子。理解最重要。不要去死记硬背 哪些什么。定义啊。逻辑啊。理解才是最重要滴

* 3.1 单向链表

A->B->C->D->E->F->G->H. 这就是单向链表 H 是头 A 是尾 像一个只有一个头的火车一样 只能一个头拉着跑

* 3.2 双向链表

数组和链表区别：  
数组：数组元素在内存上连续存放，可以通过下标查找元素；插入、删除需要移动大量元素，比较适用于元素很少变化的情况  
链表：链表中的元素在内存中不是顺序存储的，查找慢，插入、删除只需要对元素指针重新赋值，效率高

* 3.3 循环链表

循环链表是与单向链表一样，是一种链式的存储结构，所不同的是，循环链表的最后一个结点的指针是指向该循环链表的第一个结点或者表头结点，从而构成一个环形的链。发挥想象力 A->B->C->D->E->F->G->H->A. 绕成一个圈。就像蛇吃自己的这就是循环 不需要去死记硬背哪些理论知识。

#### 二叉树／平衡二叉树

* 4.1 什么是二叉树

树形结构下,两个节点以内 都称之为二叉树 不存在大于2 的节点 分为左子树 右子树 有顺序 不能颠倒 ，懵逼了吧，你肯定会想这是什么玩意，什么左子树右子树 ，都什么跟什么鬼？ 现在我以普通话再讲一遍，你把二叉树看成一个人 ，人的头呢就是树的根 ，左子树就是左手，右子树就是右手，左右手可以都没有（残疾嘛，声明一下，绝非歧视残疾朋友，勿怪，勿怪就是举个例子，i am very sorry） , 左右手呢可以有一个，就是不能颠倒。这样讲应该明白了吧

**二叉树有五种表现形式**

1.空的树（没有节点）可以理解为什么都没 像空气一样  
2.只有根节点。 （理解一个人只有一个头 其他的什么都没，说的有点恐怖）  
3.只有左子树 （一个头 一个左手 感觉越来越写不下去了）  
4.只有右子树  
5.左右子树都有

二叉树可以转换成森林 树也可以转换成二叉树。这里就不介绍了 你做项目绝对用不到  
数据结构大致介绍这么多吧。理解为主, 别死记，死记没什么用

**1、不用中间变量,用两种方法交换A和B的值**

// 1.中间变量

void swap(int a, int b) {

int temp = a;

a = b;

b = temp;

}

// 2.加法

void swap(int a, int b) {

a = a + b;

b = a - b;

a = a - b;

}

// 3.异或（相同为0，不同为1. 可以理解为不进位加法）

void swap(int a, int b) {

a = a ^ b;

b = a ^ b;

a = a ^ b;

}

**2、求最大公约数**

/\*\* 1.直接遍历法 \*/

int maxCommonDivisor(int a, int b) {

int max = 0;

for (int i = 1; i <=b; i++) {

if (a % i == 0 && b % i == 0) {

max = i;

}

}

return max;

}

/\*\* 2.辗转相除法 \*/

int maxCommonDivisor(int a, int b) {

int r;

while(a % b > 0) {

r = a % b;

a = b;

b = r;

}

return b;

}

// 扩展：最小公倍数 = (a \* b)/最大公约数

**3、模拟栈操作**

* 栈是一种数据结构，特点：先进后出 -
* 练习：使用全局变量模拟栈的操作

\*\*#include <stdio.h>

# include <stdbool.h>

# include <assert.h>\*\*

//保护全局变量：在全局变量前加static后，这个全局变量就只能在本文件中使用  
static int data[1024];//栈最多能保存1024个数据  
static int count = 0;//目前已经放了多少个数(相当于栈顶位置)

//数据入栈 push

void push(int x){

assert(!full());//防止数组越界

data[count++] = x;

}

//数据出栈 pop

int pop(){

assert(!empty());

return data[--count];

}

//查看栈顶元素 top

int top(){

assert(!empty());

return data[count-1];

}

//查询栈满 full

bool full() {

if(count >= 1024) {

return 1;

}

return 0;

}

//查询栈空 empty

bool empty() {

if(count <= 0) {

return 1;

}

return 0;

}

int main(){

//入栈

for (int i = 1; i <= 10; i++) {

push(i);

}

//出栈

while(!empty()){

printf("%d ", top()); //栈顶元素

pop(); //出栈

}

printf("\n");

return 0;

}

**4、排序算法**

选择排序、冒泡排序、插入排序三种排序算法可以总结为如下：  
都将数组分为已排序部分和未排序部分。

1.选择排序将已排序部分定义在左端，然后选择未排序部分的最小元素和未排序部分的第一个元素交换。

2.冒泡排序将已排序部分定义在右端，在遍历未排序部分的过程执行交换，将最大元素交换到最右端。

3.插入排序将已排序部分定义在左端，将未排序部分元的第一个元素插入到已排序部分合适的位置。

**4.1、选择排序**

* 【选择排序】：最值出现在起始端
* 第1趟：在n个数中找到最小(大)数与第一个数交换位置
* 第2趟：在剩下n-1个数中找到最小(大)数与第二个数交换位置
* 重复这样的操作...依次与第三个、第四个...数交换位置
* 第n-1趟，最终可实现数据的升序（降序）排列。

void selectSort(int \*arr, int length) {

for (int i = 0; i < length - 1; i++) { //趟数

for (int j = i + 1; j < length; j++) { //比较次数

if (arr[i] > arr[j]) {

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

**4.2、冒泡排序**

* 【冒泡排序】：相邻元素两两比较，比较完一趟，最值出现在末尾
* 第1趟：依次比较相邻的两个数，不断交换（小数放前，大数放后）逐个推进，最值最后出现在第n个元素位置
* 第2趟：依次比较相邻的两个数，不断交换（小数放前，大数放后）逐个推进，最值最后出现在第n-1个元素位置
* …… ……
* 第n-1趟：依次比较相邻的两个数，不断交换（小数放前，大数放后）逐个推进，最值最后出现在第2个元素位置

void bublleSort(int \*arr, int length) {

for(int i = 0; i < length - 1; i++) { //趟数

for(int j = 0; j < length - i - 1; j++) { //比较次数

if(arr[j] > arr[j+1]) {

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

#### 5、折半查找（二分查找）

折半查找：优化查找时间（不用遍历全部数据）  
折半查找的原理：

* 1> 数组必须是有序的
* 2> 必须已知min和max（知道范围）
* 3> 动态计算mid的值，取出mid对应的值进行比较
* 4> 如果mid对应的值大于要查找的值，那么max要变小为mid-1
* 5> 如果mid对应的值小于要查找的值，那么min要变大为mid+1

// 已知一个有序数组, 和一个key, 要求从数组中找到key对应的索引位置

int findKey(int \*arr, int length, int key) {

int min = 0, max = length - 1, mid;

while (min <= max) {

mid = (min + max) / 2; //计算中间值

if (key > arr[mid]) {

min = mid + 1;

} else if (key < arr[mid]) {

max = mid - 1;

} else {

return mid;

}

}

return -1;

}

#### 字符串反转

给定字符串 "hello,world",实现将其反转。输出结果：dlrow,olleh

- (void)charReverse

{

NSString \* string = @"hello,world";

NSLog(@"%@",string);

NSMutableString \* reverString = [NSMutableString stringWithString:string];

for (NSInteger i = 0; i < (string.length + 1)/2; i++) {

[reverString replaceCharactersInRange:NSMakeRange(i, 1) withString:[string substringWithRange:NSMakeRange(string.length - i - 1, 1)]];

[reverString replaceCharactersInRange:NSMakeRange(string.length - i - 1, 1) withString:[string substringWithRange:NSMakeRange(i, 1)]];

}

NSLog(@"reverString:%@",reverString);

//C

char ch[100];

memcpy(ch, [string cStringUsingEncoding:NSUTF8StringEncoding], [string length]);

//设置两个指针，一个指向字符串开头，一个指向字符串末尾

char \* begin = ch;

char \* end = ch + strlen(ch) - 1;

//遍历字符数组，逐步交换两个指针所指向的内容，同时移动指针到对应的下个位置，直至begin>=end

while (begin < end) {

char temp = \*begin;

\*(begin++) = \*end;

\*(end--) = temp;

}

NSLog(@"reverseChar[]:%s",ch);

}

#### 序数组合并

将有序数组 {1,4,6,7,9} 和 {2,3,5,6,8,9,10,11,12} 合并为  
{1,2,3,4,5,6,6,7,8,9,9,10,11,12}

- (void)orderListMerge

{

int aLen = 5,bLen = 9;

int a[] = {1,4,6,7,9};

int b[] = {2,3,5,6,8,9,10,11,12};

[self printList:a length:aLen];

[self printList:b length:bLen];

int result[14];

int p = 0,q = 0,i = 0;//p和q分别为a和b的下标，i为合并结果数组的下标

//任一数组没有达到s边界则进行遍历

while (p < aLen && q < bLen) {

//如果a数组对应位置的值小于b数组对应位置的值,则存储a数组的值，并移动a数组的下标与合并结果数组的下标

if (a[p] < b[q]) result[i++] = a[p++];

//否则存储b数组的值，并移动b数组的下标与合并结果数组的下标

else result[i++] = b[q++];

}

//如果a数组有剩余，将a数组剩余部分拼接到合并结果数组的后面

while (++p < aLen) {

result[i++] = a[p];

}

//如果b数组有剩余，将b数组剩余部分拼接到合并结果数组的后面

while (q < bLen) {

result[i++] = b[q++];

}

[self printList:result length:aLen + bLen];

}

- (void)printList:(int [])list length:(int)length

{

for (int i = 0; i < length; i++) {

printf("%d ",list[i]);

}

printf("\n");

}

#### HASH算法

* 哈希表  
  例：给定值是字母a，对应ASCII码值是97，数组索引下标为97。  
  这里的ASCII码，就算是一种哈希函数，存储和查找都通过该函数，有效地提高查找效率。
* 在一个字符串中找到第一个只出现一次的字符。如输入"abaccdeff"，输出'b'字符(char)是一个长度为8的数据类型，因此总共有256种可能。每个字母根据其ASCII码值作为数组下标对应数组种的一个数字。数组中存储的是每个字符出现的次数。

- (void)hashTest

{

NSString \* testString = @"hhaabccdeef";

char testCh[100];

memcpy(testCh, [testString cStringUsingEncoding:NSUTF8StringEncoding], [testString length]);

int list[256];

for (int i = 0; i < 256; i++) {

list[i] = 0;

}

char \*p = testCh;

char result = '\0';

while (\*p != result) {

list[\*(p++)]++;

}

p = testCh;

while (\*p != result) {

if (list[\*p] == 1) {

result = \*p;

break;

}

p++;

}

printf("result:%c",result);

}

###### 查找两个子视图的共同父视图

思路:分别记录两个子视图的所有父视图并保存到数组中，然后倒序寻找,直至找到第一个不一样的父视图。

- (void)findCommonSuperViews:(UIView \*)view1 view2:(UIView \*)view2

{

NSArray \* superViews1 = [self findSuperViews:view1];

NSArray \* superViews2 = [self findSuperViews:view2];

NSMutableArray \* resultArray = [NSMutableArray array];

int i = 0;

while (i < MIN(superViews1.count, superViews2.count)) {

UIView \*super1 = superViews1[superViews1.count - i - 1];

UIView \*super2 = superViews2[superViews2.count - i - 1];

if (super1 == super2) {

[resultArray addObject:super1];

i++;

}else{

break;

}

}

NSLog(@"resultArray:%@",resultArray);

}

- (NSArray <UIView \*>\*)findSuperViews:(UIView \*)view

{

UIView \* temp = view.superview;

NSMutableArray \* result = [NSMutableArray array];

while (temp) {

[result addObject:temp];

temp = temp.superview;

}

return result;

}

###### 求无序数组中的中位数

中位数：当数组个数n为奇数时，为(n + 1)/2，即是最中间那个数字；当n为偶数时，为(n/2 + (n/2 + 1))/2,即是中间两个数字的平均数。  
首先要先去了解一些几种排序算法:iOS排序算法  
思路:

* 1.排序算法+中位数  
  首先用冒泡排序、快速排序、堆排序、希尔排序等排序算法将所给数组排序，然后取出其中位数即可。
* 2.利用快排思想

#### 十二、网络相关

**HTTP协议：超文本传输协议**

是一种详细规定了浏览器和万维网(WWW = World Wide Web)服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。  
HTTP是基于TCP的应用层协议  
(OSI网络七层协议从上到下分别是 应用层、表示层、会话层 、传输层、网络层 、数据链路层、物理层）

* 请求/响应报文
* 连接建立流程
* HTTP的特点

#### 一、请求报文和响应报文

**1、请求报文**

如下:

POST /somedir/page.html HTTP/1.1

//以上是请求行:方法字段、URL字段和HTTP版本字段

Host: www.user.com

Content-Type: application/x-www-form-urlencoded

Connection: Keep-Alive

User-agent: Mozilla/5.0.

Accept-lauguage: fr

//以上是首部行

（此处必须有一空行） //空行分割header和请求内容

name=world 请求体

Host：指明了该对象所在的主机  
Connection：Keep-Alive首部行用来表明该浏览器告诉服务器使用持续连接  
Content-Type: x-www-form-urlencoded首部行用来表明 HTTP会将请求参数用key1=val1&key2=val2的方式进行组织，并放到请求实体里面  
User-agent：首部行用来指明用户代理，即向服务器发送请求的浏览器类型  
Accept-lauguage：首部行表示用户想得到该对象的法语版本（如果服务器中有这样的对象的话），否则，服务器应发送它的默认版本

**2、响应报文**

如下：

HTTP/1.1 200 OK

//以上是状态行：协议版本字段、状态码、相应状态信息

Connection：close

Server:Apache/2.2.3(CentOS)

Date: Sat, 31 Dec 2005 23:59:59 GMT

Content-Type: text/html

Content-Length: 122

//以上是首部行

（此处必须有一空行） //空行分割header和实体主体

(data data data data)//响应实体主体

**状态码及其相应的短语指示了请求的结果。**

一些常见的状态码和对应的短语：

* **200 OK：**请求成功，信息在返回的响应报文中
* **301 Moved Permanently**：请求的对象已经被永久转移了，新的URL定义在响应报文中的Location：首部行中。客户软件将自动获取新的URL
* **400 Bad Request**：一个通用差错代码，指示该请求不能被服务器理解
* **404 Not Found**：被请求的文件不在服务器上
* **505 HTTP Version Not Supported**：服务器不支持请求报文使用的HTTP协议版本  
  <4开头的状态码通常是客户端的问题，5开头的则通常是服务端的问题>

Connection：close首部行告诉客户，发送完报文后将关闭TCP连接。  
Date:指的不是对象创建或最后修改的时间，而是服务器从文件系统中检索到该对象，插入到响应报文，并发送该响应报文的时间。  
Server: 首部行指示该报文是由一台Apache Web服务器产生的，类似于HTTP请求报文里的User-agent  
Content-Length：首部行指示了被发送对象中的字节数  
Content-Type：首部行指示了实体体中的对象是HTML文本

#### 二、HTTP的请求方式

**GET、POST、PUT、DELETE、HEAD、OPTIONS**

**1、GET和POST方式的区别**

**从语法角度来看，最直观的区别就是**

* GET的请求参数一般以?分割拼接到URL后面，POST请求参数在Body里面
* GET参数长度限制为2048个字符，POST一般是没限制的
* GET请求由于参数裸露在URL中， 是不安全的，POST请求则是相对安全  
  之所以说是相对安全，是因为，如果POST虽然参数非明文，但如果被抓包，GET和POST一样都是不安全的。(HTTPS该用还是得用)

**而从语义的角度来看：**

GET：获取资源是 安全的，幂等的(只读的，纯粹的)， 可缓存的  
POST：获取资源是 非安全的，非幂等的，不可缓存的

* 这里的安全是指不应引起Server端的任何状态变化  
  GET的语义就是获取数据，是不会引起服务器的状态变化的，即是安全的。（HEAD，OPTIONS也是安全的）  
  而POST语义则是提交数据，是可能会引起服务器状态变化的，即是不安全的
* 幂等:同一个请求方法执行多次和执行一次的效果完全相同  
  显然GET请求是幂等而POST请求是非幂等的。  
  这里用幂等形容GET还不够，因为GET不止是执行多次和执行一次的效果完全相同，而且是执行一次和执行零次的效果也是完全相同的。
* 可缓存的  
  请求是否可以被缓存。  
  GET请求会主动进行Cache

以上特性，并非并列，正是因为GET是幂等的只读的，即GET请求除了返回数据不会有其他副作用，所以GET才是安全的，从而可以直接由CDN缓存，大大减轻服务器的负担，也就是可缓存的。  
而POST是非幂等的，即除了返回数据还会有其他副作用，所以POST是不安全的，必须交由web服务器处理，即是 不可缓存的

**GET和POST本质上就是TCP链接，并无差别。但是由于HTTP的规定和浏览器/服务器的限制，导致他们在应用过程中体现出一些不同。**

在响应时，GET产生一个TCP数据包；POST产生两个TCP数据包:  
对于GET方式的请求，浏览器会把Header和实体主体一并发送出去，服务器响应200（返回数据）；  
而对于POST，浏览器先发送Header，服务器响应100 Continue，浏览器再发送实体主体，服务器响应200 OK（返回数据）。

**2、GET 相对 POST 的优势是什么？**

1、最大的优势就是方便。GET 的URL可以直接手输，从而GET请求中的URL可以被存在书签里，或者历史记录里  
2、可以被缓存，大大减轻服务器的负担  
所以大多数情况下，还是用GET比较好。

#### 三、HTTP的特点

无连接、 无状态  
HTTP的持久连接、Cookie/Session

**1、HTTP的无状态**

即协议对于事务处理没有记忆能力。  
每次的请求都是独立的，它的执行情况和结果与前面的请求和之后的请求时无直接关系的，它不会受前面的请求应答情况直接影响，也不会直接影响后面的请求应答情况  
也就是说**服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器**  
标准的HTTP协议指的是不包括cookies，session，application的HTTP协议

**2、HTTP的持久连接**

* 非持久连接：每个连接处理一个请求-响应事务。
* 持久连接：每个连接可以处理多个请求-响应事务。

持久连接情况下，服务器发出响应后让TCP连接继续打开着。同一对客户/服务器之间的后续请求和响应可以通过这个连接发送。  
HTTP/1.0使用非持久连接。HTTP/1.1 默认使用持久连接<keep-alive>。

非持久连接的每个连接，TCP得在客户端和服务端分配TCP缓冲区，并维持TCP变量，会严重增加服务器负担。而且每个对象都有2个RTT(Round Trip Time，也就是一个数据包从发出去到回来的时间)的延迟，由于TCP的拥塞控制方案,每个对象都遭受TCP缓启动，因为每个TCP连接都起始于缓启动阶段

**HTTP持久连接怎么判断一个请求是否结束的？**

* Content-length：根据所接收字节数是否达到Content-length值
* chunked(分块传输):Transfer-Encoding。当选择分块传输时，响应头中可以不包含Content-Length，服务器会先回复一个不带数据的报文（只有响应行和响应头和\r\n），然后开始传输若干个数据块。当传输完若干个数据块后，需要再传输一个空的数据块，当客户端收到空的数据块时，则客户端知道数据接收完毕。

#### 一、HTTPS和HTTP的区别

HTTPS协议 = HTTP协议 + SSL/TLS协议  
SSL的全称是Secure Sockets Layer，即安全套接层协议，是为网络通信提供安全及数据完整性的一种安全协议。TLS的全称是Transport Layer Security，即安全传输层协议。  
即HTTPS是安全的HTTP。

#### 二、HTTPS的连接建立流程

HTTPS为了兼顾安全与效率，同时使用了对称加密和非对称加密。在传输的过程中会涉及到三个密钥：

* 服务器端的公钥和私钥，用来进行非对称加密
* 客户端生成的随机密钥，用来进行对称加密

如上图，HTTPS连接过程大致可分为八步:

**1、客户端访问HTTPS连接。**

客户端会把安全协议版本号、客户端支持的加密算法列表、随机数C发给服务端。

**2、服务端发送证书给客户端**

服务端接收密钥算法配件后，会和自己支持的加密算法列表进行比对，如果不符合，则断开连接。否则，服务端会在该算法列表中，选择一种对称算法（如AES）、一种公钥算法（如具有特定秘钥长度的RSA）和一种MAC算法发给客户端。  
服务器端有一个密钥对，即公钥和私钥，是用来进行非对称加密使用的，服务器端保存着私钥，不能将其泄露，公钥可以发送给任何人。  
在发送加密算法的同时还会把数字证书和随机数S发送给客户端

**3、客户端验证server证书**

会对server公钥进行检查，验证其合法性，如果发现发现公钥有问题，那么HTTPS传输就无法继续。

**4、客户端组装会话秘钥**

如果公钥合格，那么客户端会用服务器公钥来生成一个前主秘钥(Pre-Master Secret，PMS)，并通过该前主秘钥和随机数C、S来组装成会话秘钥

**5、客户端将前主秘钥加密发送给服务端**

是通过服务端的公钥来对前主秘钥进行非对称加密，发送给服务端

**6、服务端通过私钥解密得到前主秘钥**

服务端接收到加密信息后，用私钥解密得到主秘钥。

**7、服务端组装会话秘钥**

服务端通过前主秘钥和随机数C、S来组装会话秘钥。  
至此，服务端和客户端都已经知道了用于此次会话的主秘钥。

**8、数据传输**

客户端收到服务器发送来的密文，用客户端密钥对其进行对称解密，得到服务器发送的数据。  
同理，服务端收到客户端发送来的密文，用服务端密钥对其进行对称解密，得到客户端发送的数据。

**总结：**

会话秘钥 = random S + random C + 前主秘钥

* HTTPS连接建立过程使用非对称加密，而非对称加密是很耗时的一种加密方式
* 后续通信过程使用对称加密，减少耗时所带来的性能损耗
* 其中，对称加密加密的是实际的数据，非对称加密加密的是对称加密所需要的客户端的密钥。

#### 三、对称加密和非对称加密

**1、对称加密**

用同一套密钥来进行加密解密。  
对称加密通常有 DES,IDEA,3DES 加密算法。

**2、非对称加密**

用公钥和私钥来加解密的算法。  
公钥（Public Key）与私钥（Private Key）是通过一种算法得到的一个密钥对（即一个公钥和一个私钥），公钥是密钥对中公开的部分，私钥则是非公开的部分,私钥通常是保存在本地。

* 用公钥进行加密，就要用私钥进行解密；反之，用私钥加密，就要用公钥进行解密（数字签名）。
* 由于私钥是保存在本地的，所以非对称加密相对与对称加密是安全的。  
  但非对称加密比对称加密耗时(100倍以上),所以通常要结合对称加密来使用。

常见的非对称加密算法有：RSA、ECC（移动设备用）、Diffie-Hellman、El Gamal、DSA（数字签名用）

而为了确保客户端能够确认公钥就是想要访问的网站的公钥，引入了数字证书的概念，由于证书存在一级一级的签发过程，所以就出现了证书链，在证书链中的顶端的就是根CA。

#### 一、分别用C语言、python、GCDAsyncUdpSocket来实现UDP通信

**1、C语言方式**

* 首先初始化socket对象，Udp要用SOCK\_DGRAM
* 然后初始化sockaddr\_in网络通信对象，如果作为服务端要绑定socket对象与通信链接，来接收消息
* 然后开启一个循环，循环调用recvfrom来接收消息
* 收到消息后，保存下发消息对象的地址，以便之后回复消息

- (void)initCSocket

{

char receiveBuffer[1024];

\_\_uint32\_t nSize = sizeof(struct sockaddr);

if ((\_listenfd = socket(AF\_INET, SOCK\_DGRAM, 0)) == -1)

{

perror("socket() error. Failed to initiate a socket");

}

bzero(&\_addr, sizeof(\_addr));

\_addr.sin\_family = AF\_INET;

\_addr.sin\_port = htons(\_destPort);

if(bind(\_listenfd, (struct sockaddr \*)&\_addr, sizeof(\_addr)) == -1)

{

perror("Bind() error.");

}

\_addr.sin\_addr.s\_addr = inet\_addr([\_destHost UTF8String]);//ip可是是本服务器的ip，也可以用宏INADDR\_ANY代替，代表0.0.0.0，表明所有地址

while(true){

long strLen = recvfrom(\_listenfd, receiveBuffer, sizeof(receiveBuffer), 0, (struct sockaddr \*)&\_addr, &nSize);

NSString \* message = [[NSString alloc] initWithBytes:receiveBuffer length:strLen encoding:NSUTF8StringEncoding];

\_destPort = ntohs(\_addr.sin\_port);

\_destHost = [[NSString alloc] initWithUTF8String:inet\_ntoa(\_addr.sin\_addr)];

NSLog(@"来自%@---%zd:%@",\_destHost,\_destPort,message);

}

}

* **由于开启while循环来一直接收消息，所以为了避免阻塞主线程，这里要将initCSocket函数放在子线程中调用**

dispatch\_async(dispatch\_get\_global\_queue(0, 0), ^{

[self initCSocket];

});

* 调用sendto方法来发送消息

- (void)sendMessage:(NSString \*)message

{

NSData \*sendData = [message dataUsingEncoding:NSUTF8StringEncoding];

sendto(\_listenfd, [sendData bytes], [sendData length], 0, (struct sockaddr \*)&\_addr, sizeof(struct sockaddr));

}

**2、GCDAsyncUdpSocket方式**

* 首先初始化Socket对象
* 绑定端口，调用beginReceiving：方法来接收消息

- (void)initGCDSocket

{

\_receiveSocket = [[GCDAsyncUdpSocket alloc] initWithDelegate:self

delegateQueue:dispatch\_get\_global\_queue(0, 0)];

NSError \*error;

// 绑定一个端口(可选),如果不绑定端口, 那么就会随机产生一个随机的唯一的端口

// 端口数字范围(1024,2^16-1)

[\_receiveSocket bindToPort:test\_port error:&error];

if (error) {

NSLog(@"服务器绑定失败");

}

// 开始接收对方发来的消息

[\_receiveSocket beginReceiving:nil];

}

* 在代理方法里获取到对方发过来的消息，记录下主机和端口，以便之后回复消息

#pragma mark - GCDAsyncUdpSocketDelegate

- (void)udpSocket:(GCDAsyncUdpSocket \*)sock didReceiveData:(NSData \*)data fromAddress:(NSData \*)address withFilterContext:(id)filterContext {

NSString \*message = [[NSString alloc] initWithData:data encoding:NSUTF8StringEncoding];

\_destPort = [GCDAsyncUdpSocket portFromAddress:address];

\_destHost = [GCDAsyncUdpSocket hostFromAddress:address];

NSLog(@"来自%@---%zd:%@",\_destHost,\_destPort,message);

}

* 调用sendData:(NSData \*)data toHost:(NSString \*)host port:(uint16\_t)port withTimeout:(NSTimeInterval)timeout tag:(long)tag方法来发送消息

- (void)sendMessage:(NSString \*)message

{

NSData \*sendData = [message dataUsingEncoding:NSUTF8StringEncoding];

[\_receiveSocket sendData:sendData toHost:\_destHost port:\_destPort withTimeout:60 tag:500];

}

**3、python方式**

python方式就比较简单了

* 初始化socket，绑定端口

socket = socket(AF\_INET, SOCK\_DGRAM)

socket.bind(('', port))

* 循环接收消息

while True:

message, address = socket.recvfrom(2048)

print address,message

* 发送消息

socket.sendto(message, address)

#### 二、利用python实现Udp通信demo

* 创建两个python文件，分别作为客户端和服务端，然后同时运行
* 客户端

from socket import \*

host = '127.0.0.1'

port = 12000

socket = socket(AF\_INET, SOCK\_DGRAM)

while True:

message = raw\_input('input message ,print 0 to close :\n')

socket.sendto(message, (host, port))

if message == '0':

socket.close()

break

receiveMessage, serverAddress = socket.recvfrom(2048)

print receiveMessage,serverAddress

* 服务端

from socket import \*

port = 12000

socket = socket(AF\_INET, SOCK\_DGRAM)

socket.bind(('', port))

print 'server is ready to receive'

count = 0

while True:

message, address = socket.recvfrom(2048)

print address,message

count = count + 1

if message == '0':

socket.close()

break

else:

message = raw\_input('input message ,print 0 to close :\n')

socket.sendto(message, address)

* 客户端打印

/usr/local/bin/python2.7 /Users/wangyong/Desktop/other/python/UDPClient.py

input message ,print 0 to close :

hello，服务端

hello，客户端 ('10.208.61.53', 12000)

input message ,print 0 to close :

结束通信吧我们

好的 ('10.208.61.53', 12000)

input message ,print 0 to close :

0

Process finished with exit code 0

* 服务端打印

/usr/local/bin/python2.7 /Users/wangyong/Desktop/other/python/UDPServer.py

server is ready to receive

('10.208.61.53', 53500) hello，服务端

input message ,print 0 to close :

hello，客户端

('10.208.61.53', 53500) 结束通信吧我们

input message ,print 0 to close :

好的

('10.208.61.53', 53500) 0

Process finished with exit code 0

#### 三、iOS端基于UDP的简易聊天demo

**1、UdpManager**

Udp通信用C语言版和GCDAsyncUdpSocket都可以，封装在UdpManager中

* initSocketWithReceiveHandle:(dispatch\_block\_t)receiveHandle：初始化socket相关，receiveHandle是接收到消息后的回调
* sendMessage:(NSString \*)message：发送消息
* messageArray:消息列表，包括接收到的和发送出去的消息

+ (void)initSocketWithReceiveHandle:(dispatch\_block\_t)receiveHandle;

+ (void)sendMessage:(NSString \*)message;

+ (NSMutableArray \*)messageArray;

消息内容用MessageModel,其中role代表消息发送对象，为0即是接收到的消息，1为自己发送的消息

@interface MessageModel:NSObject

@property (nonatomic, copy) NSString \*message;

@property(nonatomic,assign) NSInteger role;

@end

**2、ViewController**

控制器里调用UdpManager初始化socket

[UdpManager initSocketWithReceiveHandle:^{

dispatch\_async(dispatch\_get\_main\_queue(), ^{

self.title = [NSString stringWithFormat:@"%@---%@",[[UdpManager shareManager] valueForKey:@"\_destHost"],[[UdpManager shareManager] valueForKey:@"\_destPort"]];

[self reloadData];

});

}];

在代理方法textFieldShouldReturn即点击键盘的发送按钮时发送编辑好的消息

#pragma mark - UITextFieldDelegate

- (BOOL)textFieldShouldReturn:(UITextField \*)textField

{

if (self.textField.text.length == 0) return YES;

[UdpManager sendMessage:self.textField.text];

[self reloadData];

self.textField.text = nil;

return YES;

}

发送或者接收到新消息后都会将消息添加到messageArray里，并刷新页面

- (void)sendMessage:(NSString \*)message

{

NSData \*sendData = [message dataUsingEncoding:NSUTF8StringEncoding];

[self.messageArray addObject:[[MessageModel alloc] initWithMessage:message role:1]];

#ifdef UseGCDUdpSocket

// 该函数只是启动一次发送 它本身不进行数据的发送, 而是让后台的线程慢慢的发送 也就是说这个函数调用完成后,数据并没有立刻发送,异步发送

[\_receiveSocket sendData:sendData toHost:\_destHost port:\_destPort withTimeout:60 tag:500];

#else

sendto(\_listenfd, [sendData bytes], [sendData length], 0, (struct sockaddr \*)&\_addr, sizeof(struct sockaddr));

#endif

}

UI就不多做介绍了，控制器里只有一个显示接收和发送消息内容列表的tableView及一个编辑消息的输入框textField。大概就这些内容，只是个简易的demo，只实现了接收发送文字消息的功能，并没有做更多优化

**3、测试**

分别用模拟器和真机运行，或者可以配合刚才的python程序测试.  
test\_host就直接用电脑ip即可

* 然后手机先发送消息到模拟器上，模拟器就可以根据记录下的手机的主机和端口回复消息了。这里手机连外网也是可以的

**TCP（Transmission Control Protocol 传输控制协议）和UDP（User Datagram Protocol 用户数据报协议）同属传输层协议**

#### 一、UDP的特点

UDP是是面向非连接的协议，传送数据不需要和服务器连接，只需要知道ip和监听端口，不需要链接没有目的的socket，只是将数据报投递出去，不管接收方是否成功接收到，是一种不可靠的传输。

既然UDP是不可靠数据传输协议，那为什么那么多应用去选择UDP呢？

**1、关于何时、发送什么数据的应用层控制更加精细**

* 只要应用将数据传递给UDP，UDP就会将此数据打包进UDP报文段并立刻将其传递给网络层。
* 而TCP则是有个拥塞控制机制，以确保数据能够安全传输，而不管可靠传输成功需要用多少时间。
* 所以有些实时应用，比如微信视频、语音都是更希望数据能够及时发送，为此可以容忍一部分数据丢失，比较适合用UDP

**2、无需连接建立**

* 众所周知，TCP在数据传输前需要经过三次握手，UDP却不需要做任何的准备即可进行数据传输，因此UDP不会引入建立连接的时延。
* 这也是DNS运行在UDP而不是TCP上的主要原因。
* 而HTTP协议之所以使用TCP，是因为对于HTTP协议来说，可靠性是至关重要的。

**3、无连接状态**

* TCP需要维护连接状态。此连接状态包括接收和发送缓存、拥塞控制参数以及序号与确认号的参数。（后面如果有时间，会详细说下TCP的拥塞控制方案，对该方案来说，这些状态信息都是必要的）
* 而UDP不需要维护连接状态，也不用跟踪这些参数

**4、分组首部开销小**

每个TCP报文段都有20字节的首部开销，而UDP仅有8字节的开销

所以，如非必要，比如电子邮件，远程终端服务，web，以及文件传输，需要可靠地数据传输，会去采用TCP。其余的尤其是对实时性要求高的应用，比如实时视频会议，网络电话，一般都会选用UDP

#### 二、UDP的报文结构

应用层数据占用UDP报文段的数据字段。UDP首部只有4个字段，每个字段由2个字节组成，即UDP首部仅有8字节。

* 端口号：可以使目的主机将应用数据交给运行在目的端系统中端相应进程，执行分用功能。
* 长度：该字段指示了在UDP报文段中的字节数（首部+数据）
* 检验和：接收方使用检验和来检查在该报文段中是否出现了差错，即差错检测。

###### 三、UDP差错检测

UDP检验和提供了差错检测功能。  
检验和相当于用于确定当UDP报文段从源到达目的地移动时，其中的比特是否发生了改变（比如，由于链路中的噪声干扰或存储在路由器中时的引入问题）。  
发送方的UDP对报文段中的所有16比特字对和进行反码运算，求和时遇到的任何溢出都被回卷。得到的结果被放在UDP报文段中的检验和字段。

比如，假定有下面三个16比特的字:

0110011001100000

0101010101010101

1000111100001100

这些16比特字的前两个之和是：

1011101110110101

再将该和与第三个16比特字相加，得出:

10100101011000001

发现溢出了，该和就要被回卷，即把首位的1加到最后一位去，得出：

0100101011000010

然后对其进行反码运算，所谓反码运算，即是将所有的1换成0，0换成1

1011010100111101

这就是得出的检验和。而在接收方，全部的4个16比特字（包括检验和）加在一起。如果分组中没有引入差错，显然在接收处该和将是1111111111111111。而如果这些比特之一是0，那我们就知道该分组中出现了差错。  
UDP在端到端基础上在运输层提供差错检测，这就是在系统设计中被称颂的端到端原则

而UDP虽然提供差错检测，但它对差错恢复无能为力。这就需要用到可靠数据传输--TCP了

#### 一、TCP的特点和报文结构

**1、面向连接、可靠传输、面向字节流、全双工服务**

**2、TCP的报文结构**

TCP报文段由**首部字段**和一个**数据字段**组成。  
数据字段包含一块应用数据。最大报文长度MSS（Maximum Segment Size）限制了报文段数据字段的最大长度。MSS选项用于在TCP连接建立时，收发双方协商通信时每一个报文段所能承载的最大数据长度。  
所以当TCP发送一个大文件（比如一张高清图片）时，通常是将该文件划分为MSS长度的若干块（最后一块除外，通常会小于MSS）。而实际交互式应用通常传送长度小于MSS的数据块。

如图，与UDP一样，首部包括**源端口号**和**目的端口号**，用于多路复用/分解来自上层或送到上层应用的数据。TCP首部也同样包括**检验和字段**

TCP首部还包含下列字段:

* 32比特的**序号字段Seq(sequence number field)** 和32比特的**确认号字段Ack(acknowledge number field)**
* 16比特的**接收窗口字段RW(receive window field)**,该字段用于流量控制，用于指示接收方愿意接收的字节数量。
* 4比特的**首部长度字段（header length field）**，该字段指示了以32比特的字为单位的TCP首部长度。由于TCP选项字段的原因，TCP首部长度是可变的。（通常，选项字段为空，所以TCP首部的典型长度就是20字节）
* 可选和变长的**选项字段（option field）**，该字段用于发送方和接收方协商最大报文段长度（MSS）时，或用作窗口调节因子时使用。
* 6比特的**标志字段（flag field）**。ACK比特用于指示确认字段中的值是有效的，即该报文段包括一个对已被接收报文段的确认。**RST、SYN、FIN**比特用于连接建立和拆除。  
  **PSH**比特指示接收方应立即将数据交给上层。**URG**比特用于指示报文段里存在着被发送端的上层实体置为“紧急”的数据。紧急数据的最后一个字节由16比特的紧急数据指针字段指出。当紧急数据存在并给出指向紧急数据尾的指针的时候，TCP必须通知接收端的上层实体。在实践中，PSH、URG和紧急数据指针并没有使用。

**3、序号字段Seq和确认号字段Ack**

* 在TCP通讯中，无论是建立连接，数据传输，友好断开，强制断开，都离不开Seq值和Ack值，它们是TCP传输的可靠保证。

**序号Seq**：  
TCP把数据看成一个无结构的、有序的字节流。一个**报文段的序号**因此是该报文段的首字节的字节流编号。  
比如数据流由一个包含100000字节的文件组成，其MSS是1000字节，数据流的首字节编号是0。该TCP将为该数据流构建100个报文段。给第一个报文段分配序号0，第二个则是1000，第三个是2000，以此类推。每一个序号被填入到相应TCP报文段首部的序号字段中。

**确认号Ack：**  
TCP是全双工服务的，因此主机A在向主机B发送数据的同时，也许也在接收主机B的数据。  
主机A填充进报文段的确认号是主机A期望从主机B收到的下一个字节的序号。

在上个例子中，假如服务端已经接收包含字节0-999的报文段和包含字节2000-2999的报文段，但由于某种原因，还未收到包含字节1000-1999的报文段，那么将仍会等待字节1000（及其后的字节）。因此服务端发给客户端的下一个报文段将在**确认号Ack**字段中包含1000。  
因为TCP只确认该流中至第一个丢失字节为止的字节，所以TCP被称为**累积确认**。

#### 二、三次握手

-数据开始传输前，需要通过 三次握手来建立连接  
事实上我认为，这里称呼三步握手（three-way handshake）才更贴切些

**第一步：**

* 客户端的TCP首先向服务端的TCP发送一条特殊的TCP报文段。该报文段不包含应用层数据，该报文段首部中的一个标志位（SYN比特）被置为1，所以该报文段被称为**SYN报文段**。另外，客户会随机选择一个初始序号client\_isn，并将该序号放置于该起始的TCP SYN报文段的序号字段中。
* 客户端和服务端最开始都处于CLOSED状态，发送过该 SYN报文段后，客户端TCP进入SYN\_SENT状态，等待服务端确认并将SYN比特置为1的报文段。

**第二步：**

* 收到SYN报文段后，服务端会为该TCP连接分配TCP缓存和变量，服务端TCP会进入SYN\_RCVD状态，等待客户端TCP发送确认报文段。
* 并向该客户端TCP发送允许连接的报文段，该报文段同样不包含应用层数据。该报文段首部的SYN比特被置为1，确认号字段被置为client\_isn+1。服务端还会选择自己的初始序号server\_isn，放到报文段首部的序号段中。该连接被称为**SYNACK报文段**。

**第三步：**

* 收到SYNACK报文段后，客户端也要为该TCP连接分配缓存和变量，客户端TCP进入ESTABLISHED状态，在此状态，客户端就能发送和接收包含有效载荷数据的报文段了。
* 并向服务端TCP发送一个报文段：这最后一个报文段对服务端的允许连接的报文表示了确认（将server\_isn + 1放到报文段首部的确认字段中）。因为连接已经建立了，所以该SYN比特被置为0。这个阶段，可以在报文段负载中携带应用层数据。
* 收到客户端该报文段后，服务端TCP也会进入ESTABLISHED状态，可以发送和接收包含有效载荷数据的报文段。

1. List item

#### 三、四次挥手

**参与TCP连接的两个进程中的任何一个都能终止该连接，当连接结束后，主机中的资源（缓存和变量）会被释放。**

上边说到，SYN和FIN标志位分别对应着TCP连接的建立和拆除。

**第一步：**

* 客户应用进程发出一个关闭连接的指令。会引起客户端TCP向服务端发送一个特殊的TCP报文段。该报文段即是将首部的一个标志位FIN比特置为1。
* 同时，客户端进入FIN\_WAIT\_1状态，等待服务端的带有确认的TCP报文段。

**第二步：**

* 收到该报文段后会向客户端发送一个确认报文段。
* 服务端TCP进入CLOSE\_WAIT状态，对应客户端的TIME\_WAIT，表示被动关闭。
* 客户端收到该报文段后，进入FIN\_WAIT\_2状态，等待服务端的FIN比特置为1的报文段。

**第三步：**

* 服务端发送自己的终止报文段，同样是把报文段首部的标志位FIN比特置为1。
* 服务端TCP进入LAST\_ACK状态，等待服务端最后的确认报文段。

**第四步：**

* 客户端收到服务端的终止报文段后，向服务端发送一个确认报文段。同时，客户端进入TIME\_WAIT状态。
* 假如ACK丢失，TIME\_WAIT状态使TCP客户重传最后的确认报文，TIME\_WAIT通常会等待2MSL（Maximum Segment Lifetime 最长报文段寿命）。经过等待后，连接就正式关闭，重新进入CLOSED状态，客户端所有资源将被释放。
* 服务端收到该报文段后，同样也会关闭，重新进入CLOSED状态，释放所有服务端TCP资源。

###### 一些问题

**1、问：为什么建立连接只用三次握手，而断开连接却要四次挥手？**

* 首先，当客户端数据已发送完毕，且知道服务端也全部接收到了时，就会去断开连接即向服务端发送FIN
* 服务端接收到客户端的FIN，为了表示接收到了，就会向客户端发送ACK
* 但此时，服务端可能还在发送数据，并没有关闭TCP窗口的意思，所以服务端的FIN和ACK并不是同步发的，只有当数据发送完了，才会发送FIN
* **答：服务端的FIN和ACK需要分开发，并不是像三次握手中那样，SYN可以和ACK同步发，所以就需要四次挥手**

**2、在四次挥手中，客户端为什么在TIME\_WAIT后必须等待2MSL时间呢？**

这个ACK报文段有可能丢失，因而使处在LAST\_ACK端的服务端收不到对已发送的FIN报文段的ACK报文段，从而服务端会去不断重传FIN报文段。  
而客户端就能在2MSL时间内收到重传的FIN报文段。接着客户端重传一次确认，重新启动2MSL计时器。直至服务端收到后，客户端和服务端就都会进入CLOSED状态，关闭TCP连接。  
而如果客户端不等待2MSL时间，而是在发送完ACK确认后立即释放资源，关闭连接，那么就无法收到服务端重传的FIN报文段，因而也不会再发送一次ACK确认报文段，这样，服务端就无法正常进入CLOSED状态，资源就一直无法释放了。

* 答：为了保证客户端发送的最后一个ACK报文段能够到达服务端。

**3、TCP在创建连接时，为什么需要三次握手而不是两次或四次？**

一个简单的例子：

* 三次握手：  
  “喂，你听得到吗？”  
  “我听得到呀，你听得到我吗？”  
  “我能听到你，今天balabala……”
* 两次握手：  
  “喂，你听得到吗？”  
  “我听得到呀，你听得到我吗？”  
  “喂，你听得到吗？”  
  “……谁在说话？”  
  “喂，你听得到吗？”  
  “……”
* 四次握手：  
  “喂，你听得到吗？”  
  “我听得到呀”“你能听到我吗？”  
  “……不想跟傻逼说话”

之所以不用四次握手的原因很容易理解，就是浪费资源，服务端的SYN和ACK可以一起发，完全没必要分开两次。

而如果是两次握手：  
客户端发出的第一个连接请求SYN报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达服务端。本来这是一个早已失效的报文段。但服务端收到此失效的连接请求SYN报文段后，就误认为是客户端再次发出的一个新的连接请求SYN报文段。于是就向客户端发出ACK确认报文段，同意建立连接。假设不采用三次握手，那么只要服务端发出确认，新的连接就建立了。  
由于现在客户端并没有发出建立连接的SYN请求，因此不会理睬服务端的确认，也不会向服务端发送数据。但服务端却以为新的运输连接已经建立，并一直等待客户端发来数据。这样，服务端的很多资源就白白浪费掉了。

事实上：TCP对有数据的TCP报文段必须确认的原则，所以，客户端对服务端的SYN报文段必须回复一个ACK报文段表示确认。并且，TCP不会为没有数据的ACK超时重传，那么当服务端没收到客户端的ACK确认报文段时，会超时重传自己的SYN报文段，一直到收到客户端的ACK为止。

* **答：两次握手会可能导致已失效的连接请求报文段突然又传送到了服务端产生错误，四次握手又太浪费资源**

###### 代码实现

参考UDP的代码，其实TCP在代码实现上也很相似，首先·socket·初始化时不再用·SOCK\_DGRAM·，而是用·SOCK\_STREAM·

fd = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

TCP服务端要多了一道监听、接受连接的过程：

int listen\_ret = listen(fd,5);

int listen\_socket = accept(\_fd,(sockaddr \*)&addr,&addr\_len);

UDP则是多了一道连接的过程：

int ret = connect(\_fd, (struct sockaddr \*) &addr, sizeof(addr));

然后就是在接收和发送数据时，不用再传主机和端口了。即由recvfrom、sendto改为recv和send。

send(\_fd, [buffer bytes], [buffer length], 0);

recv(\_fd, receiveBuffer, sizeof(receiveBuffer), 0);

python的客户端代码如下：

from socket import \*

serverName = '127.0.0.1'

serverPort = 12000

clientSocket = socket(AF\_INET,SOCK\_STREAM)

clientSocket.connect((serverName,serverPort))

sentence = raw\_input('Input lowercase:\n')

clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1029)

print 'From server:\n',modifiedSentence

clientSocket.close()

服务端代码：

from socket import \*

serverPort = 12000

serverSocket = socket(AF\_INET,SOCK\_STREAM)

serverSocket.bind(('',serverPort))

serverSocket.listen(1)

print 'server is ready to receive'

connectionSocket,addr = serverSocket.accept()

sentence = connectionSocket.recv(1029)

capitalizeSentence = sentence.upper()

print capitalizeSentence

connectionSocket.send(capitalizeSentence)

connectionSocket.close()

#### 一、可靠数据传输

**网络层服务（IP服务）是不可靠的。IP不保证数据报的交付，不保证数据报的按序交付，也不保证数据报中数据的完整性。**

TCP则是在IP服务上创建了一种**可靠数据传输服务**  
TCP的**可靠数据传输服务**确保一个进程从其接收缓存中读出的数据流是无损坏、无间隔、无冗余、按序的数据流。即该字节流与连接的另一端发出的字节流是完全相同的。  
作为TCP接收方，有三个与发送和重传有关的主要事件

**1、从上层应用数据接收数据**

将数据封装到一个报文段中，并把报文段交付给IP。每个报文段都包含一个序号Seq，即该报文段第一个数据字节的字节流编号。如果定时器还没有为其他报文段而运行，则启动定时器(即不是每条报文段都会启动一个定时器，而是一共只启动一个定时器)，定时器的过期间隔是TimeoutInterval  
是由EstimatedRTT和DevRTT计算得来的:TCP的往返时间的估计与超时

**2、超时**

TCP通过重传引起超时的报文段来响应超时事件，然后重启定时器。

而发送端超时有两种情况：发送数据超时，接收端发送ACK超时。这两种情况都会导致发送端在TimeoutInterval内接收不到ACK确认报文段。

* 1、如果是发送数据超时，直接重传即可。
* 2、而如果是接收端发送ACK超时，这种情况接收端实际上已经接收到发送端的数据了。那么当发送端超时重传时，接收端会丢弃重传的数据，同时再次发送ACK。

而如果在TimeoutInterval后接收到了ACK，会收下ACK，但不做任何处理

* TCP不会为没有数据的ACK超时重传

以下两种情况：

* 1、如果在发送两条或多条数据报文段都超时，那么只会重传序号最小的那个，并重启定时器。只要其余报文段的ACK在新重启的定时器超时前到达，就不会重传。
* 2、如果发送序号为100和120的两条数据报文段，序号100的ACK丢失，但收到了序号120的ACK，由于累积确认机制，可以得出接收方已经接收到了序号100的报文段，这种情况也不会去重传。

**3、接收到ACK**

用TCP状态变量SendBase指最早未被确认的字节的序号。则SendBase - 1 指接收方已正确按序接收到的数据的最后一个字节的序号。  
当收到ACK确认报文段后，会将ACK的值Y与SendBase比较。TCP采用**累计确认**的方法，所以Y确认来字节编号在Y之前的所有字节都已经收到。如果Y比SendBase小，不用理会；而如果Y比SendBase大，则该ACK是在确认一个或多个先前未被确认的报文段，因此要更新SendBase变量，如果当前还有未被确认的报文段，TCP还要重启定时器。  
**通过超时重传，能保证接收到的数据是无损坏、无冗余的数据流，但并不能保证按序。**

而通过TCP滑动窗口，能够有效保证接收数据有序

#### 二、流量控制

TCP连接的双方主机都会为该TCP连接分配缓存和变量。当该TCP连接收到正确、按序的字节后，就将数据放入接收缓存。上层的应用进程会从该缓存中读取数据，但不必是数据一到达就立即读取，因为此时应用程序可能在做其他事务。而如果应用层读取数据相对缓慢，而发送方发送得太多、太快，发送的数据就会很容易地使该连接的接收缓存溢出。

所以，TCP为应用程序提供了**流量控制服务**（flow-control service），以消除发送方使接收方缓存溢出的可能性。

流量控制是一个速度匹配服务，即发送方的发送速率与接收方应用程序的读取速率相匹配。  
作为全双工协议，TCP会话的双方都各自维护一个**发送窗口**和一个**接收窗口**（receive window）的变量来提供流量控制。而发送窗口的大小是由对方接收窗口来决定的，接收窗口用于给发送方一个指示--该接收方还有多少可用的缓存空间。

**1、发送窗口**

发送方的发送缓存内的数据都可以被分为4类:

* 已发送，已收到ACK
* 已发送，未收到ACK
* 未发送，但允许发送
* 未发送，但不允许发送

则2和3属于发送窗口

* 发送窗口只有收到发送窗口内字节的ACK确认，才会移动发送窗口的左边界

**2、接收窗口**

接收方的缓存数据分为3类：  
1.已接收  
2.未接收但准备接收  
3.未接收而且不准备接收  
则2 属于接收窗口（这里的接收指接收数据并确认）

* 接收窗口只有在前面所有的报文段都确认的情况下才会移动左边界。当在前面还有字节未接收但收到后面字节的情况下，会先接收下来，接收窗口不会移动，并不对后续字节发送ACK确认报文，以此确保发送端会对这些数据重传。

我们定义以下变量：

* LastByteRead：接收方应用程序读取的数据流的最后一个字节编号。可以得知，这是接收缓存的起点
* LastByteRcvd：从网络中到达的并且已放入接收缓存中的数据流的最后一个自己的的编号。

可以得知：LastByteRcvd - LastByteRead <= RcvBuffer(接收缓存大小)  
那么接收窗口rwnd =RcvBuffer - （LastByteRcvd - LastByteRead）  
rwnd是随时间动态变化的，如果rwnd为0，则意味着接收缓存已经满了。  
接收端在回复给发送端的ACK中会包含该rwnd，发送端则会根据ACK中的接收窗口的值来控制发送窗口。

有一个问题，如果当发送rwnd为0的ACK后，发送端停止发送数据。等待一段时间后，接收方应用程序读取了一部分数据，接收端可以继续接收数据，于是给发送端发送报文告诉发送端其接收窗口大小，但这个报文不幸丢失了，我们知道，**不含数据的ACK是不会超时重传的**，于是就出现发送端等待接收端的ACK通知||接收端等待发送端发送数据的死锁状态。

为了处理这种问题，TCP引入了持续计时器（Persistence timer），当发送端收到对方的rwnd=0的ACK通知时，就启用该计时器，时间到则发送一个1字节的探测报文，对方会在此时回应自身的接收窗口大小，如果结果仍未0，则重设持续计时器，继续等待。

#### 三、拥塞控制

TCP除了可靠传输服务外，另一个关键部分就是拥塞控制。  
TCP让每一个发送方根据所感知到的网络拥塞程度来限制其能向连接发送流量的速率。  
可能有三个疑问：  
1、TCP发送方如何感知网络拥塞？  
2、TCP发送方如何限制其向连接发送流量的速率？  
3、发送方感知到网络拥塞时，采用何种算法来改变其发送速率？  
这就是TCP的**拥塞控制机制**。  
前边说到，TCP连接的每一端都是由一个接收缓存、一个发送缓存和几个变量（LastByteRead、LastByteRcvd、rwnd等）组成。而运行在发送方的TCP拥塞控制机制会跟踪一个额外的变量，即**拥塞窗口cwnd**（congestion window）。它对一个TCP发送方能向网络中发送流量的速率进行了限制。  
发送方中未被确认的数据量不会超过cwnd和rwnd的最小值:min(rwnd,cwnd)

**1、TCP发送方如何感知网络拥塞？**

**冗余ACK（duplicate ACK）**：就是再次确认某个报文段的ACK，而发送方先前已经收到对该报文段的确认。

**冗余ACK的产生原因：**

* 1.当接收端接收到失序报文段时，即该报文段序号大于下一个期望的、按序的报文段，检测到数据流中的间隔，即由报文段丢失，并不会对该报文段确认。TCP不使用否定确认，所以不能向发送方发送显式的否定确认，为了使接收方得知这一现象，会对上一个按序字节数据进行重复确认，这也就产生了一个冗余ACK。
* 2.因为发送方经常发送大量的报文段，如果其中一个报文段丢失，可能在定时器过期前，就会收到大量的冗余ACK。一旦收到3个冗余ACK（3个以下很可能是链路层的乱序引起的，无需处理），说明在这个已被确认3次的报文段之后的报文段已经丢失，TCP就会执行**快速重传**，即在该报文段的定时器过期之前重传丢失的报文段。

**将TCP发送方的丢包事件定义为：要么出现超时，要么收到来自接收方的3个冗余ACK。**

当出现过度的拥塞时，路由器的缓存会溢出，导致一个数据报被丢弃。丢弃的数据报接着会引起发送方的丢包事件。那么此时，发送方就认为在发送方到接收方的路径上出现了网络拥塞。

**2、TCP发送方如何限制其向连接发送流量的速率？**

* 当出现丢包事件时：应当降低TCP发送方的速率。
* 当对先前未确认报文段的确认到达时，即接收到非冗余ACK时，应当增加发送方的速率。

**3、发送方感知到网络拥塞时，采用何种算法来改变其发送速率？**

即**TCP拥塞控制算法（TCP congestion control algorithm）**  
包括三个主要部分：**慢启动、拥塞避免、快速恢复**，其中快速恢复并非是发送方必须的，慢启动和拥塞避免则是TCP强制要求的

* **1、慢启动**  
  当一条TCP连接开始时，拥塞窗口cwnd的值通常置为一个MSS的较小值，这就使初始发送速率大约为**MSS/RTT**（RTT：往返时延，报文段从发出到对该报文段的确认被接收之间的时间量）。  
  而对TCP发送方来说，可用带宽可能比MSS/RTT大得多,TCP发送方希望迅速找到可用带宽的数量。因此，在慢启动状态，cwnd以一个MSS的值开始并且每当收到一个非冗余ACK就增加一个MSS。

最初cwnd值为1MSS，发送一个报文段M1。收到M1的确认后，cwnd增加为2MSS，这时可以发送两个报文段M2，M3。收到这两个报文段的确认后，cwnd则增加为4MSS，可以发送四个报文段，以此类推...

**因此，TCP虽然发送速率起始慢，但在慢启动阶段以指数增长。**

这种指数增长很显然不是无限制的，那么何时结束呢？  
如果出现丢包事件，TCP发送方将ssthresh（慢启动阈值）设置为cwnd/2

* 发生由超时引起的丢包事件，并将cwnd重置为1MSS，重启慢启动
* 当TCP发送方的cwnd值达到或超过ssthresh，再继续翻倍显然不合适。这时将结束慢启动转移到拥塞避免模式。
* TCP发送方检测到3个冗余ACK，会结束慢启动，并快速重传，即在该报文段的定时器过期之前重传丢失的报文段。且进入快速恢复状态。
* **2、拥塞避免**

一旦进入拥塞避免状态，cwnd的值大约是上次遇到拥塞时的值的一半，即距离拥塞并不遥远。因此，TCP无法每过一个RTT就将cwnd翻倍。而是每个RTT只增加1MSS，即每收到一个非冗余ACK，就将cwnd增加1/cwnd。即假如此时cwnd为10MSS，则每收到一个非冗余ACK，cwnd就增加1/10MSS，在10个报文段都收到确认后，拥塞窗口的值就增加了1MSS。

那么何时结束拥塞避免的线性增长（每RTT 1MSS）呢？  
和慢启动一样，如果出现丢包事件，TCP发送方将ssthresh（慢启动阈值）设置为cwnd/2（加法增大， 乘法减小）

* 发生由超时引起的丢包事件，拥塞避免和慢启动处理的方式相同。即TCP发送方将ssthresh（慢启动阈值）设置为cwnd/2，并将cwnd重置为1MSS，重启慢启动
* TCP发送方检测到3个冗余ACK，cwnd为原来的一半加上3MSS，进入快速恢复状态。
* **3、快速恢复**

快速恢复是由3个冗余ACK引起的。  
在快速恢复中，对引起TCP进入快速恢复状态的缺失报文段，对收到的每个冗余ACK，cwnd增加1个MSS。最终，当对丢失报文段的一个ACK到达时，TCP在降低cwnd后进入拥塞避免状态。  
如果出现超时，和之前一样，即TCP发送方将ssthresh（慢启动阈值）设置为cwnd/2，并将cwnd重置为1MSS，重启慢启动

快速恢复并非是必须的。

TCP的拥塞控制是：每个RTT内cwnd线性（加性增）增加1MSS，然后出现3个冗余ACK事件时cwnd减半（乘性减），因此TCP拥塞控制常被称为**加性增，乘性减**拥塞控制方式。

#### 一、DNS

因特网上的主机，可以使用多种方式标识，比如主机名或IP地址。

* 一种标识方法就是用它的**主机名（hostname）**，比如·[www.baidu.com](https://links.jianshu.com/go?to=http%3A%2F%2Fwww.baidu.com)、[www.google.com](https://links.jianshu.com/go?to=http%3A%2F%2Fwww.google.com)、[gaia.cs.umass.edu](https://links.jianshu.com/go?to=http%3A%2F%2Fgaia.cs.umass.edu)等。这方式方便人们记忆和接受，但是这种长度不一、没有规律的字符串路由器并不方便处理。
* 还有一种方式，就是直接使用定长的、有着清晰层次结构的IP地址，路由器比较热衷于这种方式。

为了折衷这两种方式，我们需要一种能进行主机名到IP地址转换的目录服务。这就是\***域名系统（Domain Name System，DNS）**的主要任务。

**DNS**是：  
1、一个由分层的**DNS服务器**实现的分布式数据库  
2、一个使得主机能够查询分布式数据库的**应用层协议**

而DNS服务器通常是运行BIND软件的UNIX机器，DNS协议运行在**UDP**上，使用53号端口  
DNS通常是由其他应用层协议所使用的，包括HTTP、SMTP等。其作用则是：**将用户提供的主机名解析为IP地址**

DNS的一种简单设计就是在因特网上只使用一个DNS服务器，该服务器包含所有的映射。很明显这种设计是有很大的问题的：

* **单点故障**：如果该DNS服务器崩溃，全世界的网络随之瘫痪
* **通信容量**：单个DNS服务器必须处理所有DNS查询
* **远距离的集中式数据库**：单个DNS服务器必须面对所有用户，距离过远会有严重的时延。
* **维护**：该数据库过于庞大，还需要对新添加的主机频繁更新。

所以，DNS被设计成了一个**分布式、层次数据库**

#### 二、DNS服务器

为了处理扩展性问题，DNS使用了大量的DNS服务器，它们以层次方式组织，并且分布在全世界范围内。

域名服务器是提供域名解析的服务器，在有基本的知识下，任何人都可以搭建域名服务器，甚至是根域名服务器，有名的软件有：BIND

目前的DNS服务器大致分为3种类型的DNS服务器：根DNS服务器、顶级域DNS服务器、权威DNS服务器

**1、 根DNS服务器**

因特网上有13个根DNS服务器（标号A到M），1个为主根服务器在美国。其余12个均为辅根服务器，其中9个在美国，欧洲2个，位于英国和瑞典，亚洲1个位于日本。这里的个并不是指物理意义上的单个服务器，它是一个逻辑概念，根DNS服务器可以由分布在全球的多个服务器组成，形成一个集群，对外统一为1台逻辑的根DNS服务器（即每个标号下的根DNS服务器的IP地址是一样的）。而实际物理意义上的根DNS服务器，已超过千台  
[https://root-servers.org/](https://links.jianshu.com/go?to=https%3A%2F%2Froot-servers.org%2F) 这里可以查到所有根DNS服务器的分布

截止发这篇文章之前，中国有19台根DNS服务器，分别位于：

* 北京：F，I，J，L
* 杭州：J
* 澳门：F
* 香港：A、D、E、F、F、I、J
* 台北：E、F、F、I、K、L

**2、顶级域（TLD）DNS服务器**

这些服务器负责顶级域名如com、org、net、edu和gov，以及所有国家的顶级域名如uk、fr、ca和cn

**3、权威DNS服务器**

在因特网上具有公共可访问主机的每个组织机构必须提供公共可访问的DNS记录，这些记录将这些主机的名字映射为IP地址。一个组织机构的权威DNS服务器收藏了这些DNS记录。

除此之外，还有一种很重要的DNS，成为**本地DNS服务器**，其严格来说不属于该服务器的层次结构，但它对DNS层次结构是重要的。每个**ISP** （互联网服务提供商），比如一个大学，一个公司或一个居民区的ISP，都有一台本地DNS服务器。

#### 三、DNS解析过程

**1、迭代查询和递归查询**

很清晰地显示出了一条DNS查询链：本地DNS服务器-->根DNS服务器-->顶级域DNS服务器-->权威DNS服务器 ，所有查询都是递归的。  
这是**递归查询**。

这种利用了**迭代查询**和**递归查询**，从Client与本地DNS之间是**递归查询**，其余则是**迭代查询**。  
所谓 **递归查询**过程 就是 “查询的递交者” 更替, 而 **迭代查询**过程 则是 “查询的递交者”不变。  
从理论上讲，任何DNS查询既可以是迭代的也能是递归的。  
而在实际过程中，更常用的是图上 **从请求主机到本地DNS服务器的查询是递归，其余查询是迭代的**这种方式。

**2、DNS缓存**

**DNS缓存（DNS Caching）**：为了改善时延性能并减少在因特网上到处传输的DNS报文数量，DNS广泛使用了缓存技术。  
在一个请求链中，当某DNS服务器接收一个DNS回答时，它能将该回答中的信息缓存在本地存储器中。那么另一个对相同主机名的查询到达该DNS服务器时，该DNS服务器就可以提供所要求的IP地址，即使它不是该主机名的权威服务器。  
由于IP和主机名的映射并不是永久的，DNS服务器在一段时间后就会丢弃缓存的信息。  
本地DNS服务器也能够缓存TLD服务器的IP地址，从而允许本地DNS绕过查询链中的根DNS服务器。  
**而事实上，有DNS的地方，就有缓存。浏览器、操作系统、本地 DNS服务器、根DNS服务器，它们都会对DNS结果做一定程度的缓存。**

**3、DNS解析过程**

大致分为8步：

* 1、发起基于域名的请求后，首先检查本地缓存（浏览器缓存-->操作系统的hosts文件）
* 2、如果本地缓存中有，直接返回目标IP地址，否则将域名解析请求发送给本地DNS服务器
* 3、如果本地DNS服务器中有，直接返回目标IP地址，到这一步基本能解析80%的域名。如果没有，本地DNS服务器将解析请求发送给根DNS服务器
* 4、根DNS服务器会返回给本地DNS服务器一个所查询的TLD服务器地址列表
* 5、本地DNS服务器再向上一步返回的TLD服务器发送请求，TLD服务器查询并返回域名对应的权威域名服务器的地址
* 6、本地DNS服务器再向上一步返回的权威域名服务器发送请求，权威域名服务器会查询存储的域名和IP的映射关系表，将IP连同一个TTL（过期时间）值返回给本地DNS服务器
* 7、本地DNS服务器会将IP和主机名的映射保存起来，保存时间由TTL来控制
* 8、本地DNS服务器把解析的结果返回给用户，用户根据TTL值缓存在本地系统缓存中，域名解析过程结束

#### 四、DNS记录和报文

* **1、资源记录**

所有DNS服务器都存储了**资源记录（Resource Record，RR）**，其提供了主机名到IP的映射。  
资源记录是一个包含以下字段的四元组：  
**（Name，Value，Type，TTL）**  
TTL是该记录的生存时间，决定了资源记录应当从缓存中删除的时间。  
Name和Value的值取决于Type(以下涉及的foo,bar均为伪变量)：

* Type = A，则Name是主机名，Value使其对应的IP地址。这也是一个标准的主机名到IP地址的映射。如(replay1.bar.foo.com,145.37.93.126,A)
* Type = NS，则Name是个域（如foo.com），而Value是个知道如何获取该域中主机IP地址的权威DNS服务器的主机名，如(foo.com,dns.foo.com,NS)
* Type = CNAME,则Name是别名为Name的主机对应的规范主机名。该记录能够向查询的主机提供一个主机名对应的规范主机名，如（foo.com,replay1.bar.foo.com,CNAME）
* Type = MX，则Value是个别名为Name的邮件服务器的规范主机名。如（foo.com,main.bar.foo.com,MX）

**2、DNS报文**

DNS只有查询和回答两种报文，这两种报文格式是一样的。

* 前12个字节是**首部区域**。  
  标识符用于标识该查询，这个标识符会被复制到对查询的回答报文中，以便让客户用它来匹配发送的请求和接收到的回答。  
  标志字段中含有若干标志。1比特的“查询/回答”标志位指出报文是查询报文（0）还是回答报文（1）。当某DNS服务器是所请求名字的权威DNS服务器时，1比特的“权威的”标志位被置在回答报文中。此外，还有“希望递归”、“递归可用”等标志位。  
  在首部中，还有4个数量相关的字段，指出来在首部后的4类数据区域出现的数量，其中RR是资源记录的意思。
* **问题区域**包含着正在进行的查询信息。该区域包括：️名字字段，指出正在被查询的主机名字；️类型字段，指出有关该名字的正被查询的问题类型，即上边说的四元组中的Type
* **回答区域**包含了对最初请求的名字的资源记录。在回答区域中可以包含多条RR，因此一个主机名理论上能够有多个IP地址（不同用户在不同地点访问同一个域名，可能会访问到不同的IP地址）
* **权威区域**中包含了其他权威服务器的记录
* **附加区域**包含了其他有帮助的记录

得知DNS的报文格式后，我们也就可以手动发送DNS查询包了。  
一个在线DNS查询的工具：[https://tool.lu/dns/index.html](https://links.jianshu.com/go?to=https%3A%2F%2Ftool.lu%2Fdns%2Findex.html)

#### 五、DNS解析安全问题

**1、DNS劫持**

一种可能的域名劫持方式即黑客侵入了宽带路由器并对终端用户的本地DNS服务器进行篡改，指向黑客自己伪造的本地DNS服务器，进而通过控制本地DNS服务器的逻辑返回错误的IP信息进行域名劫持。

另一方面，由于DNS解析主要是基于UDP协议，除了上述攻击行为外，攻击者还可以监听终端用户的域名解析请求，并在本地DNS服务器返回正确结果之前将伪造的DNS解析响应传递给终端用户，进而控制终端用户的域名访问行为。

**2、缓存污染（DNS污染）。**

我们知道在接收到域名解析请求时，本地DNS服务器首先会查找缓存，如果缓存命中就会直接返回缓存结果，不再进行递归DNS查询。这时候如果本地DNS服务器针对部分域名的缓存进行更改，比如将缓存结果指向第三方的广告页，就会导致用户的访问请求被引导到这些广告页地址上。

**3、如何解决DNS劫持？**

DNS解析发生在HTTP协议之前，DNS解析和DNS劫持和HTTP没有关系，DNS协议使用的是UDP协议向服务器的53端口进行请求。  
要想解决DNS劫持：

* 可以使用HttpDNS的方案：**使用 HTTP协议向DNS服务器的80端口进行请求**,来规避DNS劫持  
  比如：http://119.29.29.29/d?dn=domain&ip=clientIp
* 在终端上，可以更换DNS服务器，不管手机还是电脑，都能手动配置DNS

#### 一、Cookie

这里有说到，HTTP协议是无状态的，服务器中没有保存客户端的状态，客户端必须每次带上自己的状态去请求服务器  
基于HTTP这种特点，就产生了cookie/session

**1、用户与服务器的交互：Cookie**

cookie主要是用来记录用户状态，区分用户，**状态保存在客户端**。

* 1.首次访问amazon时，客户端发送一个HTTP请求到服务器端 。服务器端发送一个HTTP响应到客户端，其中包含Set-Cookie头部
* 2.客户端发送一个HTTP请求到服务器端，其中包含Cookie头部。服务器端发送一个HTTP响应到客户端
* 3.隔段时间再去访问时，客户端会直接发包含Cookie头部的HTTP请求。服务器端发送一个HTTP响应到客户端

cookie技术有4个组件：

* 1.在HTTP响应报文中的一个cookie首部行
* 2.在HTTP请求报文中的一个cookie首部行
* 3.在用户端系统中保留一个cookie文件，并由用户的浏览器进行管理
* 4.位于Web站点的一个后端数据库

也就是说，cookie功能需要浏览器的支持。如果浏览器不支持cookie（如大部分手机中的浏览器）或者把cookie禁用了，cookie功能就会失效。

**2、cookie的修改和删除**

在修改cookie的时候，只需要新cookie覆盖旧cookie即可，在覆盖的时候，由于Cookie具有不可跨域名性，注意name、path、domain需与原cookie一致  
删除cookie也一样，设置cookie的过期时间expires为过去的一个时间点，或者maxAge = 0(Cookie的有效期,单位为秒)即可

**3、cookie的安全**

事实上，cookie的使用存在争议，因为它被认为是对用户隐私的一种侵害，而且cookie并不安全  
HTTP协议不仅是无状态的，而且是不安全的。使用HTTP协议的数据不经过任何加密就直接在网络上传播，有被截获的可能。使用HTTP协议传输很机密的内容是一种隐患。

* 如果不希望Cookie在HTTP等非安全协议中传输，可以设置Cookie的secure属性为true。浏览器只会在HTTPS和SSL等安全协议中传输此类Cookie。
* 此外，secure属性并不能对Cookie内容加密，因而不能保证绝对的安全性。如果需要高安全性，需要在程序中对Cookie内容加密、解密，以防泄密。
* 也可以设置cookie为**HttpOnly**，如果在cookie中设置了**HttpOnly**属性，那么通过js脚本将无法读取到cookie信息，这样能有效的防止**XSS（跨站脚本攻击）**攻击

#### 二、Session

除了使用Cookie，Web应用程序中还经常使用**Session**来记录客户端状态。Session是服务器端使用的一种记录客户端状态的机制，使用上比Cookie简单一些，相应的也增加了服务器的存储压力。

Session是另一种记录客户状态的机制，不同的是**Cookie保存在客户端浏览器中，而Session保存在服务器上**。  
客户端浏览器访问服务器的时候，服务器把客户端信息以某种形式记录在服务器上。这就是Session。客户端浏览器再次访问时只需要从该Session中查找该客户的状态就可以了。

* 当程序需要为某个客户端的请求创建一个session时，服务器首先检查这个客户端的请求里是否已包含了一个session标识（称为SessionId）
* 如果已包含则说明以前已经为此客户端创建过session，服务器就按照SessionId把这个session检索出来，使用（检索不到，会新建一个）
* 如果客户端请求不包含SessionId，则为此客户端创建一个session并且生成一个与此session相关联的SessionId，SessionId的值应该是一个既不会重复，又不容易被找到规律以仿造的字符串，这个SessionId将被在本次响应中返回给客户端保存。
* 保存这个SessionId的方式可以采用cookie，这样在交互过程中浏览器可以自动的按照规则把这个标识发送给服务器。但cookie可以被人为的禁止，则必须有其他机制以便在cookie被禁止时仍然能够把SessionId传递回服务器。

#### 三、Cookie 和Session 的区别：

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。  
2、cookie相比session不是很安全，别人可以分析存放在本地的cookie并进行cookie欺骗,考虑到安全应当使用session。  
3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能,考虑到减轻服务器性能方面，应当使用cookie。  
4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。而session存储在服务端，可以无限量存储  
5、所以：将登录信息等重要信息存放为session;其他信息如果需要保留，可以放在cookie中

之前有说到OSI七层协议中的应用层（HTTP协议）、传输层（TCP协议、UDP协议），在传输层之上就是网络层，网络层负责IP数据报的产生以及IP数据包在逻辑网络上的路由转发，网络层分为三个组件：

###### 1、IP协议

###### 2、路由选择协议，它决定了数据报从源到目的地所流经的路径

###### 3、ICMP协议 (Internet Control Message Protocol, 因特网控制报文协议)，报告数据报中的差错和对某些网路层信息请求进行响应对设施。

###### 网络层和传输层的区别

* 网络层只是根据网络地址将源结点发出的数据包传送到目的结点（点到点），其主要任务是：通过路由选择算法，为报文或分组通过通信子网选择最适当的路径。该层控制数据链路层与传输层之间的信息转发，建立、维持和终止网络的连接。具体地说，数据链路层的数据在这一层被转换为数据包，然后通过路径选择、分段组合、顺序、进/出路由等控制，将信息从一个网络设备传送到另一个网络设备。

###### 即网络层提供了主机之间的逻辑通信

* 而传输层则负责将数据可靠地传送到相应的端口（端到端），传输层提供了主机应用程序进程之间的端到端的服务。传输层利用网络层提供的服务，并通过传输层地址提供给高层用户传输数据的通信端口，使高层用户看到的只是在两个传输实体间的一条端到端的、可由用户控制和设定的、可靠的数据通路。

###### 即传输层为运行在不同主机上的进程之间提供了逻辑通信

#### 一、IP协议

IP协议是TCP/IP核心协议。

###### 1、IP协议的数据报格式（IPv4）

* 版本号  
  规定了数据报的IP协议版本（IPv4还是IPv6）。不同的IP版本使用不同的数据报格式 ，上图是IPv4的数据报格式
* 首部长度  
  大多数IP数据报不包含选项，所以一般IP数据报具有20字节的首部。
* 服务类型  
  使不同类型的IP数据报能相互区别开来。
* 数据报长度  
  **整个IP数据报的长度**，**利用首部长度和总长度就可以是算出IP数据报中数据内容的起始地址**。该字段长度为16比特，所以IP数据报最长可达2^16=65535字节，而事实上，数据报很少有超过1500字节的
* 标识、标志、片偏移字段  
  这三个字段与IP分片有关。此外，IPv6不允许在路由器上对分组分片
* 生存时间TTL  
  用来确保数据报不会永远在网络中循环。设置该数据报可以经过的最多的路由器数。指定了数据报的生存时间，经过一个路由器，它的值就减1，当该字段为0时，数据报就被丢弃
* 协议  
  该字段只有在一个IP数据报到达其目的地才有用。该字段值指示了IP数据报的数据部分应交给哪个特定的传输层协议，比如，值为6表明要交给TCP，而值为17则表明要交给UDP
* 首部检验和  
  与UDP/TCP的检验和不同，这个字段只检验数据报的首部，不包括数据部分。
* 选项字段  
  是一个可变长字段，选项字段一直以4字节作为界限。这样就可以保证首部始终是4字节的整数倍。很少被用到
* 源IP和目的IP

记录源IP地址，目的IP地址

* 数据

#### 二、IP数据报分片

一个链路层帧能承载的最大数据量叫做**最大传送单元（Maximun Transmission Unit,MTU）**,即链路层的MTU限制着IP数据报的长度。  
问题是在不同的链路上，可能使用不同的链路层协议，且每种协议可能具有不同的MTU。  
假定在某条链路上收到一个IP数据报，通过检查转发表确定出链路，且出链路的MTU比该IP数据报的长度要小，如何将这个过大的IP数据报压缩进链路层帧的有效载荷字段呢？

解决方案就是**分片**：将IP数据报中的数据分片为两个或更多个较小的IP数据报，用单独的链路层帧封装这些较小的IP数据报，然后向出链路上发送这些帧，每个这些较小的数据都称为**片（fragment）**。

###### 片在到达目的地传输层前需要重新组装。

实际上，TCP和UDP都希望从网络层上收到完整的未分片的报文。IPv4的数据报重组工作是在端系统中，而不是在网络路由器中。  
当一台目的主机从相同源收到一系列数据时，需要确定这些数据报中的某些是否是一些原来较大的数据报中的片。而如果是片的话，需要进一步确定何时收到最后一片，并且如何将这些片拼接到一起以形成初始的数据报。从而就用到了前边说到的IPv4数据报首部中的**标识、标志、片偏移** 字段。

* 1、当生成一个数据报时，发送主机在为该数据报设置源和目的地址的同时在贴上标识号,发送主机通常将为它发送的每个数据报标识号加1
* 2、当某路由器需要对一个数据报分片时，形成的每个数据报（即片）具有初始数据报的源地址、目的地址和标识号
* 3、当目的地从同一发送主机收到一系列数据报时，它能够检查数据报的标识号以确定哪些数据报实际上是同一较大数据报的片
* 4、由于IP协议是不可靠服务，一个或者多个片可能永远到达不了目的地。为了让目的主机绝对相信它已收到初始数据报的最后一个片，最后一个片的标志比特被设为0，其余被设为1
* 5、为了让目的主机确定是否丢失了一个片，并且能按照正确的顺序重新组装片，使用偏移字段指定该片应放在初始IP数据报的哪个位置>  
  此外，如果有一个片或者多个片未能到达，则该不完整的数据报将会被丢弃且不会交给传输层。但如果传输层正使用着TCP，则TCP将通过让源以初始数据来重传数据。因为IP层没有超时重传机制，所以会重传整个数据报，而不是某个片

#### 三、IPv4编址

###### 1、IP地址

一台主机通常只有一条链路连接到网络，当主机上的IP想发送一条数据报时，就在该链路上发送。主机与物理链路之间的边界叫做**接口（interface）**。  
而路由器的任务是从链路上接收数据报并从某些其他链路转发出去，路由器必须有两条或更多链路与其连接，路由器与它的任意一条链路之间的边界也叫做接口。即一台路由器会有多个接口，每个接口有其链路。  
因为每台主机与路由器都能发送和接收IP数据报，IP要求每台主机和路由器接口都有自己的IP地址。因此，一个IP地址技术上是与一个接口相关联的，而不是与包括该接口的主机或路由器相关联的。

###### 2、子网

每个IP地址（IPv4）长度为32比特（4字节），按**点分十进制记法**书写，即地址中的每个字节都用它的十进制形式书写，各字节间以点.隔开，比如193.32.122.30  
在因特网上的每台主机和路由器上的每个接口，必须有一个全球唯一的IP地址（NAT后的接口除外）。这些地址不能自由选择，一个接口的IP地址的一部分需要由其连接的子网来决定。

如图，一台路由器有三个接口，连接7台主机。左侧的三台主机以及连接它们的路由器的接口，都有一个形如223.1.1.x的IP地址。即在它们的IP地址中，最左侧的24比特是相同的。

互联左侧这三个主机接口与1个路由器接口的网络形成1个**子网（subnet）**（也被称为IP网络或直接称为网络）。IP编址为这个子网分配一个地址：223.1.1.0/24，其中的/24记法，有时称为**子网掩码（network mask）**，指示了32比特中的最左侧24比特定义了子网地址。任何连接到该子网的主机都要求其地址具有223.1.1.x的形式。同样图中下侧和右侧也是子网，分别为223.1.3.0/24和223.1.2.0/24

上图显示了3台通过点对点链路彼此互联的路由器，这里出现了6个子网。  
一个具有多个以太网段和点对点链路的组织将具有多个子网，在给定子网上的所有设备都具有相同的子网地址。  
虽然在理论上来说，不同子网可以有完全不同的子网地址。但上图可以看出，这6个子网在前16个比特是一致的，都是223.1

###### 3、无类别域间路由选择（CIDR）

因特网的地址分配策略被称为**无类别域间路由选择（CIDR）**(也被称为无分类编址，以区分于分类编址)。对于子网寻址，32比特的IP地址被分为两部分，也是点分十进制数形式a.b.c.d/x，其中x指示了地址的第一部分中的比特数，又被称为该地址的**前缀（prefix）**。  
一个组织通常被分配一块连续的地址，即具有相同前缀的地址。在该组织外的的路由器仅考虑前面的前缀比特x，这相当大地减少了在这些路由器中转发表的长度，形式为a.b.c.d/x单一表项足以将数据报转发到该组织内的任何目的地。

如图，200.23.16.0/20下有8个组织，分别是200.23.16.0/23到200.23.30.0/23，每个组织有自己的子网。而外界不需要知道这8个组织。这种使用单个网络前缀通告多个网络的能力通常称为**地址聚合**，也称为**路由聚合**或**路由摘要**

###### 4、分类编址

在CIDR被采用之前，IP地址的网络部分被限制长度为8、16、24比特，也就是**分类编址（classful addressing）**。具有8、16、24比特子网地址的子网被称为A、B和C类网络。  
一个C类（/24）子网既能容纳**2的8次方 - 2 = 254**台主机（其中两个地址预留用于特殊用途），这对于很多组织来说都太小了。  
而一个B类（/16）子网可支持多达**2的16次方 - 2 = 65534**  
台主机，又太大了。  
在分类编址方法下，一个有2000台主机的组织通常被分给一个B类（/16）地址，那么剩下的6万多个地址就浪费掉了。这就会导致**B类地址空间的迅速损耗以及所分配的地址空间的利用率低**。

此外，255.255.255.255是IP广播地址，当一台主机发出一个目的地址为该地址的数据报时，该报文会交付给同一个网络中的所有主机。

###### 5、获取主机地址

某组织一旦获得了一块地址，它就可为本组织内的主机与路由器逐一分配IP地址。系统管理员通常手工配置路由器中的IP地址。主机地址也能手动配置，但更多使用的是**动态主机配置协议（DHCP）**。DHCP允许主机自动获取IP地址。网络管理员可以配置DHCP，以使某给定主机每次与网络连接时能得到一个相同的IP地址，或者某主机将被分配一个**临时的IP地址**，该地址在每次与网络连接时也许是不同的。

###### 6、网络地址转换

每个IP地址（IPv4）长度为32比特（4字节），因此总共有2[图片上传失败...(image-524f6a-1565267828082)]

个可能的IP地址，约为40亿个。在互联网越来越普及的当下，个人计算机及智能手机等越来越多，这些IP地址显然无法满足人们的需求。  
为了解决IP地址不足的问题，于是就有了**网络地址转换(Network Address Translation， NAT)**，它的思想就是给一个局域网络分配一个IP地址就够了，对于这个网络内的主机，则分配私有地址，这些私有地址对外是不可见的，他们对外的通信都要借助那个唯一分配的IP地址。

如果从广域网到达NAT路由器的所有数据报都有相同的目的IP地址，那么该路由器如何知道是发送给哪个内部主机的呢？其原理就是使用在NAT路由器上的一张**NAT转换表**，并在该表内包含了端口号及其IP地址。

假设一台主机向广域网请求数据，NAT路由器收到该数据报，会为该数据报生成一个新的端口号替换掉源端口号，并将源IP替换为其广域网一侧接口的IP地址。当生成一个新的源端口号时，该端口号可以是任意一个当前未在NAT转换表中的源端口号（端口号字段是16比特，意味着NAT协议可以支持超过60000个并行使用路由器广域网一侧IP地址的连接），路由器中的NAT也在其NAT转换表中增加一表项。

该NAT路由器收到广域网返回的数据时，路由器使用目的IP地址与目的端口号从NAT转换表中检索出该主机使用的IP地址和目的端口号，改写该数据报的目的IP地址和目的端口号，并向该主机转发该数据报

NAT虽然在近几年得到了很广泛的应用，但也被很多人反对。  
主要是：

* 1、端口号是用来进程编址的，而不是主机编址的（NAT协议类似 NAT路由器将该家庭网络的主机都当做进程处理，并通过NAT转换表为其分配端口号）
* 2、路由器通常仅应当处理高达第三层的分组
* 3、违背端到端原则，即主机彼此之间应当相互直接对话，结点不应当介入修改IP地址与端口号。
* 4、应当用IPv6来解决IP地址短缺问题

但不管反对与否，NAT终究已成为当今因特网的一个重要组件

由于新的子网和IP结点以惊人的增长率连到因特网上，并被分配唯一的IP地址，32比特的IPv4地址空间即将用尽，为了解决这一问题，**IPv6**也应运而生。而事实上在20多年前，因特网工程任务组就开始致力于开发一种替代IPv4的协议，即**IPv6**

#### 一、IPv6数据报格式

###### 1、IPv6数据报格式

image

* **版本**（4比特）  
  该字段用于标识IP版本号，IPv6将该字段值设为6。而如果将该字段设为4并不能创建一个合法的IPv4数据报
* **流量类型**（8比特）  
  类似于IPv4数据报中的服务类型（TOS）
* **流标签**（20比特）  
  流标签字段是IPv6数据报中新增的一个字段，用来标识一条数据报的流类型，以便在网络层区分不同的报文。
* **有效载荷长度**（16比特）

IPv6数据报中在40定长字节数据报首部后的字节数量，即除了IPv6的数据报首部以外的其他部分的总长度

* **下一个首部**（8比特）  
  当IPv6没有扩展报头时，该字段的作用和IPv4的协议字段一样。当含有扩展报头时，该字段的值即为第一个扩展报头的类型
* **跳限制**（8比特）  
  与IPv4报文中的TTL字段类似，转发数据报的每台路由器将对该字段的内容减1.如果跳限制计数到达0，则该数据报将被丢弃
* **源地址和目的地址**（各128比特）  
  记录源IP地址，目的IP地址
* **数据**

可以看出，在IPv4数据报中出现的几个字段在IPv6数据报中已不复存在：

* **分片/重新组装**  
  IPv6不允许在中间路由器上进行分片和重新组装。这种操作只能在源与目的地上执行。如果路由器收到的IPv6数据报因太大不能转发出链路上的话，路由器会丢掉该数据报，并回一个“分组太大”的ICMP差错报文
* **首部检验和**  
  因为运输层和数据链路层协议执行了检验操作，该项功能在网络层就没有必要了，从而更快速处理IP分组
* **选项**  
  选项字段不再是标准IP首部的一部分了。但并没有消失，而是可能出现在IPv6首部中由“下一个首部”指出的位置上。即就像TCP或UDP协议首部能够是IP分组中的“下一个首部”，选项字段也能是“下一个首部”

IPv6相对IPv4最重要的变化如下：

* **扩大的地址容量**  
  IPv6将IP地址长度由32比特增加到128比特，这使得理论可存在的IP地址增加到2[图片上传失败...(image-a67865-1565268140183)]

个，约340万亿亿亿亿个，这是一个非常大的数字，确保全世界再也不会用尽IP地址，甚至可以为地球上每一粒沙子都分配一个唯一的IP地址  
除了单播和多播地址外，IPv6没有广播这一说法，而是引入了一种称为**任播地址**的新型地址，这种地址可以使数据报交付给一组主机中的任意一个

* **简化高效的40字节首部**  
  除去共32字节的源地址和目标地址外，首部其余字段只占了8字节
* **流标签与优先级**  
  给属于特殊流的分组打上标签，这些特殊流是发送方要求进行特殊处理的流，如一种非默认服务质量或需要实时服务的流

###### 2、IPv6书写和表达方式

表述和书写时，把长度为128比特的IPv6地址分成8个16位的二进制段、每一个16位的二进制段用4位的16进制数表示，段间用“：”（冒号）隔开（其书写方法和IPv4的十进制数加“.”不同）。

例如：1000:0000:0000:0000:000A:000B:000C:000D就是每一个16位的二进制数的段用4位16进制数的段来表示、段间用“：”（冒号）隔开的一个IPv6地址；其中：各个4位16进制数的段中的高位0允许省略；因此，上面的IPv6地址也可以缩写成：1000:0:0:0:A:B:C:D。

为了更进一步简化，IPv6的地址规范中还规定，可以在一个IPv6地址中**最多使用一次双冒号（::）**来取代IPv6地址中紧密相连的多个全0的16进制数的段（因为如果允许在一个IPv6地址中使用一次以上的双冒号时将无法判断IPv6地址的长度，所以IPv6的地址规范中才规定：在一个IPv6地址中最多只能使用一次双冒号），这样上面的IPv6地址还可以缩写成：1000::A:B:C:D。

双冒号使用的地点可以在IPv6地址的前面、后面或者是中间；例如：对于1000:0:0:0:A:B:0:0这样的一个IPv6地址，可以写成1000::A:B:0:0，也可以写成1000:0:0:0:A:B::；但是不能写成1000::A:B::。

带有端口号的IPV6地址字符串形式，地址部分应当用“[]”括起来，在后面跟着‘：’带上端口号，如 [A01F::0]:8000

#### 二、从IPv4到IPv6的迁移

基于IPv4的公共因特网如何迁移到IPv6呢？这是个非常现实的问题  
虽然IPv6使能系统可做成向后兼容，即能接收、发送和路由IPv4数据报，但已部署的IPv4使能系统却不能处理IPv6数据报

###### 1、双协议栈

引入IPv6使能结点的最直接方式是**双栈**方法，即使用该方法的IPv6结点还有完整的IPv4实现，即**IPv6/IPv4结点**，具有接收和发送IPv4和IPv6两种数据报的能力。  
当与IPv4结点互操作时，IPv6/IPv4结点可使用IPv4数据报；当与IPv6结点互操作时，IPv6/IPv4结点又可使用IPv6数据报。

IPv6/IPv4结点必须有IPv6与IPv4两种地址。此外，它们还必须能确定另一个结点是否是IPv6使能的或仅IPv4使能的。

可以使用**DNS**来解决，若要解析的结点名字是IPv6使能的，则DNS会返回一个IPv6地址，否则返回一个IPv4地址。如果发出DNS请求的结点是仅IPv4使能的，则只返回一个IPv4地址。

两个IPv6使能的结点不应相互发送IPv4数据报，而如果发送方或接收方任意一个仅为IPv4使能的,则必须使用IPv4数据报。  
这样就会有下面这种情况：

如图，假如结点A、B、E、F都是IPv6使能的结点，而结点C和D是仅IPv4使能的结点，那么当按A->B->C->D->E->F顺序发送数据报时,AB之间会发IPv6数据报，BC会发IPV4数据报， 由于IPv6数据报特定的字段在IPv4数据报中无对应的部分，这些字段将会丢失。因此，即使E和F之间能发IPv6数据报，从D到达E的IPv4数据报并未含有从A发出的初始IPv6数据报中的所有字段。

###### 2、隧道

**建隧道**是另一种双栈方法，该方法能解决上述问题。  
假定两个IPv6结点要使用IPv6数据报进行交互，但是它们是经由中间IPv4路由器互联的。将两台IPv6路由器中间的IPv4路由器的集合成为一个隧道，如B->C->D->E。

如图，借助于隧道，在隧道发送端的IPv6结点可将整个IPv6数据报放到一个IPv4数据报的数据字段中。于是，该IPv4数据报的地址设为指向隧道接收端的IPv6结点，再发送给隧道中的第一个结点。而隧道中的IPv4路由器在它们之间为该数据报提供路由，就像对待其他IPv4数据报一样，完全不知道该数据报自身就含有一个完整的IPv6数据报。而隧道接收端的IPv6结点最终收到该IPv4数据报，并确定该IPv4数据报中含有一个IPv6数据报，于是提取出该IPv6数据报，然后再为该IPv6数据报提供路由

###### 3、NAT-PT

除了双栈和隧道方案外，还有一种**NAT-PT(Network Address Translator - Protocol Translator)附带协议转换器的网络地址转换器**方案  
[IPv4 IPv6共存技术-----NAT-PT](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.csdn.net%2Fshengmingbi%2Farticle%2Fdetails%2F70160161)  
可以参考下这篇文章

[IPv6 测试](https://links.jianshu.com/go?to=%255Bhttp%3A%2F%2Ftest-ipv6.com%2F%255D%28http%3A%2F%2Ftest-ipv6.com%2F%29)可在这测试是否支持IPv6

#### 十三、性能优化

# iOS 性能优化面试题

在性能优化中一个最具参考价值的属性是FPS:Frames Per Second,其实就是屏幕刷新率，苹果的iphone推荐的刷新率是60Hz，也就是说GPU每秒钟刷新屏幕60次，这每刷新一次就是一帧frame，FPS也就是每秒钟刷新多少帧画面。静止不变的页面FPS值是0，这个值是没有参考意义的，只有当页面在执行动画或者滑动的时候，FPS值才具有参考价值，FPS值的大小体现了页面的流畅程度高低，当低于45的时候卡顿会比较明显。

**图层混合：**

每一个layer是一个纹理，所有的纹理都以某种方式堆叠在彼此的顶部。对于屏幕上的每一个像素，GPU需要算出怎么混合这些纹理来得到像素RGB的值。

当Sa = 0.5时，RGB值为(0.5, 0, 0)，可以看出，当两个不是完全不透明的CALayer覆盖在一起时,GPU大量做这种复合操作，随着这中操作的越多，GPU越忙碌，性能肯定会受到影响。

**公式：**

R = S + D \* ( 1 – Sa )

结果的颜色是源色彩(顶端纹理)+目标颜色(低一层的纹理)\*(1-源颜色的透明度)。

当Sa = 1时，R = S,GPU将不会做任何合成，而是简单从这个层拷贝，不需要考虑它下方的任何东西(因为都被它遮挡住了)，这节省了GPU相当大的工作量。

#### 一、入门级

1、用ARC管理内存  
2、在正确的地方使用 reuseIdentifier  
3、尽量把views设置为透明  
4、避免过于庞大的XIB  
5、不要阻塞主线程

6、在ImageViews中调整图片大小。如果要在UIImageView中显示一个来自bundle的图片，你应保证图片的大小和UIImageView的大小相同。在运行中缩放图片是很耗费资源的，特别是UIImageView嵌套在UIScrollView中的情况下。如果图片是从远端服务加载的你不能控制图片大小，比如在下载前调整到合适大小的话，你可以在下载完成后，最好是用background  
thread，缩放一次，然后在UIImageView中使用缩放后的图片。

7、选择正确的Collection。

* Arrays: 有序的一组值。使用index来lookup很快，使用value lookup很慢， 插入/删除很慢。
* Dictionaries: 存储键值对。 用键来查找比较快。
* Sets: 无序的一组值。用值来查找很快，插入/删除很快。

8、打开gzip压缩。app可能大量依赖于服务器资源，问题是我们的目标是移动设备，因此你就不能指望网络状况有多好。减小文档的一个方式就是在服务端和你的app中打开gzip。这对于文字这种能有更高压缩率的数据来说会有更显著的效用。  
iOS已经在NSURLConnection中默认支持了gzip压缩，当然AFNetworking这些基于它的框架亦然。

#### 二、中级

1、重用和延迟加载(lazy load) Views

* 更多的view意味着更多的渲染，也就是更多的CPU和内存消耗，对于那种嵌套了很多view在UIScrollView里边的app更是如此。
* 这里我们用到的技巧就是模仿UITableView和UICollectionView的操作: 不要一次创建所有的subview，而是当需要时才创建，当它们完成了使命，把他们放进一个可重用的队列中。这样的话你就只需要在滚动发生时创建你的views，避免了不划算的内存分配。

2、Cache, Cache, 还是Cache!

* 一个极好的原则就是，缓存所需要的，也就是那些不大可能改变但是需要经常读取的东西。
* 我们能缓存些什么呢？一些选项是，远端服务器的响应，图片，甚至计算结果，比如UITableView的行高。
* NSCache和NSDictionary类似，不同的是系统回收内存的时候它会自动删掉它的内容。

3、权衡渲染方法.性能能还是要bundle保持合适的大小。

4、处理内存警告.移除对缓存，图片object和其他一些可以重创建的objects的strong references.

5、重用大开销对象

6、一些objects的初始化很慢，比如NSDateFormatter和NSCalendar。然而，你又不可避免地需要使用它们，比如从JSON或者XML中解析数据。想要避免使用这个对象的瓶颈你就需要重用他们，可以通过添加属性到你的class里或者创建静态变量来实现。

7、避免反复处理数据.在服务器端和客户端使用相同的数据结构很重要。

8、选择正确的数据格式.解析JSON会比XML更快一些，JSON也通常更小更便于传输。从iOS5起有了官方内建的JSON deserialization 就更加方便使用了。但是XML也有XML的好处，比如使用SAX 来解析XML就像解析本地文件一样，你不需像解析json一样等到整个文档下载完成才开始解析。当你处理很大的数据的时候就会极大地减低内存消耗和增加性能。

9、正确设定背景图片

* 全屏背景图，在view中添加一个UIImageView作为一个子View
* 只是某个小的view的背景图，你就需要用UIColor的colorWithPatternImage来做了，它会更快地渲染也不会花费很多内存：

10、减少使用Web特性。想要更高的性能你就要调整下你的HTML了。第一件要做的事就是尽可能移除不必要的javascript，避免使用过大的框架。能只用原生js就更好了。尽可能异步加载例如用户行为统计script这种不影响页面表达的javascript。注意你使用的图片，保证图片的符合你使用的大小。

11、Shadow Path 。CoreAnimation不得不先在后台得出你的图形并加好阴影然后才渲染，这开销是很大的。使用shadowPath的话就避免了这个问题。使用shadow path的话iOS就不必每次都计算如何渲染，它使用一个预先计算好的路径。但问题是自己计算path的话可能在某些View中比较困难，且每当view的frame变化的时候你都需要去update shadow path.

12、优化Table View

* 正确使用reuseIdentifier来重用cells
* 尽量使所有的view opaque，包括cell自身
* 避免渐变，图片缩放，后台选人
* 缓存行高
* 如果cell内现实的内容来自web，使用异步加载，缓存请求结果
* 使用shadowPath来画阴影
* 减少subviews的数量
* 尽量不适用cellForRowAtIndexPath:，如果你需要用到它，只用-一次然后缓存结果
* 使用正确的数据结构来存储数据
* 使用rowHeight, sectionFooterHeight 和 sectionHeaderHeight来设定固定的高，不要请求delegate

13、选择正确的数据存储选项

* NSUserDefaults的问题是什么？虽然它很nice也很便捷，但是它只适用于小数据，比如一些简单的布尔型的设置选项，再大点你就要考虑其它方式了
* XML这种结构化档案呢？总体来说，你需要读取整个文件到内存里去解析，这样是很不经济的。使用SAX又是一个很麻烦的事情。
* NSCoding？不幸的是，它也需要读写文件，所以也有以上问题。
* 在这种应用场景下，使用SQLite 或者 Core Data比较好。使用这些技术你用特定的查询语句就能只加载你需要的对象。
* 在性能层面来讲，SQLite和Core Data是很相似的。他们的不同在于具体使用方法。
* Core Data代表一个对象的graph model，但SQLite就是一个DBMS。
* Apple在一般情况下建议使用Core Data，但是如果你有理由不使用它，那么就去使用更加底层的SQLite吧。
* 如果你使用SQLite，你可以用FMDB这个库来简化SQLite的操作，这样你就不用花很多经历了解SQLite的C API了。

#### 三、高级

1、加速启动时间。快速打开app是很重要的，特别是用户第一次打开它时，对app来讲，第一印象太太太重要了。你能做的就是使它尽可能做更多的异步任务，比如加载远端或者数据库数据，解析数据。避免过于庞大的XIB，因为他们是在主线程上加载的。所以尽量使用没有这个问题的Storyboards吧！一定要把设备从Xcode断开来测试启动速度

2、使用Autorelease Pool。NSAutoreleasePool`负责释放block中的autoreleased objects。一般情况下它会自动被UIKit调用。但是有些状况下你也需要手动去创建它。假如你创建很多临时对象，你会发现内存一直在减少直到这些对象被release的时候。这是因为只有当UIKit用光了autorelease pool的时候memory才会被释放。消息是你可以在你自己的@autoreleasepool里创建临时的对象来避免这个行为。

3、选择是否缓存图片。常见的从bundle中加载图片的方式有两种，一个是用imageNamed，二是用imageWithContentsOfFile，第一种比较常见一点。

4、避免日期格式转换。如果你要用NSDateFormatter来处理很多日期格式，应该小心以待。就像先前提到的，任何时候重用NSDateFormatters都是一个好的实践。如果你可以控制你所处理的日期格式，尽量选择Unix时间戳。你可以方便地从时间戳转换到NSDate:

- (NSDate\*)dateFromUnixTimestamp:(NSTimeInterval)timestamp {

return[NSDate dateWithTimeIntervalSince1970:timestamp];

}

这样会比用C来解析日期字符串还快！需要注意的是，许多web API会以微秒的形式返回时间戳，因为这种格式在javascript中更方便使用。记住用dateFromUnixTimestamp之前除以1000就好了。

**平时你是如何对代码进行性能优化的？**

* 利用性能分析工具检测，包括静态 Analyze 工具，以及运行时 Profile 工具，通过Xcode工具栏中Product->Profile可以启动,
* 比如测试程序启动运行时间，当点击Time Profiler应用程序开始运行后.就能获取到整个应用程序运行消耗时间分布和百分比.为了保证数据分析在统一使用场景真实需要注意一定要使用真机,因为此时模拟器是运行在Mac上，而Mac上的CPU往往比iOS设备要快。
* 为了防止一个应用占用过多的系统资源，开发iOS的苹果工程师门设计了一个“看门狗”的机制。在不同的场景下，“看门狗”会监测应用的性能。如果超出了该场景所规定的运行时间，“看门狗”就会强制终结这个应用的进程。开发者们在crashlog里面，会看到诸如0x8badf00d这样的错误代码。

**优化Table View**

* 正确使用reuseIdentifier来重用cells
* 尽量使所有的view opaque，包括cell自身
* 如果cell内现实的内容来自web，使用异步加载，缓存请求结果  
  减少subviews的数量
* 尽量不适用cellForRowAtIndexPath:，如果你需要用到它，只用一次然后缓存结果
* 使用rowHeight, sectionFooterHeight和sectionHeaderHeight来设定固定的高，不要请求delegate

**UIImage加载图片性能问题**

* imagedNamed初始化
* imageWithContentsOfFile初始化
* imageNamed默认加载图片成功后会内存中缓存图片,这个方法用一个指定的名字在系统缓存中查找并返回一个图片对象.如果缓存中没有找到相应的图片对象,则从指定地方加载图片然后缓存对象，并返回这个图片对象.
* imageWithContentsOfFile则仅只加载图片,不缓存.
* 加载一张大图并且使用一次，用imageWithContentsOfFile是最好,这样CPU不需要做缓存节约时间.
* 使用场景需要编程时，应该根据实际应用场景加以区分，UIimage虽小，但使用元素较多问题会有所凸显.
  + 不要在viewWillAppear 中做费时的操作：viewWillAppear: 在view显示之前被调用，出于效率考虑，方法中不要处理复杂费时操作；在该方法设置 view 的显示属性之类的简单事情，比如背景色，字体等。否则，会明显感觉到 view 有卡顿或者延迟。
  + 在正确的地方使用reuseIdentifier：table view用 tableView:cellForRowAtIndexPath:为rows分配cells的时候，它的数据应该重用自UITableViewCell。
  + 尽量把views设置为透明：如果你有透明的Views你应该设置它们的opaque属性为YES。系统用一个最优的方式渲染这些views。这个简单的属性在IB或者代码里都可以设定。
  + 避免过于庞大的XIB：尽量简单的为每个Controller配置一个单独的XIB，尽可能把一个View Controller的view层次结构分散到单独的XIB中去, 当你加载一个引用了图片或者声音资源的nib时，nib加载代码会把图片和声音文件写进内存。
  + 不要阻塞主线程：永远不要使主线程承担过多。因为UIKit在主线程上做所有工作，渲染，管理触摸反应，回应输入等都需要在它上面完成,大部分阻碍主进程的情形是你的app在做一些牵涉到读写外部资源的I/O操作，比如存储或者网络。  
    dispatch\_async(dispatch\_get\_global\_queue(DISPATCH\_QUEUE\_PRIORITY\_DEFAULT, 0), ^{  
    // 选择一个子线程来执行耗时操作  
    dispatch\_async(dispatch\_get\_main\_queue(), ^{  
    // 返回主线程更新UI  
    });  
    });
  + 在Image Views中调整图片大小  
    如果要在UIImageView中显示一个来自bundle的图片，你应保证图片的大小和UIImageView的大小相同。在运行中缩放图片是很耗费资源的.

**讲讲你用Instrument优化动画性能的经历吧（别问我什么是Instrument）**

Apple的instrument为开发者提供了各种template去优化app性能和定位问题。很多公司都在赶feature，并没有充足的时间来做优化，导致不少开发者对instrument不怎么熟悉。但这里面其实涵盖了非常完整的计算机基础理论知识体系，memory，disk，network，thread，cpu，gpu等等，顺藤摸瓜去学习，是一笔巨大的知识财富。动画性能只是其中一个template，重点还是理解上面问题当中CPU GPU如何配合工作的知识。

**facebook启动时间优化**

1.瘦身请求依赖  
2.UDP启动请求先行缓存  
3.队列串行化处理启动响应

光栅化是将几何数据经过一系列变换后最终转换为像素，从而呈现在显示设备上的过程，光栅化的本质是坐标变换、几何离散化

我们使用 UITableView 和 UICollectionView 时经常会遇到各个 Cell 的样式是一样的，这时候我们可以使用这个属性提高性能：

cell.layer.shouldRasterize=YES;

cell.layer.rasterizationScale=[[UIScreenmainScreen]scale];

## 日常如何检查内存泄露？

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，感兴趣的可以看一下这两篇文章：

* [MLeaksFinder：精准 iOS 内存泄露检测工具](https://links.jianshu.com/go?to=http%3A%2F%2Fwereadteam.github.io%2F2016%2F02%2F22%2FMLeaksFinder%2F)
* [MLeaksFinder 新特性](https://links.jianshu.com/go?to=http%3A%2F%2Fwereadteam.github.io%2F2016%2F07%2F20%2FMLeaksFinder2%2F)

## 如何高性能的画一个圆角？

视图和圆角的大小对帧率并没有什么卵影响，数量才是伤害的核心输出

label.layer.cornerRadius = 5

label.layer.masksToBounds = true

首先上面的方式是不可取的，会触发离屏渲染。

* 如果能够只用 cornerRadius 解决问题，就不用优化。
* 如果必须设置 masksToBounds，可以参考圆角视图的数量，如果数量较少（一页只有几个）也可以考虑不用优化。
* UIImageView 的圆角通过直接截取图片实现，其它视图的圆角可以通过 Core Graphics 画出圆角矩形实现。

image

## 如何提升 tableview 的流畅度？

本质上是降低 CPU、GPU 的工作，从这两个大的方面去提升性能。

* CPU：对象的创建和销毁、对象属性的调整、布局计算、文本的计算和排版、图片的格式转换和解码、图像的绘制
* GPU：纹理的渲染

#### 卡顿优化在 CPU 层面

* 尽量用轻量级的对象，比如用不到事件处理的地方，可以考虑使用 CALayer 取代 UIView
* 不要频繁地调用 UIView 的相关属性，比如 frame、bounds、transform 等属性，尽量减少不必要的修改
* 尽量提前计算好布局，在有需要时一次性调整对应的属性，不要多次修改属性
* Autolayout 会比直接设置 frame 消耗更多的 CPU 资源
* 图片的 size 最好刚好跟 UIImageView 的 size 保持一致
* 控制一下线程的最大并发数量
* 尽量把耗时的操作放到子线程
  + 文本处理（尺寸计算、绘制）
  + 图片处理（解码、绘制）

#### 卡顿优化在 GPU层面

* 尽量避免短时间内大量图片的显示，尽可能将多张图片合成一张进行显示
* GPU能处理的最大纹理尺寸是 4096x4096，一旦超过这个尺寸，就会占用 CPU 资源进行处理，所以纹理尽量不要超过这个尺寸
* 尽量减少视图数量和层次
* 减少透明的视图（alpha<1），不透明的就设置 opaque 为 YES
* 尽量避免出现离屏渲染

[iOS 保持界面流畅的技巧](https://links.jianshu.com/go?to=https%3A%2F%2Fblog.ibireme.com%2F2015%2F11%2F12%2Fsmooth_user_interfaces_for_ios%2F)

##### 1.预排版，提前计算

在接收到服务端返回的数据后，尽量将 CoreText 排版的结果、单个控件的高度、cell 整体的高度提前计算好，将其存储在模型的属性中。需要使用时，直接从模型中往外取，避免了计算的过程。

尽量少用 UILabel，可以使用 CALayer 。避免使用 AutoLayout 的自动布局技术，采取纯代码的方式

##### 2.预渲染，提前绘制

例如圆形的图标可以提前在，在接收到网络返回数据时，在后台线程进行处理，直接存储在模型数据里，回到主线程后直接调用就可以了

避免使用 CALayer 的 Border、corner、shadow、mask 等技术，这些都会触发离屏渲染。

##### 3.异步绘制

##### 4.全局并发线程

##### 5.高效的图片异步加载

## 如何优化 APP 的电量？

###### 程序的耗电主要在以下四个方面：

* CPU 处理
* 定位
* 网络
* 图像

#### 优化的途径主要体现在以下几个方面：

* 尽可能降低 CPU、GPU 的功耗。
* 尽量少用 定时器。
* 优化 I/O 操作。
  + 不要频繁写入小数据，而是积攒到一定数量再写入
  + 读写大量的数据可以使用 Dispatch\_io ，GCD 内部已经做了优化。
  + 数据量比较大时，建议使用数据库
* 网络方面的优化
  + 减少压缩网络数据 （XML -> JSON -> ProtoBuf），如果可能建议使用 ProtoBuf。
  + 如果请求的返回数据相同，可以使用 NSCache 进行缓存
  + 使用断点续传，避免因网络失败后要重新下载。
  + 网络不可用的时候，不尝试进行网络请求
  + 长时间的网络请求，要提供可以取消的操作
  + 采取批量传输。下载视频流的时候，尽量一大块一大块的进行下载，广告可以一次下载多个
* 定位层面的优化
  + 如果只是需要快速确定用户位置，最好用 CLLocationManager 的 requestLocation 方法。定位完成后，会自动让定位硬件断电
  + 如果不是导航应用，尽量不要实时更新位置，定位完毕就关掉定位服务
  + 尽量降低定位精度，比如尽量不要使用精度最高的 kCLLocationAccuracyBest
  + 需要后台定位时，尽量设置 pausesLocationUpdatesAutomatically 为 YES，如果用户不太可能移动的时候系统会自动暂停位置更新
  + 尽量不要使用 startMonitoringSignificantLocationChanges，优先考虑 startMonitoringForRegion:
* 硬件检测优化
  + 用户移动、摇晃、倾斜设备时，会产生动作(motion)事件，这些事件由加速度计、陀螺仪、磁力计等硬件检测。在不需要检测的场合，应该及时关闭这些硬件

## 如何有效降低 APP 包的大小？

降低包大小需要从两方面着手

##### 可执行文件

* 编译器优化
  + Strip Linked Product、Make Strings Read-Only、Symbols Hidden by Default 设置为 YES
  + 去掉异常支持，Enable C++ Exceptions、Enable Objective-C Exceptions 设置为 NO， Other C Flags 添加 -fno-exceptions
* 利用 AppCode 检测未使用的代码：菜单栏 -> Code -> Inspect Code
* 编写LLVM插件检测出重复代码、未被调用的代码

##### 资源

资源包括 图片、音频、视频 等

* 优化的方式可以对资源进行无损的压缩
* 去除没有用到的资源： [https://github.com/tinymind/LSUnusedResources](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2Ftinymind%2FLSUnusedResources)

## 什么是 离屏渲染？什么情况下会触发？该如何应对？

离屏渲染就是在当前屏幕缓冲区以外，新开辟一个缓冲区进行操作。

###### 离屏渲染出发的场景有以下：

* 圆角 （maskToBounds并用才会触发）
* 图层蒙版
* 阴影
* 光栅化

###### 为什么要避免离屏渲染？

CPU GPU 在绘制渲染视图时做了大量的工作。离屏渲染发生在 GPU 层面上，会创建新的渲染缓冲区，会触发 OpenGL 的多通道渲染管线，图形上下文的切换会造成额外的开销，增加 GPU 工作量。如果 CPU GPU 累计耗时 16.67 毫秒还没有完成，就会造成卡顿掉帧。

圆角属性、蒙层遮罩 都会触发离屏渲染。指定了以上属性，标记了它在新的图形上下文中，在未愈合之前，不可以用于显示的时候就出发了离屏渲染。

* 在OpenGL中，GPU有2种渲染方式
  + On-Screen Rendering：当前屏幕渲染，在当前用于显示的屏幕缓冲区进行渲染操作
  + Off-Screen Rendering：离屏渲染，在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作
* 离屏渲染消耗性能的原因
  + 需要创建新的缓冲区
  + 离屏渲染的整个过程，需要多次切换上下文环境，先是从当前屏幕（On-Screen）切换到离屏（Off-Screen）；等到离屏渲染结束以后，将离屏缓冲区的渲染结果显示到屏幕上，又需要将上下文环境从离屏切换到当前屏幕
* 哪些操作会触发离屏渲染？
  + 光栅化，layer.shouldRasterize = YES
  + 遮罩，layer.mask
  + 圆角，同时设置 layer.masksToBounds = YES、layer.cornerRadius大于0
  + 考虑通过 CoreGraphics 绘制裁剪圆角，或者叫美工提供圆角图片
  + 阴影，layer.shadowXXX，如果设置了 layer.shadowPath 就不会产生离屏渲染

# iOS 性能优化面试题（如何检测离屏渲染？）

[全网iOS面试题总结](https://www.jianshu.com/u/0ac1433a3074)关注

0.3312019.08.10 20:19:29字数 852阅读 1,453

1、模拟器debug-选中color Offscreen - Renderd离屏渲染的图层高亮成黄 可能存在性能问题

2、真机Instrument-选中Core Animation-勾选Color Offscreen-Rendered Yellow

**离屏渲染的触发方式**

设置了以下属性时，都会触发离屏绘制：

1、layer.shouldRasterize（光栅化）

光栅化概念：将图转化为一个个栅格组成的图象。

光栅化特点：每个元素对应帧缓冲区中的一像素。

2、masks（遮罩）

3、shadows（阴影）

4、edge antialiasing（抗锯齿）

5、group opacity（不透明）

6、复杂形状设置圆角等

7、渐变

8、drawRect

例如我们日程经常打交道的TableViewCell,因为TableViewCell的重绘是很频繁的（因为Cell的复用）,如果Cell的内容不断变化,则Cell需要不断重绘,如果此时设置了cell.layer可光栅化。则会造成大量的离屏渲染,降低图形性能。

如果将不在GPU的当前屏幕缓冲区中进行的渲染都称为离屏渲染，那么就还有另一种特殊的“离屏渲染”方式：CPU渲染。如果我们重写了drawRect方法，并且使用任何Core Graphics的技术进行了绘制操作，就涉及到了CPU渲染。整个渲染过程由CPU在App内同步地完成，渲染得到的bitmap最后再交由GPU用于显示。

现在摆在我们面前得有三个选择：当前屏幕渲染、离屏渲染、CPU渲染，该用哪个呢？这需要根据具体的使用场景来决定。

**尽量使用当前屏幕渲染**

鉴于离屏渲染、CPU渲染可能带来的性能问题，一般情况下，我们要尽量使用当前屏幕渲染。

**离屏渲染 VS CPU渲染**

由于GPU的浮点运算能力比CPU强，CPU渲染的效率可能不如离屏渲染；但如果仅仅是实现一个简单的效果，直接使用CPU渲染的效率又可能比离屏渲染好，毕竟离屏渲染要涉及到缓冲区创建和上下文切换等耗时操作

UIButton 的 masksToBounds = YES又设置setImage、setBackgroundImage、[button setBackgroundColor:[UIColor colorWithPatternImage:[UIImage imageNamed:@"btn\_selected"]]];

下发生离屏渲染，但是[button setBackgroundColor:[UIColor redColor]];是不会出现离屏渲染的

关于 UIImageView,现在测试发现(现版本: iOS10),在性能的范围之内,给UIImageView设置圆角是不会触发离屏渲染的,但是同时给UIImageView设置背景色则肯定会触发.触发离屏渲染跟 png.jpg格式并无关联

日常我们使用layer的两个属性，实现圆角

imageView.layer.cornerRaidus = CGFloat(10);

imageView.layer.masksToBounds = YES;

这样处理的渲染机制是GPU在当前屏幕缓冲区外新开辟一个渲染缓冲区进行工作，也就是离屏渲染，这会给我们带来额外的性能损耗。如果这样的圆角操作达到一定数量，会触发缓冲区的频繁合并和上下文的的频繁切换，性能的代价会宏观地表现在用户体验上——掉帧

# iOS 性能优化面试题（怎么检测图层混合？）

1、模拟器debug- 选中 color blended layers红色区域表示图层发生了混合

2、Instrument-选中Core Animation-勾选Color Blended Layers

**避免图层混合：**

1、确保控件的opaque属性设置为true，确保backgroundColor和父视图颜色一致且不透明

2、如无特殊需要，不要设置低于1的alpha值

3、确保UIImage没有alpha通道

**UILabel图层混合解决方法：**

iOS8以后设置背景色为非透明色并且设置label.layer.masksToBounds=YES让label只会渲染她的实际size区域，就能解决UILabel的图层混合问题

iOS8 之前只要设置背景色为非透明的就行

为什么设置了背景色但是在iOS8上仍然出现了图层混合呢？

UILabel在iOS8前后的变化，在iOS8以前，UILabel使用的是CALayer作为底图层，而在iOS8开始，UILabel的底图层变成了\_UILabelLayer，绘制文本也有所改变。在背景色的四周多了一圈透明的边，而这一圈透明的边明显超出了图层的矩形区域，设置图层的masksToBounds为YES时，图层将会沿着Bounds进行裁剪 图层混合问题解决了