Swift基础知识

## 1、swift关键字dynamic的含义

加了**@objc标识的方法、属性无法保证都会被运行时调用**，因为**Swift会做静态优化**。要想完全被动态调用，必须使用**dynamic修饰**。使用dynamic修饰将会隐式的加上@objc标识

## 2、swift的消息传递机制

**直接派发 (Direct Dispatch)**

直接派发是最快的, 不止是因为需要调用的指令集会更少, 并且编译器还能够有很大的优化空间, 例如函数内联等,. 直接派发也有人称为静态调用.

**函数表派发 (Table Dispatch )**

函数表派发是编译型语言实现动态行为最常见的实现方式. 函数表使用了一个数组来存储类声明的每一个函数的指针. 大部分语言把这个称为 "virtual table"(虚函数表), Swift 里称为 "witness table". 每一个类都会维护一个函数表, 里面记录着类所有的函数, 如果父类函数被 override 的话, 表里面只会保存被 override 之后的函数. 一个子类新添加的函数, 都会被插入到这个数组的最后. 运行时会根据这一个表去决定实际要被调用的函数.

**消息机制派发 (Message Dispatch )**

消息机制是调用函数最动态的方式. 也是 Cocoa 的基石, 这样的机制催生了 KVO, UIAppearence 和 CoreData 等功能. 这种运作方式的关键在于开发者可以在运行时改变函数的行为. 不止可以通过 swizzling 来改变, 甚至可以用 isa-swizzling 修改对象的继承关系, 可以在面向对象的基础上实现自定义派发.

Final 直接派发 @inline 直接派发

@objc Dynamic 使用消息机制

@nonobjc 不使用消息机制

**值类型总是会使用直接派发, 简单易懂**

**Swift协议和类的 extension 都会使用直接派发**

**NSObject 的 extension 会使用消息机制进行派发**

**NSObject 生命作用域里的函数都会使用函数表进行派发.**

**协议里声明的, 并且带有默认实现的函数会使用函数表进行派发**

## 3、如何理解swift的面向协议编程

1、**Swift协议的特性，可以添加相关默认方法，同时也支持泛型协议**。

2、**更加强调了接口隔离原则**。

3、**协议在Swift中是first-class的，可以认为是一个特殊的对象**

4、**Swift是单继承，可以使用协议进行扩展，**而且协议可以使用&符号来进行聚合，是多继承的一个实现方式

## 4、swift中出现循环引用的情况，并说明怎么解决

用weak或者unowned引用代替其中一个的强引用，来打破循环引用

闭包则使用capture list来进行解决

Unowned不会保持强引用（生命周期相同时使用）

## 5、声明一个静态属性或者函数，我们常常使用值类型的static修饰符，static和class有什么区别？

struct Sun{ static func illuminate(){} }

**对类来说，使用static或者class修饰符，都是可以的。它们使用后的效果是一样的，但是本质上是不同的**。static修饰的属性或者修饰的函数都不可以重写。但是使用class修饰符的，你可以重写属性或者函数。当static在类中应用的时候，**static就成为class final的一个别名**。

## 6、const int number = 0 和 let number = 0两者之间有什么不同吗？如果有，请说明原因。

const常量是一个在**编译时或者编译解析时**被初始化的变量。通过let创建的是一个**运行时常量**，是不可变的

## 7、typealias和associateType作用

associatedType**用来实现泛型协议，**关联类型为协议中的某个类型提供了一个**占位名**（或者说别名），其代表的实际类型在协议**被采纳时才会被指定**。

typealias为**常用数据类型起一个别名**，也可以定义associateType实际定义的类型，实现一个泛型协议。如：typealias SuccessWithMsg = ((\_ msg:String) -> Void)

## 8、对一个optional变量拆包有多少种方法？并在安全方面进行评价。

**Force unwrap** ！操作符——不安全

**Implicite uwrap**——大多数情况下不安全

**Optinal binding**——安全

**optional chaining**——安全

**nil coalescing**运算符(空值合并运算符??)——安全

**guard语句**——安全（也是Optional Binding的一种）

## 9、哪些情况下你不得不使用隐式拆包？说明原因。

对optional变量使用隐式拆包最常见的原因如下：

1、**对象属性在初始化的时候不能nil，否则不能被初始化。**典型的例子是**Interface Builder outlet**类型的属性，它总是在它的拥护者初始化之后再初始化。在这种特定的情况下，假设它在Interface Builder中被正确的配置outlet被使用之前，保证它不为nil。

2、**解决强引用的循环问题——当两个实例对象相互引用，并且对引用的实例对象的值要求不能为nil的时候**。在这种情况下，引用的一方可以标记为unowned, 另一方使用隐式拆包。

**建议：**除非必要，不要对option类型使用隐式拆包。使用不当会增加运行时崩溃的可能性。在某些情况下，崩溃可能是有意的行为，但有更好的方法来达到相同的效果，例如，通过使用fatalError()函数

## 10、Swift如何管理C对象？（unmanaged关键字）

CoreFoundation如果**标识由Swift自动内存管理的API返回的对象可以直接使用**，而不用调用CFRetain、CFRelease

**自己的C函数，可以使用插入相关宏来进行内存管理《内存托管对象》**

**CF\_RETURNS\_RETAINED、CF\_RETURNS\_NOT\_RETAINED**

可以使用 **CF\_IMPLICIT\_BRIDGING\_ENABLED、CF\_IMPLICIT\_BRIDGING\_DISABLED**来包含符合CoreFoundation命名规则的c函数

**进行手动管理内存策略《内存非托管对象》**

在获得对象的方法中没有使用CF\_RETURNS\_RETAINED或CF\_RETURNS\_NOT\_RETAINED注释声明，编译器无法帮助管理内存。内存非托管对象使用起来有些麻烦，要根据获得所有权方法，进行相应的处理。

如果一个函数名中包含Create或Copy，则调用者获得这个对象的同时也获得对象所有权，返回值Unmanaged<T>需要调用**takeRetainedValue()方法获得对象，会进行减一操作，这样就可以消耗掉不平衡的retain而不需要在swift中进行调用CRRealse**。

如果一个函数名中包含Get，则调用者获得这个对象的同时不会获得对象所有权，返回值Unmanaged<T>调用**takeUnretainedValue()方法获得对象，并不进行retainCount的减一操作**。

Unmanaged<Instance>结构体有两个方法防Unmanaged转换为managed对象 —takeUnretainedValue() and takeRetainedValue().也可以在Unmanaged对象调用 retain(), release(),autorelease()

**Unmanaged**

当使用此类型时，用户有部分责任保持对象存活

主要的方法分为四类

**第一类**是创建Unmanaged Reference： passRetained(Instance)、passUnretained(Instance)

**第二类**是进行指针和Unmanaged Reference的转化：fromOpaque(UnsafeRawPointer)、 toOpaque()

**第三类**是对引用进行引用计数进行操作：autorelease()、release()、retain()

**第四类**是获取Unmanaged Reference的值并对该值引用计数操作：takeRetainedValue()、takeUnretainedValue()

## 11、try、try？、try!的区别？

当调用可能抛出异常的方法时候（throws | rethrow）需要使用try开头的方法。

**Try?将一个运行时error转化为optional value**

**Try!则是禁止抛出error**（即认为不会抛出Error），如果此时抛出error，则会有运行时错误。

## 12、assertions和preconditions的区别？

**Assertions只在debug下工作，preconditions在debug和production下都工作**

如果使用**unchecked mode编译模式。Preconditions不会被检查，而被假定一直true**。

## 13、Swift4.0引入了哪些新特性？

**引入Substring类型，Substring可以重用原string的部分内存**，但不适合长期存储，因为如果长期存储Substring，则需要原String一直存在。需要转化为String进行长期存储。

String和Substring都实现了stringprotocol协议

**添加Codable协议**，方便自定义数据类型序列化

**改进key-value coding的keypath**

**String 又变成了集合类型**

**Associated Type 可以追加 Where 约束语句**

**Xcode 9 引入了 New Build System**，预编译 Bridging Headers，编译器会在预编译阶段把 Bridging Headers 编译一次，然后插入到每个 Swift 文件中，这样就大大提高了编译速度。

**减少隐式 @objc 自动推断**

为了优化性能引入了 COW Existential Containers，这里的 COW 就代表 "Copy-On-Write"，当存在多个相同的值时，他们会共用 buffer 上的空间，直到某个值被修改时，这个被修改的值才会被拷贝一份并分配内存空间。

**Codable协议**

## 14、Swift的Codable协议

Codable是个**组合协议**，public **typealias Codable** = Decodable & Encodable

同时与序列化相关的还有一个叫 **CodingKey 的协议**，用来表示编码和解码的 key。

主要方法：

func encode(to encoder: Encoder) throws

init(from decoder: Decoder) throws

如果我们需要用不同的名称, 只需要提供我们自己的 CodingKey：

private enum CodingKeys:String, CodingKey {

case name = "person\_name"

case age

}

## 15、Swift标准库中的常见函数和协议有哪些

**Sequence**

可以迭代地访问其元素，实现此协议可以使用for-in来进行循环访问数据元素，Sequence protocol提供了很多默认操作的实现，如contains(\_:)

1、Sequence协议并不保证其元素是否被无损地迭代访问，也就是说多个for-in循环进行访问一个Sequence时，不能保证从头开始或者从其中一个Sequence循环 break的元素开始。

2、如果需要保证循环访问无损地从头开始，可以实现**Collection**协议。

**Collection** 概念是因为 Sequence 无法确定集合里的位置而在 Sequence 的基础上实现了 **Indexable** 协议。有了 Collection 就可以确定元素的位置，包括开始位置和结束位置，这样就能够确定哪些元素是已经访问过的，从而避免多次访问同一个元素。还能够通过一个给定的位置直接找到那个位置的元素。

**IteratorProtocol**

一个迭代器需要实现的协议，IteratorProtocol与Sequence Protocol很是紧密地绑定在一起的，实现IteratorProtocol需要实现mutating func next()方法。

**AnyIterator** 实现了 **IteratorProtocol** 和 **Sequence** 协议

**Hashable**

最常见的协议了，它提供一个哈希值，可以在set中使用，或者作为Dictionary的key值。很多类型都默认提供了hash值。自定义的类型也可以被hash，当定义个没有associated value的枚举时，该类型会自动进行实现Hashable协议。，if a == b then a.hashValue == b.hashValue.反过来是不正确的。**Hashable协议继承于Equatable**协议，所以要实现Hashable协议除了提供hashvalue外还需要定义的类型添加 == 函数。

==：value相等

===: class对象完全相同

**CustomStringConvertible**

实现此协议来提供一个description属性，这个属性用来将一个实例转化为String。String(describing:)方法是将实例转化为String的首选方法，当实现CustomSStringConvertible时，String(describing:)和print(\_:)将会使用此description属性。

**CustomDebugStringConvertible**

实现此协议来提供一个debugDescription属性，这个属性将一个实例转化为String。如果一个类型实现了CustomDebugStringConvertible协议，String(reflecting:)方法和debugPrint(\_:)函数将使用此debugDescription属性

**CustomReflectable：**

返回自定义的Mirror。

**CustomLeafReflectable**

只返回自己的Mirror，子类的mirror不会体现，除非也复写了customMirror。

StaticString：在编译期就已经知道其内容的String，是不可变的，只能通过指针来进行访问。

## 16、Swift的Switch和OC的有什么不同？

**默认不Fallthrough**

**可以区间匹配**

**可以条件匹配**

**可以进行值绑定（value binding）**

**可以进行组合case（compound case），一个case对应多个匹配**

## 17、Swift如何进行检验API是否可用？

**#available() 运行时方法**

**@avaliable()编译期属性**

## 18、什么是trailing closures？

如果closure是函数最后一个参数，在调用的时候可以将closure参数逻辑写在函数括号的外边

## 19、escaping closure如何使用？

如果将**closure存储在了函数外，超出了该方法的作用域**，则需要使用@escaping关键字，使用@escaping关键字后，需要显式地使用self来进行引用变量和调用方法。

## 20、说一下@autoclosure的作用

自动将一个表达式封装程一个闭包传入，可以将**表达式执行进行延迟操作**

## 21、Swift中Enum的RawValue、AssociatedValue

RawValue是Enum的defaultvalue，所有的RawValue都是同一个类型，初始化时绑定

AssocaiateValue 是与case额外相关联的值，可以是不同类型

## 22、Swift中Class和Struct的异同？

**相同点**：

都可以定义store property、compute property、subscripts、方法、extensions

**不同点**：

**Class是引用类型，struct为值类型，Class可以有多个引用**

**Class可以被继承，struct不可以**

## 23、属性可以被声明为lazy常量吗？lazy property是线程安全的吗？全局常量和变量是lazy吗?

**不可以，因为初始化完成前，常量必须要进行初始化**

Lazy Property不是线程安全的

Global常量和变量都是lazy的，不必要使用lazy来进行标识

**单例中，静态常量是线程安全的，静态变量并不是线程安全的**

**另一种应用就是惰性求值**

**为了支持惰性求值，在 collection 通过 lazy 属性可以获取到一个惰性的集合。**

// a huge collection

let giant = 0..< Int.max

// lazily map it: no work is done yet

let mapped = giant.lazy.map { $0 \* 2 }

// sum the first few elements

let sum = mapped.prefix(10).reduce(0, +)

// sum == 90

**上面的代码中，mapped 赋值时计算并没有发生，因为 lazy 返回的 LazyCollecion 上的 map 操作是惰性的**

## 24、Swift的initializer和OC的有什么区别？

**不返回初始化的对象，如果是可失败的initialer，可以根据返回的optional来判定是否成功**

**不会默认继承父类的初始化方法**。（**当没有designated initializer，会自动继承父类designated initalizer，当通过自动继承或者全部实现父类的designated initializer，同时也会自动继承父类所有convenience initalizer**）

## 25、如何编写自定义初始化方法，同时可以使用默认初始化或者memberwise初始化方法？

将自定义方法写在exntentions中

## 26、Swift初始化方法的代理规则是什么？

Designated initializer delegate up， Convenience initializer delegate across

## 27、Swift 类初始化的两阶段处理，和OC有什么不同？

为了保证属性在访问前都已经初始化，OC会对所有属性进行赋值0或者nil，Swift可以更灵活地处理0或者nil不是有效默认值的类型

**第一阶段**：

Designated initializer方法调用，类实例分配内存，但内存还没有被初始化，此时应保证所有stored properties 都有值，此时**stored properties的内存被初始化**。

同时向上初始化父类的stored properties，当所有stored perperties都被初始化，此时**实例类存就被完全初始化**。

**第二阶段**：

从**继承链自上而下自定义相关属性**

## 28、在Swift 类初始化过程中，编译器的四步检查？

1、一个designated initializer必须要在调用superclass的初始化方法前保证类的所有属性都已经初始化。

2、**在对继承的属性进行赋值前，designated initializer必须要先调用superclass的初始化方法，防止赋值后被父类重写**。

3、一个convenience initializer必须要先调用其他初始化方法，然后才能进行对属性赋值。如果不这样，一些convenience initializer可能覆盖了这些属性值。

4、**在第一阶段初始化完成前不能使用属性和self**。

## 29、Swift中的Any和AnyObject

Any可以代表**所有类型，包括函数、optional types，AnyObject可以代表所有Class类型的实例**

**一个协议通过继承AnyObject来约束协议只能被类实现（以前是继承Class）**

## 30、Swift访问控制

open、public、internal、fileprivate、private

open可以被其他module进行继承，public不可以

内部权限不能被外部大，如private不能有public属性

类可以比被实现的protocol权限大

## 31、Swift如何重载操作符函数和自定义操作符

**重载操作符**使用static (prefix、infix、postfix) func 直接定义

**自定义操作符**需要先进行声明，使用operator关键字，同时使用prefix，infix、postfix修饰符，如prefix operator +++；然后使用static (prefix、infix、postfix) func 来进行定义

## 32、Swift中有哪些指针？

const Type \* UnsafePointer<Type>

Type \* UnsafeMutablePointer<Type>

Type \* const \* UnsafePointer<Type>

Type \* \_\_strong \* UnsafeMutablePointer<Type>

Type \*\* AutoreleasingUnsafeMutablePointer<Type>

const void \* UnsafeRawPointer

void \* UnsafeMutableRawPointer

**Null pointers的对应关系是：**

**const Type \* \_Nonnull UnsafePointer<Type>**

**const Type \* \_Nullable UnsafePointer<Type>?**

**const Type \* \_Null\_unspecified UnsafePointer<Type>!**

## 33、Swift的一些条件编译指令

os() macOS, iOS, watchOS, tvOS, Linux

arch() i386, x86\_64, arm, arm64

swift() >= followed by a version number

**canImport()**  模块名

**targetEnvironment()** simulator

## 34、swift和OC如何混编？

**在工程中：**

**Swift使用OC**：在Bridging header引入OC头文件

**OC使用Swift**： 在OC中引入Swift generated 文件（**moduleName-Swift.h**）

**在Framework中：**

**Swift使用OC**：在Bridging header引入OC头文件

**OC使用Swift**： 在OC中引入Swift generated 文件（moduleName-Swift.h），引入时前面加framework module名字

其他相关宏

**NS\_REFINED\_FOR\_SWIFT**： 保持原函数的可用性，同时新生成一个新的函数别名

初始化方法、subscripting导入为以\_\_双下划线为开头的函数

**NS\_SWIFT\_NAME**： 指明swift函数名

**NS\_SWIFT\_UNAVAILABLE**： Swfit不可用

## 35、Swift编译优化等级？

1、-Onone

适用于开发阶段，编译器进行了最低限度的优化并保留了debug信息。

2、-O

这适合于绝大多数的线上环境代码，编译器进行了单文件编译优化，使得XCode可以并行编译多个文件，但是并行编译使得编译器的一些特定优化无法进行。

3、-O -whole-module-optimization

将整个工程作为一个文件进行编译，编译时间更长，但优化后运行的性能更高。

**新的编译系统：**

No optimization(-Onone)

Optimize for Speed (-O)

Optimize for Size (-Osize)

**-Osize会降低效率**：官网5%

## 36、Swift的枚举与OC的不同？

**不会自动添加interger 值，只有在Int类型下才会自动赋值。**

**有AssociatedValue**

**可以定义computed 属性和方法**

**可以有嵌套。Indirect**

## 37、Swift的异常捕获？

swift通常都是通过对应的signal来捕获crash

//对于OC的exception采取如下方式捕获

**NSSetUncaughtExceptionHandler(UncaughtExceptionHandler)**

//对于Swift则捕获相关signal，一般来说如下几种已经能够捕获大部分crash。(**其中SIGTRAP一定要捕获, swift大量的crash都会通过它**）

signal(SIGABRT, SignalExceptionHandler)

signal(SIGSEGV, SignalExceptionHandler)

signal(SIGBUS, SignalExceptionHandler)

signal(SIGTRAP, SignalExceptionHandler)

signal(SIGILL, SignalExceptionHandler)

**捕获信号并处理后，要恢复对信号的默认处理**

## 38、Swift为什么不使用print而使用os\_log?

原来都存在硬盘的/etc/system.log， 同意的logging 系统将信息存储在内存和数据中心，而不是写成文件。

Swift使用os\_log函数统一了log系统。

## 39、Swift如何使用可变参数函数？

使用**getVaList**函数和**withVaList**函数。

**getVaList**使用CVarArg作为参数并返回一个CVaListPointer值

**withVaList**将**CValistPointer**提供在闭包里而不是直接返回。

调用 withVaList 来获取 CVaListPointer ，这个指针指向你的参数列表( getVaList 也可以用但是文档推荐尽量不使用它)。让我们看看一个使用 vprintf 的例子：

withVaList(["a", "b", "c"]) { ptr -> Void in

vprintf("Three strings: %s, %s, %s\n", ptr)

}

## **40、@convention(c)关键字是干什么的**？

C函数指针在swift中被导入为Closure，为了表示这个closure的调用使用**c函数的方式则使用此关键字**。

**int (\*)(void) => @convention(c) () -> Int32.**

**指针函数**：本质时函数

float \*fun();

float \*p;

p = fun(a)

**函数指针**：是指向函数的指针变量，即本质是一个指针变量。

void (\*fptr)();

把函数的地址赋值给函数指针，可以采用下面两种形式：

fptr=&Function;

fptr=Function;

两种方式来通过指针调用函数：

x=(\*fptr)();

x=fptr();

## 41、什么是ArraySlice?

类似于Substring和String的关系，不可以用来长时间存储，因为它依赖于产生ArraySlice的Array、ContiguousArray或者另一个ArraySlice。它可以有效提供操作Array的效率，它与产生ArraySlice的Array、ContiguousArray共享存储的元素Contiguous

## 42、什么是Array？

**Array在存储的时候，如果是Class或者@objc协议，则存储为NSArray，其他的则存储在一块连续的内存，用来提供访问和操作效率。ContiguousArray是不管什么类型都存储在一块连续内存的容器，如果存储Class或者@objc协议，则比Array效率高，如果存储struct等，则和Array效率相同。**

## 43、Optional的实现?

enum Optional<T>: \_Reflectable, NilLiteralConvertible {

case None

case Some(T)

…其他函数

}

## 44、Swift实现一个泛型栈？

protocol Stackable{

//声明一个关联类型，使用typealias关键字

associateType ItemType

mutating func push(\_ item: ItemType)

mutating func pop() -> ItemType

}

struct Stack<T>: Stackable {

typealias ItemType = T;

var store = [T]()

mutating func push(item: T){//实现协议的push方法要求

store.append(item)

}

mutating func pop() -> T {//实现协议的pop方法要求

return store.removeLast()

}

}

## 45、什么是泛型协议？

支持泛型的协议，

使用associateType来进行占位，使用typealias来进行确定实际类型参数。

Swift4.0 associateType后可以跟where限定，泛型协议的where放在类、protocol的最后，放在方法返回值后面。

## 46、Swift中function是first-class， first-class function指的什么？

**语言中把类型分为三类**：

**First Class**。该类型的值可以作为函数的参数和返回值，也可以赋给变量。

**Second Class**。该类型的值可以作为函数的参数，但不能从函数返回，也不能赋给变量。

**Third Class**。该类型的值作为函数参数也不行。

所以说Swift中的function可以作为参数、返回值、和变量值。

## 47、如何支持Swift的条件编译？

1、以前，在Build Settings 中，找到 **Swift Compiler - Custom Flags，在 Other Swift Flags** 加条件变量。

2、目前，Swift 中没有宏的概念，但提供了 **Active Compilation Conditions** ,这个设置可替代预编译宏的方式。

## 48、Swift中的泛型擦除是什么？

**编译检查泛型约束、编译后无泛型**

**在使用关联类型的时候需要注意当声明一个使用了关联属性的协议作为属性时**，因为该协议使用了关联类型，则无法通过编译，会提示‘**can only be used as a generic constraint because it has Self or associated type requirements’**，

需要通过类型擦除来解决，主要思路就是加个中间层在代码中让这个抽象的类型具体化。实际上在 Swift 的标准库里就有类型擦除很好的运用，比如 **AnySequence 的协议**。

**利用一个具体实现的通用泛型类（参看系统库的AnySequence），去包装具体实现了该泛型协议的类**。用以**解决不能直接使用泛型协议进行变量定义的问题**。

## 49、如何使用C第三方库？

创建module.map, 一般是创建一个shim头文件，shim头文件引入C头文件，然后再module.map进行引入shim文件。

module {module名字} [system] [extern\_c] {

header ‘头文件’

link ‘link名字’

export \*

}

在**SWIFT\_INCLUDE\_PATHS**进行设置**module.map路径。**

## 50、函数式编程解决什么问题？

**函数式编程强调没有"副作用"**，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值

在函数式编程中，由于数据全部都是不可变的，所以**没有并发编程的问题，是多线程安全的**

函数式编程的表达方式更加符合人类日常生活中的语法，代码可读性更强

由于所有的**数据都是不可变的**，所以所有的变量在程序运行期间都是一直存在的，非常占用运行资源

面向对象程序设计可以看作一种在程序中包含各种独立而又互相调用的对象的思想.

**函数式编程关心数据的映射**

## 51、Swift和OC有什么不同？和Java相比有什么不同？

Swift提供了tuple、Optional等现代语言细节

常量、变量进行区分，便于**静态调用优化**

**静态调用优化**

Swift更加安全，它是类型安全的语言。

Swift代码更少，简洁的语法，可以省去大量冗余代码

Swift速度更快，运算性能更高。

## 52、Swift如何进行性能优化？

1、编译优化：把每个文件的编译时间写入文件，并按照编译耗时降序排序

xcodebuild -workspace XXX.xcworkspace -scheme XXX clean build OTHER\_SWIFT\_FLAGS="-Xfrontend -debug-time-function-bodies" | grep .[0-9]ms | grep -v ^0.[0-9]ms | sort -nr > culprits.txt

**减少动态派发（Dynamic Dispatch）**

建议：使用 final，如果代码没有复写的可能

建议：使用 private 和 fileprivate ，如果没有被外部访问的可能

**高效使用容器类型**

容器类型：Array、Dictionary、Set 等等

建议：在数组中使用值类型

建议：使用 ContiguousArray 存放引用类型，如果不需要 NSArray 的 bridging

建议： 尽量同一个容器对象做操作，而不是新生成容器对象操作完后再赋值。因为会触发 copy on write

**未检查操作（Unchecked operations）**

建议：当你知道不会发生溢出的时，使用未经检查的整数运算

**泛型**

建议：把泛型声明和使用的代码放在同一个 module 里。下面的 Whole-module optimization 内有说明原因

建议：使用 @\_specialize 指引编译器知道具体的泛型类型

大型值类型的消耗

建议：使用 copy-on-write 的语意。有时候需要做一些封装

**Unsafe code**

**建议：**有时候可以手动管理引用计数

**协议**

**建议：**如果协议只会用在类对象中，那么声明协议继承 AnyObject

protocol Pingable : class { func ping() -> Int }

Whole-module optimization

Swift 3 的编译器新提供的优化选项。可以在 Build Setting 中开启

在编译项目中不同模块的文件时，编译器是每个文件独立编译。每次编译一个文件，编译器都要读取并解析源文件（已经做一下类型检查等等的事情），然后编译器对 Swift 代码进行优化，生成机器码并写入对象文件。最后链接所有对象文件并生成分享库或可执行文件。

单个文件的编译方式，编译器只能在文件范围内做编译优化，如果遇到跨函数优化，比如函数内联或者泛型类型的具体化问题，编译器需要做很多事情去确定最终的编译方式。

**开启 Whole-module optimization**

开启 Whole-module optimization 编译优化项后，编译器会直接把整个模块的文件一起做编译优化。**两个好处**：

1、编译器可以知道整个模块中的函数实现，所以跨函数调用编译有不确定因素的问题就解决了。

2、编译器可以知道所有只有模块内可以使用的函数，编译器会根据函数的使用情况，重新生成优化后的函数。

编译时间对比

编译器的编译流程：解析、类型检查、SIL（Swift Intermediate Language）优化、LLVM backend

**SIL 优化这一步处理所有重要的 Swift 优化工作。**

**开启 Whole-module optimization 编译优化项**后，只在 SIL 这步把整个模块的文件一起做编译，其它步骤还是会通过并行的方式去编译。

## 53、Swift自省的方法有哪些？

is as as? As!

OC与C基础知识

## 1、为什么在 Objective-C 中给 nil 发送消息程序不会崩溃？

teq a1, #0

moveq a2, #0

bxeq lr

对应 objc\_msgSend，第一个参数为 self，返回值也是 self，都放在 r0

moveq 指令说明：如果self为空，则将 selector 也设置为空。

bx 指令说明：

在 arm 中 bx lr 用来返回到调用子程序的地方（即：返回到调用者），此处是：**如果 self 为空，就返回到调用 objc\_msgSend 的地方继续执行。**

**总之：**如果传递给 objc\_msgSend 的 self 参数是 nil，该函数不会执行有意义的操作，直接返回

## 2、什么是KVC和KVO？

**KVC**: 键值编码（key-value-coding)，一个非正式的Protocol,提供一种机制来间接访问对象的属性。而不是通过调用Setter、Getter方法访问。

**KVO**：键值观察者（key-value-observing）。KVO提供了一种观察者的机制，通过对某个对象的某个属性添加观察者，当该属性改变，就会调用"**observeValueForKeyPath:"**方法。

**KVC按顺序使用如下技术**:

检查是否存在getter方法-<key>或者setter方法-set<key>:的方法;

如果没有上述方法,则检查是否存在名字为-\_<key>、<key>的实例变量;

如果仍未找到,则调用 **valueForUndefinedKey:** 和 **setValue: forUndefinedKey:** 方法。这些方法的默认实现都是抛出异常,我们可以根据需要重写它们。

重写**setNilValueForKey**等方法

Setdic 模型序列化的时候 用到之外 还有 通过valueForKey给容器中的每一个对象发送操作消息 @sum @avg

实现countOfVal，或者objectInValAtIndex:与valAtIndexes其中之一，这个组合会使KVC返回一个代理数NSKeyValueArray。一般需要KVO监听 数组insert remove 相关 原理就是搜索 insert remove <key> 相关的方法， 如果找到就返回代理合集 找不到就搜索set<key>方法 将mutableArrayValueForKey 返回的代理合集 调用setkey 重新赋值给对象 触发KVO

@distinctUnionOfObjects 返回一个由操作符右边的key path所指定的对象属性组成的数组，不对数组去重

@unionOfObjects 返回一个由操作符右边的key path所指定的对象属性组成的数组，并对数组去重

## 3、load和initialize的异同

**执行时间**：

load在类被加载时候调用，所有category的load都会被调用，此时framework都已经加载

当向一个类第一次发送消息时候，initialize方法会被调用（通过 runtime 的消息机制，objc\_sendMsg 方法），lazy run

**继承链、类、category的调用顺序**：

Load先调用父类的load，保证只调用一次，Load先调用类的load，然后调用 category的load

Initializer也是调用父类，但当子类没有实现时，可能调用父类两次，所以要加self的class判断，**分类中的initializer覆盖类中的方法，只执行分类的实现**

## 4、OC Class的载入过程？（从Load和Initializer来进行分析）

1、当程序启动时，就会加载项目中所有的类和分类。而且加载后会调用每个类的load犯法和分类的＋load方法。只会调用一次

2、先加载父类，再加载子类（先调用父类的＋load，在调用子类的＋load方法）

3、先初始化父类，再初始化子类（先调用父类的＋initialize方法，再调用子类的＋initialize方法），当第一次使用某个类时，就会调用当前类的＋initialize方法

## 5、OC集合遍历方法？

1、for循环 2、NSEnumerator（NSEnumerator提供的nextObject方法） 3、for in 快速遍历 4、基于block的遍历：enumerateObjectsUsingBlock、enumerateKeysAndObjectsUsingBlock

## 6、如何在struct中使用OC对象？

如果想要把一个OC object放到一个C的struct当中去，需要将这个OC object转换为void \*

## 7、对于Objective-C，你认为它最大的优点和最大的不足是什么？对于不足之处，如何绕过这些不足实现需求？

**OC优点：** 1) Cateogies 2) 动态绑定和运行时特性

**OC缺点 :** 1) 不支援命名空间 2) 不支持运算符重载 3）不支持多重继承 4）使用动态运行时类型，所有的方法都是函数调用，所以很多编译时优化方法都用不到。（如内联函数等），性能低劣。不足是没有命名空间，对于命名冲突，可以使用长命名法或特殊前缀解决

如果是引入的第三方库之间的命名冲突，可以使用link命令及flag解决冲突。

## 8、category的作用？优缺点有哪些？

**1、给已经存在的类添加方法。**

**2、把类的实现放在不同的文件里面**。

（1）可以减少单个文件的体积。

（2）可以把不同的功能组合到不同的category里

（3）可以按需要加载想要的category

**3、声明私有方法**

4、**模拟多继承**

5、把framework的私有方法公开

**优点**：

**不需要通过增加子类而增加现有类的行为(方法),且类目中的方法与原始类方法基本没区别;**

**通过类目可以将庞大一个类的方法进行划分,从而便于代码的日后的维护、更新以及提高代码的阅读性;**

**缺点**：**加载慢、可能覆盖，不易追查**

无法向类目添加实例变量,如果需要添加实例变量,只能通过定义子类的方式;

类目中的方法与原始类以及父类方法相比具有更高优先级,如果覆盖父类的方法,可能导致super消息的断裂。因此,最好不要覆盖原始类中的方法

## 9、id和instancetype的区别？

**instancetype == id == 万能指针 == 指向一个对象**

1、**id在编译的时候不能判断对象的真实类型，instancetype在编译时可判断对象的真实类型**(一个在编译时不知道真实类型, 一个在编译时知道真实类型)

2、**id可以用来定义变量, 可以作为返回值, 可以作为形参，instancetype只能用于作为返回值**,它会进行类型检查,如果创建出来的对象,赋值了不相干的对象就会有一个警告信息,防止出错

## 10、id存在的意义是什么？

id是一个数据类型, 并且是一个动态数据类型，动态数据类型的特点:

1、在编译的时候编译器并不知道变量的真实类型, **只有在运行的时候才知道它的真实类型**

2、如果通过动态数据类型定义变量, 调用属性和方法时只有在运行时才能知道是否能够响应, 编译器不会报错

这样**就可以实现动态绑定**

## 11、OC自省相关的方法有哪些？

判断对象类型:

-(BOOL) **isKindOfClass:** 判断是否是这个类或者这个类的子类的实例

-(BOOL) **isMemberOfClass:** 只能判断是否是这个类的实例

判断对象or类是否有这个方法

-(BOOL) **respondsToSelector:** 判断**实例是否有这样方法**

+(BOOL) **instancesRespondToSelector:** **判断类（产生实例）是否有这个方法**

## 12、Nullability中\_Nonnull、\_Nullable 和 nonnull、nullable、nullunspecific的区别？为什么要加这个标识？

1、在 Swift 中，我们会使用 ? 和 ! 去显式声明一个对象或者方法的参数是optional 还是 non-optional ，而在 Objective-C 中则没有这一区分，这样就会带来一个问题：在 Swift 与Objective-C 混编时，Swift 编译器并不知道一个 Objective-C 对象或者一个方法的参数到底是 **optional** 还是 **non-optional**

2、**通过知晓是否为NULL可以做编译器的优化**

**null\_resettable：** setter方法可以为空,getter方法不能为空

**\_Null\_unspecified：** 被swift导入为implicit optinal

**使用规范：**

1、对于OC**属性、方法返回值、方法参数的修饰**，使用： nonnull/nullable ；

2、对于 **C 函数的参数、Block 的参数、Block 返回值的修饰**，使用： \_Nonnull/\_Nullable

另外还有**NS\_ASSUME\_NONNULL\_BEGIN**、**NS\_ASSUME\_NONNULL\_END** 宏块标识

## 13、\_\_kindof 关键字

用\_\_kind修饰的变量表示是某个类或者这个类的子类，在方法的返回值中加\_\_kindof修饰,返回值类型以及它的子类都可以认为是此类。

**也可以结合OC轻泛型使用**：@property (nonatomic, strong) NSArray<\_\_kindof Car \*> \*cars;

## 14、谈一下OC的轻泛型？

使用容器类时限制存储元素类型，存储的元素是声明时的类型，不再是id

可以使用**协变性修饰符、逆变性修饰符**来自定义泛型

// \_\_covariant 协变性修饰 \_\_covariant ObjectTape为泛型名

@interface Car<\_\_covariant ObjectTape > : NSObject

## 15、\_\_covariant、\_\_contravariant修饰符的意义？如何使用？

**不指定泛型类型的 Stack 可以和任意泛型类型转化**

**但指定了泛型类型后，两个不同类型间是不可以强转的**

如果希望主动控制转化关系，就需要使用泛型的协变性\_\_covariant和逆变性\_\_contravariant修饰符：

**\_\_covariant : 协变性**，子类型可以强转到父类型（里氏替换原则） Stack<NSMutableString>就可以转换为Stack<NSString>

**\_\_contravariant: 逆变性**，父类型可以强转到子类型 Stack<NS bString>就可以转换为Stack<NSMutableString>

## 16、NS\_ENUM和NS\_OPTIONS做了哪些工作？

**NS\_ENUM枚举项的值为NSInteger，NS\_OPTIONS枚举项的值为NSUInteger；**

NS\_ENUM定义通用枚举，NS\_OPTIONS定义位移枚举.

根据是否要将代码按C++模式编译，NS\_OPTIONS宏的定义方式也有所不同。

**如果不按C++编译**，**那么其展开方式就和NS\_ENUM相同。**

**若按C++编译**，则展开后 的代码略有不同。原因在于，用按位或运算来操作两个枚举值时，C++编译模式的处理办法与非C++模式不一样。在用或运算操作两个枚举值时，**C++认为运算结果的数据类型应该是枚举的底层数据类型，也就是NSUInteger，且C++不允许它隐式转换为枚举类型本身**，**所以C++模式下定义了NS\_OPTIONS宏以保证枚举结果的类型正确**

## 17、OC中designated Initializer和convenience initializer是什么？

**designated initializer :** 负责调用superclass的初始化方法以及初始化自己的实例变量的初始化方法, 使用宏NS\_DESIGNATED\_INITIALIZER

**convenience initializers :** 非designated initializer都被称为**convenience** initializer。这些initializer内部实现一般都是调用另外一个initializer，然而最终一系列链式调用之后，最终都会调用某一个designated initializer 方法来进行初始化行为。

## 18、常用的OC宏标识有哪些？

NS\_UNAVALABLE 不可用

NS\_REQUIRES\_NIL\_TERMINATION 可变参数，以nil结尾

NS\_AVAILABLE

NS\_ENUM\_AVAILABLE\_IOS、

\_\_IPHONE\_OS\_VERSION\_MIN\_REQUIRED

\_\_IPHONE\_OS\_VERSION\_MAX\_ALLOWED

\_\_IPHONE\_8\_0

\_\_IPHONE\_7\_0

FOUNDATION\_EXTERN = extern

**NS\_REQUIRES\_SUPER**：用来修饰所有方法，表示子类override父类的方法时，必须在方法内调用super的方法

## 19、responseToSelector和instanceResponseToSelector的区别？

一个是实例方法，一个是类方法，类方法用来判定类的实例是否可以响应某个方法

## 20、NSHashTable和NSSet的区别？

NSHashTable效仿了NSSet(NSMutableSet)，但提供了比NSSet更多的操作选项，**尤其是在对弱引用关系的支持上，NSHashTable在对象/内存处理时更加的灵活。**

1.NSHashTable**是可变的，它没有不可变版本**。

2.它可以通过**NSPointerFunctionsOptional**持有元素的**弱引用**，而且**在对象被销毁后能正确地将其移除**。而这一点在NSSet是做不到的。

3.它的成员可以使用**NSPointerFunctionsCopyIn** 选项标志添加时被拷贝。

4.它的**成员可以使用指针来标识是否相等及做hash检测。**

5.**它可以包含任意指针，其成员没有限制为对象**。我们**可以配置一个NSHashTable实例来操作任意的指针，而不仅仅是对象。**

场景:某工具类需要持有多个代理对象，方便后续逐一回调。比如某个订阅器订阅了某个通知，然后通知到来时需要下发给每一个需要响应的页面，这些页面肯定是要实现订阅器的代理方法的。所以，遇到这种场景时，我们可能要注意了。不能使用常用数据类型来管理多个代理者了(因为代理者不能被强引用，会有循环引用问题),此时我们可以采用NSHashTable的弱引用特性。

## 21、NSMapTable和NSDictionary的区别？

NSMapTable是更广泛意义上的NSDictionary。和NSDictionary/NSMutableDictionary相比具有如下特性：

NSMapTable对象类似与NSDictionary的数据结构，但是NSMapTable功能比NSDictionary对象要多的功能就是**可以设置key和value的****NSPointerFunctionsOptions特性!**其他的用法与NSDictionary相同

**NSPointerFunctionsOptions**来初始化对象的持有方式,或者,用系统提供的相应的类方法。

+ (NSMapTable<KeyType, ObjectType> \*)strongToStrongObjectsMapTable NS\_AVAILABLE(10\_8, 6\_0);

+ (NSMapTable<KeyType, ObjectType> \*)weakToStrongObjectsMapTable NS\_AVAILABLE(10\_8, 6\_0); // entries are not necessarily purged right away when the weak key is reclaimed

+ (NSMapTable<KeyType, ObjectType> \*)strongToWeakObjectsMapTable NS\_AVAILABLE(10\_8, 6\_0);

+ (NSMapTable<KeyType, ObjectType> \*)weakToWeakObjectsMapTable NS\_AVAILABLE(10\_8, 6\_0);

1.**NSDictionary/NSMutableDictionary会复制keys并且通过强引用values来实现存储**；

**2.NSMapTable是可变的，没有不可变版本；**

3**.NSMapTable可以通过弱引用来持有keys和values**，所以**当key或者value被deallocated的时候，所存储的实体也会被移除；**

4.**NSMapTable可以在添加value的时候对value进行复制**；

## 22、NSPointerArray和NSArray的区别？

NSPointerArray初始化方法，可以**创建强引用，弱引用对象**的数组

+ (NSPointerArray \*)**strongObjectsPointerArray** NS\_AVAILABLE(10\_8, 6\_0);

+ (NSPointerArray \*)**weakObjectsPointerArray** NS\_AVAILABLE(10\_8, 6\_0);

## 23、static的作用？

作用于变量：

**用static声明局部变量时**，则改变变量的存储方式（生命期），使**变量成为静态的局部变量，即编译时就为变量分配内存，直到程序退出才释放存储单元。**这样，使得该局部变量有记忆功能，可以记忆上次的数据，**不过由于仍是局部变量，因而只能在代码块内部使用（作用域不变）。**

**用static声明外部变量**，**编译时分配内存，程序结束时释放内存单元**。同时 其作用域很广，整个文件都有效甚至别的文件也能引用它

**使用static用于函数定义时**，对函数的连接方式产生影响，使得函数只在本文件内部有效，对其他文件是不可见的。这样的函数又叫作静态函数。使用静态函数的好处是，不用担心与其他文件的同名函数产生干扰，另外也是对**函数本身的一种保护机制**

**如果想要其他文件可以引用本地函数，则要在函数定义时使用关键字****extern，表示该函数是外部函数**

## 24、extern的作用

用来获取全局变量(包括全局静态变量)的值

**extern工作原理:先在当前文件查找有没有全局变量，没有找到，才会去其他文件查找。**

## 25、new、delete、malloc、free关系？

（1）他们都是在堆(heap)上进行动态的内存操作。

（2）new、delete用于C++中分配和释放内存，需要成对出现，注意他们**只是操作符，不是函数，调用系统的构造函数和析构函数，new分配内存并初始化。**

（3）malloc、free是C中用于分配内存和释放内存，**也需要成对出现，并且malloc函数需要指定内存分配的字节数并且不能初始化对象**，他们是是库函数。

## 26、int a；int \*a；int \*\*a； int a[10]； int \*a[10]；int (\*a)[10]； int (\*a)(int)；int (\*a[10])(int); 的区别？

a) int a; 一个整型

b) int \*a; 一个指向整型数的指针

c) int \*\*a; 一个指向指针的的指针，它指向的指针是指向一个整型数

d) int a[10] 一个有10个整型数的数组

e) int \*a[10]; 一个有10个指针的数组，该指针是指向一个整型数的

f) int (\*a)[10]; 一个指向有10个整型数数组的指针

g)  **int (\*a)(int);**  一个指向**函数的指针**，该函数有一个整型参数并返回一个整型数

i) int (\*a[10])(int); **一个有10个指针的数组，该指针指向一个函数，该函数有一个整型参数并返回一个整型数**

## 27、strlen 与 sizeof 的区别

**1、****sizeof是运算符，****strlen是库函数。**

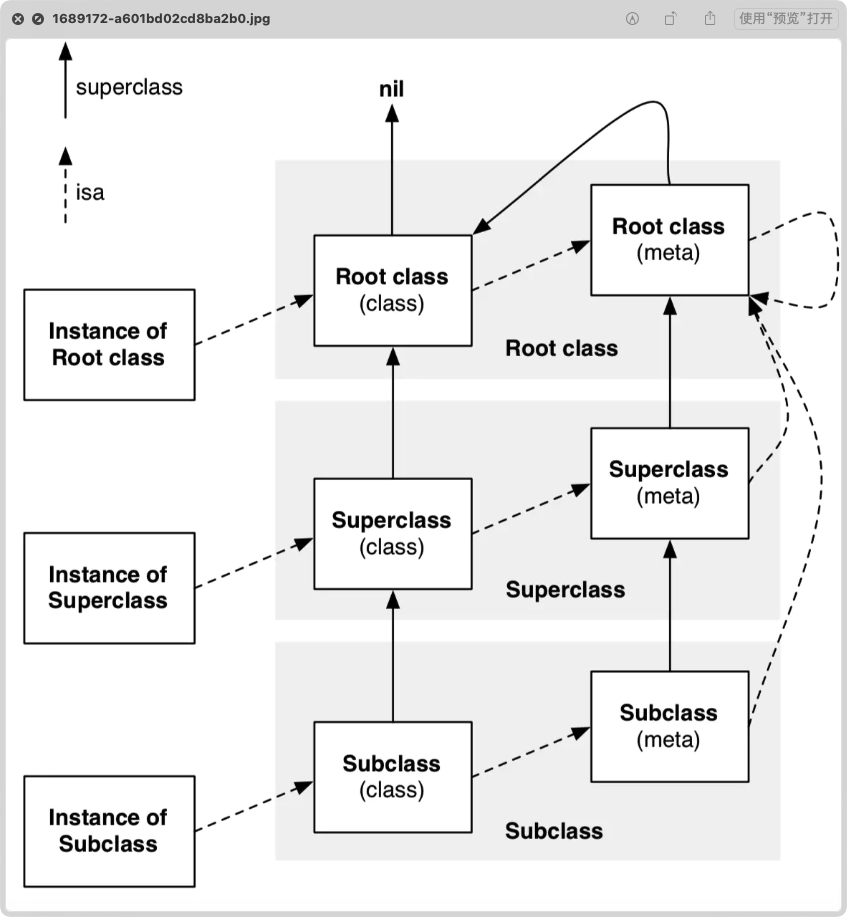
**2、sizeof可以用类型、变量做参数，而****strlen只能用 char\* 变量做参数，且必须以\0结尾。**

**3、sizeof是在编译的时候计算类型或变量所占内存的大小，而****strlen的结果要在运行的时候才能计算出来，用来计算字符串的长度。**

**4、数组做sizeof的参数不退化，传递给strlen就退化为指针了**

## 28、为什么要有元类？将类方法存在元类的好处?

OC对象都有一个isa指针，发送消息时会在isa的对象进行寻找方法响应，类方法则在类的isa指针中进行寻找，即元类，最终的根元类的superclass指针指向NSObject isa指向自己形成闭环，使得OC一切皆对象。



## 29、c++复制构造函数的形参为什么要是const引用？

**为什么要是引用**

CBox(CBox initB);//复制构造函数一开始想到的原型

CBox cigar;

CBox myBox(cigar); //如果编写这样一条语句

//那么将生成这样一条对复制构造函数的调用CBox::CBox(cigar);

因为实参是通过按值传递机制传递的。在可以传递对象cigar之前，编译器需要安排创建该对象的副本。因此，编译器为了处理复制构造函数的这条调用语句，需要调用复制构造函数来创建实参的副本。但是，由于是按值传递，第二次调用同样需要创建实参的副本，因此还得调用复制构造函数，就这样持续不休。最终得到的是对复制构造函数的无穷调用。（其实就是创建副本也是需要调用复制构造函数的）

**为什么要用const**

其实，这里，如果不去改变实参的值的话，不加const的效果和加const的效果是一样的，而且不加const编译器也不会报错，因为函数的形参是引用，则调用函数时不需要复制实参，函数是直接访问调用函数中的实参变量的。但是为了整个程序的安全，还是加上const，防止对实参的意外修改

## 30、const跟在函数后的用法（C++中）

常成员函数。常成员函数可以理解为是一个“只读”函数，它既不能更改数据成员的值，也不能调用那些能引起数据成员值变化的成员函数，只能调用const成员函数

## 31、如何正确停止timer？

**正确地调用invalidate方法**：该方法用来停止timer并且请求将timer从runloop中移除对timer的强引用

必须再将timer添加到runloop的线程中进行调用，如果不是，则可能无法正确移除timer，线程也可能不能正确地退出

## 32、属性修饰符有哪些?

1.原子性：nonatomic 、atomic

2.读写： readwrite 、readonly

3.方法名：getter=<name>、setter=<name>

4.内存：strong、retain、copy、weak、assign 、unsafe\_unretained

## 33、layoutSubView何时调用？

1、 初始化方法事不会调用

2、滚动UIScrollview触发

3、旋转屏幕时触发

4、改变View的值时候触发，前提是frame改变了

5、改变UIView的大小时触发

## 34、define和const定义的常量的区别？

define定义宏的指令，程序在预处理阶段将用#define所定义的内容只是进行了替换。因此程序运行时，常量表中并没有用#define所定义的宏，系统并不为它分配内存，而且在编译时不会检查数据类型，出错的概率要大一些。

const定义的常量，在程序运行时是存放在常量表中，系统会为它分配内存，而且在编译时会进行类型检查。

# OC内存管理

## 1、\_\_bridge、\_\_bridge\_retained、\_\_bridge\_transfer

**\_\_bridge只做类型转换**，但是不修改对象（内存）管理权；如果Core Foundation对象被释放，Foundation对象也同时不能使用了，需要手动管理Core Foundation对象。

**\_****\_bridge\_retained（也可以使用CFBridgingRetain）将Objective-C的对象转换为Core Foundation的对象**，同时将对象（内存）的管理权交给我们，**后续需要使用CFRelease或者相关方法来释放对象；**

**\_\_****bridge\_transfer（也可以使用CFBridgingRelease）将Core Foundation的对象转换为Objective-C的对象**，同时将对象（内存）的管理权交给ARC。系统会自动管理2个对象

## 2、OC内存相关的属性有哪些？区别？

assign、retain、copy、unsafe\_unretained、strong、weak

**assign:**用于对**基本数据类型进行赋值操作，不更改引用计数, 被assign修饰的对象在释放后，指针的地址还是存在的，也就是说指针并没有被置为nil，成为野指针.**

**weak：**修饰Object类型，**修饰的对象在释放后，指针地址会被置为nil，**是一种弱引用。在ARC环境下，为避免循环引用，往往会把delegate属性用weak修饰；在MRC下使用assign修饰.

**copy：**会在内存里拷贝一份对象, 在MRC下，block在创建的时候，它的内存是分配在栈(stack)上的，而不是在堆(heap)上，可能被随时回收。他本身的作于域是属于创建时候的作用域，一旦在创建时候的作用域外面调用block将导致程序崩溃。通过copy可以把block拷贝（copy）到堆，保证block的声明域外使用。在ARC下写不写都行，编译器会自动对block进行copy操作.

## 3、如何解决循环引用？

MRC：**使用\_\_block** ，用来修饰block内部需要使用并改变的外部局部变量，\_block关键字，然后再block快结束的时候，**把\_\_block引用的变量设置为nil。**

ARC：**用\_\_weak和unsafe\_unretained来解决循环引用。Swift使用weak， unown， capture List来进行**

## 4、ARC是什么时期的特性？相对于MRC是如何达到优化的

**编译期属性**。

在以前我们使用MRC的时候：

生成对象的地方，使用autorelease

对象使用时，先retain再autorelease

对象在函数中返回时，使用return [[object retain] autorelease];

ARC下为了**免除autorelease和retain**这两步**多余的操作**来提高性能，**使用函数objc\_autoreleaseReturnValue来替代autorelease、函数objc\_retainAutoreleasedReturnValue来替代retain**

objc\_autoreleaseReturnValue会通过检测caller的retain指令，**来判断caller是否想持有创建的对象。若是的话，就不会执行autorelease，而是将对象存储到TLS(线程局部存储)**中

**该函数会先判断方法返回的对象和TLS中存储的是否一样，若一样，表示对象没有被加到AutoreleasePool中，则可以直接使用，否则就要执行一次retain操作。如此一来，就可以免除autorelease和retain这两步操作，从而提高性能。**

## 5、\_\_block和\_\_weak有什么区别？

\_\_weak 是ARC下使用, \_\_block 在ARC和MRC下都可以使用

\_\_weak 只能修饰对象信息 不能修饰基本类型, \_\_block 对象和基本类型都可以修饰

\_\_block 主要功能是修改外部值, \_\_weak主要是为了解决循环引用

在ARC环境下，\_\_Block会对修饰的对象强引用，在MRC环境下对修饰的对象不会强引用。而且\_\_block修饰局部变量，表示这个对象是可以在block内部修改

## 6、WeakSelf 和 StrongSelf的区别，为什么在block外先Weak后Strong？

进行weak是为了防止循环引用，进行**strong**是防止weak对象被提前置为nil

在block里面使用的\_\_strong修饰的weakSelf是为了在函数生命周期中防止self提前释放。strongSelf是一个自动变量当block执行完毕就会释放自动变量strongSelf不会对self进行一直进行强引用。

## 7、\_\_weak对象是何时被释放的？

**手动添加的AutoreleasePool对象中的对象，被添加到了当前最近的 autoreleasepool 中的，只有当这个 autoreleasepool 自身** **drain 的时候，autoreleasepool 中的 autoreleased 对象才会被 release**，

A**pp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。

**所以是在runloop准备进入休眠时和即将退出时被释放的**

## 8、\_\_weak修饰符的作用以及如何被自动置为nil的？

功能：

若**使用\_\_weak修饰符的变量引用对象被废弃时，则将nil赋值给该变量， 用来解决循环引用。**

使用**objc\_storeWeak**来将\_\_weak变量存在**weak hash表**，**key是引用的对象的地址，value是\_\_weak变量（指针）地址数组（**数组的原因是：因为一个对象可能被多个弱引用指针指向**）**

当引用的object 进行release的时候，执行的过程如下:

1. **objc\_release**

2. 因为引用计数为0所以执行**dealloc**

3. **\_objc\_rootDealloc**

4. **object\_dispose（一个object开头）**

5. **objc\_****destructInstance**

6. **objc\_clear\_deallocating**

而，**调用objc\_clear\_deallocating**的动作如下：

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. 从weak表中删除记录

## 9、在@autoreleasepool可以使用\_\_weak对象吗？

**在使用附有\_\_weak修饰符变量的情形下，增加了对****objc\_****loadWeakRetained函数和objc\_autorelease函数的调用。**这些函数的动作如下：

1. **objc\_loadWeakRetained**函数取出附有\_\_weak修饰符变量所引用的对象并retain

2. **objc\_autorelease**函数将对象注册到autorelease中。

由此可知**，因为附有\_\_weak修饰符变量所引用的对象这样被注册到autorelease中，所以在@autoreleasepool块结束之前都可以放心使用**

## 10、autoreleasepool的底层实现原理及使用场景？

autoreleasepool 是没有单独的内存结构的，它是通过**以 AutoreleasePoolPage 为结点的双向链表**来实现的。

每一个线程的 autoreleasepool 其实就是一个**指针的堆栈**；

每一个指针代表一个需要 release 的对象或者 POOL\_SENTINEL（哨兵对象，代表一个 autoreleasepool 的边界）；

一个 pool token 就是这个 pool 所对应的 POOL\_SENTINEL 的内存地址。当这个 pool 被 pop 的时候，所有内存地址在 pool token 之后的对象都会被 release ；

这个堆栈被划分成了一个以 page 为结点的双向链表。pages 会在必要的时候动态地增加或删除；

Thread-local storage（线程局部存储）指向 hot page ，即最新添加的 autoreleased 对象所在的那个 page

每个 event loop 开始前，系统会自动创建一个 autoreleasepool ，并在 event loop 结束时 drain



magic 用来校验 AutoreleasePoolPage 的结构是否完整；

**next** 指向最新添加的 autoreleased 对象的下一个位置，初始化时指向 begin() ；

**thread** 指向当前线程；

**parent** 指向父结点，第一个结点的 parent 值为 nil ；

**child** 指向子结点，最后一个结点的 child 值为 nil ；

depth 代表深度，从 0 开始，往后递增 1；

hiwat 代表 high water mark

**单个 autoreleasepool 的运行过程可以简单地理解为 objc\_autoreleasePoolPush()、[对象 autorelease] 和 objc\_autoreleasePoolPop(void \*) 三个过程**。

**使用场景**：

**如果你编写的程序不是基于 UI 框架的，比如说命令行工具；**

**如果你编写的循环中创建了大量的临时对象；**

**如果你创建了一个辅助线程**。

## 11、ARC下Dealloc的过程？

父类的dealloc的方法将在子类dealloc方法返回后自动调用

NSObject执行**dealloc**时

调用**\_objc\_rootDealloc**

调用**object\_dispose**

随后调用**objc\_destructInstance**方法。

**objc\_destructInstance 调用object\_cxxDestruct**，执行**\_object\_remove\_assocations**去除和这个对象assocate的对象

执行**objc\_clear\_deallocating**，清空引用计数表并**清除弱引用表，将所有weak引用指nil**

cxxDestruct，它遍历当前对象所有的实例变量（Ivars)，调用objc\_storeStrong(&obj,nil)，ARC下对象的成员变量于编译器插入的.cxx\_desctruct方法自动释放

## 12、GC与ARC的不同？

**GC是运行时，ARC是编译时，GC依赖可达性判断，ARC依赖引用计数；**

## 13、weak的底层实现原理？

weak 实现原理的概括

Runtime维护了一个**weak表**，用于存储指向某个对象的所有weak指针。weak表其实是一个hash（哈希）表，Key是所指对象的地址，Value是weak指针的地址（这个地址的值是所指对象的地址）数组。

weak 的实现原理可以概括一下三步：

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。

## 14、IBOutlet为什么要设置成weak、设置成strong行不行？

UIViewController的view属性是strong，因为controller要直接拥有view。而添加到view上的subviews，作为IBOutlet只需要设置为weak就可以了，因为他们不是controller直接拥有的。直接拥有subviews的是controller的view，ARC会帮助管理内存。

总结归纳为：当控件的父view销毁时，如果你还想继续拥有这个控件，就用srtong；如果想保证控件和父view拥有相同的生命周期，就用weak。

## 15、assign和weak有什么区别？

**assign**是指针赋值，不对引用计数操作，使用之后如果没有置为nil，可能就会产生野指针；而weak一旦不进行使用后会自动置为nil，就不会产生野指针！ weak和assign都是引用计数不变，两个的差别在于，weak用于object type，就是指针类型，

**assign**还可以用于简单的数据类型，如int BOOL 等。 assign看起来跟weak一样，其实不能混用的，assign的变量在释放后并不设置为nil（和weak不同），当你再去引用时候就会发生错误，崩溃，EXC\_BAD\_ACCESS

## 16、OC是否支持垃圾回收机制？

Objective-C在Mac OS是支持垃圾回收机制的（Garbagecollection简称GC），但是apple得移动终端中，是不支持GC的。

## 17、assign、weak、unsafe\_unretained的区别？

assign：用于值类型(如int，float等)

weak： 用于修饰引用类型

unsafe\_unretained：只修饰引用类型

区别：

三者修饰效果相同，即都不会更改所赋新值的引用计数，也不改变旧值的引用计数

assign一般只修饰值类型，虽然也可以修饰引用类型，但是修饰的对象释放后，指针不会自动被置空，此时向对象发消息会崩溃。

weak 不会产生野指针问题。因为weak修饰的对象释放后（引用计数器值为0），指针会自动被置nil，之后再向该对象发消息也不会崩溃。 weak是安全的。

**unsafe\_unretained与assign的区别在于，其只修饰引用类型。会造成 野指针**

**unsafe\_unretained 相对于weak 性能上 稍有差别，微乎其微 ，当明确对象生命周期的时候**

举个例子，当A拥有B对象，A消亡B也消亡，这样当B存在，A也一定会存在的时候，此时B要调用A的接口，就可以通过\_\_unsafe\_unretained 保持对A的引用关系。

## 18、strong、retain、copy的区别？

strong：用于引用类型，强引用。

retain ：用于引用类型，强引用。

copy：修饰属性会在内存里拷贝对象。

区别：

三者都用于修饰引用类型。

strong用于ARC，retain、copy用于MRC。

copy分为浅层复制和深复制两种，NSString、NSArray、NSDictionary等不可变类型都为浅层复制，即其引用计数会+1，而不会创建新的内存。

Block代码块

## 1、\_\_block有什么用？

MRC下可以修改block的变量，也可以进行避免循环引用

## 2、block的copy过程？

在 ARC 下，对 block 变量进行 copy 始终是安全的，无论它是在栈上，还是全局数据段，还是已拷贝到堆上。

**对栈上的 block 进行 copy 是将它拷贝到堆上；**

**对全局数据段中的 block 进行 copy 不会有任何作用；**

**对堆上的 block 进行 copy 只是增加它的引用记数**。

如果栈上的 block 中引用了\_\_block 类型的变量，该 \_\_block 变量在堆上还没有对应的拷贝的话，在将该 block 拷贝到堆上时也会将 \_\_block 变量拷贝到堆上，否则就增加堆上对应的拷贝的引用记数。

**栈上的block拷贝时，会执行辅助函数，同时植入BLOCK\_NEEDS\_FREE标记位，再次拷贝如果有此标记则增加引用计数。**

## 3、为什么使用\_\_block修饰的变量可以被block修改？

**没有\_\_block标记的变量，其值会被只读一份到block私有内存区**

**有\_\_block标记的变量，编译时候会提供两个辅助函数，其地址会被记录在block私有内存区，同时使用一个结构体指针\_\_forwording,**

当一个\_\_block变量从栈上被复制到堆上时，栈上的那个**\_\_Block\_****byref**\_i\_0结构体中的\_\_forwarding指针也会指向堆上的结构。

**\_\_forwarding指针的作用：无论\_\_block变量配置在栈上还是在堆上都能够正确地访问该变量**。通过block的复制，\_\_block变量也从栈上复制到堆上，此时可同时访问栈上的\_\_block变量和堆上的\_\_block变量。

**复制之前，栈上的block的\_\_forwarding指针指向栈上的\_\_block变量，将block从栈上复制到堆上后，栈上的block的\_\_forwarding指针指向堆上的\_\_block变量，堆上的block也指向堆上的\_\_block变量**

## 4、block的底层实现是什么样的？（Block的内存分布是什么样的？）

使用clang的rewrite-objc命令查看

struct BlockLiteral {

void \*isa;

int flags;

int reserved;

void (\*invoke)(void \*, ...);

**struct BlockDescripto**r \*descriptor;

};

**struct BlockDescriptor** {

unsigned long int reserved;

unsigned long int size;

void (\***copy\_helper**)(void \*dst, void \*src);

void (\***dispose\_helper**)(void \*src);

const char \*signature;

};

isa 其实指向了一个类，每一个 block 指向的类可能是 **\_\_NSGlobalBlock\_\_**、**\_\_NSMallocBlock\_\_** 或者 **\_\_NSStackBlock\_\_**

## 5、block为什么要用copy？为什么不用retain?

在栈中而不是在堆中

Runtime运行时

## 1、简述一下OC 消息转发流程

1. Runtime 检查了你的类和所有父类的 class cache 和分发表，但是没找到指定的方法。2. 动态方法解析（dynamic method resolution）

Objective\_C 的 Runtime 会在你的类上调用：

+(BOOL)resolveInstanceMethod:(SEL)selector

+ (BOOL)resolveClassMethod:(SEL)selector'

2、**替换消息接受者**： Runtime 然后调用 – (id)**forwardingTargetForSelector**:(SEL)aSelector。因为 – (void)forwardInvocation:(NSInvocation \*)anInvocation代价昂贵

3、**完整的消息转发**：

如果没有将不能处理的消息转发给其他对象进行处理，则启动了完整的消息转发。运行时会将所有关于此消息的细节封装为一个NSInvocation对象，一旦你有了一个 NSInvocation 你可以改变这个消息的一切，包括目标对象，selector 和参数。

- (NSMethodSignature \*)methodSignatureForSelector:(SEL)aSelector这个函数让重载方有机会抛出一个函数的签名，再由后面的forwardInvocation:去执行。

4、如果你继承自 NSObject，默认它的 – (void)forwardInvocation:(NSInvocation \*)anInvocation实现只是简单的调用 -doesNotRecognizeSelector:

## 2、为什么说OC是一门动态语言（什么是动态绑定）

动态绑定，对象类型，调用的方法均是在运行时确定的，使用消息机制。

## 3、OC是如何响应一条消息的

objc\_msgSend ->Cache查询->方法列表查询->父类查询->消息转发

## 4、说一说消息转发流程，你在开发过程中有哪些应用

异常容错，转发流程见上。

## 5、常用的Runtime方法？

Objc\_ Class\_ Ivar\_ Property\_ Method\_ 开头的

objc\_getClassList

property\_getName

objc\_getClass 获取Class对象

objc\_getMetaClass 获取MetaClass对象

objc\_allocateClassPair 分配空间,创建类(仅在 创建之后,注册之前 能够添加成员变量)

objc\_registerClassPair 注册一个类(注册后方可使用该类创建对象)

objc\_disposeClassPair 注销某个类

objc\_allocateProtocol 开辟空间创建协议

objc\_registerProtocol 注册一个协议

objc\_constructInstance 构造一个实例对象(ARC下无效)

objc\_destructInstance 析构一个实例对象(ARC下无效)

objc\_setAssociatedObject 为实例对象关联对象

objc\_getAssociatedObje\*ct 获取实例对象的关联对象

objc\_removeAssociatedObjects 清空实例对象的所有关联对象

objc\_msgSend 发送ObjC消息

更多查看：

<https://www.jianshu.com/p/e99a80aee2f>

## 6、Objc\_msgSend类似的方法

objc\_msgSend\_stret。如果待发送的消息要返回结构体，那么可交由此函数处理。只有当CPU的寄存器能够容纳得下消息返回类型时， 这个函数才能处理此消息。

objc\_msgSend\_fpret。如果消息返回的是浮点数，那么可交由此函数处理。在某些架构的CPU中调用函数时，需要对“浮点数寄存器” （floating-point register）做特殊处理，

objc\_msgSendSuper。如果要给超类发消息，例如[super message:parameter]，那么就交由此函数处理。也有另外两个与objc\_msgSend\_stret和 objc\_msgSend\_fpret等效的函数，用于处理发给super的相应消息。

## 7、Category里面的方法是如何被加载的？

1.函数**\_objc\_init**为runtime的加载入口，由**libSystem**调用，进行初始化操作。

2.之后调用objc-runtime-new.mm -> **map\_images**加载map到内存

3.之后调用objc-runtime-new.mm->**\_read\_images**初始化内存中的map, 这个时候将会**load所有的类**，协议还有Category。

NSObject的+load方法就是这个时候调用的

1) 循环调用了 **\_****getObjc2CategoryList**方法，这个方法的实现是:

**GETSECT**(\_getObjc2CategoryList, category\_t \*, "\_\_objc\_catlist");

方法中最后一个参数**\_\_objc\_catlist就是编译器刚刚生成的category**数组

2) **load完所有的categories之后，开始对Category进行处理**。

实例方法被加入到了当前的类对象中, 类方法被加入到了当前类的Meta Class中

Step 1. 调用addUnattachedCategoryForClass方法

Step 2. 调用remethodizeClass方法, 在remethodizeClass的实现里调用attachCategoryMethods，这里把一个类的category\_list的所有方法取出来生成了method list。这里是倒序添加的，也就是说，新生成的category的方法会插到旧的category的方法前面。

之后调用**attachMethodLists将所有方法前序添加进类的method list中，category方法在前**

**注意：**

**runtime加载完成后，Category的原始信息在类结构中将不会存在**

**category其实并不是完全替换掉原来类的同名方法，只是category在方法列表的前面而已，所以我们只要顺着方法列表找到最后一个对应名字的方法，就可以调用原来类的方法**

## 8、OC有虚函数表vtable吗？

有，保存了NSObject最常用的方法，可以提升你的应用的速度，因为它使用了比 objc\_msgSend 更少的指令（fewer instructions），仅保存了NSObject的常用方法

static const char \* const defaultVtable[] = {

"allocWithZone:",

"alloc",

"class",

"self",

"isKindOfClass:",

"respondsToSelector:",

"isFlipped",

"length",

"objectForKey:",

"count",

"objectAtIndex:",

"isEqualToString:",

"isEqual:",

"retain",

"release",

"autorelease",

};

## 9、如何hook一个class的实例方法，如何hook一个class方法？

method\_**exchangeImplementations**(Method m1,Method m2);

Method **class\_getClassMethod**(Class aClass, SEL aSelector);

Method **class\_getInstanceMethod**(Class aClass, SEL aSelector);

**先进行class\_addmethod()方法判定是否已经存在此方法，class\_addmethod会进行覆盖父类，但不会覆盖将要进行添加方法的类。**

**如果添加成功，则进行class\_replaceMethod**

**如果没有成功，则进行 method\_exchangeImplementations**

## 10、\_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。

## 11、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；**

原因：

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\_setIvarLayout 或 class\_setWeakIvarLayout** 来处理 strong weak 引用。所以不能向存在的类中添加实例变量；

运行时创建的类是可以添加实例变量，调用 class\_addIvar 函数。**但是得在调用 objc\_allocateClassPair 之后，objc\_registerClassPair 之前，原因同上**。

OC底层数据

## 1、id的底层数据结构

**typedef struct objc\_object {**

**Class isa;**

**} \*id**

## 2、Category的数据结构

旧runtime

typedef struct objc\_category \*Category;

struct objc\_category

{

const char\* category\_name;

const char\* class\_name;

struct objc\_method\_list \***instance\_methods**;

struct objc\_method\_list \***class\_methods**;

struct objc\_protocol\_list \*protocols;

};

新runtime

Typedef struct category\_t \*Category

typedef struct category\_t {

    const char \*name; //类名而不是category名字

classref\_t cls;

struct method\_list\_t \*instanceMethods; /\*定义的实例方法. NULL没有实例方法.

struct method\_list\_t \*classMethods; /\*定义的类方法. NULL表示

struct protocol\_list\_t \*protocols; /\* 遵循的协议表 \*/

struct property\_list\_t \*instanceProperties; //通过objc\_setAssociatedObject和objc\_getAssociatedObject增加实例变量

} category\_t;

## 3、Class的数据结构

typedef struct objc\_class \*Class;

旧runtime：

struct objc\_class {

**Class isa;**

**Class super\_class;**

**const char \*name;**

**long version;**

**long info;**

**long instance\_size;**

**struct objc\_ivar\_list \*ivars;**

**struct objc\_method\_list \*\*methodLists;**

**struct objc\_cache \*cache;**

**struct objc\_protocol\_list \*protocols;**

}

新runtime：

struct objc\_class {

struct objc\_class\* class\_pointer; /\* 指向元类的指针. \*/

struct objc\_class\* super\_class; /\* 指向父类的指针. 对于NSObject来说是NULL.\*/

const char\* name; /\* 类的名称. \*/

long version; /\* 未知. \*/

unsigned long info; /\* 比特蒙板. 参考下面类的蒙板定义. \*/

long instance\_size; /\* 类的字节数.包含类的定义和所有父类的定义

struct objc\_ivar\_list\* ivars; /\* 指向类中定义的实例变量的列表结构. NULL代表没有实例变量.不包括父类的变量. \*/

struct objc\_method\_list\* methods; /\* 链接类中定义的实例方法. \*/

struct sarray \* dtable; /\* 指向实例方法分配表. \*/

struct objc\_class\* subclass\_list; /\* 父类列表 \*/

struct objc\_class\* sibling\_class;

struct objc\_protocol\_list \*protocols; /\* 要实现的原型列表 \*/

void\* gc\_object\_type;

};

## 4、Runtime中SEL和IMP的底层数据结构及区别

typedef struct objc\_selector \*SEL;

struct objc\_selector

{

**void \*sel\_id;**

**const char \*sel\_types;**

};

typedef id (\*IMP)(id, SEL, ...);

IMP本质就是一个函数指针

## 5、MetaClass的底层数据结构及使用方法

和Class的数据结构相同

Meta Class理解为 一个Class对象的Class

当我们发送一个消息给一个NSObject对象时，这条消息会在对象的类的方法列表里查找

当我们发送一个消息给一个类时，这条消息会在类的Meta Class的方法列表里查找，如果没有找到匹配的响应函数则在该 metaclass 的父类中的 methodlists 去查找

## 6、Method的底层数据结构

typedef struct objc\_method \*Method;

struct objc\_method

{

**SEL method\_name;** /\* 这个变量就是方法的名称.编译器使用在这里使用一个`char\*`,当一个方法被注册,名称在运行时被使用真正的SEL替代 \*/

**const char\* method\_types;** /\* 描述方法的参数列表. 在运行时注册选择器时使用.那时候方法名就会包含方法的参数列表.\*/

**IMP method\_imp;**  /\* 方法执行时候的地址. \*/

};

## 7、Cache的底层数据结构和作用

objc\_cache 用来缓存用过的方法，提高性能。

typedef struct objc\_cache \*Cache

struct objc\_cache {

**unsigned int mask /\* total = mask + 1 \*/**

**unsigned int occupied**

**Method buckets[1]**  //方法指针

};

mask: 指定分配cache buckets的总数。在方法查找中，Runtime使用这个字段确定数组的索引位置

occupied: 实际占用cache buckets的总数

buckets: 指定Method数据结构指针的数组。这个数组可能包含不超过mask+1个元素。需要注意的是，指针可能是NULL，表示这个缓存bucket没有被占用，另外被占用的bucket可能是不连续的。这个数组可能会随着时间而增长。

objc\_msgSend每调用一次方法后，就会把该方法缓存到cache列表中，下次的时候，就直接优先从cache列表中寻找，如果cache没有，才从methodLists中查找方法。

## 8、objc\_method\_list底层数据

struct objc\_method\_list

{

struct objc\_method\_list\* method\_next; /\* 这个变量用来链接另一个单独的方法链表 \*/

size\_t method\_count; /\* 结构中定义的方法数量 \*/

struct objc\_method method\_list[1]; /\* 可变长度的结构 \*/

};

## 9、objc\_protocol\_list的底层数据结构

struct objc\_protocol\_list

{

struct objc\_protocol\_list \*next;

size\_t count;

struct objc\_protocol \*list[1];

};

## 10、Ivar的底层数据结构

typedef struct objc\_ivar \*Ivar;

struct objc\_ivar

{

**const char\* ivar\_name;**  /\* 类中定义的变量名. \*/

**const char\* ivar\_type;** /\* 描述变量的类型.调试时非常有用. \*/

**int ivar\_offset;** /\* 实例结构的基地址偏移字节 \*/

};

对象地址 ＋ 基类大小 + ivar偏移字节的方法来计算出ivar相应的地址，并访问到相应的ivar

## 11、objc\_super的底层数据结构

**struct objc\_super**

**{**

**id self; /\* 消息的接受者 \*/**

**Class super\_class; /\* 接受者的父类 \*/**

**};**

## 12、Nil和nil的底层定义

#ifndef Nil

#if \_\_has\_feature(cxx\_nullptr)

#define Nil nullptr

#else

#define Nil \_\_DARWIN\_NULL

#endif

#endif

#ifndef nil

#if \_\_has\_feature(cxx\_nullptr)

#define nil nullptr

#else

#define nil \_\_DARWIN\_NULL

#endif

#endif

NULL (void \*)0 **C指针的字面零值**

nil (id)0 **Objective-C对象的字面零值**

Nil (Class)0 **Objective-C类的字面零值**

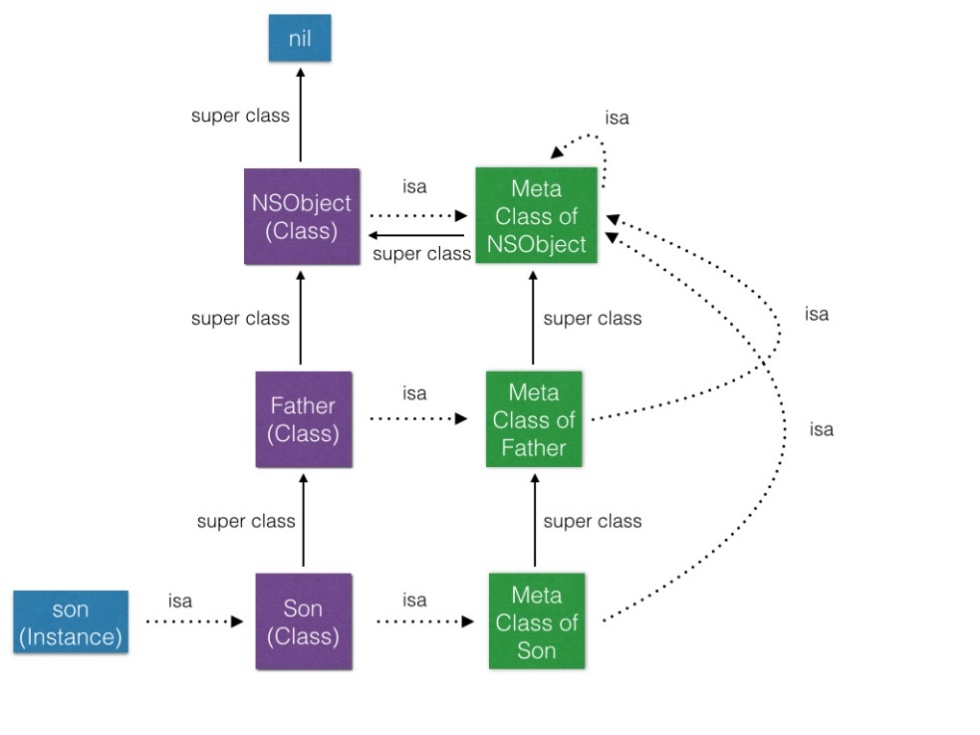
NSNull [NSNull null] **用来表示零值的单独的类对象**

## 13、什么是Non Fragile Ivars

**在 fragile ivar 下你不得不重新编译你继承自苹果类的类来恢复兼容性**

**当 runtime 检测到一个重叠的超类时，它调整你在这个类中新增的 ivar 的偏移量，这样在子类中新增加的内存部分就可以进行使用了。**

## 14、OC的isa指针和super指针，OC语言设计是如何成为一个闭环的



每个Class都有一个isa指针指向一个唯一的Meta Class

**每一个Meta Class的isa指针都指向最上层的Meta Class**（图中的NSObject的Meta Class）

最上层的Meta Class的isa指针指向自己，形成一个回路

每一个Meta Class的super class指针指向它原本Class的 Super Class的Meta Class。但是最上层的Meta Class的 Super Class指向NSObject Class本身

最上层的NSObject Class的super class指向 nil

iOS基础知识

## 1、iOS应用的生命周期

**Not running** ：未运行 程序没启动

**Inactive：** 未激活，程序在前台运行，不过没有接收到事件。在没有事件处理情况下程序通常停留在这个状态

**Active** ：激活，程序在前台运行而且接收到了事件。这也是前台的一个正常的模式

**Backgroud**：后台，程序在后台而且能执行代码，大多数程序进入这个状态后会在在这个状态上停留一会。时间到之后会进入挂起状态(Suspended)。有的程序经过特殊的请求后可以长期处于Backgroud状态

**Suspended**：挂起，程序在后台不能执行代码。系统会自动把程序变成这个状态而且不会发出通知。当挂起时，程序还是停留在内存中的，当系统内存低时，系统就把挂起的程序清除掉，为前台程序提供更多的内存。

顺序：Not running、 inactive、active、inactive、background、suspended

相关方法

application:willFinishLaunchingWithOptions: - 这个方法是你在启动时的第一次机会来执行代码

application:didFinishLaunchingWithOptions: - 这个方法允许你在显示app给用户之前执行最后的初始化操作

applicationDidBecomeActive: - app已经切换到active状态后需要执行的操作

applicationWillResignActive: - app将要从前台切换到后台时需要执行的操作

applicationDidEnterBackground: - app已经进入后台后需要执行的操作

applicationWillEnterForeground: - app将要从后台切换到前台需要执行的操作，但app还不是active状态

applicationWillTerminate: - app将要结束时需要执行的操作

## 2、UIViewController的生命周期

**initWithNibName:Bundle(xib) 或者initWithCoder(storyboard)**  :

非StoryBoard创建UIViewController都会调用这个方法。如果使用StoryBoard进行调用

initWithCoder

**awakeFromNib:**

当awakeFromNib方法被调用时**，所有视图的outlet和action已经连接**

**LoadView** ：

loadView方法在UIViewController对象的view被访问且为空的时候调用

**Viewdidload：**

当loadView将view载入内存中，会进一步调用viewDidLoad方法来进行进一步设置

**viewWillAppear：**

系统在载入所有的数据后，将会在屏幕上显示视图，这时会先调用这个方法

**viewWillLayoutSubViews：**

view 即将布局其Subviews。 比如view的bounds改变了, 要调整Subviews的位置

**ViewDidLayoutSubViews:**

view已经布局其Subviews，这里可以放置调整完成之后需要做的工作

**ViewDidAppear:**

在view被添加到视图层级中以及多视图，上下级视图切换时调用这个方法

**ViewWillDisappear:**

在视图切换时，当前视图在即将被移除、或被覆盖是，会调用该方法，此时还没有调用 removeFromSuperView

**ViewDidDisappear:**

view已经消失或被覆盖，此时已经调用removeFromSuperView

**dealloc:**

视图被销毁，此次需要对你在init和viewDidLoad中创建的对象进行释放

## 2、UIView的生命周期？

**纯代码创建UIView**，调用方式为initWithFrame:[UIScreen mainScreen].bounds];那么它的生命周期为：

-initWithFrame:

-layoutSubviews

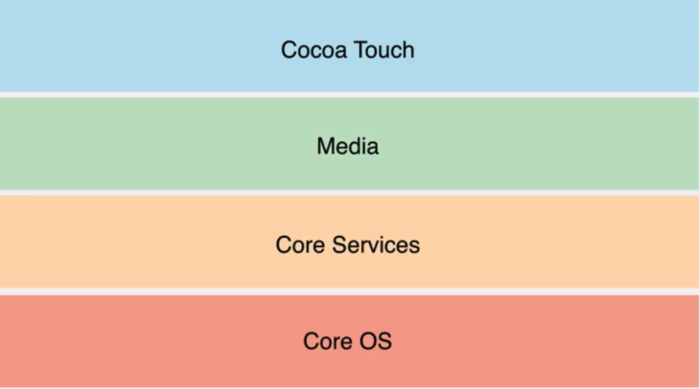
**通过XIB创建UIView**，调用方式为NSArray \*array = [[NSBundle mainBundle]loadNibNamed:@"YFView" owner:nil options:nil];[array lastObject];那么它的生命周期为：

- initWithCoder:

-awakeFromNib

-layoutSubviews

## 3、iOS与Android系统架构有什么不同？



**Cocoa Touch 触摸层**：包括Address Book UI Framework、Event Kit UI Framework、Game Kit Framework、iAd Framework、Map Kit Framework、Message UI Framework、UIKit Framework等等

**Media媒体层**：包含Core Graphics、Core Animation、OpenGL ES、Core Text、Image I/O、Assets Library Framework、Media Player Framework、AV Foundation、OpenAL、Core Audio Frameworks、AV Foundation、Core Media等等

**Core Services核心服务层**：包含Address Book Framework、CFNetwork Framework、Core Data Framework、Core Foundation Framework、Core Location Framework、Core Media Framework、Core Telephony Framework、Event Kit Framework、Foundation Framework、Mobile Core Services Framework、Quick Look Framework、Store Kit Framework、System Configuration Framework、Block Objects、Grand Central Dispatch 、In App Purchase、Location Services、SQLite、XML Support等一些框架，也基本都是基于c语言的接口

**Core OS 核心层**：包含Accelerate Framework、External Accessory Framework、Security Framework、System等几个框架，基本都是基于C语言的接口

## 4、iOS事件传递和响应机制

**触摸事件传递链：**

通过**hitTest:withEvent**方法来确定hit test view来进行事件传递：UIKit –> active app’s event queue –> window->UIViewController

hitTest:withEvent:方法通过传递进来CGPoint和UIEvent返回hit test view。该方法调用**pointInside:withEvent:**方法，如果传入hitTest:withEvent:的point在view的边界范围内，**则pointInside:withEvent:返回YES**。然后，这个方法会在view的所有子view中递归的调用hitTest:withEvent

**Motion或者remote control事件直接发送给first responder**

**响应链：**

Hit-test view是处理触摸事件的第一选择，如果hit-test view不能处理事件，该事件将从事件响应链中寻找响应器，直到系统找到一个处理事件的对象

initial view –> super view –> …..–> view controller –> window –> Application

注意：**sendEvent方法、注意motion方法的UIKit处理和CoreMotion框架**

**排除的视图：**

视图的hidden等于YES。

视图的alpha小于等于0.01。

视图的userInteractionEnabled为NO。

**传递过程：**

1、UIApplication接收到事件，将事件传递给keyWindow。

2、keyWindow遍历subViews的hitTest:withEvent:方法，找到点击区域内合适的视图来处理事件。

3、UIView的子视图也会遍历其subViews的hitTest:withEvent:方法，以此类推。

4、直到找到点击区域内，且处于最上方的视图，将视图逐步返回给UIApplication。

5、在查找第一响应者的过程中，已经形成了一个响应者链。

6、应用程序会先调用第一响应者处理事件。

7、如果第一响应者不能处理事件，则调用其nextResponder方法，一直找响应者链中能处理该事件的对象。

8、最后到UIApplication后仍然没有能处理该事件的对象，则该事件被废弃。

**模拟代码：**

- (UIView \*)hitTest:(CGPoint)point withEvent:(UIEvent \*)event {

if (self.alpha <= 0.01 || self.userInteractionEnabled == NO || self.hidden) {

return nil;

}

BOOL inside = [self pointInside:point withEvent:event];

if (inside) {

NSArray \*subViews = self.subviews;

// 对子视图从上向下找

for (NSInteger i = subViews.count - 1; i >= 0; i--) {

UIView \*subView = subViews[i];

CGPoint insidePoint = [self convertPoint:point toView:subView];

UIView \*hitView = [subView hitTest:insidePoint withEvent:event];

if (hitView) {

return hitView;

}

}

return self;

}

return nil;

}

## 5、UIView和CALayer的区别

UIView和CALayer使用了组合模式进行设计，有着类似的树形结构，在iOS中UIView则是layer-backed View，都对应着一个CaLayer。

区别：

1、**实际上UIView可以进行绘制图像，而CALayer实际上只是一个数据结构**，并不在上面进行渲染，而是管理UIView绘制的，而是缓存了UIView上的位图。

2、**UIView可以处理响应事件，而CALayer不可以**。

3、CALayer维护着三个layer tree,分别是presentLayer Tree、modeLayer Tree、Render Tree,在做动画的时候，我们修改动画的属性，其实是修改presentLayer的属性值,而最终展示在界面上的其实是提供UIView的modelLayer

## 6、iOS 持久化的几种方式

**NSKeyedArchiver归档、NSUserDefault、文件、数据库、网络**

不是所有的对象都可以直接用这种方法进行归档，只有遵守了**NSCoding协议**的对象才可以

**NSCoding协议有2个方法**：

encodeWithCoder:

initWithCoder:

[NSKeyedArchiver archiveRootObject:arr toFile:path];

NSArray \*arr = [NSKeyedUnarchiver unarchiveObjectWithFile:path];

## 7、如何调试EXC\_BAD\_ACCESS

**1、开启NSZombieEnabled**

**2、直接添加Exception Breakpoint**

## 8、GCD任务可以取消吗？如何取消

1、类似NSOperation的cancel机制，你只要加一个**外边变量**，用于标记block是否需要取消。然后block中通过及时的检测这个外部变量的状态，当发现需要取消时，停止block中的后续操作，释放资源。就能达到及时取消block的目的.

2、iOS 8 之后有**dispatch\_block\_cancel**可以进行取消任务（非已开始）

## 9、iOS的应用内通信方式有哪些？

Delegate、block、KVO、Notification

Delegate：委托代理，特点：一般是1对1，效率高

Block：也是委托，代码更紧凑，很容易获取上下文相关信息

KVO：观察者模式，与Notication相比，不是中心式的，效率高，但也缺乏了些特性。也是同步的，在触发的线程执行逻辑，一对多

Notification：中心式，一对多，Notification在哪个线程中Post, 就是在那个线程分发，也就在同一个线程中被observer处理，更为解耦。

## 10、iOS有哪些应用间通讯方式？有哪些进程间通信方式？

**URL Scheme**：配置url types和白名单

**Keychain**：同一个group的app可以通信

**UIPasteboard**：

**UIActivityViewController**：主要是用来实现同设备上app之间的共享文档，以及文档预览、打印、发邮件和复制等功能

**Local Socket**：

进程间：

**Mach Ports：**

Mach内核API，通过一个Mach端口发送一个消息调用一次mach\_msg\_send方法，CFMessagePort确实非常适合用于简单的一对一通讯。简简单单几行代码，一个本地端口就被附属到runloop源上，只要获取到消息就执行回调。

**Distributed Notification: CFNotificationCenterGetDistributedCenter**进行系统级别范围的通信

**Pasteboard**：

**XPC**：

## 11、如何手动通知KVO？

重写+ (BOOL)**automaticallyNotifiesObserversForKey:**(NSString \*)key

在set方法中**在赋值的前后分别调用：willChangeValueForKey和didChangeValueForKey**

## 12、NSTimer是否精准？如何实现一个精准的timer？有哪些定时器？

不精准，当runloop中有耗时任务，NSTimer会延迟。

其他定时器：**CADisplayLink， GCD dispatch\_source\_t实现Timer （YYTimer 用的就是这个）**

更精准：mach\_absolute\_time, 并且将线程优先级提高到最高优先级

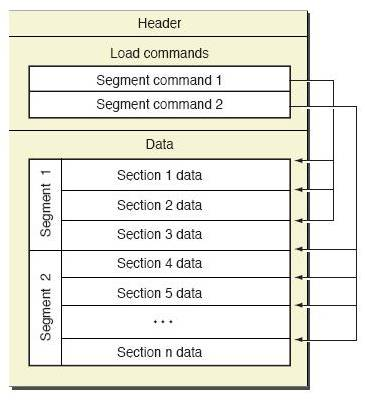
## 13、如何判断某个cell是否在屏幕上

进行监听ScrollView的**ContentOffset**，在observeValueForKeyPath中判断tableView的visibleCell是否含该cell

## 14、使用过哪些脚本语言？如何在编译前和编译后分别添加一些脚本

Shell脚本进行自动打包。在**Build phase**中的**Run Script**进行直接添加脚本， 或者运行外部脚本文件，可以上线拖动，移到Compile Source 文件上面测试在编译前，否则则是在编译后。

## 15、mach-o可执行文件的内存结构



**Header：**保存了Mach-O的一些**基本信息，包括了平台、文件类型、LoadCommands**的个数等等。

**LoadCommands**：这一段紧跟Header，**加载Mach-O文件时会使用这里的数据来确定内存的分布。**

**Data：**每一个segment的具体数据都保存在这里，这里包含了具体的代码、数据等等。

**Segment: 一个segment定义了一些Mach-O文件的数据、地址和内存保护属性**，这些数据在**动态链接器**加载程序时被映射到了虚拟内存中。因此，segments通常都是虚拟内存页对齐的，一个segment包含了0或多个sections

**Section:** TEXT和DATA段可能包含了一系列的标准sections

## 16、NSCache和NSDictionary的区别？

NSCache 是一个容器类，类似于NSDIctionary,通过key-value 形式存储和查询值，用于临时存储对象。和NSDictionary区别就是：

1、**NSCache 中的key不必实现copy，NSDictionary中的key必须实现copy，**在键不支持拷贝操作的情况下，该类用起来比字典更方便。

2、**NSCache中存储的对象也不必实现NSCoding协议，**因为毕竟是临时存储，类似于内存缓存，程序退出后就被释放了。

3、**NSCache是线程安全的，而NSDictionary则绝对不具备此优势**。

存储在NSCache中的通用数据类型通常是实现了NSDiscardableContent协议的对象。在缓存中存储这类对象是有好处的，在内存吃紧时会自动释放。如果一个NSDiscardableContent被放进缓存，则在对象被移除时，缓存会调用discardContentIfPossible方法

## 17、KVO底层实现原理？

使用isa-swizzling技术，当添加观察者之后生成一个**NSKVONotifying\_**开头的隐藏类，

**KVO重写了class方法, 将class中结构体的name指定为最初的类名，object\_setClass**

**同时重写dealloc来释放相关资源。**

## 18、如何适配iPhoneX？

换启动图，解决刘海和home键位置

导航栏：44+44（statusbar44）

使用Safe Area进行适配

竖屏

UIEdgeInsets

- top : 44.0

- left : 0.0

**- bottom : 34.0**

- right : 0.0

横屏

- top : 0.0

- left : 44.0

**- bottom : 21.0**

- right : 44.0

**Autolayout的relation secondItem使用Safe Area，而不是使用Top Layout或者BottomLayout**

## 19、iOS为什么必须在主线程中操作UI？

因为UIKit不是线程安全的

## 20、基于source的角度解释下事件响应？

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。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**。

**iOS应用在runloop上注册了一个 Source1 (基于 mach port 的) 用来接收系统事件，其回调函数为 \_\_IOHIDEventSystemClientQueueCallback()**。硬件事件触发时，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。

## 21、iOS异常有哪些异常？如何处理

OC异常：NSException一般只在OC当中被捕获，一般情况下在捕获**NSException异常后同时也会捕获到一个对应的signal异常**。

**NSSetUncaughtExceptionHandler(UncaughtExceptionHandler)，**但不会捕获到Swift异常

Mach异常和Swift异常： 使用signal处理相关信号

signal(SIGABRT, SignalExceptionHandler)

C++语言层面的错误，比如野指针、除零、内存非法访问等；

未捕获异常（Uncaught Exception），在 iOS 中最常见的就是通过 @throw 抛出的 NSException（常见的错误，比如数组访问越界）

对于第一种问题，由于 iOS 和 Android 底层系统都是 Unix 或者类 Unix 系统，**可以采用信号机制来捕获 signal 或 sigaction，通过设置的回调函数来收集信号的上下文信息。**

**第二种问题可以通过 NSSetUncaughtExceptionHandler 设置异常处理回调函数来收集异常的调用堆栈。**

**不足的地方是，并不是所有的程序崩溃都是由于发生可以捕捉的异常的，有些时候引起崩溃的大多数原因如：内存访问错误，重复释放等错误就无能为力了，因为这种错误它抛出的是Signal，所以必须要专门做Signal处理。**

**忽略信号，但 SIGKILL 和 SIGSTOP 信号不可忽略；**

**使用默认的处理函数 SIG\_DFL，大多数信号的默认动作是终止进程；**

**捕获信号，执行用户定义的函数。**

这里有两个特殊的常量：

**SIG\_IGN：向内核表示忽略此信号。对于不能忽略的两个信号SIGKILL和SIGSTOP，调用时会报错；**

**SIG\_DFL：执行该信号的系统默认动作.**

**信号处理流程分三步：**

**注册信号处理回调函数**；

**在回调函数中收集调用堆栈信息**；

**恢复信号默认处理函数；**

Mach 异常 Signal 异常 C++ 异常 OC 异常 死锁 用户抛出的异常

## 22、iOS抛出异常的过程？

iOS系统自带的 Apple’s Crash Reporter 记录在设备中的Crash日志，Exception Type项通常会包含两个元素： **Mach异常 和 Unix信号**

所有**Mach异常都在host层被ux\_exception转换为相应的Unix信号**，并通过threadsignal将信号投递到出错的线程。iOS中的 POSIX API 就是通过 Mach 之上的 BSD 层实现的

捕获Mach异常或者Unix信号都可以抓到crash事件，**优选Mach异常，因为Mach异常处理会先于Unix信号处理发生**，如果Mach异常的handler让程序exit了，那么Unix信号就永远不会到达这个进程了。

因为**硬件产生的信号(通过CPU陷阱)被Mach层捕获**，然后才转换为对应的Unix信号；苹果为了统一机制，**于是操作系统和用户产生的信号(通过调用kill和pthread\_kill)也首先沉下来被转换为Mach异常，再转换为Unix信号**

**使用Signal函数捕获Unix异常信号。**

## 23、OC Exception Handing机制有什么缺陷？

苹果也在文档中**明确标识不要使用Exception Handling机制，而要使用Error Object和防御性编程**。

对于这些异常的捕获处理，OC的Exception Handling机制其实是具有缺陷的，iOS系统使用Unix进程模型，大部分系统信号的默认处理都是直接Crash

使用Exception Handing机制进行catch的实现在32位和64位是不同的。

OC的Exception Handling机制的32位实现是非常耗费资源，在32位处理器上**，OC进行处理@try block时会调用setjmp()函数**，并且**进行执行额外的审查操作**。但是在64位处理器上抛出一个Exception却会消耗很多的资源。为了在64位处理器上达到最好的性能，在只有绝对需要抛出exception的时候才可以抛出（这句话苹果官网文档也有多出描述）

**64位是使用类似C++的zero-cost Exception handling机制**，C++的Exception目前在GNU C++和VC++64下使用的Zero-Cost Model，**在这个模型中编译器会产生一个静态表来决定一个被throw出来的exception被哪个handler处理**。

OC Runtime也重新为64位可执行文件重新实现了Exception机制，提供了一个zero-cost并且与C++的Exception具有可互操作性的@try blocks。**64位处理器处理zero-cost的@try block不会导致严重性能损失，与modern C++的Error handler机制一样，编译期，但catch操作仍然比较耗费资源。 使用了stack Unwind机制。**

## 24、NSURLProtocol如何拦截网络请求？

[NSURLProtocol registerClass:]

实现+ (BOOL)**canInitWithRequest:(NSURLRequest \*)request**等代理方法，注意在+ (BOOL)**canInitWithRequest:(NSURLRequest \*)request**不要造成循环，可以在header加一个字段。

## 25、Today Extention与Host通信的方式有哪些？

1、**Shared User Default**: 同一个AppGroup

let shared =NSUserDefaults(suiteName: “group.com.weshest.easyword")

2、**Shared Data Container**：同一个AppGroup

let groupUrl =NSFileManager.defaultManager().containerURLForSecurityApplicationGroupIdentifier(“\*\*\*\*")

3、**openURL API**

在Host App中创建 URL Type，选中选中Host App Target -> Info， 找到URL Types， 点击加号新建一个

## 26、序列化相关的NSSecureCoding和NSCoding的区别？

区别在于解数据时要指定Class，用**-decodeObjectOfClass:forKey:方法代替了-decodeObjectForKey:**。这样做更安全，因为序列化后的数据有可能被篡改，若不指定Class，-decode出来的对象可能不是原来的对象，有潜在风险。相关API：

[NSKeyedArchiver archiveRootObject: toFile:

[NSKeyedUnarchiver unarchiveObjectWithFile:

NSKeyedUnarchiver setRequiresSecureCoding:YES]

**+ (BOOL)supportsSecureCoding{**

return YES;

}

## 27、intrinsicContentSize是什么？content Hugging和Content Compression Resistance呢？size class呢？

**intrinsicContentSize**：固有大小。在AutoLayout中，它作为UIView的属性（不是语法上的属性），如果没有指定大小，则暗intrinsicContentSize来设定自己的大小。

**content Hugging／content Compression Resistance**都是为了解决intrisicContentSize冲突而引进的属性：

**1. Content Hugging** 约束（不想变大约束）表示：如果组件的此属性优先级比另一个组件此属性优先级高的话，那么这个组件就保持不变，另一个可以在需要拉伸的时候拉伸。属性分横向和纵向2个方向。Hug自己

**2. Content Compression Resistance** 约束（不想变小约束）表示：如果组件的此属性优先级比另一个组件此属性优先级高的话，那么这个组件就保持不变，另一个可以在需要压缩的时候压缩。 Compress 别人

**size class** 是把屏幕的宽和高分别分成三种情况：（**Compact, Regular, Any**），是对设备类型的一个抽象

## 28、如何使用autolayout实现动画？

几个相关的API

removeConstraints

addConstraints

updateConstraintsIfNeeded

setNeedUpdateConstraints

layoutIfNeed

## 29、简述一下NSCoping协议、深复制与浅赋值？

NSCopying是一个与对象拷贝有关的协议。如果想让一个类的对象支持拷贝，就需要让该类实现NSCopying协议。NSCopying协议中需要- (id)copyWithZone:(NSZone \*)zone。

当我们的类实现了NSCopying协议，通过类的对象调用copy方法时，copy方法就会去调用我们实现的- (id)copyWithZone:(NSZone \*)zone方法，实现拷贝功能

一般我们会自定义一个deepCopy和shallowCopy

## 30、沙盒应用目录是什么样?

**documents:**保存应用程序运行时生成的**持久化数据**,iTunes会备份和恢复此文件

**library:**用来存储应用程序的**文件数据,**默认设置,或者其他动态信息,其中包含两个文件夹(**Caches,Preferences**)

**library/Caches:**存储缓存文件,这个目录用于保存那些可再生文件,如网络请求数据(图片,音频,视频,)鉴于此,应用程序最好还是负责删除这些文件,不会被iTunes同步

**library/Preferences**:用来存储用户的**偏好设置**,会将APP的偏好设置放在这个文件中(夜间模式,无图模式,永久存储是否会或者去地理位置),注意:我们用NSUserDefaults存储的用户数据会被保存在该目录下的文件中,会被iTunes同步.

**tmp:**用来存储临时文件,保存用应用程序在运行时的临时数据,使用完毕后,再将其从相应的文件夹下删除,应用程序没有运行时,系统也有可能删除该目录下的文件,不会被iTunes同步,而且,在iPhone重启时,该目录下单文件会被删除。

**app:**获取.app文件:该目录包含了**应用程序本身的数据,包含资源文件和可执行文件**.程序启动以后会根据需求动态加载(用到了懒加载机制)代码到或者资源到内存中.而且整个目录是只读的,不会被iTunes同步.

**升级时候.app、library/preferences会变化**

## 31、iOS应用的内存分配是什么样的？

**栈区：**局部变量和实参

**堆：**OC中使用new方法创建的对象、程序员手动申请空间

**BSS段：**没有初始化的全局变量和静态变量

**常量区/数据段：**用来存储已经初始化的全局变量、静态变量、常量数据

**代码段：**代码的二进制文件（用来存储程序的代码和指令）

## 32、iOS中广播的总类有哪些？新的推送API有哪些？

UILocalNotification、UINotification

iOS8：UIUserNotificationSettings进行注册

iOS10：UNAuthorizationOptions 使用UserNotification框架。

**NSUserNotificationCenter \*center = [UNUserNotificationCenter currentNotificationCenter];**

**[center requestAuthorizationWithOptions:**UNAuthorizationOptions completionHandler:^(BOOL granted, NSError \*\_Nullable error) {

If(granted){

} else {

}

}

## 33、isKindOfClass和isMemberOfClass的联系与区别？

**联系：**两者都能检测一个对象是否是某个类的成员

**区别：**isKindOfClass不仅用来确定一个对象是否是一个类的成员，也可以用来确定一个对象是否派生自该类的类的成员，而isMemberOfClass只能做到第一点

## 34、什么是野指针？什么是悬挂指针？什么是安全释放？

**野指针**：Wild Pointers 所导致的错误和悬挂指针非常相似，但野指针的问题更容易被发现。

任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。

**悬挂指针：**Dangling Pointers 当所指向的对象被释放或者收回，但是对该指针没有作任何的修改，以至于该指针仍旧指向已经回收的内存地址，此情况下该指针便称迷途指针（悬挂指针）

**安全释放：**先释放，然后置为NULL

## 35、什么是Bitcode？如何解决部分SDK不支持bitcode的问题？

Bitcode见Bitcode工作原理

1、暂时关闭对bitcode的支持（建议），方法：选中工程文件-->选中project下的工程名-->Build Settings-->找到BuildOptions-->Enable Bitcode-->置为NO，即可。

2、替换和移除不支持bitcode的平台SDK。

## 36、什么是toll-free？？

Toll-Free Bridging用于在Foundation对象与Core Foundation对象之间相互转化,俗称桥接。

相关\_\_bridge、\_\_bridge\_retain、\_\_bridge\_transfer见内存知识

## 37、autorelease对象是何时被释放的？

手动干预释放和系统自动释放

**手动干预释放**就是指定autoreleasepool,当前作用域大括号结束就立即释放

**系统自动去释放:** 不手动指定autoreleasepool,Autorelease对象会在当前的 runloop 迭代结束时释放

**kCFRunLoopEntry(1):** 第一次进入会自动创建一个autoreleasepoll

**kCFRunLoopBeforeWaiting(32):** 进入休眠状态前会自动销毁一个autoreleasepoll,然后重新创建一个新的autoreleasepoll

**CFRunLoopExit(128)** 退出runloop时会自动销毁最后一个创建的autoreleasepoll

## 38、iOS应用是如何实现后台多任务处理(Multitasking)？

后台存活的几种方式：

**Background Audio，VoIP**

**Location Services**：

**Newsstand**： 定时更新

**Background Task Completion**： 开屏下可以运行10分钟

**Background Fetch**：不可控

**Remote Notification**： Silent Notification

**Background Transfer Service**：下载上传

注意 应用被**Background Fetch**、**Remote Notification**、**Background Transfer Service**三类唤醒时，只被给予几秒钟时间来处理更新

## 39、HandOff是什么？如何实现iOS、Mac/网页应用互通的？

Handoff的支持有一些硬性的要求：

互通的所有设备**必须支持 Buletooth LE 4.0**，Handoff使用BLE信号来传递用户活动数据。

设备处于联网状态，有时候有些数据还是会通过互联网来传递的，比如Mail App的邮件内容的同步。所有设备必须连到同一个iCloud账户。

当然你还得**保证当前设备的Handoff功能打开了(iOS:设置->通用->Handoff 与建议的应用程序。 Mac:系统偏好设置->通用,倒数第二栏有个选项,"允许这台Mac和iCloud设备之间使用Handoff")**

## 40、int retVal=UIApplicationMain(argc,argv,nil,nil) 参数和返回值是什么意思？

Argc: main参数的个数

Argv: main函数的参数

第一个nil：UIApplication类或者子类名字

第二个nil：UIApplication类或者子类的Delegate，如果使用的UIApplication子类，你可能需要将delegate赋予子类

返回值：尽管返回了值，但此函数不会返回，而是等到应用退出。

这个函数初始化princle class并且设置delegate，并且设置main runloop，如果应用的Info.plist通过NSMainNibFileKey指定了一个main nib文，这个函数会加载这个nib文件。

## 41、什么是AppThinning?

**iOS9**开始简单, Apple推出了一项新功能称为App Thinning, 字面理解就是App瘦身, 说来就是开发者将整个App包传到App Store，当不同用户设备下载App的时候，App Store通过根据不同设备特性分发经过剪裁的App，从而减少了用户包大小.

**App Thinning有三种特性：AppSlicing, bitcode, on-demond resource(ODR)**

## 42、什么是AppSlicing？

Slicing通过将app bundle资源根据不同的设备特性分为不同的版本。对于图片资源，会根据设备(如iphone6 plus**需要@3x,iphone 6需要@2x)所需图片分辨率不同分发给对应设备所需对应的图片资源.**

对于Slicing的使用很简单只需要满足下面几个条件：

当前项目的development target版本 iOS7及以上

图片资源通过asset catalog进行管理

设备版本必须是iOS9以后，确切的是9.0.2以后

## 43、bitcode工作原理？

在进行语法、语义分析后，会进行**生成IR**，然后进行IR优化，这是常规流程。

在前后端之间传递的就是IR（中间码），而bitcode就是一种特殊形式的中间码。原本前后端的工作都是在本地LLVM中完成，虽然Apple没有给出具体的Bitcode实现，但是通过他们的文档可以猜测，是将一部分后端的工作移到了服务器进行

指令集优化。根据你设备的状态去做编译优化，进而提升性能。

开启Bitcode编译后，可以使得开发者上传App时只需上传Intermediate Representation(中间件)，而非最终的可执行二进制文件。 在用户下载App之前，AppStore会自动编译中间件，产生设备所需的执行文件供用户下载安装。也就是当我们提交程序到 App Store上时， Xcode 会将程序编译为一个中间表现形式( bitcode )。然后 App store 会再将这个 Bitcode 编译为可执行的64位或32位程序。苹果会根据下载应用的用户的手机指令集类型生成只有该指令集的二进制，进行下发。

**实际上是不再下发fat-binary文件，而是支持单个指令集的二进制文件。**

**Bitcode注意点**

**1.**Xcode 7默认开启 Bitcode ，如果应用开启 Bitcode，那么其集成的其他第三方库也需要是 Bitcode 编译的包才能真正进行 Bitcode 编译 ，可以在**”Build Settings”->”Enable Bitcode**”开启enable bitcode。

**2.**开启 Bitcode 编译后，编译产生的 .app 体积会变大(中间代码，不是用户下载的包)，且 .dSYM 文件不能用来崩溃日志的符号化(用户下载的包是 Apple 服务重新编译产生的，有产生新的符号文件)

**3.**通过 Archive 方式上传 AppStore 的包，可以在Xcode的Organizer工具中下载对应安装包的新的符号文件

**需要注意的是bitcode只默认在archive下编译。在debug和release下并不会**。

如果您开发的是app那么走正常的打包archive流程就好了。如果你正在开发.a静态库或者framework，请注意打**包方式设置为archive,**或者在打包脚本中加入-fembed-bitcode参数。如果需要的话，需要在Build Settings中打开 DEPLOYMENT\_POSTPROCESSING=YES，设置Strip Style为debugging

**附：简短的过程**

编译器编译的过程：

**Lexer** ：读入源文件，并将其转化成字符流。

**Parser ：将**字符流转换成 AST（抽象语法树）。

**Semantic Analysis** ：对输入的 AST 进行语法检查。

**Code Generation ：**代码生成，将 AST 转换成低层次的IR指令。

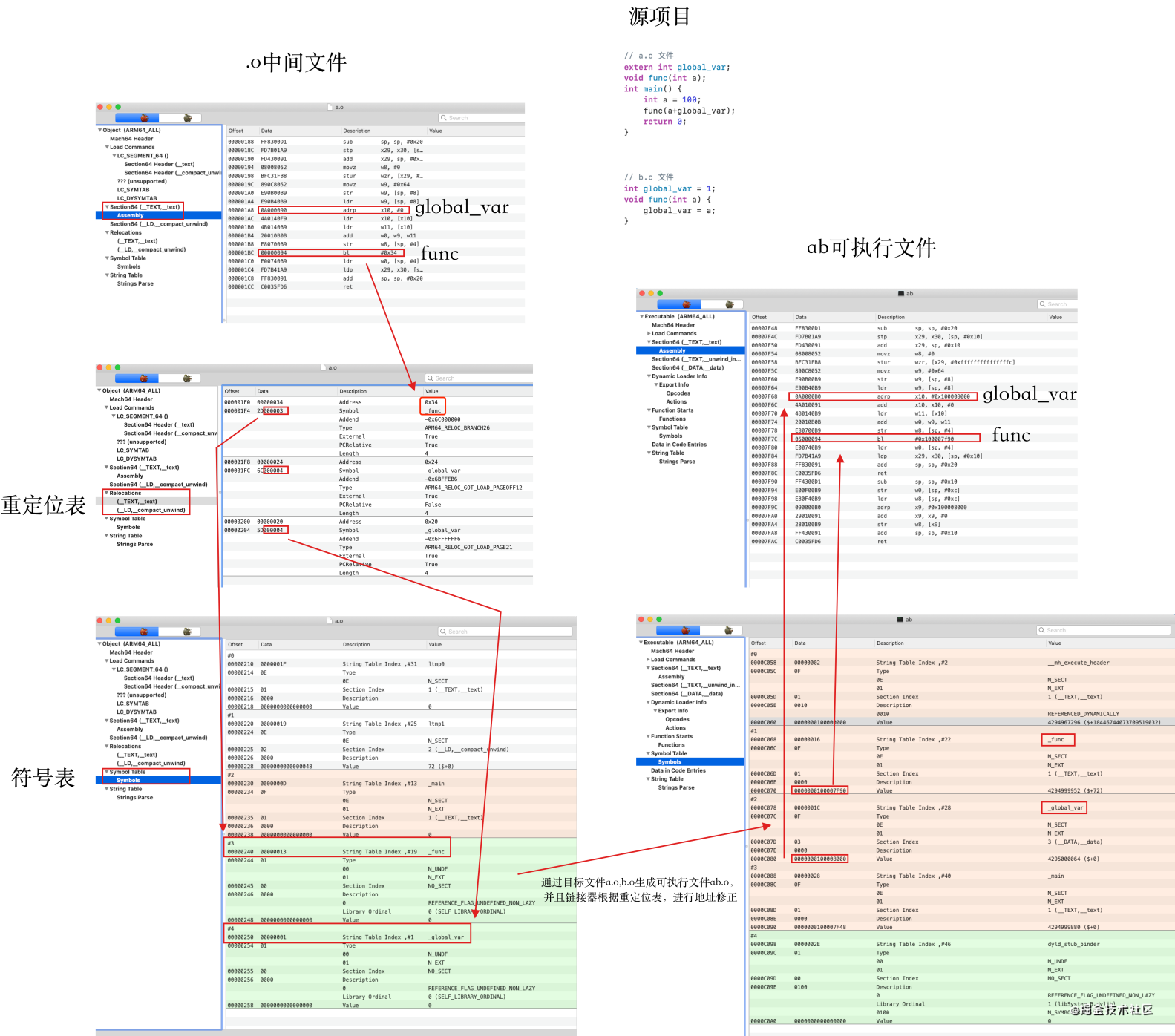
**Optimization ：**分析 IR 指令，将其中潜在会拖慢运行速度的指令干掉。

**AsmPrinter ：**通过 IR（中间码）生成特定 CPU 架构的汇编代码。

**Assemble ：**将汇编代码转化成二进制。

**Linker ：**通常程序会引用其他的二进制文件（.a 或者 framework)，但是这些链接在程序中没有正确的地址，只是个占位符。Linker 的工作就是给这些占位符正确的地址。

https://juejin.cn/post/6844903912198127623



## 44、什么是on-Demand Resources？

**Xcode build setting下 开启Enable On Demand Resources**

On-Demand资源指的是和应用下载包分离并且存在App Store上的应用内容，应用请求一系列的on-demand资源，操作系统管理下载和存储。可以使用on-demand资源获得优势的应用包扩。

在研发过程中，开发者可以为on-demand资源分配一个或者多个标签，也就是你用来辨识特定资源的字符串。

**在 Xcode的Resource Tag中进行添加tag分组，也可以在每个imageasset中进行添加**

**使用NSBundleResourceRequest 的beginAccessingResourcesWithCompletionHandler方法进行下载**

## 45、CoreText如何实现混排？如何进行优化？《暂停》，记忆相关类即可

framesetter framesetter对应的类型是 **CTFramesetter，通过CFAttributedString进行初始化，它作为CTFrame对象的生产工厂，负责根据path生产对应的CTFrame**

CTFrame **CTFrame**是可以通过**CTFrameDraw**函数直接绘制到context上的，当然你可以在绘制之前，操作CTFrame中的**CTLine**，进行一些参数的微调

**CTLine** 可以看做Core Text绘制中的一行的对象 通过它可以获得当前行的line ascent,line descent ,line leading,还可以获得Line下的所有Glyph Runs

**CTRun** 或者叫做 Glyph Run，是一组**共享想相同attributes（属性）的字形的集合体**

## 46、iOS8-iOS11有什么新特性？

iOS8：Spotlight、Handoff、UIVisualEffectView毛玻璃效果

iOS9：使用基于TLS1.2的HTTPS，强制ATS增强，URL Scheme白名单， AppThinning功能

IOS10：UserNotificaiton，CoreSpotlight 的增强

iOS11：iOS11导航栏，设置导航栏大标题： navigationBar.prefersLargeTitles，SafeArea，iOS11弃用了automaticallyAdjustsScrollViewInsets属性，新增contentInsetAdjustmentBehavior。

**UITableView新增separatorInsetReference属性**

## 47、什么是Provision Profile文件？

**Provisioning Profile文件包含了上述的所有内容：证书、App ID和设备**。

一个Provisioning Profile对应一个**Explicit App ID**或**Wildcard App ID**（一组相同Prefix/Seed的App IDs）。在网站上手动创建一个Provisioning Profile时，需要依次指定App ID（单选）、证书（Certificates，可多选）和设备（Devices，可多选）。用户可在网站上删除（Delete）已注册的Provisioning Profiles。

**Provisioning Profile决定Xcode用哪个证书（公钥）/私钥组合（Key Pair/Signing Identity）来签署应用程序（Signing Product）,将在应用程序打包时嵌入到.ipa包里。安装应用程序时，Provisioning Profile文件被拷贝到iOS设备中，运行该iOS App的设备也通过它来认证安装的程序。**

如果要**打包或者在真机上运行一个APP，一般要经历以下三步**：

**首先**，需要证书对应的私钥来进行签名，用于标识这个APP是合法、安全、完整的；

**其次，**需要指明它的App ID，并且验证Bundle ID是否与其一致；

**然后，**如果是真机调试，需要确认这台设备是否授权运行该APP。

Provisioning Profile把这些信息全部打包在一起，方便我们在调试和发布程序打包时使用。这样，只要在不同的情况下选择不同的Provisioning Profile文件就可以了。

## 48、png和jpg的区别。png一定是无损的吗？

**1、Gif: GIF有着不可忽视的特点：体积小，有着极好的压缩效果，支持动画，并且支持透明效果，但是不能像PNG那样让图片在被放大后仍然保持透明**

**优点**

优秀的压缩算法使其在一定程度上保证图像质量的同时将体积变得很小。

可插入多帧，从而实现动画效果。

可设置透明色以产生对象浮现于背景之上的效果。

**缺点**

由于采用了**8位压缩，最多只能处理256种颜色**，故不宜应用于真彩图像

**2、PNG（Portable Network Graphics）**

便携式网络图片（Portable Network Graphics），简称PNG，是一种无损数据压缩位图图形文件格式。

**PNG格式是无损数据压缩的**，允许使用类似于GIF格式的调色板技术，支持真彩色图像，**并具备Alpha**（半透明）等特性

**优点**

\* 支持256色调色板技术以产生小体积文件

\* 最高支持48位真彩色图像以及16位灰度图像。

\* 支持Alpha通道的半透明特性。

\* 支持图像亮度的gamma校正信息。

\* 支持存储附加文本信息，以保留图像名称、作者、版权、创作时间、注释等信息。

\* 使用无损压缩。

\* 渐近显示和流式读写，适合在网络传输中快速显示预览效果后再展示全貌。

\* 使用CRC循环冗余编码防止文件出错。

\* 最新的PNG标准允许在一个文件内存储多幅图像。

**缺点**

但也有一些软件不能使用适合的预测，而造成过分臃肿的PNG文件。

**3、JPG（Joint Photographic Experts Group）**

**JPEG**是一种针对相片影像而广泛使用的一种**失真压缩标准方法**

**优点**

JPEG/JFIF是最普遍在万维网（World Wide Web）上被用来储存和传输照片的格式。JPEG在色调及颜色平滑变化的相片或是写实绘画（painting）上可以达到它最佳的效果。在这种情况下，它通常比完全无失真方法作得更好，仍然可以产生非常好看的影像（事实上它会比其他一般的方法像是GIF产生更高品质的影像，因为GIF对于线条绘画（drawing）和图示的图形是无失真，但针对全彩影像则需要极困难的量化）。

**缺点**

它并不适合于线条绘图（drawing）和其他文字或图示（iconic）的图形，因为它的压缩方法用在这些图形的型态上，会得到不适当的结果

**4、WebP**

优势体现在它具有更优的图像数据压缩算法，能带来更小的图片体积，而且拥有肉眼识别无差异的图像质量；同时具备了无损和有损的压缩模式、Alpha 透明以及动画的特性，在 JPEG 和 PNG 上的转化效果都相当优秀、稳定和统一。

**让SDWebImage支持WebP，设置如下Build Settings -- Preprocessor Macros , add SD\_WEBP=1**

## 49、NSURLSession 和 NSURLConnection 的区别？

**1、普通任务和上传**

NSURLSession针对下载/上传等复杂的网络操作提供了专门的解决方案，针对普通、上传和下载分别对应三种不同的网络请求任务：NSURLSessionDataTask,NSURLSessionUploadTask和NSURLSessionDownloadTask。创建的task都是挂起状态，需要resume才能启动。

当服务器返回的数据较小时,NSURLSession与NSURLConnection执行普通任务的操作步骤没有区别。

执行上传任务时，NSURLSession与NSURLConnection一样需要设置POST请求的请求体进行上传

**2、下载任务方式**

NSURLConnection下载文件时，先是将整个文件下载到内存，然后再写入到沙盒，如果文件比较大，就会出现内存暴涨的情况。

而使用NSURLSessionDownloadTask下载文件，会默认下载到沙盒中的tem文件中，不会出现内存暴涨的情况，但是在下载完成后会把tem中的临时文件删除，需要在初始化任务方法时，在completionHandler回调中增加保存文件的代码

**3、请求方法的控制**

**NSURLConnection实例化对象，实例化开始，默认请求就发送(同步发送),不需要调用start方法。而cancel可以停止请求的发送，停止后不能继续访问，需要创建新的请求。**

**NSURLSession有三个控制方法，取消(cancel)、暂停(suspend)、继续(resume)，**暂停以后可以通过继续恢复当前的请求任务。

**4、断点续传的方式**

NSURLConnection进行断点下载，通过设置访问请求的HTTPHeaderField的Range属性，开启运行循环runloop，NSURLConnection的代理方法作为运行循环的事件源，接收到下载数据时代理方法就会持续调用，并使用NSOutputStream管道流进行数据保存。

NSURLSession进行断点下载，当暂停下载任务后，如果downloadTask（下载任务）为非空，调用**cancelByProducingResumeData:(void (^)(NSData \*resumeData))completionHandle**r这个方法，这个方法接收一个参数，完成处理代码块，这个代码块有一个NSData参数resumeData，如果resumeData非空，我们就保存这个对象到视图控制器的resumeData属性中，在点击再次下载时，通过调用[ [self.session **downloadTaskWithResumeData:self.resumeData]resume]方法进行继续下载操作**

经过以上比较可以发现，使用NSURLSession进行断点下载更加便捷

**5、配置信息**

NSURLSession的构造方法（sessionWithConfiguration:delegate:delegateQueue）中有一个NSURLSessionConfiguration类的参数可以设置配置信息，其决定了cookie，安全和高速缓存策略，最大主机连接数，资源管理，网络超时等配置。NSURLConnection不能进行这个配置，相比较与NSURLConnection依赖与一个全局的配置对象，缺乏灵活性而言，NSURLSession有很大的改进了。(关于配置信息，后面会讲解到)

通过以上几点，大概知道了NSURLSession和NSURLConnection的区别，想必下载再遇到这样的问题不会支支吾吾了。下面来点NSURLSession的干货

**NSURLSessionConfiguration的创建方法**

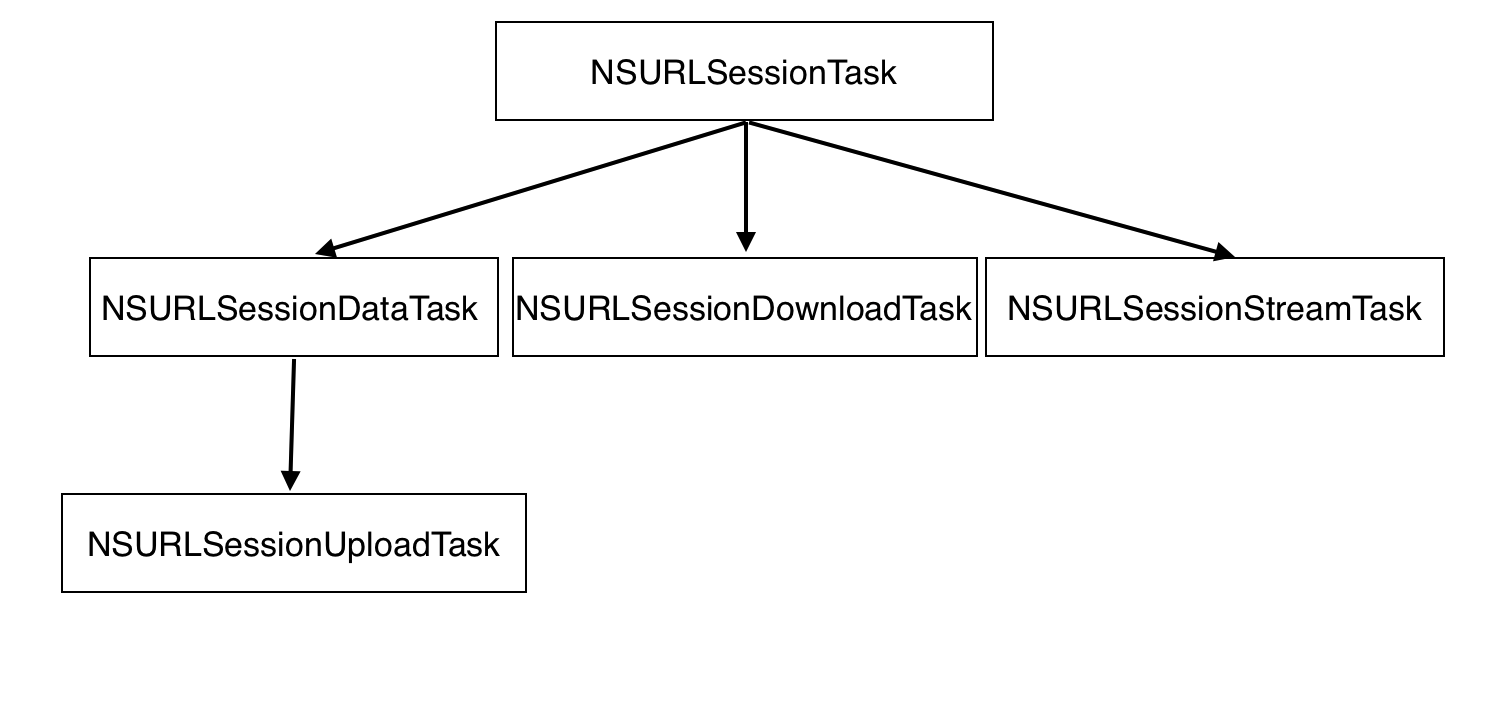
**defaultSessionConfiguration** 使用全局的cache，cookie,使用硬盘来缓存数据。

**ephemeralSessionConfiguration** 临时session配置，与默认配置相比，这个配置不会将缓存、cookie等存在本地，只会存在内存里，所以当程序退出时，所有的数据都会消失

**backgroundSessionConfiguration** 后台session配置，与默认配置类似，不同的是会在后台开启另一个线程来处理网络数据**。**

**URLSessionTask**

NSURLSessionTask是一个抽象类，其下有4个实体子类可以直接使用：**NSURLSessionDataTask**、**NSURLSessionUploadTask**、**NSURLSessionDownloadTask、NSURLSessionStreamTask**。这四个子类封装了现代程序四个最基本的网络任务：获取数据，比如JSON或者XML，上传文件和下载文件还有数据流的获取**。**



**NSURLSessionStreamTask： 创建TCP/IP连接**

## 52、单例和全局变量的区别？

**错误，创建时机**：全局变量已经初始化，单例可以使用的时候再初始化

**单例的特点：**

保证某类只存在唯一实例。

该类本身完成自身的初始化。

获取该唯一实例的方式非常明确，可以通过该类本身定义的静态方法getInstance()获取该类的唯一实例引用。

**线程安全**：全局变量不是线程安全的，单例加了dispatchonce 是线程安全的

**静态变量定义某类的实例引用特点：（注意此处是静态变量而不是全局变量）**

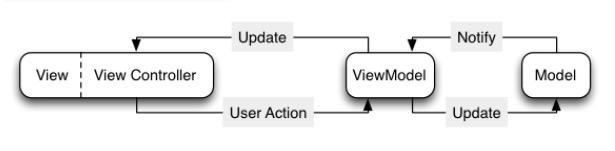
该类的实例引用的静态变量可定义在任何文档类当中。

获取该类的实例引用的静态变量，可以通过定义该静态变量的类名通过点语法进行访问该引用。

任何位置可以对该静态变量进行重新赋值。

**静态成员变量实际上是全局变量， 所以需要像全局变量一样定义以分配存储空间。**

## 53、MVC 和 MVVM 的区别？



**MVVM 的优势**

**低耦合**：View 可以独立于Model变化和修改，一个 viewModel 可以绑定到不同的 View 上

**可重用性：**可以把一些视图逻辑放在一个 viewModel里面，让很多 view 重用这段视图逻辑

**独立开发：**开发人员可以专注于业务逻辑和数据的开发 viewModel，设计人员可以专注于页面设计

**可测试：**通常界面是比较难于测试的，而 MVVM 模式可以针对 viewModel来进行测试

**MVVM 的弊端**

数据绑定使得Bug 很难被调试。你看到界面异常了，有可能是你 View 的代码有 Bug，也可能是 Model 的代码有问题。数据绑定使得一个位置的 Bug 被快速传递到别的位置，要定位原始出问题的地方就变得不那么容易了。

**对于过大的项目，数据绑定和数据转化需要花费更多的内存（成本**）。主要成本在于：

**数组内容的转化成本较高**：数组里面每项都要转化成Item对象，如果Item对象中还有类似数组，就很头疼。

**转化之后的数据在大部分情况是不能直接被展示的**，为了能够被展示，还需要第二次转化。

只有在API返回的数据高度标准化时，这些对象原型（Item）的可复用程度才高，否则容易出现类型爆炸，提高维护成本。

调试时通过对象原型查看数据内容不如直接通过NSDictionary/NSArray直观。

同一API的数据被不同View展示时，难以控制数据转化的代码，它们有可能会散落在任何需要的地方。

https://www.cnblogs.com/machao/p/5667867.html

## 54、如何编写单元测试，比如写了一个网络库，如何测试该网络库，用例怎么写的更全面一些？

Setup、teardown

## 55、Block和Protocol的区别，Block是为了解决什么问题而使用的？

答案**1. block的提出是为了解决在不同的对象间除了传递值之外还可以传递一个操作而提出的。**

答案**2. block的设计是为了解决方法中方法可做为参数传入当前上下文中，将方法具体的实现进行抽象，一种策略设计模式的具体表现。**

**两者区别：**

1. **delegate 运行成本低 只是对对象指针的保存 直接回调。 block 出栈将使用的数据从栈copy到堆上**
2. **Delegate 一对一 同一个协议 一个对象只能设置一个代理， block 写法更简单一些 不需要写 protocol 函数之类的**
3. **公共接口 方法较多的时候用delegate，比如 tableviewdelegate， 异步 简单的回调 用block，比如 afnetworking**

## 56、NSLog性能低下的原因？

**NSLog会向****ASL写log，同时向Terminal写log，而且同时会出现在****Console.app中**（Mac自带软件，用NSLog打出的log在其中全部可见）；不仅如此，**每一次NSLog都会新建一个ASL client并向ASL守护进程发起连接，log之后再关闭连接。**所以说，当这个过程出现N次时，消耗大量资源导致程序变慢也就不奇怪了。

## 58、NSTimer在dispatch\_asyc中如何使用？

**1、NSTimer只有在主线程（dispatch\_get\_main\_queue（））才会自动添加到runloop里面去，在其他线程需要主动添加**

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

NSTimer \*timer = [NSTimer scheduledTimerWithTimeInterval:3 target:self selector(methodOne) userInfo:nil repeats:YES];

**[NSRunloop currentRunloop] addTimer:timer Mode:NSDefaultRunloopMode];**

**[ [NSRunloop currentRunloop] run];** 切忌，这里我们不能将currentRunloop改成mainRunLoop，这两句代码就是主动将timer添加到runloop

})

**2、不用scheduled方式初始化的，需要手动****addTimer:forMode: 将timer添加到一个runloop中。**

**而****scheduled的初始化方法将以默认mode直接添加到当前的runloop中.**

**3、NSTimer上的定时任务是在创建NSTimer的线程上执行的。NSTimer的销毁和创建必须在同一个线程上操作**

**4、NSTimer要被添加到当前线程的 Runloop 里面且 Runloop 被启动，定时任务（selector或者invocation）才会触发。**

**正确使用：**

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

NSLog("NSTimer will be scheduled...");

//Define a NSTimer

let timer:NSTimer = NSTimer.scheduledTimerWithTimeInterval(1, target: self, selector: #selector(self.\_backgroundTimerAction(\_:)), userInfo: nil, repeats: true);

//Get the current RunLoop

let runLoop:NSRunLoop = NSRunLoop.currentRunLoop();

//Add the timer to the RunLoop

runLoop.addTimer(timer, forMode: NSDefaultRunLoopMode);

//Invoke the run method of RunLoop manually

runLoop.run();

NSLog("NSTimer scheduled...");

**}**

**59、为什么要写<NSObject>？**

**NSObject协议定义了一套所有根类可以实现的基础方法。**

**NSObjec类也是遵循NSObject协议的，也就是说NSObject类也实现了NSObject协议所声明的方法。**

**NSProxy同样也遵循NSObjec协议**

@protocol MyProtocol<NSObject>

这样对象不仅遵从了MyProtocol，而且也同时遵从了NSObject协议，可以响应这两个协议中的消息。正因为你应用中所有的对象都继承于NSObject类并且遵循NSObject协议，这样在实现MyProtocol的实例时不需要更多条件就可以使用NSObject的一般方法。

**遵循 <NSObject> 是为了确保实现了这个方法**，这样在调用的时候就可以直接用这个方法检测是否能响应这个 SEL 了。

其实在 **ObjC 1.0** 的时候，protocol 的这个 @optional 选项是不存在的，所有的 protocol 方法都是必须实现的。所以不遵循 <NSObject> 也没关系，只要判断指针是否存在然后直接调用就完了。但是 **ObjC 2.0** 加入了　@optional 特性，于是乎必须使用 的 respondsToSelector: 方法先做一次判断了。

## 59、Objective-C 中和Swift中的协议继承？

OC: 协议也是可以继承的，就如所有类的根类是NSObject一样,协议的根类协议也是NSprotocol；满足协议规则的类,在类的后面跟上<协议名字>

Swift: protocol \*\*: &&&;

## 60、NSURLRequest的缓存策略有哪些?

**NSURLRequestUseProtocolCachePolicy** = 0 //默认的缓存策略， 如果缓存不存在，直接从服务端获取。如果缓存存在，会根据response中的Cache-Control字段判断下一步操作，如: Cache-Control字段为must-revalidata, 则询问服务端该数据是否有更新，无更新的话直接返回给用户缓存数据，若已更新，则请求服务端.

**NSURLRequestReloadIgnoringLocalCacheData** = 1 //忽略本地缓存数据，直接请求服务端.

**NSURLRequestIgnoringLocalAndRemoteCacheData** = 4 //忽略本地缓存，代理服务器以及其他中介，直接请求源服务端.

**NSURLRequestReloadIgnoringCacheData** = NSURLRequestReloadIgnoringLocalCacheData

**NSURLRequestReturnCacheDataElseLoad** = 2 //有缓存就使用，不管其有效性(即忽略Cache-Control字段), 无则请求服务端.

**NSURLRequestReturnCacheDataDontLoad** = 3 //只加载本地缓存. 没有就失败. (确定当前无网络时使用)

**NSURLRequestReloadRevalidatingCacheData** = 5 //缓存数据必须得得到服务端确认有效才使用

## 61、三种创建NSURLSessionConfiguration的方式？

**三种创建NSURLSession的方式**

defaultSessionConfiguration //默认配置使用的是持久化的硬盘缓存，存储证书到用户钥匙链。存储cookie到shareCookie。

ephemeralSessionConfiguration //不使用永久持存cookie、证书、缓存的配置，最佳优化数据传输。

backgroundSessionConfigurationWithIdentifier //可以上传下载HTTP和HTTPS的后台任务(程序在后台运行)。

在后台时，将网络传输交给系统的单独的一个进程,即使app挂起、推出甚至崩溃照样在后台执行。

## 62、[UIView animationWithDuration \*\*\*]方法中使用self会引起循环引用吗？

不会循环引用，但只有到动画结束后，viewController才能dealloc，可以不用转化为weakSelf，**可以不用不代表没有必要**（转）。

## 63、NSTimer不准的解决办法？

解决方法：

1、**在子线程中创建timer，在主线程进行定时任务的操作**

**在子线程中创建timer，在子线程中进行定时任务的操作，需要UI操作时切换回主线程进行操作**

2、**添加定时器到主线程的CommonMode中或者子线程中**

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

3、通过**mach\_absolute\_time()获**取CPU已运行的tick数量。将tick数经过转换变成秒或者纳秒，从而实现时间的计算

**4、CADisplayLink**

CADisplayLink是一个频率能达到屏幕刷新率的定时器类。iPhone屏幕刷新频率为60帧/秒，也就是说最小间隔可以达到1/60s

**5、GCD定时器**

我们知道，RunLoop是dispatch\_source\_t实现的timer

## 64、如何控制屏幕旋转?

1、【Targets】中设置

【General】-->【Deployment Info】-->【Device Orientation】

2、**UIWindow设置**

iOS6的UIApplicationDelegate提供了下述方法，能够指定UIWindow中的界面的屏幕方向：

- (NSUInteger)application:(UIApplication \*)application supportedInterfaceOrientationsForWindow:(UIWindow \*)window NS\_AVAILABLE\_IOS(6\_0);

该方法默认值为Info.plist中配置的Supported interface orientations项的值。

3、UIViewController设置

通过三个代理方法设置

//Interface的方向是否会跟随设备方向自动旋转，如果返回NO,后两个方法不会再调用

**- (BOOL)shouldAutorotate** {

return YES;

}

//返回直接支持的方向

**- (UIInterfaceOrientationMask)supportedInterfaceOrientations**{

return UIInterfaceOrientationMaskPortrait;

}

//返回最优先显示的屏幕方向

**- (UIInterfaceOrientation)preferredInterfaceOrientationForPresentation** {

return UIInterfaceOrientationPortrait;

}

RxSwift

## 1、什么是响应式编程？

函数式反应式编程，是一种**和事件流有关的编程方式，关注导致状态值改变的行为事件，一系列事件组成事件流**。

FRP是更加有效率的处理事件流，而无需显式去管理状态。

具体来说，FRP包括两个核心点：

1.事件流，离散事件序列

2.属性，代表模型连续的值

**函数式编程**强调通过数学语言风格的函数**、不变性和表现力来实现计算，并最小化变量和状态的使用，线程安全**

## 2、RAC中热信号与冷信号的区别？ReactiveCocoa和Rxswift对信号的处理有什么不同

热信号是主动的，即使你没有订阅事件，它仍然会时刻推送。而冷信号是被动的，只有当你订阅的时候，它才会发送消息

ReactiveCocoa主要有两个实体，signal和 SignalProducer。Signal和SignalProducer是两种不同的对象。我们必须把SignalProducer转换成Signal后才能compose由Signal实例产生的信号。

signal无论是否绑定了订阅者都可以发布事件。 热信号

SignalProducer要有一个信号或者事件产生才会触发， 冷信号

在RxSwift中，signal和 SignalProducer变成了Observable，这两个实体在Rx中是一个东西。在RxSwift里创建Observables不需要考虑是冷信号还是热信号。冷／热信号是当你subscribing／observing产生的副作用，可以轻松的用同一个操作符将Subject和Observable的实例组合起来。

## 3、Observable是什么？RxSwift提供了哪些创建Observable的操作符？

Observable是被观察者，也就是发出事件的事件（信号）源。一个Observer（观察者）只有订阅了Observable，才能接收Observable发送的事件。

**Create：**使用观察者接收指定事件的方法来创建自定义的Observable序列（观察者将要接收的事件）

**Empty：**创建一个空Observable序列，该序列只在指定的调度器上只发出completed事件

**Never：**创建一个Never序列，该序列不会发出任何事件，也不会终止，不会输出任何结果

**Just：**创建一个只有一个元素的Observable序列

**Error：**创建一个Observable序列，该序列不发出任何事件而立刻以 error事件终止

**Of：**创建一个包含可变数量元素的Observable序列

**Deferred：**为每个订阅事件的观察者都创建一个新的被观察的Observable序列，创建的多个Observable序列是相同的，可以认为是同一个Observable 序列

**Generate：**创建一个被观察的Observable序列，只要提供的条件为真，就会生成并发出事件，而且可以在指定的调度器上进行

**From：**从Array、Dictionary或Set来创建一个被观察的Observable序列

## 4、什么是Subject？RxSwift提供了哪些Subject？

在Rx中的Subject是一个桥接或者代理，它即是一个Observer(观察者)也是一个Observable(被观察序列)，它可以作为Observer(观察者)订阅多个Observables，又可以作为Observable来发出事件序列，连接了源Observable和观察者。

**PublishSubject** ：PublishSubject只会向观察者发送 观察者订阅后 源Observable发出的事件

**ReplaySubject**：当一个观察者订阅ReplaySubject后，ReplaySubject会将订阅前已经发出的BufferSize个事件重现地发送给观察者，然后才将源Observable序列新事件发给观察者

**BehaviorSubject：**当一个观察者订阅BebaviorSubject后，BehaviorSubject会将最近源Observable已经发出的事件重现地发送给观察者，然后再将源Observable发出的事件发给观察者，功能上相当于bufferSize为1的ReplaySubject，当观察者订阅时源Observable已经以Error事件结束，则BehaviorSubject仅将Error事件发送给观察者。

**Variable**：使用BehaviorSubject来进行实现的，所以它也会将最近的事件重现地发送给观察者，并且Variable保持这当前值，也永远不会发出Error事件，但它会在deinit的时候发出Complete事件，这是与PublishSubject、ReplaySubject、BehaviorSubject不同的地方。

**注意不同点：**

1、**当PublishSubject、ReplaySubject、BehaviorSubject在销毁时不会发出一个Complete事件，Complete事件取决于源Observable**

2、。**Variable与PublishSubject、ReplaySubject、BehaviorSubject不同，它在deinit的时候会进行发出一个Complete事件**

3、**使用variable实例的asObservable()方法来访问底层的BehaviorSubject**。

4、**Variable没有事件on操作符(如onNext(\_:)),它只暴露了一个value属**性来进行获取和设置值。当进行设置一个值时，这个值会添加到底层的BehaviorSubject的事件序列中。

## 5、RxSwift提供了哪些Scheduler？

**CurrentThreadScheduler**(串行) 当前线程Scheduler，默认使用的

**MainScheduler**(串行) 主线程

**SerialDispatchQueueScheduler** 封装了GCD的串行队列

**ConcurrentDispatchQueueScheduler** 封装了GCD的并行队列，这个在有任务要在后台执行的时候很有用

**OperationQueueScheduler** 封装了NSOperationQueue

## 6、RxSwift提供了哪些组合操作符？

**startWith() ：**在一个Observable sequence前插入事件

**merge() :**将多个Observable sequence整合到一个新的Observable sequence(注意不flap)

**zip():** 将多个Observable sequence整合进一个新的Observable sequence，最多可以整合8个, 并且按照整合的index将每个被整合的Observable sequence的事件进行组合 作为新Observable sequence的一个事件。

**combineLatest()** ：将多个Observable sequence整合为一个新的Observable sequence，最多可以整合8个，它在每个被整合的Observable sequence发出一个事件之后进行发送事件，当被整合的其中一个Observable sequence有新事件后，新的Observale sequence 会将新事件与其他已经发出的事件作为一个单独的事件进行发送。

**switchLatest()：**将多个Observable sequence整合成一个新Observable sequence，新Observable sequence将从最近发送事件的Observable sequence中进行发送事件。

## 7、RxSwift提供了哪些变换操作符？

**Map**：对Observable sequence发出的事件进行变换（应用一个transform closure），并且返回一个变换后的新的Observable sequence。

**flatMap**：当一个Observable sequence是另一个Observable sequence的事件时，如果想对每一个Observable sequence中的事件进行处理时很有用，它会将所有的事件降维成一个一维的Observable sequence.

**flatMapLatest**：实际上map和switchLatest运算符的结合。它只会发送最先发送事件的inner Observable sequence中的事件。

## 8、什么是Connectable Observable ？RxSwift提供了哪些连接操作符？

**可连接的序列（****Connectable Observable）**：

（1）可连接的序列和一般序列不同在于：有订阅时不会立刻开始发送事件消息，只有当调用 connect() 之后才会开始发送值。

（2）可连接的序列可以让所有的订阅者订阅后，才开始发出事件消息，从而保证我们想要的所有订阅者都能接收到事件消息。

**publish** 方法会将一个正常的序列转换成一个可连接的序列。同时该序列不会立刻发送事件，只有在调用 connect 之后才会开始

**replay：**同上面的 publish 方法相同之处在于：会将将一个正常的序列转换成一个可连接的序列。同时该序列不会立刻发送事件，只有在调用 connect 之后才会开始。不同在于：新的订阅者还能接收到订阅之前的事件消息（数量由设置的 bufferSize 决定）

**multicast**：同样是将一个正常的序列转换成一个可连接的序列，同时 multicast 方法还可以传入一个 Subject，每当序列发送事件时都会触发这个 Subject 的发送。

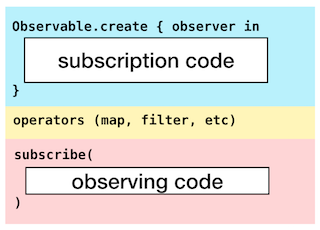
**refCount**：操作符可以将可被连接的 Observable 转换为普通 Observable即该操作符可以自动连接和断开可连接的 Observable。当第一个观察者对可连接的 Observable 订阅时，那么底层的 Observable 将被自动连接。当最后一个观察者离开时，那么底层的 Observable 将被自动断开连接。

**share(relay:)：**该操作符将使得观察者共享源 Observable，并且缓存最新的 n 个元素，将这些元素直接发送给新的观察者。简单来说 shareReplay 就是 replay 和 refCount 的组合。

## 9、RxSwift提供了哪些并发操作符？

**ObserveOn**指明了观察者在哪个调度器（Scheduler）上来观察Observable的变化，也就是在那个调度器（Scheduler）上Observable将变化通知给观察者，观察者的逻辑将在该调度器上进行执行。它只会影响后续观察者的线程。

**SubscribeOn**不仅指明了Observable在哪个调度器（Scheduler）将变化通知给观察者，同时也指定了Observable本身执行实际处理操作的线程。



默认情况下，在使用Subscribe方法后，一个Observable会在Subscribe方法调用的同一个线程通知观察者Observabl\*的变化，后续的操作符也会在此线程进行执行工作。SubscribeOn操作符不仅改变了Observable发送通知的线程，也改变了Observable实际处理操作的线程。ObserveOn仅指明了Observable发送通知消息通知观察者变化的线程，同样ObserveOn也影响了Observable操作符出现的线程，

**所以，通常在Observable的链式操作符中多次调用ObserveOn来改变操作符工作的线程。**

## 10、RxSwift提供了哪些时间操作符？

**throttle 操作符和debounce 操作符** （rxswift实现是一样的）

throttle是一个控制连续事件间隔的节流操作符，当后序事件与前一个事件大余某个时间窗口周期时，才会发出事件，这个操作符保证了result Observable序列中两个事件之间的间隔大于指定周期

sample 操作符：使用一个采样Observable事件序列对源Observable序列采样，采样返回最新的事件，如果距离上一次采样没有最新事件，则再次采样不返回事件

**interval 操作符：**用来创建Observable，周期性地进行发送事件，使用指定的调度器来运行timer和通知观察者, 在timer间隔指定时间后发送事件, 与timer操作符类似，实际上是由Implementations文件夹中的Timer来实现的，但它无法指定第一个事件在一个周期内发送的时间。

**timer 操作符：**用来创建Observable，使用指定的调度器来运行timer和通知观察者, 与interval操作符类似，但它可以指定第一个事件元素的相对时间。

**timeout 操作符：**对Observable 序列中的每个事件元素应用一个超时策略，如果没有在指定的超时时间内接收到下一个事件，将会向Observer观察者发出一个TimeoutError事件或者使用另一个Observable序列。

**delay 操作符：**将源Observable中的事件进行延迟一段事件发出从而形成一个新的Observable序列，Error事件不会被delay，在指定的调度器上运行timer。

## 11、RxSwift提供了哪些错误相关操作符？

**catchErrorJustReturn()**  ：从Error event中恢复，返回一个只发送一个Event事件的Observable Sequence，然后终止

**catchError：**从一个Error event中恢复，返回另外一个恢复后的Observable Sequence

**retry() ：**当收到Error event时通过重新订阅 Observable sequence来进行恢复，可以无限循环地执行此动作

**retry(\_:)：**当收到Error event时，通过重新订阅Observable 来恢复，可以尝试重新订阅maxAttemptCount次数

## 12、RxSwift提供了哪些条件过滤操作符？

**Filter：**仅发送符合filter中的过滤条件的事件

**distinctUtilChanged：**过滤一个observable sequence中重复的事件

**elementAt：**仅发送一个Observable sequence 中某个指定index上的元素事件

**single：**仅发送Observable sequence 中第一个元素（或者第一个符合条件的第一个元素），如果一个Observable sequence 发送的事件不止1个则会抛出异常。

**take()：**从Observable sequence 起始点算，顺序取出特定个数的事件进行发送。

**takeLast()：**从Observable sequence 结尾取出特定个数的事件进行发送。

**takeWhile：**当指定条件判断为true的时候从Observable sequence开头进行发送事件，当条件判断为false的时候停止发送。

**takeUtil()**：当一个引用的条件Observable sequence发送事件时才停止从源Observable sequence发送事件

**skip()：**从一个Observable sequence起始点算，略过特定个数的事件

**skipWhile：**从Observable sequence 事件序列开头判定，当条件判为false时发送事件。

**skipWhileWithIndex：**从Observable sequence开头开始进行判定，当条件判定为fasle时发送剩余的事件，在这个条件判定closure中index会被传入作为参数

**skipUtil()：**当一个引用的条件Observable sequence发送事件时才开始从源Observable sequence发送事件（和takeUtil相反）

## 13、RxSwift中三种Units类型

**Driver Units(驱动单位)**

（不能输出错误）\* （在主调度器上观察）\* （共享副作用,shareReplayLatestWhileConnected）

**ControlProperty/ControlEvent Units(控制属性/事件单位)**

（不能输出错误）\* （在主调度器上订阅）\* （在主调度器上观察）\* （共享副作用）

**Variable Units(变量单位)**

（不能输出错误）\* （共享副作用）

Driver（驱动单位）

驱动Driver是一个一个直观书写UI层响应式逻辑的Units，它的使用场景是使Observable序列值驱动应用。如 CoreData数据模型驱动UI展示，从其他绑定的UI元素来驱动UI展示。

## 14、RxSwift是如何实现消息转发的？

1、**添加子类**，并为子类添加一个前缀方法，这个前缀方法是原方法的实现，原方法实现变为\_objc\_msgFroward运行时消息转发接口。

2、为每个子类提供消息转发的三个方法，respondsToSelector:, class, methodSignatureForSelector: forwardInvocation

3、还有一个步骤，就是**给原类实例对象添加一个关联属性**，这个关联属性的key就是子类前缀方法，属性值value是名为MessageSentObservable实例对象的钩子，这样我们可以通过前缀方法名获取这个钩子，获取到钩子之后，调用钩子的-(void)messageSentWithParameters:(NSArray\*)parameters方法，把原方法的参数以数组的方式传递出去，用来实现个性化的 消息转发（先实现），然后才invokeWithTarget方法来实现。

4、消息转发优化：在load方法中，将这个observer根据方法的签名为key添加到optimizedObserversByMethodEncoding中。 objc\_cache类似

## 15、RxSwift如何将ViewModel数据与View进行绑定？视图于视图呢？

mySwitch.rx\_value.bindTo(testView.rx\_hidden)

**自定义可绑定属性**

extension UITextField{

var ex\_validState:AnyObserver<Bool>{

return UIBindingObserver(UIElement: self) { textfield, valid in

textfield.backgroundColor = valid ? UIColor.clearColor():UIColor.lightGrayColor()

}.asObserver()

}

}

**RxSwift中做数据绑定有三种**

利用BindTo方法

利用Driver(强烈建议使用这个，)

利用KVO来手动绑定（很少用到）

Driver是RxSwift精心制作的，专门提供给UI层的一个接口。利用Driver你可以

利用CoreData的模型来驱动UI

利用UI的状态来绑定其他UI的状态

**Driver能够保证，在主线程上监听，因为UIKit都是需要在主线程上操作**

view.rx\_observe(CGRect.self, "frame")

.subscribeNext { frame in

print("Got new frame \(frame)")

}

在rx\_observe可以使用地方都可以使用rx\_observeWeakly。rx\_observeWeakly的执行效率要低一点，因为要处理对象的dealloc关系。除此之外，rx\_observeWeakly还可以用在weak属性上。

## 16、RxSwift如何进行单元测试？

RxTest

**TestScheduler -> createHotObservable ->bind(ViewModel的一个属性)**

http://www.51testing.com/html/74/n-3721474.html

Runloop

## 1、什么是Runloop？

Runloop是一个用来调度工作以及协助事件（event）输入的事件处理循环，run loop的目的是让线程在有工作的时候忙于工作，而没有工作的时候处于休眠状态。

1、保持程序的持续运行，runloop 保证主线程不会被销毁，也就保证了程序的持续运行 ）。

2、处理App中的各种事件（ 如：touches 触摸事件、NSTimer 定时器事件、Selector事件（选择器 performSelector））。

3、节省CPU资源，提高程序性能（ 有事情就做事情，没事情就休息 (其资源释放) ）。

4、负责渲染屏幕上的所有UI

## 2、你知道哪些Runloop Mode？

Cocoa和Core Foundation定义的标准模式

1.【kCFRunLoopDefaultMode (NSDefaultRunLoopMode)】: App的默认Mode，通常主线程是在这个Mode下运行。

2.【UITrackingRunLoopMode】: 界面跟踪 Mode，用于 ScrollView 追踪触摸滑动，保证界面滑动时不受其他 Mode 影响。

3.【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。

4.【GSEventReceiveRunLoopMode : 接受系统事件的内部 Mode，通常用不到。

5.【kCFRunLoopCommonModes (NSRunLoopCommonModes)】 : 这个并不是某种具体的 Mode, 可以说是一个占位用的Mode（一种模式组合）Common modes： NSRunloopCommonModes（Cocoa），对于Cocoa程序，这个mode 包括Default mode，以及Event tracking mode。（Core Foundation 仅包括Default mode， 可以使用CFRunloopAddCommonMode来添加mode）

**用来处理事件在运行循环中的优先级和分类**

## 3、Runloop有哪些事件源？

输入源(input source)和定时源 (timer source)。输入源(input source)传递异步事件消息,通常这个消息来自于其他线程或程序。定时源(timer source) 则传递同步事件,发生在特定时间或者重复的时间间隔

**输入源包括基于端口（port-based）的输入源和自定义输入源**

基于端口的输入源（port - based Source）

Cocoa 和 Core Foundation 内置支持使用端口相关的对象和函数来创建的基于端口的输入源。

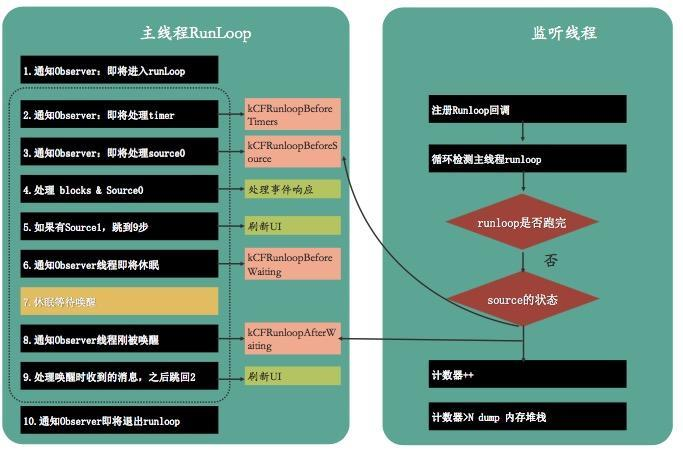
在 Cocoa中，不需要直接创建输入源。只需要简单地创建端口对象（port）,并使用 NSPort 的方法把该端口添加到 run loop。端口对象会自己处理创建和 配置输入源。

在 Core Foundation,必须手动创建端口（port）和它的 run loop 输入源.在两种情况下, 你都可以**使用端口相关的函数(CFMachPortRef,CFMessagePortRef,CFSocketRef) 来创建合适的对象**。

自定义输入源（custom input source）

**为了创建自定义输入源, 必须使用 Core Foundation 里面的 CFRunLoopSourceRef 类型相关的函数来创建**。我们可以使用回调函数来配置自定义输入源。Core Fundation会在配置源的不同地方调用回调函数,处理输入事件,在自定义输入源从 run loop 移除的时候 清理输入源的资源。

## 4、Runloop处理事件的流程？





1、通知观察者run loop已经启动

2、通知观察者任何即将要开始的定时器（**即将处理timer**）

3、通知观察者任何即将启动的非基于端口的源(**即将处理source0**)

4、启动任何准备好的非基于端口的源**(处理source0)**

5、如果基于端口的源准备好并处于等待状态，立即启动；并进入步骤9。

6、通知观察者线程进入休眠

7、将线程置于休眠直到任一下面的事件发生：

**某一事件到达基于端口的源**

**定时器启动**

**Run loop设置的时间已经超时**

**run loop被显式唤醒**

8、通知观察者线程将被唤醒。

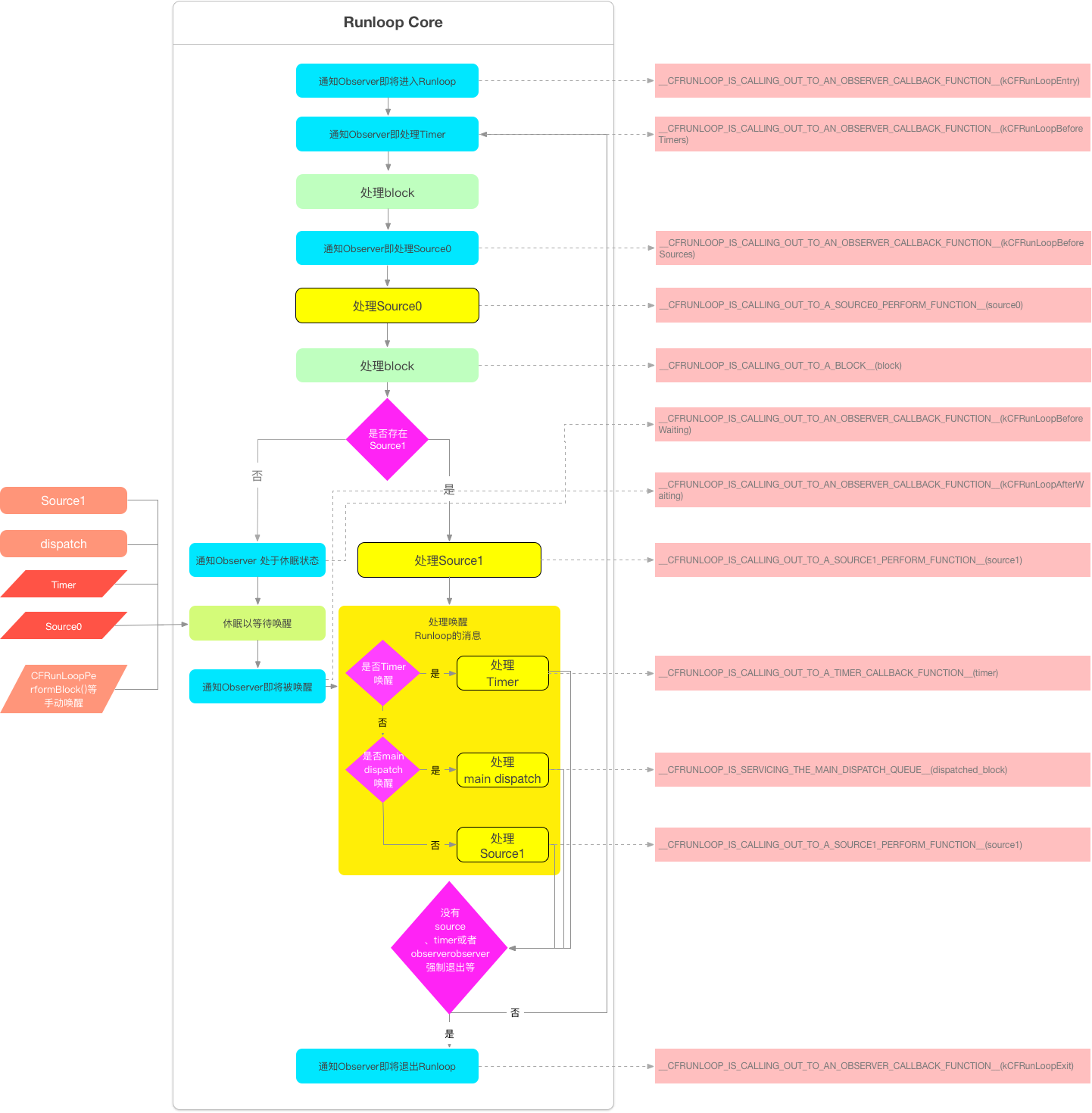
9、处理未处理的事件

**如果用户定义的定时器启动，处理定时器事件并重启run loop**。进入步骤2

如果输入源启动，传递相应的消息（**block事件和source1**）

**如果run loop被显式唤醒而且时间还没超时，重启run loop。进入步骤2**

10、通知观察者run loop结束。



## 5、Runloop与线程的关系？

一个线程对应一个RunLoop，主线程的RunLoop默认程序启动就已经创建好了。

子线程默认没有RunLoop，不过子线程可以有RunLoop，子线程的RunLoop得手动创建并且手动启动（调用run方法）

RunLoop在第一次获取时创建，在线程结束时销毁

注意：如果想给子线程添加一个与之对应的RunLoop，不能通过alloc、init方法，只能通过currentRunLoop，如果用alloc、init创建出来的RunLoop不能添加到子线程。

1.程序启动，底层会先创建一个字典。然后马上会创建一个主线程的RunLoop，并把主线程作为key，把主线程的RunLoop作为value添加到字典中。

注意：这也就是为什么一个线程对应一个RunLoop的原因，因为RunLoop是通过key-value的形式和线程以一一对应的方式保存在字典中的。

3. 每条线程都有唯一的一个与之对应的RunLoop对象。如果从子线程通过[NSRunLoop currentRunLoop]调用NSRunLoop的currentRunLoop方法的时候，系统会以子线程作为key，去字典中取对应的RunLoop对象。

4.如果取出来的RunLoop对象为空，则系统会创建一个RunLoop对象并以子线程作为key把该RunLoop对象存储到字典中去。

4、主线程的RunLoop已经自动创建，子线程的RunLoop需要主动创建。

5、RunLoop在第一次获取时创建，在线程结束时销毁。

## 6、Runloop 与autoreleasepool的关系？

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

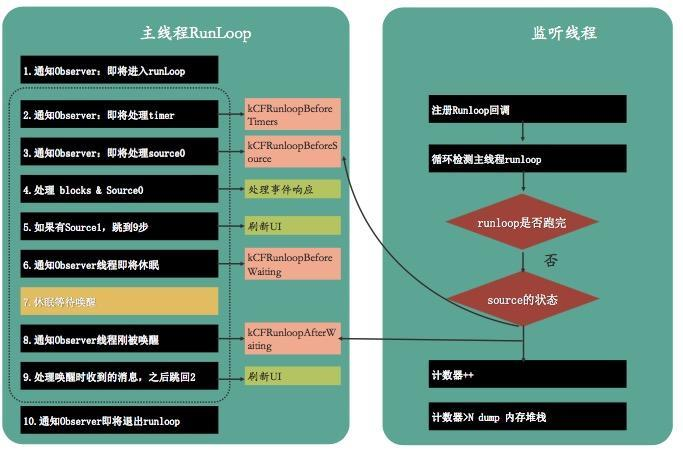
## 7、Runloop在网络请求中是如何工作的？

通常使用 NSURLConnection 时，你会传入一个 Delegate，当调用了 [connection **start**] 后，这个 Delegate 就会不停收到事件回调。实际上，**start 这个函数的内部会会获取 CurrentRunLoop，然后在其中的 DefaultMode 添加了4个 Source0 (即需要手动触发的Source)。**CFMultiplexerSource 是负责各种 Delegate 回调的，CFHTTPCookieStorage 是处理各种 Cookie 的。

当开始网络传输时，我们可以看到 NSURLConnection 创建了两个新线程：com.apple.NSURLConnectionLoader 和 com.apple.CFSocket.private。其中 CFSocket 线程是处理底层 socket 连接的。NSURLConnectionLoader 这个线程内部会使用 RunLoop 来接收底层 socket 的事件，并通过之前添加的 Source0 通知到上层的 Delegate。

NSURLConnectionLoader 中的 RunLoop 通过一些基于 mach port 的 Source 接收来自底层 CFSocket 的通知。当收到通知后，其会在合适的时机向 CFMultiplexerSource 等 Source0 发送通知，同时唤醒 Delegate 线程的 RunLoop 来让其处理这些通知。CFMultiplexerSource 会在 Delegate 线程的 RunLoop 对 Delegate 执行实际的回调。

## 8、Runloop如何检测卡顿？



会发现对于事件的处理主要就是在**kCFRunLoopBeforeSources和kCFRunLoopBeforeWaiting状态之间,还有kCFRunLoopAfterWaiting之后**。那我们就可以对在kCFRunLoopBeforeSources到kCFRunLoopBeforeWaiting再到kCFRunLoopAfterWaiting两个状态之间的超过了极限值和出现次数的场景进行监控，如果消耗时间太久，就代表着卡顿的发生。每50ms检测一次long st = dispatch\_semaphore\_wait(semaphore, dispatch\_time(DISPATCH\_TIME\_NOW, 50\*NSEC\_PER\_MSEC));

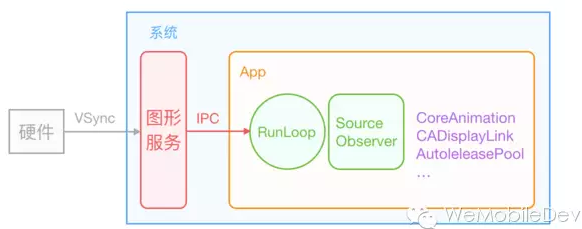
可以进行添加观察者，并统计一定时长来作为卡顿的依据

## 9、Runloop是如何实现休眠的？

RunLoop 的核心是基于 mach port 的，其进入休眠时调用的函数是 mach\_msg()

为了实现消息的发送和接收，mach\_msg() 函数实际上是调用了一个 Mach 陷阱 (trap)，即函数mach\_msg\_trap()，陷阱这个概念在 Mach 中等同于系统调用。当你在用户态调用 mach\_msg\_trap() 时会触发陷阱机制，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RunLoop 调用这个函数去接收消息，如果没有别人发送 port 消息过来，内核会将线程置于等待状态。例如你在模拟器里跑起一个 iOS 的 App，然后在 App 静止时点击暂停，你会看到主线程调用栈是停留在 mach\_msg\_trap() 这个地方。

## 10、图像是如何绘制的？



iOS 的显示系统是由 VSync 信号驱动的，VSync 信号由硬件时钟生成，每秒钟发出 60 次。**App 的 Runloop 在启动后会注册对应的 CFRunLoopSource 通过 mach\_port 接收传过来的时钟信号通知**，iOS 图形服务接收到 VSync 信号后，会通过 IPC 通知到 App 内。随后 Source 的回调会驱动整个 App 的动画与显示。

**Core Animation 在 RunLoop 中注册了一个 Observer，监听了 BeforeWaiting 和 Exit 事件**。当一个触摸事件到来时，RunLoop 被唤醒，App 中的代码会执行一些操作，比如创建和调整视图层级、设置 UIView 的 frame、修改 CALayer 的透明度、为视图添加一个动画；这些操作最终都会**被 CALayer 标记，并通过 CATransaction 提交到一个中间状态去**。当上面所有操作结束后，RunLoop 即将进入休眠（或者退出）时，关注该事件的 Observer 都会得到通知。这时 Core Animation 注册的那个 Observer 就会在回调中，把所有的中间状态合并提交到 GPU 去显示；如果此处有动画，通过 DisplayLink 稳定的刷新机制会不断的唤醒runloop，使得不断的有机会触发observer回调

在 VSync 信号到来后，系统图形服务会通过 CADisplayLink 等机制通知 App，App 主线程开始在 CPU 中计算显示内容，比如视图的创建、布局计算、图片解码、文本绘制等。随后 CPU 会将计算好的内容提交到 GPU 去，由 GPU 进行变换、合成、渲染。随后 GPU 会把渲染结果提交到帧缓冲区去，等待下一次 VSync 信号到来时显示到屏幕上。由于垂直同步的机制，如果在一个 VSync 时间内，CPU 或者 GPU 没有完成内容提交，则那一帧就会被丢弃，等待下一次机会再显示，而这时显示屏会保留之前的内容不变。这就是界面卡顿的原因。

setNeedsLayout/setNeedsDisplay方法后，这个 UIView/CALayer 就被标记为待处理，并被提交到一个全局的容器去。当Oberver监听的事件到来时，回调执行函数中会遍历所有待处理的UIView/CAlayer 以执行实际的绘制和调整，并更新 UI 界面

## 11、Runloop可监测的通知有哪些？

typedef CF\_OPTIONS(CFOptionFlags, CFRunLoopActivity) {

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

kCFRunLoopBeforeTimers = (1UL << 1), // 即将处理Timer：2

kCFRunLoopBeforeSources = (1UL << 2), // 即将处理Source：4

kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠：32

kCFRunLoopAfterWaiting = (1UL << 6), // 即将从休眠中唤醒：64

**kCFRunLoopExit** = (1UL << 7), // 即将从Loop中退出：128

kCFRunLoopAllActivities = 0x0FFFFFFFU // 监听全部状态改变

};

## 12、runloop伪代码实现？

function loop() {

initialize();

do {

var message = get\_next\_message();

process\_message(message);

} while (message != quit);

}

多线程

## 1、iOS实现多线程的方式有哪些？

NSThread， OperationQueue， GCD 、Operation

idle-time notification , NSObject asynchronous function（perform）, Timers, Separate process

Posix Thread，Mach Thread

## 2、iOS线程同步的工具有哪些？

**锁（lock）**：pthread\_mutex\_t、NSLock、@synchronized、NSRecursiveLock、NSConditionLock、NSDistributedLock

**条件（condition）**：是信号量的另外一种形式，允许在条件为真的时候线程间互相发送信号，用来同步的，而不是用来阻塞等待的。NSCondition、pthread\_cond\_t

**原子操作**（atomic operation）

**内存屏障**（memory barrier，OSMemoryBarrier函数）是一个使用来确保内存操作按照正确的顺序工作的非阻塞的同步工具

**Volatile变量**可以强制编译器每次使用变量的时候都从内存里面加载。如果一个变量的值随时可能给编译器无法检测的外部源更改，那么你可以把该变量声明为volatile变量。

## 3、iOS线程通信方式?

**Direct Messaging**：Cocoa提供了直接在其他线程上执行任务的方法，这意味着一个线程可以在另一个线程上执行方法，因为这些代码执行在目标线程，通过此方法发送的消息自动地被序列化。

**Global variable，shared memory objects**：

另一个线程间通讯的方式是使用全局变量，共享内存，共享对象，尽管这种方式比直接使用消息机制脆弱。全局变量，共享内存，对象需要使用锁或者其他同步工具进行确保资源的同步。不这样做的话会引起资源竞争问题。

**Condition:**可以使用condition来控制一个线程何时运行一段代码，只有当特定条件符合时一个线程才进行执行。

**Run loop sources:** 一个自定义的run loop 资源是用来设定处理应用特定消息的，它是事件驱动的，当没有事件处理里它会将线程置为休眠状态。

**Ports and socket:** 基于端口（port）的通信是两个线程进行通信的更为精细的方式，这是一种稳定的技术。更重要的是，端口（port）和socket可以用来与外部进行通信，如其他进程和服务，为了保持通信的高效，port是用runloop source来实现的，当没有数据时则处于休眠状态。

**Message queue：**多进程服务（multiprocessing service）定义了一个先进先出（FIFO）的队列用来管理输入和输出的数据，尽管很简单方便，但没有其他通信技术高效。

**Cocoa Distribute Objects: Distribute objects** 是对基于端口（port）通信的封装，尽管它可以用来线程间通信，但不建议这么做，因为它会浪费很多资源，Distribute Object更适合多进程间的通信。

## 4、线程成本有哪些？

多线程会占用你应用程序(和系统的)的内存使用和性能方面的资源。每个线程都需要分配一定的内核内存和应用程序内存空间的内存。管理你的线程和协调其调度所需的核心数据结构存储在使用Wired Memory的内核里面。你线程的堆栈空间和每个线程的数据都被存储在你应用程序的内存空间里面

**Kernel data structures:** 1KB的内存，这块内存被用来存储线程的数据结构和特性，大部分被分配为wired memory，所以不可以被paged to disk。

**Stack space:** iOS的主线程大约是1M，Mac主线程8M，第二个线程大约512KB。第二个线程最小为16KB，并且栈的大小是4KB的倍数。这个大小在进程空间创建线程的时候进行设置。

**Create Time：** 大约90微秒，这个时间是从开始调用线程到线程的入口函数开始执行的时间。

## 5、iOS有哪些锁?

**互斥锁（mutex）**：互斥，必须要进行等待

**递归锁（****Recursive lock）：**可以多次获取锁

**读写锁（Read-write lock）**：读比较频繁，写不频繁时候使用

**分布锁（Distributed lock）**：进程间的互斥锁，但它并不阻塞进程，而是告知资源正在使用，让进程自己决定如何处理

**自旋锁（Spin lock）**：在多处理器系统中，如果预期等待时间较短使用此锁，不断去获取condition的值

**双重检验锁（Double-checked lock）**：不安全，系统未明确支持

## 6、GCD的三种队列是什么？

1. **main**，即主线程，iOS的UI绘制、交互响应都要在此线程上执行

2. **concurrent**，并发线程队列，task的进出队列，都遵循FIFO策略，但队列中的task‘看起来’是并发执行的完成时间却可能是以任意的顺序结束。很容易理解，因为每个task的执行时间长短通常不一样。

3. **serial，**串行队列，在此队列中的task，one by one的执行，严格遵守FIFO

## 7、iOS系统提供的5中dispatch queue是什么？

Main Dispatch Queue: Serial Dispatch Queue　　Ｑos: NSQualityOfServiceUserInteractive

DISPATCH\_QUEUE\_PRIORITY\_HIGH: Concurrent Qos: NSQualityOfServiceUserInitiated

DISPATCH\_QUEUE\_PRIORITY\_DEFAULT: Concurrent Qos: NSQualityOfServiceDefault

DISPATCH\_QUEUE\_PRIORITY\_LOW: Concurrent Qos: NSQualityOfServiceUtility

DISPATCH\_QUEUE\_PRIORITY\_BACKGROUND Concurrent Qos: NSQualityOfServiceBackground

## 8、NSOperation和GCD如何选择？

1、NSOperationQueue 是在GCD基础上实现的，只不过是GCD更高一层的抽象。

2，在NSOperationQueue中，可以建立各个NSOperation之间的依赖关系和优先级。

3，NSOperationQueue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）

4，GCD 只支持FIFO 的队列，而NSOperationQueue可以调整队列的执行顺序, 可以设置异步串行（maxConcurrentOperationCount）。

OperationQueue更适合复杂逻辑下的多线程编程，GCD更简洁易用

## 9、如何使用NSOperation？

NSOperation是抽象类，必须要使用它的子类，系统提供的两个子类：

NSInvocationOperation

NSBlockOperation（也可NSOperationQueue直接addOperationWithBlock）

支持在 operation 之间建立依赖关系，只有当一个 operation 所依赖的所有 operation 都执行完成时，这个 operation 才能开始执行；

支持一个可选的 completion block ，这个 block 将会在 operation 的主任务执行完成时被调用；

支持通过 KVO 来观察 operation 执行状态的变化；

支持设置执行的优先级，从而影响 operation 之间的相对执行顺序；

支持取消操作，可以允许我们停止正在执行的 operation 。

## 10、如何自定义Operation的子类

自定义non-Concurrent Operaion：

在 main 方法写逻辑即可，需要注意isCancelled这个关键点

自定义并发执行的Operation：

**start** ：必须的，所有并发执行的 operation 都必须要重写这个方法，替换掉 NSOperation 类中的默认实现。start 方法是一个 operation 的起点，我们可以在这里配置任务执行的线程或者一些其它的执行环境。

**main** ：可选的，通常这个方法就是专门用来实现与该 operation 相关联的任务的。尽管我们可以直接在 start 方法中执行我们的任务，但是用 main 方法来实现我们的任务可以使设置代码和任务代码得到分离，从而使 operation 的结构更清晰；

**isExecuting 和 isFinished** ：必须的，并发执行的 operation 需要负责配置它们的执行环境，并且向外界客户报告执行环境的状态。生成相应的 KVO 通知，以便外界能够观察到这些状态的变化；

**is****Concurrent** ：必须的，这个方法的返回值用来标识一个 operation 是否是并发的 operation ，返回 YES

## 11、非并发NSOperation和 并发 NSOperation 的区别？

在NSOperationQueue运行NSOperation时都是并发的，两个区别在于：

Concurrent NSOperation不需要NSOperationQueue来为它提供线程，所有逻辑在自己的线程，

Non-Concurrent NSOperaion 需要NSOperaionQueue来为它提供线程。

## 12、原子操作时线程安全的吗？

atomic的操作是原子性的，但是并不意味着它是线程安全的，它会增加正确的几率，能够更好的避免线程的错误，但是它仍然是线程不安全的。

线程安全就是在多线程环境下也不会出现数据不一致，而非线程安全就有可能出现数据不一致的情况。

当使用nonatomic的时候，属性的setter，getter操作是非原子性的，所以当多个线程同时对某一属性读和写操作时，属性的最终结果是不能预测的。

仅仅使用atomic并不会使得线程安全，它仅仅保证属性的读写原子操作，不保证线程安全，我们还要为线程添加lock来确保线程的安全。

也就是要注意：**atomic所说的线程安全只是保证了getter和setter存取方法的线程安全，并不能保证整个对象是线程安全的。**

~~不过atomic可并不能保证线程安全。如果线程 A 调了 getter，与此同时线程 B 、线程 C 都调了 setter——那最后线程 A get 到的值，3种都有可能：可能是 B、C set 之前原始的值，也可能是 B set 的值，也可能是 C set 的值。同时，最终这个属性的值，可能是 B set 的值，也有可能是 C set 的值。~~

~~，getter是不会对属性进行retain的，假设当getter执行后，切换到另一个线程，执行setter，setter会对oldValue release，导致oldValue释放。再切回执行getter的线程，getter用到的是已经释放的oldValue。就会发生EXC\_BAD\_ACCESS的crash。~~

首先你得知道什么是线程不安全，线程的不安全是由于多线程访问和修改共享资源而引起的不可预测的结果(有可能crash)。可以简单理解为我们拿到的值是错的。

atomic ：系统自动生成的getter/setter方法会进行加锁操作；可以理解过读写锁，可以保证读写安全；较耗时；

nonatomic : 系统自动生成的getter/setter方法不会进行加锁操作；但速度会更快；\

虽然stringA是atomic的property，而且在取substring的时候做了length判断，线程B还是很容易crash，因为在前一刻读length的时候self.stringA = @"a very long string";，下一刻取substring的时候线程A已经将self.stringA = @"string";，立即出现out of bounds的Exception，crash，多线程不安全。

一般情况下，getter执行后，会在外部对getter获取的属性进行retain，也就是调用objc\_retain。但是也许就在getter发生之后，objc\_retain之前其他线程执行了setter。这时候，就会导致objc\_retain产生EXC\_BAD\_ACCESS，例如堆栈1，这种crash发生的概率不大，但是架不住SDWebImage被频繁使用。特别是feed上，频繁调用SDWebImage，导致多线程切换。

## 13、OSSpinLock为什么不安全了？使用什么来代替？

系统维护了 5 个不同的线程优先级/QoS: background，utility，default，user-initiated，user-interactive。高优先级线程始终会在低优先级线程前执行，一个线程不会受到比它更低优先级线程的干扰。这种线程调度算法会产生潜在的优先级反转问题，从而破坏了 Spin lock。

如果一个低优先级的线程获得锁并访问共享资源，这时一个高优先级的线程也尝试获得这个锁，它会处于 spin lock 的忙等状态从而占用大量 CPU。此时低优先级线程无法与高优先级线程争夺 CPU 时间，从而导致任务迟迟完不成、无法释放 lock。使用os\_unfaire\_lock来代替，但os\_unfair\_lock性能并不是很高

**os\_unfair\_lock\_t** unfairLock;

unfairLock = &(OS\_UNFAIR\_LOCK\_INIT);

os\_unfair\_lock\_lock(unfairLock);

os\_unfair\_lock\_unlock(unfairLock);

## 14、锁和条件有什么不同？（NSCondition和NSLock）有什么不同

锁是保证某个对象或者代码块某段时间只能被一个线程访问，条件则是满足条件后执行某块逻辑，一个更偏向处理从运行到等待的逻辑，一个偏向处理从等待到运行的逻辑，由于从等待到运行一般也需要一个条件，这个条件也需要线程安全，所以进行条件操作都需要先进行锁操作。

## 15、NSCondition和NSConditionLock有什么区别？

NSCondition提供了条件的基本API

- (void)wait;//挂起线程

- (BOOL)waitUntilDate:(NSDate \*)limit;

- (void)signal; //任意通知一个线程

- (void)broadcast; //通知所有等待的线程

NSConditionLock对条件的类型进行了绑定。比如说当和内部条件相同才会进行锁操作：

lockWithCondition:

unlockWithCondition:

## 16、如何实现固定数量的线程池？

1、使用NSOperationQueue的setMaxCocurrentOperationCount

2、自己建立多路复用队列。

循环通过pthread\_create创建线程，加入线程数组, 始化函数后，在执行函数中通过**pthread\_cond\_wait**信号将当前创建的线程挂起，程序中将会有n个挂起状态的线程，当需要执行新的task的时候查找，我们就可以查询出空闲线程，通过**pthread\_cond\_signal**重新唤醒该线程继续执行任务，如果所有线程都在使用，可放在缓冲队列中。

## 17、如何终止正在运行的工作线程？

在线程中调用exit、pthread\_exit、pthread\_kill、pthread\_cancel

**void pthread\_exit(void \*retval);**

调用了pthread\_join的线程会一直阻塞直到它指定的线程调用pthread\_exit、从启动例程中返回或者被取消。如果线程从启动它的例程返回，retval将包含返回码。若线程被取消则retval指定的内存空间会被置为PTHREAD\_CANCELED

**int pthread\_cancel(pthread\_t thread);**

**void pthread\_cleanup\_push(void (\*routine)(void \*), void \*arg);**

**void pthread\_cleanup\_pop(int execute);**

线程可以调用pthread\_cancel函数来请求取消同一进程中的其他线程。在默认情况下**pthread\_cancel函数会使由thread标识的线程的行为表现为如同调用了参数为PTHREAD\_CANCELED的pthread\_exit函数**，必须特别注意的是，pthread\_cancel并不等待线程的终止。它仅仅提出请求。后两个函数被称为线程清理处理函数，线程可以建立多个清理处理函数。处理程序记录在栈空间，所以它们的执行顺序与它们的注册时的顺序相反.

**int pthread\_kill(pthread\_t thread, int sig);**

请在创建的线程中使用**signal(SIGKILL,sig\_handler)处理信号，如果你给一个线程发送了SIGQUIT，但线程却没有实现signal处理函数，则整个进程退出。**

## 18、如何等待一个线程处理完然后再进行处理相应的逻辑？

**1、dispatch\_group\_notify ()和dispatch\_group\_async();**

dispatch\_group\_async(disgroup, disqueue, ^{

NSLog(@"任务一完成");

});

**2、NSLock和NSCondition的使用**

**3、pthread\_mutex\_lock和pthread\_cond\_t的使用**

**4、函数pthread\_join用来等待一个线程的结束,线程间同步的操作。**头文件 ： #include <pthread.h>

**函数定义：** int pthread\_join(pthread\_t thread, void \*\*retval);

**描述 ：**pthread\_join()函数，以阻塞的方式等待thread指定的线程结束。当函数返回时，被等待线程的资源被收回。如果线程已经结束，那么该函数会立即返回。并且thread指定的线程必须是joinable的。

**5、GCD进行多个异步操作，dispatch\_async\_barrier()**

**6、dispatch\_semaphore\_wait、dispatch\_semaphore\_signal**

## 19、常见的死锁情况

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

NSLog(@"on main thread!");

});

## 20、如何让主线程等待子线程结束？

pthread\_join使一个线程等待另一个线程结束。

　　代码中如果没有pthread\_join；主线程会很快结束从而使整个进程结束，从而使创建的线程没有机会开始执行就结束了。加入pthread\_join后**，主线程会一直等待直到等待的线程结束自己才结束，使创建的线程有机会执行。**

　　所有线程都有一个线程号，也就是threadid，其类型为pthread\_t。 通过调用pthread\_self()函数可以获得自身的线程号。

　　如果你的主线程，也就是main函数执行的那个线程，在你其他线程退出之前就已经退出，那么带来的bug则不可估量。通过pthread\_join函数会让主线程阻塞，直到所有线程都已经退出。

int pthread\_join(pthread\_t thread, void \*\*value\_ptr);

## 21、什么是joinable和detached 线程？

线程是可结合的**（joinable）**，或者**是分离的（detached）**。一个可结合的线程能够被其他线程收回其资源和杀死；在被其他线程回收之前，它的存储器资源（如栈）是不释放的。相反，一个分离的线程是不能被其他线程回收或杀死的，它的存储器资源在它终止时由系统自动释放。

**通过NSThread创建的线程都是Detached的。如果想要创建joinable线程，只能通过POSIX线程接口创建。**

## 22、线程有哪些参数可以配置？

**1、配置线程栈空间**

栈空间是用来存储为线程创建的本地变量的，栈空间的大小必须在线程的创建之前设定。不能使用创建线程的第一和第二种方法。在调用**NSThread的start方法之前通过setStackSize:** 设定新的栈空间大小。

**2、配置线程的本地存储**

每个线程都维护一个在线程任何地方都能获取的字典。 我们可以使用**NSThread的threadDictionary**方法获取一个NSMutableDictionary对象，然后添加我们需要的字段和数据。

**3、设置线程的Detached状态**

通过NSThread创建的线程都是Detached的。如果想要创建joinable线程，只能通过POSIX线程接口创建。

**4、设置线程的优先级**

每一个新的线程都有一个默认的优先级。系统的内核调度算法根据线程的优先级来决定线程的执行顺序。通常情况下我们不要改变线程的优先级，提高一些线程的优先级可能会导致低优先级的线程一直得不到执行，如果在我们的应用内存在高优先级线程和低优先级线程的交互的话，因为低优先级的线程得不到执行可能阻塞其他线程的执行。这样会对应用造成性能瓶颈。可以通过**NSThread的setThreadPriority**:方法设置线程优先级，优先级为0.0到1.0的double类型，1.0为最高优先级。

其他pthread线程属性：

优先级、堆栈空间、名字、线程组、detach state等

## 23、NSThread的配置和退出？

**1、配置自动释放池**

- (void)myThreadMainRoutine

{

@autoreleasepool

{

//do thread task

}

}

**2、退出**

从NSThread的threadDictionary中获取变量，通过变量来控制runloop

// Add the exitNow BOOL to the thread dictionary.

NSMutableDictionary\* threadDict = [[NSThread currentThread] **threadDictionary]**;

[threadDict setValue:[NSNumber numberWithBool:exitNow] forKey:@"ThreadShouldExitNow"];

// Install an input source.

[self myInstallCustomInputSource];

**while (moreWorkToDo && !exitNow)**

{

// Do one chunk of a larger body of work here.

// Change the value of the moreWorkToDo Boolean when done.

// Run the run loop but timeout immediately if the input source is not waiting to fire.

**[runLoop runUntilDate:[NSDate date]];**

// Check to see if an input source handler changed the exitNow value.

**exitNow = [[threadDict valueForKey:@"ThreadShouldExitNow"] boolValue];**

}

}

## 24、线程的私有数据有哪些？

**线程的私有数据**：有些数据资源，虽然在程序中定义为全局常量，但线程间不能共享，而是由每个线程自己为这些数据资源进行分配存储空间。

线程标识符、线程优先级、security information、库列表、signal blocking mask、call stack、automatic storage(本地变量存储)、错误信息

# CoreGraphic

## 1、获取Context的方法有哪些?

UIGraphicsGetCurrentContext

UIGraphicsBeginImageContextWithOptions

## 2、绘图方式有哪些?

**第一种绘图形式：**在UIView的子类方法drawRect, 使用UIKit在Cocoa为我们提供的当前上下文

**第二种绘图形式**：使用Core Graphics实现绘制

**第三种绘图形式：**将在UIView子类的drawInContext：方法中实现绘图任务。drawLayer:inContext：方法是一个绘制图层内容的代理方法。为了能够调用drawLayer:inContext：方法，我们需要设定图层的代理对象。

**第四种绘图形式**： 使用Core Graphics在drawLayer:inContext：方法中实现同样操作

**第五种绘图形式**： 使用UIKit实现

UIGraphicsBeginImageContextWithOptions(CGSizeMake(100,100), NO, 0);

\*\*\*\*\*\*\*\*\*\*\*\*

UIImage\* im = UIGraphicsGetImageFromCurrentImageContext();

UIGraphicsEndImageContext();

## 3、绘图系统框架主要有哪些?

UIKit，还有CoreGraphics、Core Animation，Core Image，OpenGL ES等多种框架

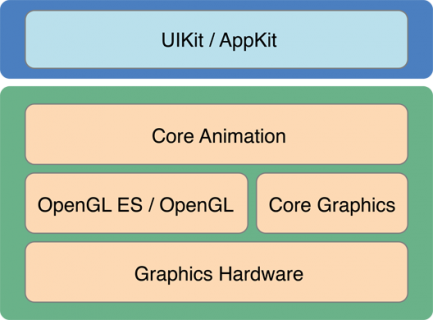
**UIKit：**最常用的视图框架，封装度最高，都是OC对象

**CoreGraphics：**主要绘图系统，常用于绘制自定义视图，纯C的API，使用Quartz2D做引擎

**CoreAnimation：**提供强大的2D和3D动画效果

**CoreImage：**给图片提供各种滤镜处理，比如高斯模糊、锐化等

**OpenGL-ES：**主要用于游戏绘制，但它是一套编程规范，具体由设备制造商实现



## 4、绘图步骤？

实际的绘图包括两部分：**视图绘制和视图布局**

**绘图周期：**

iOS在运行循环中会整合所有的绘图请求，并一次将它们绘制出来

不能在子线程中绘制，也不能进行复杂的操作，否则会造成主线程卡顿

**1.视图绘制**

调用UIView的drawRect:方法进行绘制。如果调用一个视图的setNeedsDisplay方法，那么该视图就被标记为重新绘制，并且会在下一次绘制周期中重新绘制，自动调用drawRect:方法。

**2.视图布局**

调用UIView的layoutSubviews方法。如果调用一个视图的setNeedsLayout方法，那么该视图就被标记为需要重新布局，UIKit会自动调用layoutSubviews方法及其子视图的layoutSubviews方法。

## 5、绘图状态的变化？

**1、设置绘图的上下文环境（context）**

push：UIGraphicsPushContext(context)把context压入栈中，并把context设置为当前绘图上下文

pop：UIGraphicsPopContext将栈顶的上下文弹出，恢复先前的上下文，但是绘图状态不变

**2、设置绘图的状态（state）**

save：CGContextSaveGState 压栈当前的绘图状态，仅仅是绘图状态，不是绘图上下文

restore：恢复刚才保存的绘图状态

**3、context / imageContext**

context：图形上下文，可以通过UIGraphicsGetCurrentContext:获取当前视图的上下文

imageContext：图片上下文，可以通过UIGraphicsBeginImageContextWithOptions:获取一个图片上下文，然后绘制完成后，调用UIGraphicsGetImageFromCurrentImageContext获取绘制的图片，最后要记得关闭图片上下文UIGraphicsEndImageContext。

**4、CGPathRef / UIBezierPath**

图形的绘制需要绘制一个路径，然后再把路径渲染出来，而CGPathRef就是CoreGraphics框架中的路径绘制类，UIBezierPath是封装CGPathRef的面向OC的类，使用更加方便，但是一些高级特性还是不及CGPathRe

多媒体

## 1、iOS系统提供哪些播放视频的方式？

MPMoviePlayerController、AVPlayer、**AVAssetReader+AVAssetReaderTrackOutput**自己实现、

AVPlayerViewController（iOS8）。**AVPlayerItemVideoOutput**获取image使用CADisplayLink

## 2、iOS多媒体框架有哪些？它们的关系是什么？

CoreMedia、CoreVideo、CoreAudio、AVFoundation、AudioToolbox、MediaPlayer、AVKit、Video Toolbox

**CoreVideo:** 当需要进行编辑每一帧视频、需要使用CoreImageFilter时，使用pipeline API，支持OpenGL和Metal，相关数据结构CVBuffer、CVImageBuffer、CVPixelBuffer、CVPixelBufferPool、CVPixelFormatDescription、CVTime、CVDisplayLink、

**CoreMedia**：提供给上层框架使用的数据类型

**CoreImage**： 主要是图片的ImageFilter吗，CIImage 它们可以代表两种东西：图像数据或者生成图像数据的流程。CIFilter 的输出非常轻量。它只是如何被创建的描述，并不包含任何实际的像素数据。

**AVKit：** AVPlayerViewController

**AVFoundation**：录制播放等相关API

**CoreAudio**: 操作音频流的框架

**VideoToolbox: 视频编解码**

音频播放：**AVAudioPlayer**

音频录制：**AVAudioRecorder**，录制配置可以使用**AVAudioSession**

## 3、是否使用过CoreText或者CoreImage等？如果使用过，请谈谈你使用CoreText或者CoreImage的体验。

coretext图文混排，绘制过富媒体文本，CoreImage是一个图像框架，它基于OpenGL顶层创建，底层则用着色器来处理图像，这意味着它利用了GPU基于硬件加速来处理图像。CoreImage中有很多滤镜，它们能够一次给予一张图像或者视频帧多种视觉效果。而且滤镜可以连接起来组成一个滤镜链，把滤镜效果叠加起来处理图像, CIContext, CIFilter

## 4、如何实现视频的边下边播？

一个对象实现了 **AVAssetResourceLoaderDelegate** 这个协议的方法，赋给给 AVAsset的resourceLoader属性的delegate，再把 asset 丢给 AVPlayer，AVPlayer 会在播放的时候执行 delegate方法：

func **resourceLoader**(\_ resourceLoader: AVAssetResourceLoader, **shouldWaitForLoadingOfRequestedResource** loadingRequest: AVAssetResourceLoadingRequest) -> Bool

func **resourceLoader**(\_ resourceLoader: AVAssetResourceLoader, **didCancel** loadingRequest: AVAssetResourceLoadingRequest)

在 AVAssetResourceLoadingRequest 里面，request 代表原始的请求，由于 AVPlayer 是会触发分片下载的策略**，还需要从dataRequest 中得到请求范围的信息**

AVAssetResourceLoader通过你提供的委托对象去调节AVURLAsset所需要的加载资源。而很重要的一点是，**AVAssetResourceLoader仅在AVURLAsset不知道如何去加载这个URL资源时才会被调用，就是说你提供的委托对象在AVURLAsset不知道如何加载资源时才会得到调用（AVAsset使用自定义的URL创建）**

## 5、如何录制视频？相关API和步骤有哪些？

1、**UIImagePickerController**

**2、****AVCaptureSession+****AVCaptureMovieFileOutput**

**3、AVCaptureSession+AVAssetWriter**

创建AVCaptureSession，设置拍摄分辨率

添加AVCaptureInput，如摄像头和麦克风

添加AVCaptureOutput，如AVCaptureVideoDataOutput、AVCaptureAudioDataOutput。这里AVCaptureAudioDataOutput建议在Session -startRunning后才添加，避免影响摄像头启动时间

添加AVCaptureVideoPreviewLayer，为用户提供拍摄预览界面

AVAssetWriter对象，用于写视频

开始捕捉-startRunning

**后两种进行对比：**

**相同点**：数据采集都在AVCaptureSession中进行，视频和音频的输入都一样，画面的预览一致

**不同点**：

输出不一致, **AVCaptureMovieFileOutput 只需要一个输出即可**，指定一个文件路后，视频和音频会写入到指定路径，不需要其他复杂的操作。

**AVAssetWriter 需要 AVCaptureVideoDataOutput 和 AVCaptureAudioDataOutput** 两个单独的输出，拿到各自的输出数据后，然后自己进行相应的处理。

**视频剪裁不一致**，AVCaptureMovieFileOutput 如果要剪裁视频，因为系统已经把数据写到文件中了，我们需要从文件中独到一个完整的视频，然后处理；而AVAssetWriter我们拿到的是数据流，还没有合成视频，对数据流进行处理，所以两则剪裁方式也是不一样。

## 6、iOS播放器播放音频的方法有哪些？

System Sound Services

AVAudioPlayer类

Audio Queue Services

OpenAL

## 7、视频截取和拼接相关API有哪些？

**AVAssetExportSessio**n指定timerange属性后，然后执行exportAsynchronouslyWithComletionHandler来进行裁剪导出

**相关类**：

**AVMutableComposition：** 用来操作音频和视频的组合

**AVMutableVideoComposition：** 用来对视频进行操作

**AVMutableAudioMix：** 给视频添加音频的

**AVMutableVideoCompositionInstruction：** 配合使用，用来给视频添加水印或者旋转视频方向

**AVMutableVideoCompositionLayerInstruction：**配合使用，用来给视频添加水印或者旋转视频方向

**AVAssetExportSession** ：用来进行视频导出操作的。需要值得注意的是当App进入后台之后，会对使用到GPU的代码操作进行限制，会造成崩溃，而视频处理这些功能多数会使用到GPU,所以需要做对应的防错处理。

1. 拿到视频和音频资源

2. 创建AVMutableComposition对象

3. 往AVMutableComposition对象添加视频资源，同时设置视频资源的时间段和插入点

4. 往AVMutableComposition对象添加音频资源，同时设置音频资源的时间段和插入点

5、为AVMutableVideoComposition添加instruments进行视频旋转

6、使用AVMutableVideoComposition设置animationTool来进行添加相关水印

## 8、本地视频如何加特效？GPUImage处理

**GPUImageMovie** 接收需要添加滤镜的视频

**GPUImageView** 预览视频效果

**GPUImageOutput** 视频输入输出配置，这里承载滤镜

**GPUImageMovieWriter** 添加滤镜及输出视频

**1、预览：**

[\_gpuMovie cancelProcessing];

[\_gpuMovie removeAllTargets];

\_gpuMovie = [[GPUImageMovie alloc]initWithURL:\_filePath];

[\_gpuMovie addTarget:\_pixellateFilter];

[\_pixellateFilter addTarget:\_gpuView];

[\_gpuMovie startProcessing];

**2、合成**

NSURL \*movieURL = [NSURL fileURLWithPath:self.fileSavePath];

\_movieWriter = [[GPUImageMovieWriter alloc] initWithMovieURL:movieURL size:CGSizeMake(WIDTH, WIDTH\*744/720-30)];

**[\_pixellateFilter addTarget:\_movieWriter];**

\_movieWriter.shouldPassthroughAudio = YES;

[**\_gpuMovie enableSynchronizedEncodingUsingMovieWrite**r:\_movieWriter];

[\_movieWriter startRecording];

## 9、本地图片如何使用GPUImage加特效？

**1.创建一个UIImage对象**

UIImage \*inputImage = [UIImage imageNamed:@"图片的名字"];​

**2.创建一个滤镜**

GPUImageSketchFilter \*passthroughFilter = [[GPUImageSketchFilter alloc] init];​

**3.设置参数(要渲染的区域)**

[passthroughFilter forceProcessingAtSize:inputImage.size];

[passthroughFilter useNextFrameForImageCapture];​

**4.获取数据源(将UIImage对象给GPUImagePicture)​**

**GPUImagePicture** \*stillImageSource = [[GPUImagePicture alloc] initWithImage:inputImage];​

**5.给数据源添加滤镜**

[stillImageSource addTarget:passthroughFilter];​

**6.开始渲染​**

[stillImageSource processImage];​

**7.获取渲染后的图片**

UIImage \*newImage = [**passthroughFilter** **imageFromCurrentFramebuffer**];​

**8.将渲染后的图片展示出来**

UIImageView \*imageView = [[UIImageViewalloc] initWithImage:newImage];

imageView.frame = CGRectMake(0,0,inputImage.size.width, inputImage.size.height);

[self.view addSubview:imageView];

## 10、实时视频如何加特效？

实现的基本原理是GPUImageStillCamera/GPUImageVideoCamera(照片/视频)捕获图像->GPUImageFilter滤镜->GPUImageView显示. 拍摄照片使用 GPUImageStillCamera

**1、初始化Camera**

[[**GPUImageVideoCamera** alloc] initWithSessionPreset:AVCaptureSessionPreset640x480 cameraPosition:AVCaptureDevicePositionBack];

设置显示时的方向.outputImageOrientation = UIInterfaceOrientationPortrait;

**2、addTarget添加滤镜**

**3、滤镜addTarget设置显示的GPUImageView**

**4、设置camera的audioEncodingTarget = GPUImageMovieWriter；**

**5、设置GPUImageMovieWriter的encodingLiveVideo为YES**

**6、camera 调用startCameraCapture来进行录像，****GPUImageMovieWriter调用startRecording开始录像**

**7、调用GPUImageMovieWriter的finishRecording来停止录制，使用ALAssetsLibrary来进行保存**

## 11、GPUImage的实现原理是什么？

滤镜处理的原理就是把静态图片或者视频的每一帧进行图形变换再显示出来。它的本质就是像素点的坐标和颜色变化，即顶点着色器和片段着色器

## 12、如何进行硬编码H264格式的视频？

1、调用**VTCompressionSessionCreate**创建编码session，并制定编码函数，然后调用VTSessionSetProperty设置参数，最后调用VTCompressionSessionPrepareToEncodeFrames开始编码；

2、开始视频录制，在代理的captureOutput:(AVCaptureOutput \*)captureOutput didOutputSampleBuffer:(CMSampleBufferRef)sampleBuffer fromConnection方法中利用CMSampleBufferGetImageBuffer获取到摄像头的视频帧（**将CMSampleBufferRef转化成CVImageBufferRef**），调用VTCompressionSessionEncodeFrame传入需要编码的视频帧，如果返回失败，调用VTCompressionSessionInvalidate销毁session，然后释放session；

3、每一帧视频编码完成后会调用预先设置的编码函数didCompressH264，如果是关键帧需要用CMSampleBufferGetFormatDescription获取CMFormatDescriptionRef，然后用

CMVideoFormatDescriptionGetH264ParameterSetAtIndex**取得PPS和SPS**；

最后把每一帧的所有NALU数据前四个字节变成0x00 00 00 01之后再写入文件；

4、调用VTCompressionSessionCompleteFrames完成编码，然后销毁session：VTCompressionSessionInvalidate，释放session

## 13、如何进行硬解码H264格式的视频？

如果输出时图像数据的话，它可能是纯像素的 NSData，一个 CGImage， 一个 CVPixelBuffer，或者是一个 OpenGL 纹理。

**CVImageBuffer**：这是 CVPixelBuffer (CPU) 和 CVOpenGLESTexture (GPU) 的抽象父类.

**CMSampleBuffer**：Core Media 采样缓冲 (Sample Buffer) 是 CMBlockBuffer 或者 CVImageBuffer 的包装，也包括了元数据。

**CVPixelBuffer:** 包含未压缩的像素数据，包括图像宽度、高度等， 像素缓冲 (Pixel Buffer) 是基于 CPU 的。

**CVPixelBufferPool**：CVPixelBuffer的缓冲池，因为CVPixelBuffer的创建和销毁代价很大；

CMBlockBuffer： 未压缩的图像数据；

**CMSampleBuffer**： 存放一个或者多个压缩或未压缩的媒体文件；即有多个CMBlockBuffer

用NSInputStream读入原始H.264码流，用CADisplayLink控制显示速率，用NALU的前四个字节识别SPS和PPS并存储，当读入IDR帧的时候初始化VideoToolbox，并开始同步解码；解码得到的CVPixelBufferRef会传入OpenGL ES类进行解析渲染；

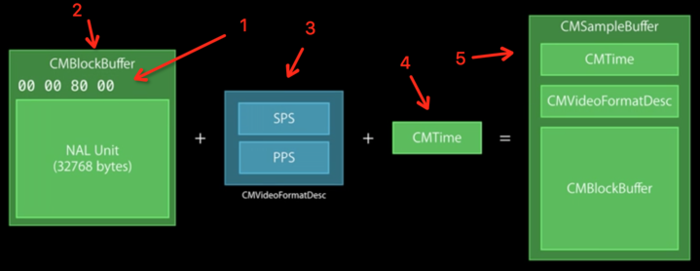
**1、替换头字节长度**

**2、用CMBlockBuffer把NALUnit包装起来；**

**3、把SPS和PPS包装成CMVideoFormatDescription；**

**4、添加CMTime时间；**

**5、创建CMSampleBuffer**；



1、**在VTDecompressionSessionDecodeFrame传入 CMSampleBuffer**

**2、回调解码函数didDecompress**

**3、OpenGLView调用displayPixelBuffer来显示解码的结果**

当遇到IDR帧时，更合适的做法是通过

VTDecompressionSessionCanAcceptFormatDescription判断原来的session是否能接受新的SPS和PPS，如果不能再新建session。

## 14、简单介绍下H264的I、B、P、IDR帧？

H.264由视讯编码层(Video Coding Layer，VCL)与网络提取层(Network Abstraction Layer，NAL)组成。

H.264包含一个内建的NAL网络协议适应层，由NAL来提供网络的状态，让VCL有更好的编译码弹性与纠错能力。

**I帧:**帧内编码帧是一种自带全部信息的独立帧，无需参考其它图像便可独立进行解码，视频序列中的第一个帧始终都是I帧。

**I和IDR帧都是使用帧内预测的**。它们都是同一个东西而已,在编码和解码中为了方便，要首个I帧(0x65)和其他I帧(0x41)区别开，所以才把第一个首个I帧叫IDR，这样就方便控制编码和解码流程。

**P帧**: 前向预测编码帧：在针对连续动态图像编码时，将连续若干幅图像分成P,B,I三种类型，P帧由在它前面的P帧或者I帧预测而来，它比较与它前面的P帧或者I帧之间的相同信息或数据，也即考虑运动的特性进行帧间压缩

**B帧**：双向预测内插编码帧，B帧法是双向预测的帧间压缩算法。当把一帧压缩成B帧时，它根据相邻的前一帧、本帧以及后一帧数据的不同点来压缩本帧

## 15、简单介绍下CoreAudio框架的使用

iOS的CoreAudio是通过callback函数与App交互的。其中需要设置回调函数有以下几种情况：

CoreAudio会向回调函数给App传入PCM音频数据，然后App需要在回调函数中将音频数据写入文件文件系统。（录音时候）

CoreAudio会需要向App请求一些音频数据，App通过从文件系统中读取音频数据，然后通过callback函数传递给CoreAudio。（播放时候）

通过注册属性观察者，监听CoreAudio的属性，注册回调函数

使用AudioStreamBasicDescription和AudioStreamPacketDescription这两个类型描述了通用的音频数据类型

使用AudioQueueAddPropertyListener注册指定状态（如kAudioQueueProperty\_IsRunning）的回调函数。

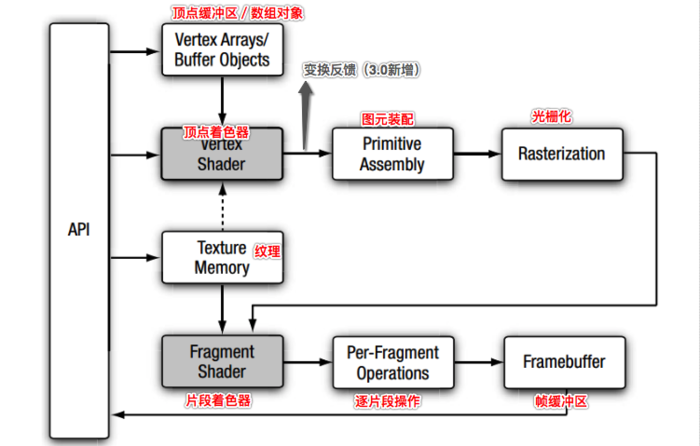
在整个Core Audio中可能会用到三种不同的packets：

CBR (constant bit rate) formats：例如 linear PCM and IMA/ADPCM，所有的packet使用相同的大小。

VBR (variable bit rate) formats：例如 AAC，Apple Lossless，MP3，所有的packets拥有相同的frames，但是每个sample中的bits数目不同。

VFR (variable frame rate) formats：packets拥有数目不同的的frames。

## 16、OpenGL绘制过程及片段着色器、定点着色器？



1、构建顶点缓冲区或数组对象

2、**顶点着色器处理**：处理顶点

**3、图元装配：**图元 (Primitive) 在OpenGL ES 中就是基本的绘图对象，三角形、直线和点，在 图元装配期间，这些顶点被组合成图元

4、**光栅化**：光栅化用来将图元映射到像素设备

5、**片段着色器处理**：片段着色器可以抛弃片段，也可以生成一个或者多个颜色值作为输出，输出最终的像素颜色

6、**逐片段操作**：

**裁剪测试**：把坐标（x, y）的像素如果超出OpenGL ES的裁剪范围，就会被抛弃。

**模版测试**：对输入Fragment Data的模版值进行测试，看这个是否应该拒绝

**深度测试**：对输入Fragment Data的深度值进行测试，比如3D中的远近效果，查看Frame Buffer中的数据和当前数据的深度进行比较，远的丢掉，近的保留

**混合**：将输入Fragment Data中的颜色值 和 保存在Frame Buffer中的颜色值，进行混合

**抖动**：用于因为使用有限的精度值在帧缓冲中保存颜色而产生的伪像

7、**帧缓冲区**

## 17、OpenGL中的uniform是什么？怎么编写一个定点着色器？

uniform类型的变量在整个绘制调用中始终使用同一个变量。这意味着你在绘制调用前加载的值在每个vertex shader调用时能访问到相同的值。uniform变量在存储光照参数（光照位置、方向等）、变换矩阵、纹理对象句柄等这一类型的数据时非常有用

1、使用#version标识openGL ES的版本号

2、使用in、out定义输入输出变量

3、在main函数进行对输入变量进行变换成输出变量

## 18、AVPlayer有哪些坑?

**当status变为AVPlayerStatusReadyToPlay后，我们调用play方法真的就能保证视频正常播放吗？**

众所周知，AVPlayer支持的视频、音频格式非常广泛，抛开那些无法正常编解码的情况，在某些情况下其可能就是无法正常播放。

AVPlayer在进行播放时，会预先解码一些内容，而此时如果我们的App使用CPU过多，I/O读写过多时，有可能导致视频播放声/画不同步，这点尤其在iPhone4上面表现更为明显。

而如果是发生在AVPlayer初始化解码视频的时候，有可能导致视频直接无法播放，这时，我们再调用play或者seekToTime:方法都无法正常播放。

建议不要在CPU或者I/O很频繁的情况下使用AVPlayer,例如刚登录App加载各种数据的情况下，可以等App预热以后再使用。

当rate属性的值大于0后，真的就在播放视频了吗？

答案是否定的，当发生上面所讲的情况时，我打印了当前的rate情况，是大于0的，但是页面上显示的情况却还是什么也没有。

有时候我们如果想要在视频一播放的时候去做一些事情，例如设置一下播放器的背景色，如果我们仅仅是监听这个rate可能无法100%保证有效，而如果我们真的要监听这种情况的话，有一个取巧的方法

id \_timerObserver = [self.player addBoundaryTimeObserverForTimes:@[[NSValue valueWithCMTime:CMTimeMake(1, 30)]] queue:dispatch\_get\_main\_queue()

usingBlock:^{

//do something

}];

另外如果不需要监听播放进度的时候可以调

[self.player removeTimeObserver:\_timerObserver];

**AVPlayer前后台播放的那些问题**

[[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(appBecomeActive:)

name:UIApplicationDidBecomeActiveNotification

object:nil];

[[NSNotificationCenter defaultCenter] addObserver:self

selector:@selector(appWillResignActive:)

name:UIApplicationWillResignActiveNotification

object:nil];

先在appWillResignActive:方法中记录当前播放的时间CMTime

等到切回前台的时候再继续播放

**音频通道的抢占引起的无法播放视频问题**

App运行的时候通常只能使用一种声音播放模式，而如果我们在录制视频或者录制声音的时候，把模式设置成了kAudioSessionCategory\_RecordAudio,这个时候如果我们使用AVPlayer播放视频，可能就无法播放视频。

这个时候我们需要把模式切换成kAudioSessionCategory\_MediaPlayback或者其它合适的模式

首先，我们需要先向设备注册激活声音打断AudioSessionSetActive(YES);,当然我们也可以通过

[AVAudioSession sharedInstance].otherAudioPlaying;这个方法来判断还有没有其它业务的声音在播放。

当我们播放完视频后，需要恢复其它业务或App的声音，这时我们可以调用如下方法:

OSStatus ret = AudioSessionSetActiveWithFlags(NO, kAudioSessionSetActiveFlag\_NotifyOthersOnDeactivation);

**在用户插入和拔出耳机时，有可能也会导致视频暂停**

AudioSessionAddPropertyListener(kAudioSessionProperty\_AudioRouteChange, audioRouteChangeListenerCallback, (\_\_bridge void\*)self);

void audioRouteChangeListenerCallback

**内存泄漏问题**

当我们释放一个正在播放的视频时，需要先调用pause方法，如果由于某些原因，例如切前后台时，导致又调用了play方法，那么有可能会hold住内存空间而导致内存泄漏

AVAssetImageGenerator: 获取视频缩略图  
获取首帧视频截图的方法如下

## 19、iOS audio session有哪些category？

**kAudioSessionCategory\_AmbientSound**：用于非以语音为主的应用，使用这个category的应用会随着静音键和屏幕关闭而静音。并且不会中止其它应用播放声音，可以和其它自带应用如iPod，safari等同时播放声音，注意：**该Category无法在后台播放声音**

**kAudioSessionCategory\_SoloAmbientSound**：类似于AVAudioSessionCategoryAmbient 不同之处在于它会中止其它应用播放声音。 这个category为默认category。该**Category无法在后台播放声音**

**kAudioSessionCategory\_RecordAudio**

用于需要录音的应用，设置该category后，除了来电铃声，闹钟或日历提醒之外的其它系统声音都不会被播放。该Category只提供单纯录音功能。

**kAudioSessionCategory\_MediaPlayback**

用于以语音为主的应用，使用这个category的应用不会随着静音键和屏幕关闭而静音。可在后台播放声音

**kAudioSessionCategory\_PlayAndRecord**

——用于既需要播放声音又需要录音的应用，语音聊天应用(如微信）应该使用这个category。该Category提供录音和播放功能。如果你的应用需要用到iPhone上的听筒，该category是你唯一的选择，在该Category下声音的默认出口为听筒（在没有外接设备的情况下）。

注意：并不是一个应用只能使用一个category，程序应该根据实际需要来切换设置不同的category，举个例子，录音的时候，需要设置为AVAudioSessionCategoryRecord，当录音结束时，应根据程序需要更改category为AVAudioSessionCategoryAmbient，AVAudioSessionCategorySoloAmbient或AVAudioSessionCategoryPlayback中的一种

## 20、AVPlayer中的一些坑？

播放异常问题

1.1 开始播放时黑屏/花屏一下

**原因**：下载数据不够就开始播放，导致卡顿和解码异常，影响播放体验。

**解决办法**：收到ReadyToPlay后不要立即播放，多缓存1到2秒的数据后再播放。

1.2 绿屏

**原因**：iOS和Mac系统播放器不支持mp4文件有多个分辨率，如果有多分辨率，在播放到分辨率变化的地方会有绿屏现象。（Android和Windows可以正常播放）

1.3 拖动后花屏

**原因**：某些mp4文件头缺失了stss字段，拖动进度条后会导致无法定位需要播放的帧，出现花屏问题。（Android和Windows可以正常播放）

1.4 画面卡住

**现象**：有些视频出现播放几秒后卡住的问题（iOS & Mac), 检查mp4格式没有异常，但是Android和Windows可以正常播放。

HLS

2.1 一直播放不起来

**原因**：1）m3u8（直播）文件中的ts片不连续 2. 将automaticallyWaitsToMinimizeStalling设置为NO，抓包显示AVPlayer一直在请求同一片数据十多次。

**解决方法**：保证ts片是连续的（dns解析到多个ip时，AVPlayer会随机向所有ip请求文件；不同ip返回的片可能不连续）；使用ip直出可以避免该问题。

2.2 首次缓冲时间长

**原因**：调用了AVPlayer的setRate, 导致首片地址下载两次。

2.3 播放到一半卡住

**原因**：中间某个ts片404了。

2.4 网速对HLS的影响

**网速远大于码率**：开始加载首片后立即开始播放（首次缓冲耗时0.5s左右）

**网速接近码率**：首片下载完毕后，第二片还下载了一部分才开始播放（首次缓冲耗时8s左右）

**网速小于码率**：一直在下载，永远播不起来。

**ResourceLoader的问题**

在iOS 9上，我们使用ResourceLoader来绕开ATS。但是在播放长视频时，拖动进度条后，视频会一直loading播不起来。 原因：在拖动时，AVPlayer会触发多次offset-length数据请求，但不会把前一次请求取消，导致重复下载很多数据。（模拟器上没有这个问题，发起新请求之前会把老的请求取消掉。）

**解决方法**：当检测到后一次请求是前一次请求数据的子集时，手动取消前次请求。

性能问题

AVPlayer播放单个视频并没有太大的性能问题，但在tableView中滚动时，自动切换和播放视频会造成滑动卡顿，主要是播放器资源的创建和销毁阻塞UI。需要将AVPlayer放到工作线程中。但是需要注意：

**4.1 replaceCurrentItemWithPlayerItem只能在主线程调用**。

**4.2 AVPlayer和对应AVPlayerItem的释放操作要放在同一线程，否则会导致crash。**

其他未解决的问题

iOS 10发布后，每天都有用户反馈视频无法播放，并且一旦出现这种情况，所有（使用AVPlayer）的app都无法播放但不报任何错误，需要重启手机恢复（有时候不重启也会恢复）。我们怀疑是iOS 10的bug，某些情况下系统媒体服务被阻塞。这个问题我们用自动化工具跑了很久一直无法重现，暂时也没有找到办法解决。

## 21、AVPlayer的文件缓存如何设计？多个视频文件缓存呢？

在 AVAssetResourceLoadingRequest 里面，request 代表原始的请求，由于 AVPlayer 是会触发分片下载的策略，还需要从dataRequest 中得到请求范围的信息。有了请求地址和请求范围，我们就可以重新创建一个设置了请求 Range 头的 NSURLRequest 对象，让下载器去下载这个文件的 Range 范围内的数据。

当 AVPlayer 触发下载时，总是会先发起一个 Range 为 0-2 的数据请求，这个请求的作用其实是用来确认视频数据的信息，如文件类型、文件数据长度。当下载器发起这个请求，收到服务端返回的 response 后，我们要把视频的信息填充到 AVAssetResourceLoadingRequest 的 contentInformationRequest 属性中，告知下载的视频格式以及视频长度。

下载到的数据可以塞给 AVAssetResourceLoadingRequest 里的 dataRequest 。 dataRequest 里面用 - (void)respondWithData:(NSData \*)data; 专门用来接收下载的数据，这个方法可以调用多次，接收增量连续的 data 数据。

当 AVAssetResourceLoadingRequest 要求的所有数据都下载完毕，调用 - (void)finishLoading 完成下载，AVAssetResourceLoader 会继续发起之后的数据片段的请求。如果本次请求失败，可以直接调用 - (void)finishLoadingWithError:(nullable NSError \*)error; 结束下载。

**重试机制**

在实际的测试中，发现AVAssetResourceLoader 在执行加载的时候，会时不时的触发取消下载调用 - (void)resourceLoader:(AVAssetResourceLoader \*)resourceLoader didCancelLoadingRequest:(AVAssetResourceLoadingRequest \*)loadingRequest，然后重新发起加载请求的策略。如果下载了部分，那么重新发起的下载请求会从还没有下载的部分开始。

**缓存下载的实现**

我们已经知道 AVAssetResourceLoaderDelegate 的实现机制，当 AVAsset 需要加载数据时会通过 delegate 告诉外部，外部接管整个视频下载过程。

接管了视频下载，便可以对视频数据做任何事情。比如：缓存、记录下载速度、获得下载进度等等。

实现一个下载器，就是用 URLSession 开启一个 DataTask 请求数据，把接收到的数据塞给 DataRequest 并写入本地磁盘。在实现下载器时主要有三个注意的点**：1. Range 请求 2. 可取消下载 3. 分片缓存**

**1. Range 请求**

每次得到的 LoadingRequest 带有请求数据范围的信息，比如期望请求第 100 字节到 500 字节，在创建 URLRequest 时需要设置 HTTPHeader 的 Range 值。

**2、可取消下载**

AVAsset 在加载视频时，经常会在某次数据请求还没有完成时触发取消下载，然后发起一个新的 LoadingReqeust。这个机制是 AVAsset 里的黑盒，具体逻辑无法得知，比较像是 AVAsset 的一种重试机制。 作为下载器，在收到取消通知时，需要立刻停止下载。由于 DataRequest 的 cancel 操作是异步的，就有可能在 cancel 还未完成时，下一个 LoadingRequest 就已经到来，所以还需要需要保证同一个 URL 只能同时存在一个下载器在下载，否则会出现数据混乱的问题

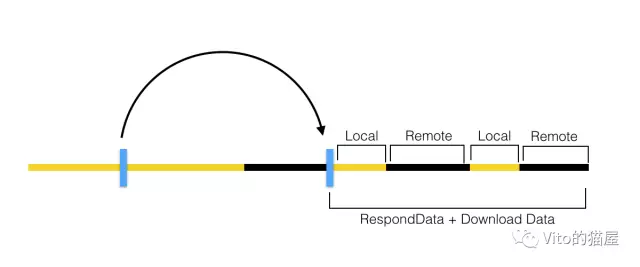
**3. 分片缓存**

如果只是单纯的下载视频，数据单调递增，缓存处理还是比较容易。然而现实是用户对 player 的 seek 操作给视频的缓存管理带来了巨大的挑战，一旦涉及到用户操作，可能性就越多，复杂度也会越高。

没有 seek 的情况：网速正常时缓存数据比播放时间走得开，正常播放；网速慢时，播放器 loading，直到有足够的数据量进行播放

为了支持 seek 操作，下载器就需要支持分片缓存。目前使用的解决方案是下载的视频数据会根据请求的 Range 值，把数据存储到文件中对应的偏移值位置，并且每个视频文件都会另外再保存一个与之对应的下载信息文件。

把已下载的分片信息分别创建一个 action，再把需要远程下载的分片数据分别创建一个 action。最终组合就可能是 LocalAction(50-100 bytes) + RemoteAction(101-200 bytes) + LocalAction(201-300 bytes) + RemoteAction(300-400 bytes)。每一个 action 会按顺序获取数据再返回给 LoadingRequest。



## 22、iOS上图像处理有哪些方式？

UIKit：drawRect:

CoreGraphic和Quartz2D（CPU）

CoreImage

GPUImage

OpenGLES

## 23、CoreImage如何使用？

**创建一个CIContext()**

**初始化CIFilter，并进行相关参数的初始化，filterWithName:**

**初始化CIImage（url、UIImage），并复制给CIFilter的参数，kCIInputImageKey**

**获取CIFilter.outputImage结果（CIImage）**

**从context.createCGImage来进行获取CGImage**

## 24、CoreImage如何进行实时性能优化？

本身有延迟处理，如果有多个CIFilter处理，则进行合并，来避免重复的计算和操作，

## 25、CoreImage如何做基于反馈的处理？

CIImageAccumulator

## 26、如何进行自定义CoreImageFilter?

继承CIFFilter

定义输入参数，以input开头，如inputImage

如果有必要覆写setDefaults方法

覆写outoutImage

## 27、如何使用CoreImage对视频加实时滤镜效果？

使用videoCompositionWithAsset:applyingCIFiltersWithHandler:创建AVVideoComposition，当**进行播放或者export**导出时候就会调用这个Filter Handler， 在这个Handler里首先是一个AVAsynchronousCIImageFilteringRequest来获取视频帧，然后对每一帧图像进行添加CIFilter。

**播放时**：使用AVPlayerItem时，需要将创建的videoComposition赋给AVPlayerItem的videoComposition属性

**导出时**：将创建的videoComposition对象赋值给AVAssetExportSession的videoComposition属性

## 28、使用CoreImage需要注意什么？

1、CoreImage Context的创建是非常耗性能的，不要重复创建和销毁，如果进行多个图片的滤镜处理，可以先进行创建，然后重复利用。

2、CoreImage 会自动管理资源，如将一个静态图片写入一个文件会自动调用writeJPEGRepresentationOfImage:toURL:colorSpace:options:error:， 当不知道渲染目的类型时候，不要使用drawImage:inRect:fromRect:方法，可以使用context的render和create开头的方法，让CoreImage来自动进行处理。

3、当进行实时渲染，或者实时更改滤镜参数时产生动画效果，尽管CIContext会自动在GPU进行处理，但也会包含在CPU和GPU之间的很多昂贵的copy操作，可以使用Metal框架

4、当使用CoreImage时候尽量不要使用Core Animation

## 29、如何打包Image Units？（使用NSBundle打包可执行和非可执行包）

Xcode创建Image Unit 工程

在工程中添加Filter 文件

自定义Load方法

修改Description Property List

编译和测试Image Unit

**loadPlugIn:allowNonExecutable:**

动画与绘制

## 1、如何自定义iOS动画？

在View的drawrect中绘制，外部需要进行调用setNeedisplay方法。

在CALayer中进行绘制。

设置引起动画的属性

在**needDisplayForKey**中返回属性

在**drawInContext**中进行绘制，或者在其代理中进行绘制**- (void)drawLayer: inContext.**进行绘制

## 2、iOS中有哪些动画

UIView Animation

UIView Block Animation

CAAnimation

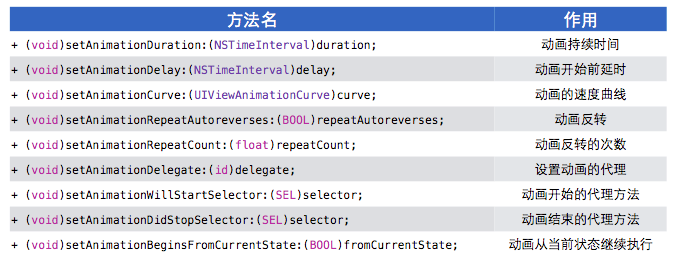
CATransition

1、UIView Animation

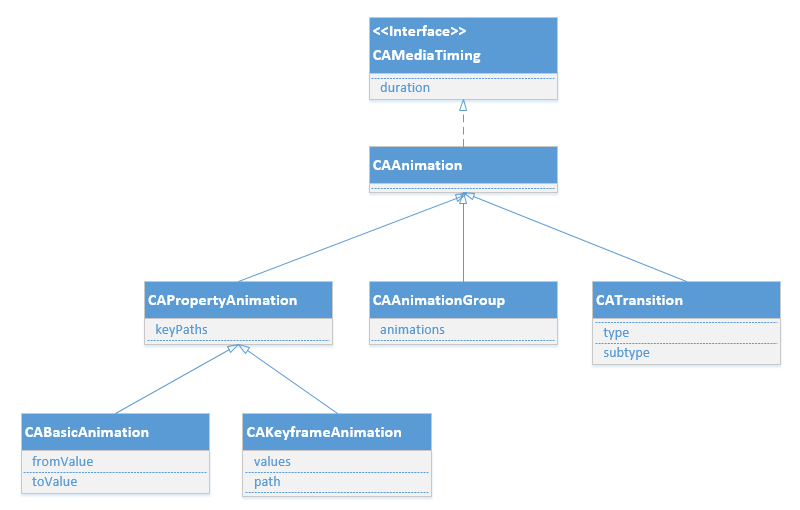
[UIView beginAnimaitons: nil context: nil]; //标记UIView动画开始

//设置UIView各种动画属性

[UIView commitAnimations];//标记UIView动画结束



## 3、iOS CAAnimation框架中的主要类结构



**CAAnimation**：核心动画的基础类，不能直接使用，负责动画运行时间、速度的控制，本身实现了**CAMediaTiming**协议。

**CAPropertyAnimation**：属性动画的基类（通过属性进行动画设置，注意是可动画属性），不能直接使用。

**CAAnimationGroup**：动画组，动画组是一种组合模式设计，可以通过动画组来进行所有动画行为的统一控制，组中所有动画效果可以并发执行。

**CATransition**：转场动画，主要通过滤镜进行动画效果设置。

**CABasicAnimation**：基础动画，通过属性修改进行动画参数控制，只有初始状态和结束状态。

**CAKeyframeAnimation**：关键帧动画，同样是通过属性进行动画参数控制，但是同基础动画不同的是它可以有多个状态控制。

## 4、CoreGraphic绘制流程是什么？

**1.获取绘图上下文：UIGraphicsGetCurrentContext()**

**2.创建并设置路径**

**3.将路径添加到上下文**

**4.设置上下文状态**

**5.绘制路径**

**6.释放路径**

## 5、CALayer的图层树？

**Model Layer**：模型树树表示每一层的对象模型值，即设定图层的属性值，值会立即改变

**Presentation Layer**：呈现树表示当前动画发生的时候，将要显示的值，则模型树中对应属性的值会立即改变，而呈现树则是在将要显示给用户的时候才进行更改。

**Render tree**： 渲染树是在渲染图层的时候使用呈现树的值。

## 6、CAAnimation的fillMode是做什么的？

**fillMode**的作用就是决定当前对象过了非active时间段的行为. 比如动画开始之前,动画结束之后。**如果是一个动画CAAnimation,则需要将其removedOnCompletion设置为NO,要不然fillMode不起作用**

**kCAFillModeRemoved** 这个是默认值,也就是说当动画开始前和动画结束后,动画对layer都没有影响,动画结束后,layer会恢复到之前的状态

**kCAFillModeForwards** 当动画结束后,layer会一直保持着动画最后的状态

**kCAFillModeBackwards** 这个和kCAFillModeForwards是相对的,就是在动画开始前,你只要将动画加入了一个layer,layer便立即进入动画的初始状态并等待动画开始

**kCAFillModeBoth**是**kCAFillModeForwards和kCAFillModeBackwards的结合**

CoreText

## 1、CoreText和UIKit坐标系如何进行转化？

//获取当前上下文

CGContextRef context = **UIGraphicsGetCurrentContext()**;

//翻转坐标系步骤

//设置当前文本矩阵

**CGContextSetTextMatrix**(context, CGAffineTransformIdentity);

//文本沿y轴移动

**CGContextTranslateCTM**(context, 0, self.bounds.size.height);

//文本翻转成为CoreText坐标系

**CGContextScaleCTM**(context, 1, -1);

**设置完属性后：**

//绘制文本

**CTFrameDraw**(frame, context);

//绘制图像

UIImage \*image = [UIImage imageNamed:@"boat.jpg"];

**CGContextDrawImage**(context, \_imageRect, image.CGImage);

## 2、CoreText如何进行绘制？

CoreText中布局大体思路是确定**文本绘制区域**，接着得到文本实际大小(frame)。

**1.**首先要确定布局时绘制的区域，其对应的类为**CG(Mutable)PathRef**

**2.**设置文本内容，其对应的类为NS(Mutable)AttributedString

**3.**根据文本内容配置其**CTFramesetterRef**

**4.**利用**CTFramesetterRef得到CTFrame**

**//1.创建绘制区域,显示的区域可以用CGMUtablePathRef生成任意的形状**

CGMutablePathRef path = CGPathCreateMutable();

CGPathAddRect(path, NULL, CGRectMake(20, 50, self.bounds.size.width - 40, self.bounds.size.height - 100));

**//2.创建需要绘制的文字**

NSMutableAttributedString \*attString = [[NSMutableAttributedString alloc] initWithString:@"\tWhen I will learn CoreText, i think it will hard for me.But it is easy.\n\tIn fact,if you bengin learn, you can know that every thing is easy when you start.you just need some knowlages"];

**//3.根据AttString生成CTFramesetterRef**

CTFramesetterRef frameSetter = CTFramesetterCreateWithAttributedString((CFAttributedStringRef)attString);

CTFrameRef frame = CTFramesetterCreateFrame(frameSetter, CFRangeMake(0, [attString length]), path, NULL);

**//4、attString设置下划线、行间距、字体间距、设置黑体**

## 3、CoreText图文如何混排？

图片宽高在工程中都需要加载后才知道，而在文本绘制中需要直接留出其位置再进行绘制，所以图片的宽高都是在数据中保存好的，此处笔者用固定值来表示其宽高。为了留出其位置我们需要用**空白的字符来做占位符**使用。为了知道其图片绘制的位置(即空白占位符位置)我们需要设置代理才能够得知图片绘制位置。具体步骤如下:

**1.创建CTRunDelegateCallbacks 回调函数**：通过回调函数来确定图片绘制的宽高

**2.创建空白占位字符**

**3.设置CTRunDeleagte：**通过代理来找到该字符串，并确定图片绘制的原点

## 4、CoreText绘制流程？

**1：获取上下文并翻转坐标系**

**2：创建NSAttributedString，使用子类NSMutableAttributedString为文字设置相应属性**

**3：根据NSAttributedString创建CTFramesetterRef**

**4：创建绘制区域CGPathRef**

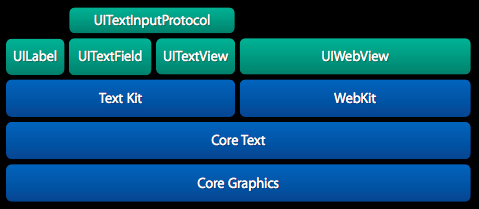
**5：根据CTFramesetterRef和CGPathRef创建CTFrame**

**6：调用CTFrameDraw进行绘制**

## 5、CoreText如何实现一个阅读器？

## 6、有哪些方式可以实现图文混排？

TextKit、CoreText、UIWebView、WKWebView、DTCoreText、JavascriptCore自己实现、TextView显示NSAttributeString



数据存储

## 1、多线程下如何使用CoreData？

**NSManagedObjectContext非线程安全，尽量一个线程对应一个NSManagedObjectContext，一个线程只读写与其对应的context**

监听NSManagedObjectContextDidSaveNotification通知，并执行对应线程NSManagedObjectContext的mergeChangesFromContextDidSaveNotification:方法。

objectWithID(\_ objectID: ) 来获取通知里的对象

可以使用一个NSPersistentStoreCoordinator实例，这个实例可以很安全的顺序访问永久存储，这是因为**NSManagedObjectContext会在便用NSPersistentStoreCoordinator前上锁**

网络编程

## 1、BSD socket API有哪些？

Socket（） 创建并初始化 socket，返回该 socket 的文件描述符，如果描述符为 -1 表示创建失败。

Bind() 将 socket 与特定主机地址与端口号绑定，成功绑定返回0，失败返回 -1。

Accept() 接受客户端连接请求并将客户端的网络地址信息保存到 clientAddress 中

Connect() 客户端向特定网络地址的服务器发送连接请求，连接成功返回0，失败返回 -1

Send() 通过 socket 发送数据，发送成功返回成功发送的字节数，否则返回 -1

Receive() 从 socket 中读取数据，读取成功返回成功读取的字节数，否则返回 -1。

Sendto() 通过UDP socket 发送数据到特定的网络地址，发送成功返回成功发送的字节数，否则返回 -1

Recvfrom() 从UDP socket 中读取数据，并保存发送者的网络地址信息，读取成功返回成功读取的字节数，否则返回 -1

Gethostbyname() 使用 DNS 查找特定主机名字对应的 IP 地址。如果找不到对应的 IP 地址则返回 NULL。

close 关闭 socket

## 2、BSD Socket工作流程？

服务器工作流程

有了上面的 socket API 讲解，下面来总结一下服务器的工作流程。

1. 服务器调用 **socket(...) 创建socket；**

2. 服务器调用 **listen(...) 设置缓冲区；**

3. 服务器通过 **accept(...)接受客户端请求建立连接；**

4. 服务器与客户端建立连接之后，就可以通过 **send(...)/receive(...)**向客户端发送或从客户端接收数据；

5. 服务器调用 **close 关闭 socket；**

客户端工作流程

1. 客户端调用 **socket(...) 创建socket；**

2. 客户端调用 **connect(...) 向服务器发起连接请求以建立连接；**

3. 客户端与服务器建立连接之后，就可以通过 **send(...)/receive(...)**向客户端发送或从客户端接收数据；

4. 客户端调用 **close 关闭 socket；**

## 3、CFNetwork相关API有哪些？

**CFStreamCreatePairWithSocketToHost** 创建一对 socket stream，一个用于读取CFReadStreamRef，一个用于写入CFWriteStreamRef。

Boolean **CFReadStreamOpen**(CFReadStreamRef stream);

Boolean **CFWriteStreamOpen**(CFWriteStreamRef stream);

Boolean **CFReadStreamSetClient**;

Boolean **CFWriteStreamSetClient**;

void **CFReadStreamScheduleWithRunLoop**;

void **CFWriteStreamScheduleWithRunLoop**

void **CFReadStreamUnscheduleFromRunLoop**

void **CFWriteStreamUnscheduleFromRunLoop**

CFIndex **CFReadStreamRead**

CFIndex **CFWriteStreamWrite**;

void **CFReadStreamClose**;

void **CFWriteStreamClose**;

Hybrid

## 1、JS和OC的交互方式有哪些？对比一下？

**通过UIWebView代理的消息拦截**

**javaScriptCore进行通信**

**WKWebView的ScriptMessageHandler和WKUserContentController**

**WebViewJavascriptBridge框架**

## 2、iOS与ReactNative的通信机制?(从5开始看)

**简述：**

宏编译进行控制方法暴露

ModuleId，methodId，args、CallbackId ，js callback -> block

MessageQueue查询Callbackid

在启动阶段，初始化JS引擎，生成Native端模块配置表存于两端，其中模块配置是同步取得，而各模块的方法配置在该方法被真正调用时懒加载。

Native和JS端分别有一个bridge，发生调用时，调用端bridge查找模块配置表将调用转换成{moduleID, methodID, args(callbackID)}，处理端通过同一份模块配置表转换为实际的方法实现。

**Native->JS**，原理上使用JSCore从Native执行JS代码，React-Native在此基础上给我们提供了通知发送的执行方式。

**JS->Native**，原理上JS并不主动调用Native，而是把方法和参数(回调)缓存到队列中，在Native事件触发并访问JS后，通过blocks回调Native。

**RCTBridgeModule协议、RCT\_EXPORT\_MODULE()、RCT\_EXPORT\_METHOD()**、RCTResponseSenderBlock、RCTPromiseResolveBlock和RCTPromiseRejectBlock、多线程实现方法- (dispatch\_queue\_t)methodQueue、依赖注入RCTBridgeDelegate、**RCTEventEmitter、RCT\_EXPORT\_VIEW\_PROPERTY**

**详细流程：**

**1、Load JS Source Code（并行）**

**2、2.Init Module（同步）**：

初始化加载React Native模块。该阶段会将所有注册的Native模块类整理保存到一个以Module Id为下标的数组对象中（同时还会保存一个以Module Name为Key的Dictionary，用于做索引方便后续的模块查找）。

整个模块的基础初始化和注册过程在系统Load Class阶段就会完成。React Native对模块注册的实现还是比较巧妙、方便，只需要对目标类添加相应的宏即可。

**注册模块**。实现RCTBridgeModule协议，并且在响应的Implemention文件中添加RCT\_EXPORT\_MODULE宏，该宏会为所在类自动添加一个+load方法，调用RCTBridge的RCTRegisterModule实现在Load Class阶段就完成模块注册工作。

**注册函数**。待注册函数所在的类必须是已注册模块，在需要注册的函数前添加RCT\_EXPORT\_MODULE宏即可。对于模块类中想要声明的方法，需要添加RCT\_EXPORT\_METHOD宏。它会给方法名添加” rct\_export “前缀。

当然这里需要注意的问题时模块初始化是一个同步任务，它必须被同步加载，所以当模块较多时势必会带来高延迟的问题，也是在新的版本中SDK将Module Method改为Lazy Load的原因之一。

**3、Setup JS Executor（并行）**

通过IOS SDK JSContext来实现相关的逻辑

requireModuleConfig：用于在JS端获取对应的Native Module

**4、Module Config（并行）**

Native模块类转换成Json，保存为remoteModuleConfig

**5、Inject Json Configration To JS**

一个setupJSExecutorAndModuleConfig的dispatchGroup中，当3和4两步都完成后，触发该步。此时将第4步中生成的**config注入到JS中，保存到名为\_\_fbBatchedBridgeConfig 的js全局变量中**，真正补全模块信息是当JS页面源码中，调用nativeRequireModuleConfig这个API使用指定模块（当然这也是非常符合Require按需加载的理念）时补全模块信息。

**6、BatchedBridge**：

在JS端也存在一个bridge模块BatchedBridge，也是与Native建立双向通信的关键所在

**BatchedBridge是一个MessageQueue实例，它在创建时传入了\_\_fbBatchedBridgeConfig值保存Native端支持的模块列表配置，BatchedBridge在创建时将自己写入全局变量\_\_fbBatchedBridge上，这样Native可以通过JSContext[@”\_\_fbBatchedBridge”]访问到JS bridge对象**

MessageQueue的几个信息：

**RemoteModules**属性，用于保存Native端模块配置

**Callbacks**属性缓存js的回调方法

**Queue**事件队列用于处理各类事件等

**7、JS Call**

当JS调用module.method时，其实调用了**self.\_\_nativeCall**(module, method, args, onFail, onSucc)，对于\_\_nativeCall方法

这段代码为每个method创建了一个闭包fn，在\_\_nativeCall方法中，并且在这里做了两件重要的工作：

把onFail和onSucc缓存到\_callbacks中，同时把callbackID添加到params

把moduleID, methodID, params放入队列中，回调Native代码.

\_\_nativeCall如何做到回调Native代码呢？看第二部分第3步，在初始化JS引擎JSExecutor Setup时，**Native端注册一个全局block回调nativeFlushedQueueImmediate**，nativeCall在处理完毕后，通过该回调把队列作为返回值传给Native。

**8、Call JS Function**

**callFunctionReturnFlushedQueue**用于实现Native调用带有返回值的JS端函数，对于**callFunctionReturnFlushedQueue方法，它最终调用的是\_\_callFunction：处会根据Native传入的module, method，调用JS端相应的模块并传入参数列表args**.

同时我们又可以获得对于MessageQueue的另一条推测，\_callableModules用来存放JS端暴露给Native的模块

**总结：**

在启动阶段，初始化JS引擎，生成Native端模块配置表存于两端，其中模块配置是同步取得，而各模块的方法配置在该方法被真正调用时懒加载。

**Native和JS端分别有一个bridge，发生调用时，调用端bridge查找模块配置表将调用转换成{moduleID, methodID, args(callbackID)}，处理端通过同一份模块配置表转换为实际的方法实现。**

**Native->JS**，原理上使用JSCore从Native执行JS代码，**React-Native在此基础上给我们提供了通知发送的执行方式**。

**JS->Native**，原理上JS并不主动调用Native，而是把方法和参数(回调)缓存到队列中，在Native事件触发并访问JS后，通过blocks回调Native。

## 3、JSPatch实现原理（头条问到）

**1、方法调用**：

Require()

1、require 做的事很简单，就是在JS全局作用域上创建一个同名变量，变量指向一个对象，对象属性 \_\_clsName 保存类名，同时表明这个对象是一个 OC Class.

2、在 OC 执行 JS 脚本前，通过正则把所有方法调用都改成调用 \_\_c() 函数，再执行这个 JS 脚本，做到了类似 OC/Lua/Ruby 等的消息转发机制

UIView.alloc().init() > UIView.\_\_c('alloc')().\_\_c('init')()

3、**\_\_c实际上调用了\_methodFunc()** 就是把相关信息传给OC，OC用 Runtime 接口调用相应方法，返回结果值，这个调用就结束了。**\_methodFunc(self.\_\_obj, self.\_\_clsName, methodName, args, self.\_\_isSuper)**

4、**JS 通过调用 JSContext 定义的Native方法把数据传给 OC，OC 通过执行evaluateScript返回值传会给 JS。（\_\_C函数中得到返回值）**

**2、方法替换**：

**获取参数**

实现一个通用的IMP，任意方法任意参数都可以通过这个IMP中转，拿到方法的所有参数回调JS的实现.

va\_list实现(32位), 依次从参数的argumentType获取类型

ForwardInvocation实现(64位)：方法通过 class\_replaceMethod() 把原来方法实现保存，然后新的实现指向 \_objc\_msgForward，这是一个全局 IMP，OC 调用方法不存在时都会转发到这个 IMP 上，这里直接把方法替换成这个 IMP，这样调用这个方法时就会走到 -forwardInvocation:，OC就可以获取相关参数来调用JS方法。

以替换 UIViewController 的 -viewWillAppear: 方法为例：

把UIViewController的 -viewWillAppear: 方法通过 class\_replaceMethod() 接口指向 \_objc\_msgForward，这是一个全局 IMP，OC 调用方法不存在时都会转发到这个 IMP 上，这里直接把方法替换成这个 IMP，这样调用这个方法时就会走到 -forwardInvocation:。

**为UIViewController添加 -ORIGviewWillAppear: 和 -\_JPviewWillAppear: 两个方法，前者指向原来的IMP实现，后者是新的实现，稍后会在这个实现里回调JS函数。**

改写UIViewController的 -forwardInvocation: 方法为自定义实现。一旦OC里调用 UIViewController 的 -viewWillAppear: 方法，经过上面的处理会把这个调用转发到 -forwardInvocation: ，这时已经组装好了一个 NSInvocation，包含了这个调用的参数。在这里把参数从 NSInvocation 反解出来，带着参数调用上述新增加的方法 -\_JPviewWillAppear: ，**在这个新方法里取到参数传给JS，调用JS的实现函数。整个调用过程就结束了**。

**新建方法**：

**现在 defineClass 定义的方法会经过 JS 包装，变成一个包含参数个数和方法实体的数组传给OC，OC会判断如果方法已存在，就执行替换的操作，若不存在，就调用 class\_addMethod() 新增一个方法，通过传过来的参数个数和方法实体生成新的 Method，把 Method 的参数和返回值类型都设为id。这里 JS 调用新增方法走的流程还是 forwardInvocation 这一套。**

内存管理（JSClass）：OC的toJSObject方法将OC对象转化为一个JS的JSClass实例

消息传递（JavascriptCore）

方法拦截（方法添加、参数获取），在-forwardInvocation:获取参数

## 4、WebViewJavascriptBridge原理是什么？

分别在OC环境和javascript环境都保存一个bridge对象，里面维持着**requestId,callbackId,**以及每个id对应的具体实现。

**1、OC-> js**

首先要在JavaScript里面注册一个方法：

bridge.registerHandler('showChat', function(data, responseCallback) {

log('ObjC called showChat with', data)

var responseData = { 'Javascript Says':'Right back atcha!' }

log('JS responding with', responseData)

responseCallback(responseData)

})

**OC的\_bridge callHandler**通过javascript环境的window.WebViewJavascriptBridge对象找到具体方法，然后执行。

[\_bridge callHandler:@"testJavascriptHandler" data:data responseCallback:^(id response) {

NSLog(@"testJavascriptHandler responded: %@", response);

}]; **传参注意javascript支持的类型**

**NSString\* javascriptCommand = [NSString stringWithFormat:@"WebViewJavascriptBridge.\_handleMessageFromObjC('%@');", messageJSON];**

**[self \_evaluateJavascript:javascriptCommand];**

**2、JS-> OC**

**OC注册一个方法：**

[\_bridge registerHandler:@"showChat" handler:^(id data, WVJBResponseCallback responseCallback) {

NSLog(@"showChat called: %@", data);

responseCallback(@"Response from showChat");

}];

**JavaScript调用方法**

bridge.callHandler('showChat', {'foo': 'bar'}, function(response) {

log('JS got response', response)

});

javascript通过改变iframe的src来触发webview的代理方法webView:(WKWebView \*)webView decidePolicyForNavigationAction:(WKNavigationAction \*)navigationAction decisionHandler:(void (^)(WKNavigationActionPolicy))decisionHandler从而实现把javascript消息发送给OC这个功能。

**变量callbackId是个字符串，responseCallBacks[]放到一个消息队列中（数组队列）让Native去取（通过evaluateJavascript去取）**

本质上是将消息拦截封装起来了，提供了标识、逻辑对注册功能

## 5、WKWebView如何与JS交互？遇到什么问题？如何进一步地优化？

OC调用JS：

**evaluateJavaScript:(NSString \*)javaScriptString completionHandler:**

**evaluatingJavaScript 是在客户端执行这条代码的时候立刻去执行当条JS代码**

**WKUserScript 是预先准备好的JS代码，当WKWebView加载Dom的时候，执行当条JS代码**

JS 调用 OC：

注册：**addScriptMessageHandler:(id<wkscriptmessagehandler>)scriptMessageHandler name:(NSString \*)name**

发送：**window.webkit.messageHandlers.<事件名>.postMessage(需要传递的数据)**

处理：**userContentController:(WKUserContentController\*)userContentController didReceiveScriptMessag**

**关于缓存**

在 WKWebsiteDataStore 出现之前（iOS 9 中），WKWebView 是没有缓存，也无从清理。WKWebView 是基于 WebKit 框架的，它会忽视**先前使用的网络存储 NSURLCache, NSHTTPCookieStorage, NSCredentialStorage等**，它也有自己的存储空间用来存储cookie和cache，其他的网络类如NSURLConnection 是无法访问到的。 同时WKWebView发起的资源请求也是不经过NSURLProtocol的，导致无法拦截或自定义新请求。

**关于 Cookie**

在使用 UIWebVIew 的时候我们并不关注 Cookie，因为在调用登录接口的时候无论是AFNetworking，还是其他，登录成功之后都会自动保存在

[NSHTTPCookieStorage sharedHTTPCookieStorage].cookies 中，以后再使用也会自动去获取，WKWebView 的存储体系与 UIWebVIew 完全不一样，只能手动给它添加 Cookie Post body丢失问题

**关于跨域**

WebKit框架对跨域进行了安全性检查限制，不允许跨域，比如从一个 HTTP 页对 HTTPS 发起请求是无效的

**关于 NSURLProtocol 拦截**

WKWebView 基于 WebKit 框架，与 UIWebView 机制不同：加载过程中所有的请求都不经过 NSURLProtocol，换句话说就是 WKWebView 无法拦截响应数据 鉴于之前大部分 Hybrid 框架的离线预加载机制都依赖拦截功能

**关于 POST 请求**

POST包使用通用的 Web&Native 交互协议，为 Web 提供 Native POST 请求的接口+回调 CallBack 即可

**关于本地 HTML 加载**

当使用 loadRequest 来读取本地 Documents 目录的 HTML 文件时，WKWebView 是无法读取成功的（Could not create a sandbox extension for /），只能通iOS 9的新接口加载

**而在iOS9的SDK中加入了以下方法来加载本地的HTML文件**：

[WKWebView loadFileURL:allowingReadAccessToURL:]

但是在iOS9以下的版本是没提供这个便利的方法的。以下为解决方案的思路，就是在iOS9以下版本时，先将本地HTML文件的数据copy到tmp目录中，然后再使用loadRequest来加载。但是如果在HTML中加入了其他资源文件，例如js，css，image等必须一同copy到temp中。

## 6、如何使用JavascriptCore进行JS与Native的交互？

JSContext 代表JS的执行环境，通过-evaluateScript:方法就可以执行JS代码

JSValue 封装了JS中与ObjC中的对应的类型，以及调用JS的API等

JSExport 是一个协议，遵守此协议，就可以定义我们自己的协议，在协议中声明的API都会在JS中暴露出来，才能调用

OC《--》都有两种方法

**OC调用JS：**

使用JSContext的evaluateScript方法，或**者**

**获取JSContext使用方法名作为标识的JSValue来** 进行调用callWithArguments

**JS调用OC：（block）**

获取js环境的jsContext，将block付给jsContext的一个名字标识，js就可以将这个标识作为方法来进行使用。

**JS调用OC：（JSExport）**

声明一个继承JSExport协议的协议，并定义相关方法和具体实现类。

将该具体**实现类的实例**付给jsContext的一个标识。

此时js可以使用window.**<实例标识**>.方法名来进行调用OC的代码

注意：

不要在JS中给OC对象增加成员变量，

OC对象不要直接强引用JSValue对象

## 7、RN如何调用Native方法？

实现RCTBridgeModule协议，使用RCT\_EXPORT\_MODULE();暴露给RN模块，

并使用RCT\_EXPORT\_METHOD暴露相关方法

然后RN就可以直接**模块名.方法名()**来调用Native方法了。

## 8、如何在Native中调用ReactNative方法？

RN进行注册相应事件的处理方法，this.listener = NativeAppEventEmitter.addListener(事件名,(data)=>{});

实现RCTBridgeModule协议，**使用RCTRootView的bridge属性**

可以使用**[bridge.eventDispatch sendAppEventWithName:**事件名 body:{}];来进行发送给RN相应的事件。

## 9、如何实现一个供RN使用的Native UI？

Native 视图是通过 **RCTViewManager** 的子类创建和操做的。这些子类的功能与视图控制器很相似，但本质上它们是单件模式——bridge只为每一个子类创建一个实例。它们将 **native 视图提供给 RCTUIManager**，它会传回到 native 视图来设置和更新的必要的视图属性。RCTViewManager 通常也是视图的代表，通过bridge将事件发送回 JavaScript。

**继承RCTViewManager（子类以manager结尾）**

**添加标记宏 RCT\_EXPORT\_MODULE()。**

**实现 -(UIView \*)view 方法，返回NativeView的实例**

**使用RCT\_EXPORT\_VIEW\_PROPERTY(pitchEnabled, BOOL)暴露Native属性（js可以进行赋值）**

RN通过

var { **requireNativeComponent** } = require('react-native');

**module.exports** = requireNativeComponent('RCTMap', null); 可以获取component

## 10、JSPatch如何处理block？

JSPatch的脚本中Block结构的定义如下:

block('ParameterAType1, ParameterBType1...', function(ParameterA, ParameterB...) {

// 函数体

})

block(parameterString, functionImpl)本身就是JavaScript函数。

Block变量在Objective-C代码经过编译后是一个函数C语言函数的结构体指针。

**JSPatch也把函数参数中的Block直接转成了函数。**

**callback block变量**

作为参数，因为JS是弱类型，直接用名字放在函数参数变量中，代表JS函数。function(url, dict, callback)

作为执行代码，JS代码中直接作为函数体执行callback({"content": content}, err)。

作为Block传递，self.setCallback(block('int, NSString \*', callback));，该句JS代码分为两部分，前半部分是setter方法self.setCallback(x)，x部分是将JS函数转化为JS中的block样式。

**handleRequestSuccess block变量**

再看self.handleRequestSuccessblock，JS代码self.handleRequestSuccess()(data);，咱们可以拆解成两部分，前半部分self.handleRequestSuccess()是一个属性获取方法，JS中获取后直接是函数，所以后面紧跟着就是函数的参数列表

**2. setter方法转义**

因为JS中不能直接获取只能通过方法获取变量，所以在hook setter方法时需要通过Key-Value Coding的方式。

当要把 JS 函数作为 block 参数给 OC时，需要先使用 block(paramTypes, function) 接口包装: 这里 block 里的参数类型用字符串表示，写上这个 block 各个参数的类型，用逗号分隔。NSObject 对象如 NSString \*, NSArray \*等可以用 id 表示，但 block 对象要用 NSBlock\* 表示。

**从 OC 返回给 JS 的 block 会自动转为 JS function，直接调用即可:**

若要把这个从 OC 传过来的 block 再传回给 OC，同样需要再用 block() 包装，因为这里 blk 已经是一个普通的 JS function，跟我们上面定义的 JS function 没有区别：

**总结：JS 没有 block 类型的变量，OC 的 block 对象传到 JS 会变成 JS function，所有要从 JS 传 block 给 OC 都需要用 block() 接口包装**

**从 JS 传 block 到 OC，有两个限制：**

A. block 参数个数最多支持6个。（若需要支持更多，可以修改源码）

B. block 参数类型不能是 double / NSBlock / struct 类型。

## 11、JSPatch对内存的管理？

可以在 JS 通过 \_\_weak() 声明一个 weak 变量，主要用于避免循环引用。

若要在使用 weakSelf 时把它变成 strong 变量，可以用 \_\_strong() 接口：

## 12、如何在多个WKWebView之间共享cookie

**1、新建一个名为 LZWKWebKitSupport 的类，用于生成一个统一的，全局使用同一个 WKProcessPool 的 WKWebView 对象。**

WKWebViewConfiguration \*configuration = [WKWebViewConfiguration new];

// 一下两个属性是允许H5视频自动播放,并且全屏,可忽略

configuration.allowsInlineMediaPlayback = YES;

configuration.mediaPlaybackRequiresUserAction = NO;

// 全局使用同一个processPool

**configuration.processPool = [[LZWKWebKitSupport sharedSupport] processPool];**

configuration.userContentController = userContentController;

// 考虑到左侧菜单栏，需要设置webView的不同frame

**WKWebView \*wk\_webView = [[WKWebView alloc] initWithFrame:**CGRectMake(0, y, width, height) configuration:configuration];

**2. 在加载H5的地方初始化 LZWKWebKitSupport，并在 WKNavigationDelegate 中获取 cookie，并设置到本地。**

- (void)webView:(WKWebView \*)webView decidePolicyForNavigationResponse:(WKNavigationResponse \*)navigationResponse decisionHandler:(void (^)(WKNavigationResponsePolicy))decisionHandler

{

decisionHandler(WKNavigationResponsePolicyAllow);

NSHTTPURLResponse \*response = (NSHTTPURLResponse \*)navigationResponse.response;

// 读取cookie，并设置到本地

NSArray \*cookies =[NSHTTPCookie cookiesWithResponseHeaderFields:[response allHeaderFields] forURL:response.URL];

for (NSHTTPCookie \*cookie in cookies) {

[[NSHTTPCookieStorage sharedHTTPCookieStorage] setCookie:cookie];

}

}

**3. 在从第一个H5页面跳转至第二个H5页面时，在发起请求时注入Cookie。**

// 注入Cookie

**[requestObj setValue:cookies forHTTPHeaderField:@"Cookie"];**

// 加载请求

**[self.wkWebView loadRequest:requestObj];**

通过以上三步就可以达到同步 Cookie 的目的，现在看来之前通过 JS脚本 注入 Cookie 失败，可能是由于后台需要同步 BJSESSIONID，而BJSESSIONID 是 HtppOnly，不允许通过js脚本修改。

**最后，需要特别注意的一点是：考虑在加载H5页前，是否需要清除某些H5页面的 Cookie ?**

## 13、RN传给Native的类型有哪些？

**参数类型**

RCT\_EXPORT\_METHOD 支持所有标准JSON类型，包括：

string (NSString)

number (NSInteger, float, double, CGFloat, NSNumber)

boolean (BOOL, NSNumber)

array (NSArray) 包含本列表中任意类型

**object (NSDictionary) 包含string类型的键和本列表中任意类型的值**

function (RCTResponseSenderBlock)

除此以外，任何RCTConvert类支持的的类型也都可以使用(参见**RCTConvert**了解更多信息)。RCTConvert还提供了一系列辅助函数，用来接收一个JSON值并转换到原生Objective-C类型或类。

**特殊参数类型处理**（Date对象）

在我们的CalendarManager例子里，我们需要把事件的时间交给原生方法。我们不能在桥接通道里传递Date对象，所以需要把日期转化成字符串或数字来传递

交叉编译

## 1、iOS如何自动打包？

xcodebuild负责编译，xcrun负责将app打成ipa

脚本步骤：

1、清理工程

2、检查依赖 cocoapods文件

3、清除并重新安装cocoapods依赖

4、设置info.plist中版本号等信息、app名称、bundle以及与bundle对应的证书uuid

5、编译project或者workspace

6、使用xcrun打包

## 2、iOS目前主流的硬件架构？

I386、x86\_64、armv7、armv7s、arm64

## 3、iOS应用程序的编译过程？

**编译信息写入辅助文件**，**创建文件架构 .app 文件**

**处理文件打包信息**

**执行 CocoaPod 编译前脚本**，checkPods Manifest.lock

**编译.m文件**，使用 CompileC 和 clang 命令

**链接需要的 Framework**

**编译 xib**

**拷贝 xib ，资源文件**

**编译 ImageAssets**

**处理 info.plist**

**执行 CocoaPod 脚本**

**拷贝标准库**

**创建 .app 文件和签名**

**预处理**

符号化 (Tokenization)

宏定义的展开

#include 的展开

**语法和语义分析**

将符号化后的内容转化为一棵解析树 (parse tree)

解析树做语义分析

输出一棵抽象语法树（Abstract Syntax Tree\* (AST)）

**生成中间代码和优化**

将 AST 转换为更低级的中间码 (LLVM IR)

对生成的中间码做优化

生成特定目标代码

输出汇编代码

**汇编器**

将汇编代码转换为目标对象文件。

**链接器**

将多个目标对象文件合并为一个可执行文件 (或者一个动态库)

编译器前端的任务是进行：语法分析，语义分析，生成中间代码(intermediate representation )。在这个过程中，会进行类型检查，如果发现错误或者警告会标注出来在哪一行

编译器后端会进行机器无关的代码优化，生成机器语言，并且进行机器相关的代码优化。iOS的编译过程，后端的处理如下

**LVVM优化器会进行BitCode的生成，链接期优化等等**，LLVM机器码生成器会针对不同的架构，比如arm64等生成不同的机器码

调试技巧

## 1、lldb有哪些命令？

Help、print、**expression**、**po**、**thread step-in, thread step-out、thread step-ove**r、process continue、thread-continue、**watchpoint**、XCode的符号断点、断点命令、**breakpoint set**

## 2、Chisel 有哪些命令？

pviews、pvc、visualize、fv、fvc、caflush等

## 3、如何进行手动解析CrashLog（6种）

.dysm（debugging symbols）调试符号表

**确定符号表和崩溃日志的一致性**

符号表可以使用dwarfdump -uuid .dysm文件来进行获取

崩溃日志比较靠下的位置有个Binary Images模块，里面可以获取代码块的起始位置、CPU指令集、应用UUID

**计算崩溃符号表位置：**

**运行时堆栈地址=运行时起始位置+偏移量（崩溃日志）**

**符号表堆栈位置 = 符号表起始地址 + 偏移量**

符号表起始位置的获取，可以使用**otool –l 命令获取vmaddr**，也可以在崩溃符号表的**Binary Images**进行获取

崩溃信息还原

1、**dwarfdump命令、atos命令、Symbolicatecrash工具、XCode、appstore上传符号表。**

**有了偏移量，也可以进行静态分析，使用Hopper、IDA**

## 4、用过些调试工具？

Lldb、解析crashllog的命令、Instrument、Chisel、FLEX、FBMemoryProfile、FBRetainCycleDetector。

## 5、什么是LinkMap文件？

LinkMap的输出是一个**纯文本格式的文件，里面包含重要的编译信息及报错信息**，

LinkMap服务的开启方式及文件目录

在 Build Settings 里设置 Write Link Map File 为 Yes 后每次编译都会在指定目录生成这样一个文件。Xcode->Project->Build Settings-> Search map -> 设置 Write Link Map Files 选项为YES

**LinkMap 各部分的作用**

**1、App的完整的目标文件列表（#Object files）**： 这个部分的**内容都是 .m 文件编译后的 .o 和需要 link 的 .a 文件**。

**2、App的段表（#Section）**：这里描述的是每个 Section 在可执行文件中的位置和大小。每个 Section 的 **Segment 的类型分为 \_\_TEXT 代码段和 \_\_DATA 数据段两种**

**3、App中具体目标文件在对应的section中的位置和大小（#Symbols）**：Symbols 是对 Sections 进行了再划分

## 6、什么是dsym文件？

在**每次编译后都会生成一个 dSYM 文件，程序在执行中通过地址来调用方法函数**，而 **dSYM 文件里存储了函数地址映射**，这样**调用栈里的地址可以通过 dSYM 这个映射表能够获得具体函数的位置**。一般都会用来处理 crash 时获取到的调用栈 .crash 文件将其符号化。

## 7、BAD\_ACCESS在什么情况下出现？

访问了悬垂指针，比如对一个已经释放的对象执行了release、访问已经释放对象的成员变量或者发消息

安全防护

## 1、Linux提供动态装载库的API有哪些？

**- dlopen，打开一个库，并为使用该库做些准备**，这个dlopen函数使 file指定的可执行文件能够被调用此方法的程序使用，**在dlopen（）函数以指定模式打开指定的动态链接库文件，并返回一个句柄给dlsym（）的调用进程。使用dlclose（）来卸载打开的库**

Dlsym 使用dlopen打开动态链接库后返回的指针和symbol（就是要求获取的函数的名称，函数 ）返回值是void\*,指向函数的地址，供调用使用

**- dlsym，在打开的库中查找符号的值**。

- dlclose，使用dlclose（）来卸载打开的库

- dlerror，返回一个描述最后一次调用dlopen、dlsym，或dlclose的错误信息的字符串

**- dladdr，从函数指针解析符号名称和所在的文件**。

- dlvsym，与dlsym类似，只是多了一个版本字符串参数

## 2、iOS相关逆向调试命令行有哪些？

**nm**: 列出.o .a .so中的符号信息，包括诸如符号的值，符号类型(在哪个段)及符号名称等

**otool** ：查看程序依赖哪些动态库信息，反编代码段

**ldid**：签名工具

**ld**命令：将几个文件（file）和库（library）结合在一起，解决引用并产生一个输出文件。ld命令可以产生一个最终链接后的镜像（final linked image），它可以是一个可执行文件（executable）、动态链接库（dylib）、资源文件（bundle）,或者使用-r 选项来产生另一个object文件。

**lipo**命令则用的比较多，可用于将不同架构的静态库合并成一个静态库，如ios模拟器的i386与真机的arm架构的静态库合并，或者将一个通用架构的文件瘦身为只支持特定架构的文件。

**top**命令可以 查看当前mac所有活动的进程，在终端敲入top，会发现统计了进程的很多列信息

**ps**命令：与Top命令不同，ps 为我们提供了进程的一次性的查看，它所提供的查看结果并不动态连续的；如果想对进程时间监控，应该用 top 工具。

## 3、iOS相关逆向工具有哪些？

**snoop-it**： web界面直接调用相关函数，跳过某些流程或流程，也可以进行运行时修改硬件标识、地理位置

**introspy**： 追踪分析，可以将追踪数据生成db和html查看

**Keychain-Dumper**： 导出keychain数据

**class-dump**： 利用runtime将存储在Mach-O文件中的头文件信息提取出来，并生成对应的.h文件

**Cycript**： 理解Objective-C语法的javascript解释器，可用于动态调试

**Reveal：**分析界面层次结构（XCode只能分析自己）

**iNalyzer**：类层次结构

**IDA和Hoppe**r

## 4、进行iOS应用攻击的主要方式有哪些？

**静态分析、动态分析、网络分析**

## 5、iOS静态分析主要有哪些手段？

**文件系统取证**：可以获取短信数据库、联系人数据库、电话记录、**keychain、键盘输入缓存、文件还原**

**关键符号**：nm命令、otool命令、hopper、IDA、class-dump、iNalyzer

## 6、

**lldb、cycript、introspy、snoop-it**

## 7、如何防护iOS静态分析？

**1、文件系统取证**：

**使用NSFileProtection文件加密选项：使用数据保护API**

**混淆相关字符串：**URL进行编码加密，防止URL被静态分析

**本地数据加密**，帮扩keychain、SQLite

**安全的文件擦除**

**使用安全键盘**：键盘缓存会被明文缓存，、输入框是密码框的数据不会被缓存、全部由数字组成的字符串不会被缓存、禁用文本框的自动更正功能可以防止数据被缓存。

**禁用应用程序屏幕快照**： [UIApplication sharedApplication].keyWindow.hidden = YES;

**2、关键符号**：

**减少编译后的符号**，编译优化，或者使用static裁掉函数符号

**混淆**：使用16位字面量字符串、异或混淆，在Buildphases添加脚本、llvm编译

花指令：

## 8、如何预防iOS动态分析？

**1、生成\_restrict section**

在Build Settings中找到“**Other Linker Flags**”

在其中加上

**-Wl,-sectcreate,\_\_RESTRICT,\_\_restrict,/dev/nul （iOS10以下）**

通过阅读[dyld](https://opensource.apple.com/source/dyld/dyld-210.2.3/src/dyld.cpp" \t "_blank)源代码，我们可以得知其大概原理

**当dylib加载路径是以 @executable\_path、@loader\_path 或者不是以 ‘/‘开头，则会抛出异常使进程结束。**

方程序生成\_\_RESTRUCT/\_restrict节，调试阶段dyld就无法在程序加载时加载对应的dylib文件。这个方法可以在程序生成过程众添加一个简单的编译参数实现

**2、保证运行时安全**：

objective-C运行时映射使得大部分的ASLR保护失效。

在用户经过认证和数据经过加密前，永远不要在内存众存储任何数据。

不要使用objective-C实例变量加密任何密钥或者重要的数据、不要在实例变量里存储指向密钥或者其他重要数据的指针。

在任何可能的情况下，当重要数据不再需要时，对其进行擦除。清除内存、memset([ntData bytes], 0, [myData length]);

**3、重要逻辑使用C而不用OC，避免采用运行时**

**4、暗桩检查，然后进行更改响应，crash掉**

**5、防止调试器依附**：**PT\_DENY\_ATTACH** 标志位可以指定单个调用的ptrace指定跟踪机制为不允许程序被调试。但可以进行在ptrace函数前加断点绕过。**ptrace\_ptr**(PT\_DENY\_ATTACH, 0, 0, 0)；

**6、检查内存地址空间**：

动态链接库中包含了一个**dladdr**的函数，这个函数用于返回特定函数的内存空间信息。通过提供一个类的方法的指针，它可以验证起始地址是否来自你的程序，苹果的基础库或恶意源。防止关键函数被hook

**7、内联函数的使用**

内联函数是编译器将函数功能插入到每处函数调用的地方，换句话说，内联函数不再是一个函数。一般来说，内联函数通过增加文件体积来提供程序性能。使用内联函数进行安全检查会迫使攻击者捕捉每次代码出现的地方，并进行内存补丁或者找到一个替代的方法来攻击， 除非使用static关键字，否则初始函数仍然会存在程序内部，并且包含在符号表中

**8、优化标记，去除符号，循环展开**

## 9、如何检测iOS是否越狱？

1、检查**cryptid**是否为0

2、CodeResources、Info.plist MD5校验

3、**fork函数**可以允许你的程序生成一个新的进程，如果沙盒未被破坏，则fork函数失败，否则，则成功。

4、**文件系统检测，检测第三方文件是否存在**，如Cydia.app,以及其他的如cydiasubstrate等文件、/bin/bash

5、**检测当前程序运行的环境变量**

void printEnv(void)

{

charchar \*env = **getenv**("**DYLD\_INSERT\_LIBRARIES**");

NSLog(@"%s", env);

}

未越狱设备返回结果是null

## 10、如何分析竞品使用了哪些函数？

静态分析即可

**nm命令、otool命令、hopper、IDA、class-dump、iNalyzer**

## 11. iOS安全机制有哪些？

**1、安全沙盒**

**进程隔离，每个程序都有自己的虚拟地址空间。应用程序在安装之后，系统就通过计算得到一个标识**，然后基于应用程序的根目录和这个标识构件一个指向应用程序目录的路径，其他应用程序都不能进行访问。iOS 的沙箱是基于TrustBSD策略框架的内核扩展模块，针对每个进程都可以制定特殊的沙箱配置文件，**沙箱配置文件编译**后以2进制的方式保存在KernelCache文件中(iOS下）

**2、代码签名**

所有的可执行文件、库文件都需要Apple签名后才可以运行在iOS中，内核会在调用**execve**之前检测Mach-o文件中的**LC\_CODE\_SIGNATURE段是否有效**和可信任的，iOS启动的时候同样也会检测KernelCache的签名是否有效，iOS最重要的安全机制。所有的二进制文件和类库在被iOS内核允许运行之前都必须经过apple的签名。此外在内存中，只有那些来自 已经签名的来源程序的页（page）才会被执行

**3、ASLR(address space layout randomisation)**：**地址空间布局随机化**。使得iOS中的二机制文件、库文件、动态链接库文件、堆、栈内存地址的位置全部是随机的。当系统同时具备DEP和ASLR机制，则意味着攻击者需要两个漏洞，一个用来获取代码执行权（绕过代码签名和DEP），另一个是用来获取内存地址（绕过ASLR）以执行ROP

**4、数据执行保护（**DEP, Data Excecution prevention）:**处理器能区分哪些内容是可执行代码，哪些内存是数据，DEP不允许数据的执行，只允许代码执行**。针对启用DEP的系统的攻击方法是利用ROP来创建一块可写入且可执行的内存区域（只有DEP就不会执行），由于iOS中的代码签名机制，攻击者无法将攻击代码写入磁盘。攻击者通常使用ROP( return-Oriented Programming ，面向返回的程序设计）技术绕过DEP。

**5、文件系统加密（数据保护API）**

Data protection APIs

NSFileProtectionNone

NSFileProtectionComplete

NSFileProtectionCompleteUnlessOpen

NSFileProtectionCompleteUntilUserAuthentication

KSecAttrAccessibleAlways

KSecAttrAccessibleWhenUnlocked

KSecAttrAccessibleAfterFirstUnlock

KSecAttrAccessibleAlwaysThisdeviceOnly

KSecAttrAccessibleWhenUnlockedThisDeviceOnly

KSecAttrAccessibleAfterFirstUnlockThisDeviceOnly

## 12、一个应用程序编译成二进制后，还有相关函数的符号吗？

根据编译选项的不同，可能有，在相关section里

## 13、一个符号可以有多个内存地址吗？

可以，多个库引用同一个动态库的一个符号，则此符号有多个地址

## 14、如何做iOS应用混淆？

使用**16位字面量字符串、异或混淆，在Buildphases添加脚本随机替换、llvm编译**

## 15、iOS应用中做过哪些安全措施？

**混淆、加解密、暗桩、符号裁剪、安全、禁用应用快照、预防gdb依附、https、越狱检测、模拟器检测、循环展开-funroll-loops**

威胁感知：模拟器检测、**越狱检测、Hook检测**

安全加固： **反调试、暗桩API、混淆**、花指令、签名检测

模拟器检测：**TARGET\_IPHONE\_SIMULATOR**

## 16、如何减少iOS中的符号表的暴露

**使用C来写业务逻辑、使用static 来裁剪符号表的暴露、编译选项：循环展开-funroll-loops**

1、**BuildSettings->Optimization Level，Xcode默认设置为“Fastest ,Smallest”**，保持默认即可。

2、**Build Settings-> Linking->Dead Code Stripping** 设置成 YES

3、**Deployment Postprocessing 设**置成YES

4、**Strip Linked Product** 设置成YES

5、**工程的Enable C++ Exceptions和Enable Objective-C Exceptions选项都设置为NO**。手动管理异常。

6、**symbols hidden by default选项设置为YES**。

7、所有没有使用C++动态特性的lib库（搜索工程没有使用dynamic\_cast关键字） Enable C++ Runtime Types 选项设置为NO。

## 17、谈一谈iOS数据保护API

可以使用NSFileManager或者NSData来设置文件的加密，NSFileManager 通过设置文件的特性来进行文件的加密，NSData通过指定创建实例时的选项来进行文件的加密，NSFileManager使用的常量以NSFileProtection开头，NSData使用的常量选项以NSDataWritingFileProtection开头;

主要有四个选项：

**none选项**：NSFileProtectionNone、NSDataWritingFileProtectionNone，不使用文件加密功能。

**complete选项**：系统锁屏10秒后所有文件会被加密，这个是最高级别的保护，当应用在后台进行运行时，使用这个选项，文件系统可能无法使用，当系统解锁后文件会被解密。

**completeUnlessOpen选项**：如果应用在后台运行，则文件不进行加密，只有在应用不运行情况下锁屏10秒钟后文件才进行加密。当系统解锁，文件会被 解密。

**completeUntilFirstUserAuthentication选项**：只有在设备重启和用户解锁中间的时间文件才会加密，解锁后文件会解密不受保护，这个选项也可以让运行在后台的应用访问文件系统。

NSFileManager例子：

[[NSFileManager defaultManager] createFileAtPath:[self filePath] contents:[@"super secret file contents" dataUsingEncoding:NSUTF8StringEncoding] attributes:[NSDictionary dictionaryWithObject:NSFileProtectionComplete forKey:NSFileProtectionKey]];

NSData例子：

[data writeToFile:path options:NSDataWritingFileProtectionComplete error:&error]

**如果应用需要知道数据保护是否可用，可以使用以下三种方法：**

1、在应用代理中实现**applicationProtectedDataWillBecomeUnavailable:** 方法和applicationProtectedDataDidBecomeAvailable: 方法。

2、监听通知 **UIApplicationProtectedDataWillBecomeUnavailable**通知和UIApplicationProtectedDataDidBecomeAvailable通知。

3、使用方法 **[[UIApplication sharedApplication] protectedDataAvailable]**

## 18、谈一谈iOS代码签名机制

所有的可执行文件、库文件都需要Apple签名后才可以运行在iOS中，**内核会在调用execve之前检测Mach-o文件中的LC\_CODE\_SIGNATURE段是否有效和可信任的，iOS启动的时候同样也会检测KernelCache的签名是否有效，iOS最重要的安全机制**。所有的二进制文件和类库在被iOS内核允许运行之前都必须经过apple的签名。此外在内存中，只有那些来自 已经签名的来源程序的页（page）才会被执行。

## 21、说说你对Keychain的了解？

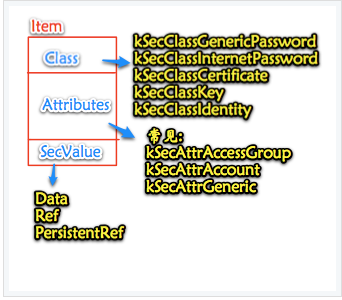
Keychain中所存储的条目，被称之为Keychain item，其实是一个字典似得数据结构，即一个key对应一个value，我们操作Keychain中的数据的时候，也是以字典的结构来进行操作的。

**Keychain item中存储的数据可以划分为三个区域：**

一个表明存储的数据类型，其key前缀为**kSecClass\***

一组描述数据信息的属性，其key前缀为**kSecAttr\***

存储敏感数据的内容，其key前缀为**kSecValue\***



**kSecClass，5种类型：**

kSecClassGenericPassword used to store a generic password

kSecClassInternetPassword used to store an internet password

kSecClassCertificate used to store a certificate

kSecClassKey used to store a kryptographic key

kSecClassIdentity used to store an identity (certificate + private key)

相关操作：**SecItemCopyMatching// 查询、SecItemAdd// 添加、SecItemUpdate// 更新SecItemDelete//删除**

操作需要指明条件来说明我们要进行操作的Keychain item。（通过kSecClass和对应的kSecAttr来指明）。

**步骤**：

1、创建一个NSMutableDictionary对象，用于存储或用来设置query条件

2、设置kSecClass，指明我需要获取什么类型数据

3、设置对应的kSecAttr属性，用来指明我需要对哪个Keychain item进行操作。

4、调用IOS中的增删改查函数，传入字典查询条件，并获得对应的结果。

对于重要的要加密的信息，IOS也是通过几个关键属性来对应生成私钥进行加解密的。对于每个class的关键属性

性能优化

## 1、Instruments有哪些profile template？

1、**Activity Monitor:** 监视CPU, 内存，磁盘和网络的使用情况。

2、**Allocations**:跟踪一个进程的虚拟内存和堆的分配，提供一个对象被retain和release的记录

3、Cocoa Layout:观察NSLayoutConstraint的变化来确定什么时候在什么地方一个layout消失或显示

4**、CoreAnimation**: 这个模板提供了**应用绘图的性能**和通过Time profile提供CPU的使用情况

5、Dispatch：监控dispatch queue的活动，记录block调用和周期。

6、Energy Diagnostics: 记录电池的使用情况和基本的开关状态

7、**GPU Driver**：**评测GPU 驱动数据和采样活动的CPU使用量**

8**、Leaks**：评测内存的使用，检查泄露的内存，并提供一个类的分配和内存地址历史纪录信息。

9、Connections：分析如何使你的应用使用TCP/IP 和UDP/IP的连接来连接的。

10、OpenGL ES Analysis: 检测OpenGL ES的活动和性能问题

11、**Sudden Termination**：分析某个进程突然奔溃的信息，并提供文件系统操作和是否闪退开关的backtraces

12、System Trace; 通过提供线程调度，并且展示所有从用户转化未系统调用和内存操作的过程。

1**3、Time Profile**: 通过对一个在CPU上进行基于时间的低功耗采样提供信息。

## 2、Instrument Time Profile采集数据的两种方式？

1、**Immediate display**(立即显示)：采集期间，Instruments应用在跟踪面板和详细面板立即显示收集的数据。

2、**Deferred display**(延迟显示)。Instruments 应用延迟显示收集的数据直到记录

停止。在测量期间，Instruments 对系统的影响非常小。选择 File > Record Options > Deferred Display。

## 3、UITableView如何进行优化

**高度计算并缓存**

预缓存机制在没有滚动的空闲时刻执行（利用Runloop空闲时间）

原理：1、在NSDefaultRunloopMode下，在Runloop进行休眠前执行高度计算相关逻辑

2、将计算分解长多个Source0任务（使用performSelector来进行发送Source0）

table cell及其子视图的opaque属性设为YES

使用**享元模式复用**cell，减少cell的创建

加载图片和数据使用**多线程**，不要阻塞主线程

**减少cell的视图层级**

**异步绘制**（自定义绘制，dispatch\_async（里面进行绘制，绘制完之后在主线程进行设置图片）

**避免圆角**等特效造成离屛渲染

## 4、如何高效地绘制图片圆角

直接使用**cornerRadius、masksToBounds**会触发离屛渲染。

ios9.0之后对UIImageView的圆角设置做了优化，UIImageView这样设置圆角，不会触发离屏渲染，而UIButton还是都会触发离屏渲染

通过混合图层，ios9.0之前还是会触发离屏渲染。

**shouldRasterize = YES设置光栅化，可以使离屏渲染的结果缓存到内存中存为位图，** 使用的时候直接使用缓存，节省了一直离屏渲染损耗的性能

此方法就是在要添加圆角的视图上再叠加一个部分透明的视图，只对圆角部分进行遮挡。图层混合的透明度处理方式与mask正好相反。此方法虽然是最优解，没有离屏渲染，没有额外的CPU计算，但是应用范围有限。

在其他线程进行通过Core Graphics重新绘制带圆角的视图

## 5、什么是Off-Screen Rendering，如何避免？

**离屏渲染，指的是在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作**

主要有创建缓冲区和上下文切换。创建新的缓冲区代价都不算大，付出最大代价的是上下文切换

**首先我要****保存当前屏幕渲染环境，然后切换到一个新的绘制环境，申请绘制资源，初始化环境，然后开始一个绘制，绘制完毕后销毁这个绘制环境，如需要切换到On-Screen-Rendering或者再开始一个新的离屏渲染重复之前的操作。**

\* shouldRasterize（光栅化）

\* masks（遮罩）

\* shadows（阴影）

\* edge antialiasing（抗锯齿）

\* group opacity（不透明）

需要注意的是，**如果shouldRasterize被设置成YES，在触发离屏绘制的同时，会将光栅化后的内容缓存起来**，如果对应的layer及其sublayers没有发生改变，在下一帧的时候可以直接复用。这将在很大程度上提升渲染性能

不使用此相关API，使用CPU进行预先绘制，即重写了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。**使用异步绘制，drawsAsynchronously， drawRect方法仍然是在主线程调用，但coregraphic相关命令在后台执行。**

## 6、离屛渲染如何检测和优化？

offscreen-render涉及的内容比较多，有offscreen-render那就有onscreen render，onscreen render指的是GPU在当前用于显示的屏幕缓冲区进行渲染，相反offscreen-render就是不在当前的屏幕缓存区，而在另外的缓冲区进行渲染，offscreen-render有两种形式

**CPU的offscreen-render**：使用CPU来完成渲染操纵，通常在你使用：drawRect (如果没有自定义绘制的任务就不要在子类中写一个空的drawRect方法，因为只要实现了该方法，就会为视图分配一个寄宿图，这个寄宿图的像素尺寸等于视图大小乘以 contentsScale的值，造成资源浪费)

使用Core Graphics上面的两种情况使用的就是CPU离屏渲染，首先分配一块内存，然后进行渲染操作生成一份bitmap位图，整个渲染过程会在你的应用中同步的进行，接着再将位图打包发送到iOS里一个单独的进程--render server，理想情况下，render server将内容交给GPU直接显示到屏幕上。

**GPU的offscreen-render**

使用GPU在当前屏幕缓冲区以外开辟一个新的缓冲区进行绘制，通常发生的情况有：

设置cornerRadius, masks, shadows,edge antialiasing等

设置layer.shouldRasterize ＝ YES

对于圆角可以使用一张中间圆形透明的图覆盖在上面，虽然这会引入blending操作，但是大部分情况下性能会比离屏渲染好。

减少视图层级，因为OpenGL在渲染layer的时候，在碰到有子层级layer的时候可能需要停下来把两者合成到一个buffer里再接着渲染

## 5、iOS程序main函数之前发生了什么？

1. 系统kernel做好启动程序的初始准备后，交给dyld负责，dyld开始将程序二进制文件初始化

2. 交由ImageLoader读取image，其中包含了我们的类、方法等各种符号

3. 由于runtime向dyld绑定了回调，当image加载到内存后，dyld会通知runtime进行处理

4. runtime接手后调用map\_images做解析和处理，接下来load\_images中调用call\_load\_methods方法，遍历所有加载进来的Class，按继承层级依次调用Class的load方法和其Category的load方法。

整个事件由dyld主导，完成运行环境的初始化后，配合ImageLoader将二进制文件按格式加载到内存，

动态链接依赖库，并由runtime负责加载成objc定义的结构，所有初始化工作结束后，dyld调用真正的main函数。

## 6、Dyld 加载的步骤

**1. Load dylibs**

**这一阶段dyld会分析应用依赖的dylib，找到其mach-o文件，打开和读取这些文件并验证其有效性，接着会找到代码签名注册到内核，最后对dylib的每一个segment调用mmap()。**

一般情况下，iOS应用会加载100-400个dylibs，其中大部分是系统库，这部分dylib的加载系统已经做了优化。

**优化点：**

尽量不使用内嵌（embedded）的dylib，加载内嵌dylib性能开销较大

合并已有的dylib和使用静态库（static archives），减少dylib的使用个数

懒加载dylib，但是要注意dlopen()可能造成一些问题，且实际上懒加载做的工作会更多

强引用（Required）的framework是一定会被加载到内存的，但是**弱引用（Optional）的framework只在需要时才会被载入内存**，这对于比较大的framework来说，在最初加载的时候会省很多时间

**2. Rebase/Bind**

在dylib的加载过程中，系统为了安全考虑，引入了ASLR（Address Space Layout Randomization）技术和代码签名。由于ASLR的存在，镜像（Image，包括可执行文件、dylib和bundle）会在随机的地址上加载，和之前指针指向的地址（preferred\_address）会有一个偏差（slide），dyld需要修正这个偏差，来指向正确的地址。

Rebase在前，**Bind在后，Rebase做的是将镜像读入内存，修正镜像内部的指针，性能消耗主要在IO。Bind做的是查询符号表，设置指向镜像外部的指针，性能消耗主要在CPU计算**。

**优化点：**

减少ObjC类（class）、方法（selector）、分类（category）的数量

减少C++虚函数的的数量（创建虚函数表有开销）

使用Swift structs（内部做了优化，符号数量更少），使用值类型

**3. Objc setup**

大部分ObjC初始化工作已经在Rebase/Bind阶段做完了，**这一步dyld会注册所有声明过的ObjC类，将分类插入到类的方法列表里，再检查每个selector的唯一性**。

**4. Initializers**

到了这一阶段，dyld开始运行程序的初始化函数，**调用每个Objc类和分类的+load方法，调用C/C++ 中的构造器函数（用attribute((constructor))修饰的函数），和创建非基本类型的C++静态全局变量**。Initializers阶段执行完后，dyld开始调用main()函数。

**优化点：**

少在类的+load方法里做事情，尽量把这些事情推迟到+initiailize

减少构造器函数个数，在构造器函数里少做些事情

**减少C++静态全局变量的个数，用dispatchonce()代替所有的\_\_attribute\_\_((constructor))函数、C++静态对象初始化、ObjC的+load**

## 7、iOS应用启动的加载过程？

**Main函数前**：见上

**Main函数后**：

UIKit初始化：如果应用的Root View Controller是由XIB实现的，也在启动时被初始化

应用回调：调用UIApplicationDeleagte的回调application:didFinishLaunchingWithOptions

第一次Core Animation调用：在启动后的方法-[UIApplication \_resportAppLaunchFinished]中调用CA::Transaction::commit实现第一帧画面的绘制。如果你的程序启动很慢，能 做的首先是将与显示第一屏画面无关的操作放到之后执行；如果是用XIB文件load第一屏，XIB文件中的View层也要如果扁平，不要有太多图层。

## 8、如何优化iOS应用的启动时间？

使用DYLD\_PRINT\_STATISTIC测试启动时间， 在Environment Variables中设置DYLD\_PRINT\_STATISTIC的值为1，可以进行打印启动时间。Main函数后的时间可使用CFAbsolutionTime来进行测量

**Main函数前的优化：**

见dyld部分的结论

**Main函数后：**

梳理业务进行延迟加载部分逻辑

使用多线程处理部分逻辑

用dispatch\_once()代替所有的\_\_attribute\_\_((constructor))函数、C++静态对象初始化、ObjC的+load

## 9、造成APP启动过慢可能的原因有哪些？

main函数前

动态库加载过多，启动慢

Objc类多，category多，启动慢

C的constructor函数多，启动慢

C++静态对象越多，启动越慢

load()方法过多

main()函数后

willFinishLaunching等执行了过多的逻辑

## 9、如何检测iOS内存问题？

**Instrument 使用Allocations**：监测内存使用/分配情况迅速膨胀的内存可以很快让程序毙命。

**Leaks**：找到引发内存泄漏的起点

在Allocations的Heapshot Analysis中有个**MarkHeap**功能，可以进行点击markHeap，进入某个页面然后再退出再次点击 **mark heap**进行对比。

另一种严重的内存使用问题是引用了已经释放的内存，直接导致应用崩溃。

而Allocation有一个选项**Enable NSZombie detection**能够在应用使用已经释放的内存时标注出来，同时显示错误发生的调用栈信息。这为解决问题提供了最直接的帮助，当然缺点是必须能够重现EXEC\_BAD\_ACCESS错误。

## 10、如何进行iOS内存优化？

避免过大的xib

使图片符合尺寸

**处理低内存警告**

**重用高消耗对象**

临时创建大量对象时使用@autoreleasepool

减少视图层级

避免单利模式泛滥

避免对象生命周期的延长

## 11、如何检测iOS卡顿并定位卡顿函数？

费时函数和绘制性能

1、Instruments使用**Time Profiler（获取代码运行时间,一般用来看CPU占用**）h和**Core Animation(获取图形绘制情况，FPS，离屏渲染等)**来进行检测

2、使用**Runloop**进行检测，同时打印函数栈。

3、造成卡顿可能是CPU，也可能是GPU，使**用Activity Monitor**， **点击耗时函数栈可前往查看**

4、直接使用**CADisplayLink显示FPS**（次数/时间就可以得出当前屏幕的刷新率，1秒的次数）

## 12、如何优化iOS应用网络请求？

**DNS解析**，可以做本地解析，并使用ping查找速度最优的IP

**网络质量检测**（根据网络质量来改变资源请求策略）

**提供网络服务优先级和依赖机制**

**提供网络服务重发机制**

**减少数据传输量**

**缓存机制（etag、cache control等）**

**启用压缩gzip**

**新技术（SPDY和QUIC）或者protobuf**

**关注网络优化主要是三个方面：**

**速度、弱网、安全**

## 13、如何进行iOS包瘦身？

1、**编译器优化级别**

Build Settings->Optimization Level有几个编译优化选项，release版应该选择Fastest, Smalllest，这个选项会开启那些不增加代码大小的全部优化，并让可执行文件尽可能小。

2、**去除符号信息**

**Strip Linked Product** / Deployment Postprocessing / Symbols Hidden by Default 在release版本应该设为yes，可以去除不必要的调试符号

3、**删除无用代码**

在项目里新建一个类，给它添加几个方法，但不要在任何地方import它，build完项目后观察linkmap，你会发现这个类还是被编译进可执行文件了。

**按C++的经验，没有被使用到的类和方法编译器都会优化掉，不会编进最终的可执行文件，但object-c不一样，因为object-c的动态特性**，它可以通过类和方法名反射获得这个类和方法进行调用，所以就算在代码里某个类没被使用到，编译器也没法保证这个类不会在运行时通过反射去调用

**4、类/方法名长度**

**5、避免使用xib、storyboard**

**6、删除冗余资源和图片（图片很占大小）（压缩相关图片）**

7、Clang的无用代码分析插件

8、**Mach-O 分析**：**从 LinkMap文件的\_\_TEXT.\_\_text找到所有的方法，使用otool命令otool -v -s \_\_DATA \_\_objc\_selrefs逆向\_\_DATA.\_\_objc\_selrefs段**，**无用的class可以通过otool命令逆向\_\_DATA.\_\_objc\_classlist段和\_\_DATA.\_\_objc\_classrefs段来获取当前所有oc类和被引用的oc类，两个集合相减就是无用oc类**

**9、App Thin技术**

**其他工具:**

**LSUnusedResources可以用来扫描工程中没有用到的图片**

**fdupes可以在指定的目录及子目录中查找重复的文件**

**TinyPNG很不错的一个图片免费在线压缩工具进行压缩图片**

## 14、可以进行iOS安装包瘦身的编译选项有哪些？

优化编译选项

1、**BuildSettings->Optimization Level**，Xcode默认设置为“Fastest ,Smallest”，保持默认即可。

2、**Build Settings-> Linking->Dead Code Stripping** 设置成 YES

3、**Deployment Postprocessing** 设置成YES

4、**Strip Linked Product** 设置成YES

5、**工程的Enable C++ Exceptions和Enable Objective-C Exceptions选项都设置为NO**。手动管理异常，可以对某些文件单独支持异常，**编译选项加上-fexceptions即可**。关键路径进行异常捕获

6、**symbols hidden by defaul**t选项设置为YES。

7、所有没有使用C++动态特性的lib库（搜索工程没有使用dynamic\_cast关键字） Enable C++ Runtime Types 选项设置为NO。

## 14、如何使用Instrument检验离屛渲染？

**CoreAnimation两个选项**

**Color Offscreen-Rendered Yellow**：开启后会把那些需要离屏渲染的图层高亮成黄色，这就意味着黄色图层可能存在性能问题。

**Color Hits Green and Misses Red**：如果shouldRasterize被设置成YES，对应的渲染结果会被缓存，如果图层是绿色，就表示这些缓存被复用；如果是红色就表示缓存会被重复创建，这就表示该处存在性能问题了

## 15、如何使用Time profile进行iOS性能优化？

使用Instrument的timer profile来进行分析代码的执行时间，找出导致程序变慢的原因。

配置相关CallTree选项来查看耗时函数。

**Separate by Thread：**

按线程分开做分析，这样更容易揪出那些吃资源的问题线程。特别是对于主线程，它要处理和渲染所有的接口数据，一旦受到阻塞，程序必然卡顿或停止响应。

**Invert Call Tree：**

反向输出调用树。把调用层级最深的方法显示在最上面，更容易找到最耗时的操作。

**Hide Missing Symbols：**

隐藏缺失符号。如果 dSYM 文件或其他系统架构缺失，列表中会出现很多奇怪的十六进制的数值，用此选项把这些干扰元素屏蔽掉，让列表回归清爽。

**Hide System Libraries：**

隐藏系统库文件。过滤掉各种系统调用，只显示自己的代码调用。

**Flattern Recursion：**

拼合递归。将同一递归函数产生的多条堆栈（因为递归函数会调用自己）合并为一条。

**Top Functions：**

找到最耗时的函数或方法。

**如果想要在TimeProfile中直观的查看方法耗时**，需要对Xcode进行设置 在Xcode->Build Setting->**Debug Information Format中设置选项为：DWARF with DSYM File**

**Time Profiler工具中的”Recod thread waiting”选项可以统计出app运行时各个线程中的阻塞系统调用情况**，例如文件读写read/write，网络读写send/recv，加锁psynch\_mutex\_wait等。Instruments中的System Trace工具则能够记录所有的底层系统调用。

**NSDateFormatter**问题凸显、图片加载API耗时，imageNamed默认加载图片成功后会内存中缓存图片,而imageWithContentsOfFile则仅只加载图片,不缓存

**语言方面：尽量不要使用runtime，swift使用final、开启Swift编译优化等级**

Swift编译优化等级：

1、-Onone

适用于开发阶段，编译器进行了最低限度的优化并保留了debug信息。

2、-O

这适合于绝大多数的线上环境代码，编译器进行了单文件编译优化，使得XCode可以并行编译多个文件，但是并行编译使得编译器的一些特定优化无法进行。

3、-O -whole-module-optimization

将整个工程作为一个文件进行编译，编译时间更长，但优化后运行的性能更高。

## 16、如何分析网络请求的瓶颈

1、**Ponnydebugerg**可以进行相关网络性能的分析。

2、使用**charles**分析模拟慢网

3、可以**Hook URLConnection CFNetwork、NSURLSession**等api

4、因为我们所使用的URLConnection、CFNetwork、NSURLSession底层都是 **BSDSocket**，所以可以尝试在socket上动手脚来实现效果，类似于通过ViewController的生命周期方法来统计页面加载时间的做法，我们Hook socket相关的方法来做，比如通过hook socket连接时的 connect方法，拿到tcp握手的起始时间，通过hook SSLHandshake方法，在SSLHandshake执行的时候拿到 SSL握手的起始时间等。

5、iOS 9 Apple 加入 **ATS** 新特性，并要求开发者使用 HTTPS，我在 iOS9、10上对 HTTPS 网络请求Hook socket方法时候，有一些方法hook 失效，猜想应该是Apple 进行了加固、加密，导致一些系统方法没办法hook

不过apple在 iOS 10 推出一个API，可以在 iOS10 版本以上进行网络信息的收集

6、- (void)**URLSession**:(NSURLSession \*)session task:(NSURLSessionTask \*)task **didFinishCollectingMetrics**:(NSURLSessionTaskMetrics \*)metrics。

## 17、drawRect方法注意什么？什么时候使用绘制

调用以下方法时候进行调用

- (void)**setNeedsDisplay**;:标记为需要重绘，异步调用drawRect，但是绘制视图的动作需要等到下一个绘制周期执行，并非调用该方法立即执行;

- (void)**setNeedsDisplayInRect**:(CGRect)rect;:标记为需要局部重绘，具体调用时机同上;

drawRect时用的UIimageView , layer图层却没有申请一个后备存储。取而代之的是使用一个 CGImageRef 作为他的内容，并且渲染服务将会把图片的数据绘制到帧的缓冲区，比如，绘制到显示屏。从帧缓冲区，去取出然后显示，省去了重新绘制，在这种情况下，将不会继续重新绘制。我们只是简单的将位图数据以图片的形式传给了 UIImageView，然后 UIImageView 传给了 Core Animation，然后轮流传给渲染服务。(其实绘图相当于cpu做渲染，也叫离屏渲染，**但是性能的影响，远小于GPU的离屏渲染，所以在GPU使用负荷较大的时候可以把部分计算和渲染交给cpu)**

需要对比CPU绘制和GPU离屛渲染绘制的性能。

## 18、如何用Core Animation检测绘制性能？

Core Animation相关的检查选项：

**Color Blended layers**

勾选这个选项后,**blended layer 就会被显示为红色,而不透明的layer则是绿色**。我们希望越少红色区域越好。

**Color Hits Green and Misses Red**

如果shouldRasterize被设置成YES，对应的渲染结果会被缓存，如果图层是绿色，就表示这些缓存被复用；如果是红色就表示缓存会被重复创建，这就表示该处存在性能问题了。如果光栅化的层变红得太频繁那么光栅化对优化可能没有多少用处。位图缓存从内存中删除又重新创建得太过频繁，红色表明缓存重建得太迟。可以针对性的选择某个较小而较深的层结构进行光栅化，来尝试减少渲染时间。

**Color copied images**

这个选项主要检查我们有无使用不正确图片格式,若是GPU不支持的色彩格式的图片则会标记为青色,则只能由CPU来进行处理。我们不希望在滚动视图的时候,CPU实时来进行处理,因为有可能会阻塞主线程。

**Color misaligned images**

这个选项检查了图片是否被放缩,像素是否对齐。被放缩的图片会被标记为黄色,像素不对齐则会标注为紫色。

**Color Offscreen-Rendered Yellow**

开启后会把那些需要离屏渲染的图层高亮成黄色，这就意味着黄色图层可能存在性能问题。

## 19、Activity Monitor 检测 CPU渲染性能？

使用Activity Monitor，**点击耗时函数栈**可前往查看

## 20、GPU driver 检测GPU渲染性能？

GPU Driver最主要的两项：

**Renderer Utilization** - 如果这个值超过了~50%，就意味着你的动画可能对帧率有所限制，很可能因为离屏渲染或者是重绘导致的过度混合。

**Tiler Utilizatio**n - 如果这个值超过了~50%，就意味着你的动画可能限制于几何结构方面，也就是在屏幕上有太多的图层占用了。

## 21、什么是IP直连，原理是什么？

HTTPDNS是客户端基于http协议向服务器A发送域名B解析请求（例如：www.baidu.com），服务器A直接返回域名B对应的ip地址（例如：119.75.217.109。

防劫持，可以绕过运营商 LocalDNS 解析过程，避免域名劫持，提高网络访问成功率

降低延迟，DNS 解析是一个相对耗时的工作，跳过这个过程可以降低一定的延迟。

实现 HTTP 协议下 IP 连接其实是很简单的，我们只需要通过 NSURLProtocol 来拦截网络请求，然后将符号条件的网络请求 URL 中的域名修改为 IP 就可以啦。单个，多个可进行预先获取。

ATS下，一个 IP 对应多个域名的情况，这种情况如果客户端 IP 直连的时候，没有告诉服务端他要请求的是哪个域名的证书，服务端就没办法返回正确的证书，从而导致客户端证书校验失败，我们需要手动的将 host 字段塞到 header 中去，方便服务器的正确识别，

SNI 的全称 **Server Name Indication**，为了解决一个服务器使用多个域名和证书的 SSL/TLS 扩展。在连接到服务器建立 SSL 连接时，客户端可以在第一次握手的 Client Hello 中的 SNI 扩展字段中填入要访问站点的域名（Hostname），

我们在进行 IP 直连的时候，面对单 IP 多域名的情况需要客户端手动配置 SNI 字段，但是上层的网络库 NSURLSession、NSURLConnection 都没有提供配置的接口，我们需要使用更加底层的 libcurl 库和 CFNetwork 来完成 SNI 字段的配置。

**问题1：服务器无法判断请求访问的内容：**

由于服务器是根据host字段来判断请求的服务，所以在发起网络请求时，用带ip的URL生成request后，手动将request中的host字段改回域名。

**问题2：域名替换并修改host后，无法通过https证书校验**操作系统

**解决：**我们hook证书校验过程中的证书中的域名校验，将请求中的ip再替换回域名后，再执行证书验证。

我们项目中使用的网络库是AFNetworking，AFN已经封装好了会在证书校验时执行的block，我们要做的就是在生成request时候，在block中写好替换host的代码就可以了。

**问题3：由于web页面的请求并不是由客户端发起，我们无法在生成request的时候修改host。**

**解决：在这里我们使用NSURLProtocol来解决。**

## 22、深层次UI视图为什么性能低？

Layer blend和Opacity（alpha）的计算

## 23、离屏渲染切换上下文为什么造成性能下降？

保存当前屏幕渲染环境，然后切换到一个新的绘制环境，申请绘制资源，初始化环境，然后开始一个绘制，绘制完毕后销毁这个绘制环境

## 24、如何进行appdelegate瘦身？

1、FRDModuleManager初始化各module，并在各appdelegate的代理方法来加入钩子

2、JSDecoupledAppDelegate，将delegate拆分成几个delegate，并分别进行设置和实现

3、AppDelegate分类(Category)

4、MGJRouter

5、Objection

## 25、如何进行网络测速？

ping 测的是点到点的网络延迟，而网速指的是点到点之间单位时间内的传输数据量

方案1：通过上传和下载数据包，使用 TotalSize / TotalTime 来计算真实的上传和下载速率是多少

方案2：通过读取网卡数据来计算

**两种方案各有优劣，**可以在合适的场合来选择对应的方案

第一种方案感觉是比较准确，这个时候是真实的在下载或上传数据，比较充分的利用了当前的带宽，计算的网速也比较接近真实的网速值。但是蜂窝网络下，会消耗用户的少量流量。

第二种方案在下载和上传东西时，计算的值和第一种方案比较接近。但是如果当前系统内没有 App 在被使用，处于静止状态的话，其实当前读取的流量值是比较小的，无法反映出网速情况，但是可以实时反映流量消耗状况。

## 26、image解压过程？

1、假设我们使用 +imageWithContentsOfFile: 方法从磁盘中加载一张图片，这个时候的**图片并没有解压缩；**

2、然后将生成的 UIImage 赋值给 UIImageView ；

3、接着一个隐式的 CATransaction 捕获到了 UIImageView 图层树的变化；

4、在主线程的下一个 run loop 到来时，Core Animation 提交了这个隐式的 transaction ，这个过程可能会对图片进行 copy 操作，而受图片是否字节对齐等因素的影响，这个 copy 操作可能会涉及以下部分或全部步骤：

**a、分配内存缓冲区用于管理文件 IO 和解压缩操作；**

**b、将文件数据从磁盘读到内存中；**

**c、将压缩的图片数据解码成未压缩的位图形式，这是一个非常耗时的 CPU 操作；**

**d、最后 Core Animation 使用未压缩的位图数据渲染 UIImageView 的图层。**

在上面的步骤中，我们提到了**图片的解压缩是一个非常耗时的 CPU 操作，并且它默认是在主线程中执行**的。

**强制解压缩的原理:**

既然图片的解压缩不可避免，而我们也不想让它在主线程执行，影响我们应用的响应性，那么是否有比较好的解决方案呢？答案是肯定的。

我们前面已经提到了，当未解压缩的图片将要渲染到屏幕时，系统会在主线程对图片进行解压缩，而如果图片已经解压缩了，系统就不会再对图片进行解压缩。因此，也就有了业内解决方案，在子线程提前对图片进行强制解压缩。

**而强制解压缩的原理就是对图片进行重新绘制，得到一张新的解压缩后的位图。其中，用到的最核心的函数是 CGBitmapContextCreate**

## 27、Hybrid如何进行优化？

**1、全局WebView（第一次启动很慢，多300ms）**

在客户端刚启动时，就初始化一个全局的WebView待用，并隐藏；

当用户访问了WebView时，直接使用这个WebView加载对应网页，并展示。

**2、客户端代理数据请求**

在客户端初始化WebView的同时，直接由native开始网络请求数据；

当页面初始化完成后，向native获取其代理请求的数据。

**3、建立连接/服务器处理**

在页面请求的数据返回之前，主要有以下过程耗费时间。

DNS、connection、服务器处理

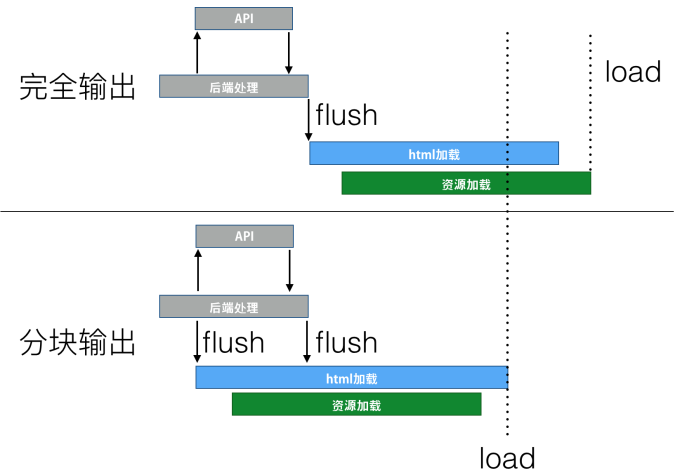
**4、同步渲染采用chunk编码**

同步渲染时如果后端请求时间过长，可以考虑采用chunk编码，将数据放在最后，并优先将静态内容flush。对于传统的后端渲染页面，往往都是使用的【浏览器】--> 【Web API】 --> 【业务 API】的加载模式，其中后端时间就指的是Web API的处理时间了。在这里Web API一般有两个作用：

确定静态资源的版本。

根据用户的请求，去业务API获取数据。

在HTTP协议中，**可以在header中设置 transfer-encoding:chunked** 使得页面可以分块输出。如果合理设计页面，让head部分都是确定的静态资源版本相关内容，而body部分是业务数据相关内容，那么我们可以在用户请求的时候，首先将Web API可以确定的部分先输出给浏览器，然后等API完全获取后，再将API数据传输给浏览器。



**5、通常情况下，CSS不会阻塞HTML的解析，但如果CSS后面有JS，则会阻塞JS的执行直到CSS加载完成（即便JS是内联的脚本），从而间接阻塞HTML的解析。**

CSS的加载会在HTML解析到CSS的标签时开始，所以CSS的标签要尽量靠前。

但是，CSS链接下面不能有任何的JS标签（包括很简单的内联JS），否则会阻塞HTML的解析。

如果必须要在头部增加内联脚本，一定要放在CSS标签之前。

**6、Native进行请求，并进行缓存，类似于offlineServer的情况**

**7、静态渲染+离线欲推**

**8、动态渲染+离线缓存**

第三方框架

## 1、SDWebImage是如何实现的？

1. 入口 setImageWithURL:placeholderImage:options: 会先把 placeholderImage 显示，然后 SDWebImageManager 根据 URL 开始处理图片。

2. 进入 **SDWebImageManager**-downloadWithURL:delegate:options:userInfo:，交给 **SDImageCache** 从缓存查找图片是否已经下载 queryDiskCacheForKey:delegate:userInfo:.

3. 先从内存图片缓存查找是否有图片，如果内存中已经有图片缓存，**SDImageCacheDelegate** 回调 imageCache:didFindImage:forKey:userInfo: 到 **SDWebImageManager**。

4. **SDWebImageManagerDelegate** 回调 webImageManager:didFinishWithImage: 到 UIImageView+WebCache 等前**端展示图片**。

5. 如果内存缓存中没有，生成 **NSInvocationOperation** 添加到队列开始从硬盘查找图片是否已经缓存。

6. 根据 URLKey 在硬盘缓存目录下尝试读取图片文件。这一步是在 NSOperation 进行的操作，所以回主线程进行结果回调 notifyDelegate:。

7. 如果上一操作从硬盘读取到了图片，将图片添加到内存缓存中（如果空闲内存过小，会先清空内存缓存）。**SDImageCacheDelegate** 回调 imageCache:didFindImage:forKey:userInfo:。进而回调展示图片。

8. 如果从硬盘缓存目录读取不到图片，说明所有缓存都不存在该图片，需要下载图片，回调 imageCache:didNotFindImageForKey:userInfo:。

9. 共享或重新生成一个下载器 **SDWebImageDownloader** 开始下载图片。

10. 图片下载由 NSURLConnection 来做，实现相关 delegate 来判断图片下载中、下载完成和下载失败。

11. connection:didReceiveData: 中利用 ImageIO 做了按图片下载进度加载效果。

12. connectionDidFinishLoading: 数据下载完成后交给 SDWebImageDecoder 做图片解码处理。

13. **图片解码处理在一个 NSOperationQueue** 完成，不会拖慢主线程 UI。如果有需要对下载的图片进行二次处理，最好也在这里完成，效率会好很多。

14. 在主线程 notifyDelegateOnMainThreadWithInfo: 宣告解码完成，imageDecoder:didFinishDecodingImage:userInfo: 回调给 SDWebImageDownloader。

15. imageDownloader:didFinishWithImage: 回调给 SDWebImageManager 告知图片下载完成。

16. 通知所有的 downloadDelegates 下载完成，回调给需要的地方展示**图片。**

**17. 将图片保存到 SDImageCache 中，内存缓存和硬盘缓存同时保存。写文件到硬盘也在一单独 NSInvocationOperation** 完成，避免拖慢主线程。

18. SDImageCache 在初始化的时候会注册一些消息通知，在内存警告或退到后台的时候清理内存图片缓存，应用结束的时候清理过期图片。

19. SDWI 也提供了 UIButton+WebCache 和 MKAnnotationView+WebCache，方便使用。

20. **SDWebImagePrefetcher** 可以预先下载图片，方便后续使用

## 2、FastImageCache的图片加载优化

iOS从磁盘加载一张图片，使用UIImageVIew显示在屏幕上，需要经过以下步骤：

从磁盘拷贝数据到内核缓冲区

从内核缓冲区复制数据到用户空间

生成UIImageView，把图像数据赋值给UIImageView

如果图像数据为未解码的PNG/JPG，解码为位图数据

CATransaction捕获到UIImageView layer树的变化

主线程Runloop提交CATransaction，开始进行图像渲染

6.1 **如果数据没有字节对齐，Core Animation会再拷贝一份数据，进行字节对齐。**

6.2 GPU处理位图数据，进行渲染。

FastImageCache分别优化了2,4,6.1三个步骤：

**使用mmap内存映射，省去了上述第2步数据从内核空间拷贝到用户空间的操作。**

**缓存解码后的位图数据到磁盘，下次从磁盘读取时省去第4步解码的操作。**

**生成字节对齐的数据，**防止上述第6.1步CoreAnimation在渲染时再拷贝一份数据。

<http://blog.cnbang.net/tech/2578/>

## 3、检测内存泄露的原理，如FBMemoryProfiler

FBMemoryProfiler 是几个组件的结合。其中包括 **FBAllocationTracker** 和 **FBRetainCycleDetector**。

**FBAllocationTracker：**可视化工具，直接嵌入到 App 中，可以起到在 App 中直接查看内存使用情况，并筛选潜在泄漏对象的作用

主要用于快速检测潜在的内存泄漏对象，并提供给 FBRetainCycleDetector 进行检测，这是一个用来主动追踪所有 NSObject 的子类的内存分配和释放操作的工具。**FBAllocationTracker 用于检测应用在运行时所有实例的分配。它的原理其实就是用 method swizzling 替换原本的 alloc 方法**。这样就可以记录下所有的实例分配了

**FBRetainCycleDetector ：接受一个运行时的实例，然后从这个实例开始遍历它所有的属性，逐级递归。 如果发现遍历到重复的实例，就说明存在循环引用，并给出报告**

## 4、AFNetworking的安全策略？

**SSL安全策略：**

**AFSSLPinningModeNone**：这个模式表示不做SSL pinning，只跟浏览器一样在系统的信任机构列表里验证服务端返回的证书。

**AFSSLPinningModeCertificate**

这个模式表示用证书绑定方式验证证书，需要客户端保存有服务端的证书拷贝，这里验证分两步，第一步验证证书的域名/有效期等信息，第二步是对比服务端返回的证书跟客户端返回的是否一致。

**AFSSLPinningModePublicKey**

这个模式同样是用证书绑定方式验证，客户端要有服务端的证书拷贝，只是验证时只验证证书里的公钥，不验证证书的有效期等信息

## 5、Aspects是如何实现的？

**aspect\_isSelectorAllowedAndTrack**：对父子类同时hook一个方法进行了一些限制

**aspect\_getContainerForObject** 通过Runtime添加关联值的方式 管理hook的方法

**aspect\_prepareClassAndHookSelector** 这是核心的实现，涉及到动态生成子类，改变isa指针的指向，改变方法的实现 一系列操作.

对于待 hook 的 selector，将其指向 **objc\_msgForward** **/ \_objc\_msgForward\_stret** ,同时生成一个新的 aliasSelector 指向原来的 IMP，并且 hook住 forwardInvocation函数，通过forwardInvocation调用到原来的IMP。

核心原理：按照上面的思路，当被 hook 的 selector 被执行的时候，首先根据 selector找到了 objc\_msgForward / \_objc\_msgForward\_stret ,而这个会触发消息转发，从而进入 forwardInvocation。同时由于forwardInvocation 的指向也被修改了，因此会转入新的 forwardInvocation函数，在里面执行需要嵌入的附加代码，完成之后，再转回**原来的 IMP**（被hook前的）。

-forwardInvocation:方法的实现给替换掉了，如果程序里真有用到这个方法对消息进行转发，原来的逻辑怎么办？首先我们在替换 -forwardInvocation:方法前会新建一个方法 -ORIGforwardInvocation:，保存原来的实现IMP，在新的 -forwardInvocation:实现里做了个判断，如果转发的方法是我们想改写的，就走我们的逻辑，若不是，就调 -ORIGforwardInvocation:走原来的流程。

**1、检查是否可以进行hook**：retain,release,autorelease,forwoardInvocation:不能被hook

**dealloc只能在方法前hook，同一个方法只能被hook一次, 类方法只能替换一次，**是在整个类的继承树上校验，而不只是单单的一个类。

**2、获取hook后的container**，将hook方法的selector添加到container

3、prepareClassAndHookSelector 动态生成子类，被hook的selectorIMP注册到内部类，保存原forwardToTarget。

4、**替换类的methodList, 将原selector的IMP替换成-forwardInvocation:,** 当调用原selector时, 将直接启动消息转发流程，获取参数，调用内部类相关方法。

5、内部类的消息转发判断是否被hook，如果被hook执行hook，没有的话执行原方法。

**如果走没有hook的方法，直接运行，如果走了hook的方法，则会在自定义的forwardInvocation中转发到我们的自定义内部类，根据hook的信息来先执行、后执行、替换逻辑，如果是真正的forwardInvocation则走我们保存的forwardInvocation**

## 6、AFNetworking3.0源码底层实现？

**网络通信模块(NSURLSession)**

**网络状态监听模块(Reachability)**

**网络通信安全策略模块(Security)**

**网络通信信息序列化/反序列化模块(Serialization)**

**对于iOS UIKit库的扩展(UIKit)**

1、系统维护了一个**NSOperationQueue**。每一次请求都会构建对应的NSOperation，然后加到该queue中。

2、执行在AFNetworking一直运行的一个Runloop中，默认情况下NSURLConnection会在RunloopMode为NSDefaultRunLoopMode时执行。这样当用户滑动ScrollView等操作时RunLoopMode为NSEventTrackingRunLoopMode，这样NSURLConnection相关的回调不会立即执行。

3、运行成功的回调也可以进行指定**completionQueue**，将一些复杂的业务回调逻辑进行异步处理，如果不指定则会在主线程调用

## 7、Reachability内部实现原理？

是对 SystemConfiguration.framework 模块中的 SCNetworkReachability.h 头文件里提供的一系列网络连接状态相关的 C 函数进行简单封装。

## 8、简述下cocoapods的实现机制？

1、它是将所有的依赖库都放到另一个名为 Pods 项目中

2、Pods 项目最终会编译成一个名为 libPods.a 的文件，主项目只需要依赖这个 .a 文件即可。这样，依赖库源码管理工作都从主项目移到了 Pods 项目中。

3、对于资源文件，CocoaPods 提供了一个名为 Pods-resources.sh 的 bash 脚本，该脚本在每次项目编译的时候都会执行，将第三方库的各种资源文件复制到目标目录中。

4、CocoaPods 通过一个名为 Pods.xcconfig 的文件来在编译时设置所有的依赖和参数

## 9、AFNetworking如何配置使用Https？

1、配置ATS的一.AllowsArbitraryLoads 白名单机制

2、免证书认证：

//允许非权威机构颁发的证书

manager.securityPolicy.allowInvalidCertificates = YES;

//也不验证域名一致性

manager.securityPolicy.validatesDomainName = NO;

//关闭缓存避免干扰测试

2、**验证方式：**单向验证和双向验证

3、**证书类型：**自签名整数vs第三方权威机构整数

**单向认证**：

**需要准备的文件**：服务端证书库 ， 服务端导出的证书

单向认证，实际上说的是只有Client端对Server端的证书进行验证，Server不需要验证Client端的证书。

// setPinnedCertificates 设置证书文件（可能不止一个证书） //设置服务器端的cer或者crt整数，不是509

[securityPolicy setPinnedCertificates:dataSet];

// allowInvalidCertificates 是否允许无效证书

[securityPolicy setAllowInvalidCertificates:NO];

// validatesDomainName 是否需要验证域名

[securityPolicy setValidatesDomainName:YES];

**4、双向认证**：

iOS和Android一样，客户端证书库类型可以是PKCS12类型的pfx证书，此类证书包含私钥，公钥和证书，并且由密码

**需要准备的文件**：服务端证书库，服务端证书信任库 ， 服务端导出的证书，客户端证书库，客户端证书

**注[1]：**服务端证书库可以和服务端信任证书库使用同一个证书库，唯一要做的是把客户端证书导入进行。

**注[2]：**客户端证书一般使用跨平台的PKCS12证书库(pfx或p12)，必须记住证书库密钥，此类证书库同时包含私钥，公钥和证书

A、信任服务器： 和单向认证一样

B、信任客户端：在NSURLSessionManager的setSessionDidReceiveAuthenticationChallengeBlock:进行处理

NSURLSessionAuthChallengeUseCredential等类

组件化

## 1、简单介绍一下异常容错组件

**解决问题**：

线上有crash、造成用户服务体验空白。

OC Exception Handing 机制有缺陷

**技术方案**：

**使用Error Object机制替代 Exception Handling 机制**

**Method Swizzle无感知地解决掉部分Crash**

要进行预见性处理而不是进行等crash上报crashlog然后再热修复

**统计的Crash类型**：

80%的Crash都是**JSON解析异常、数组边界溢出、参数类型非法、监听移除异常、消息转发异常、启动异常。**

**技术方案：**

**JSON解析安全**通过扩展JSON过滤方法，解决JSON格式异常引起的Crash，转化为空字符串、数组、字典，然后上报。

**容器类安全：**传入nil crash，大于边界会crash，通过Swizzle容器类方法、增加参数校验、边界检查，防止容器类参数非法、边界溢出的Crash。

**强制转化安全**通过扩展NSObject向相关其他类型类型相互转化方法，解决消息错误转发问题，比如NSNumber和NSString、Int等之间的转化，解决常用的消息错误转发问题，tolerantBool等方法

**消息转发安全：**通过对OC 消息转发流程进行Swizzle，swizzle doesNotRecognizeSelector方法。解决未识别的消息转发（Selector Not Recognized)）。Swizzle的doesNotRecognizeSelector方法，本来想forwordInvotion获取target，但是有jspatch，有些冲突

**通知和KVO自移除：**通过对通知和KVO自移除，解决忘记移除的情况，Notifaction使用switch dealloc来进行移除通知。实现addTolerantKVObserver等于handler紧凑的方法，并保存observer，当dealloc的时候移除observer。

**安全启动模式：**解决jspatch类似的补丁文件引起的Crash，则启动即Crash的问题无法再次更新有效补丁，用户会一直Crash，此类问题比较严重。 在介入JSPatch测试时候出现过一次，所以增加了安全启动模式。

通过对启动后特定时间内Crash计数，重置非法数据和补丁，解决最严重的启动Crash问题，这样的Crash最严重的一种情况是启动立刻Crash，无法进行下载补丁，如果在补丁下载和执行时出现问题就会出现此类严重的Crash。

1、维护一个计数变量，用于表示连续闪退的次数

**2、在启动 application:didFinishLaunchingWithOptions: 后使计数加一**

3、接着使用 dispatch\_after 方法在 5s 后清零计数，如果 App 活不过 5 秒计数就不会被清零

4、如果发现计数变量 > n，表明 App 连续 n 次连续闪退，启动保护流程，重置计数。

5、当保护流程完成后，进入 App 正常启动流程

解决的其他问题：

ARC编译问题、Class Cluster问题、功能覆盖率问题、性能损耗问题（启动时间0.18）

目前正在做一版新的 **Swift Optiona**l处理

**是否开启swizzle**，在各load方法通过config中的变量进行控制（目前是通过重启，希望通过直接恢复Swizzle）

1.如果出现KVO重复添加观察者或重复移除观察者（KVO注册观察者与移除观察者不匹配）的情况，delegate可以直接阻止这些非正常的操作。

**2.被观察对象dealloc之前**，可以通过delegate自动将与自己有关的KVO关系都注销掉，避免了KVO的被观察者dealloc时仍然注册着KVO导致的crash。

**非主线程刷UI类型crash防护（UI not on Main Thread）**

在非主线程刷UI将会导致app运行crash，有必要对其进行处理。

目前初步的处理方案是swizzle UIView类的以下三个方法：

- (void)setNeedsLayout;

- (void)setNeedsDisplay;

- (void)setNeedsDisplayInRect:(CGRect)rect;

在这三个方法调用的时候判断一下当前的线程，如果不是主线程的话，直接利用

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

//调用原本方法

});

**未来计划: 主要以Safely来进行扩展方法**

**KVO在Observer dealloc的时候如何移除？ 一个observer的强引用表，一个被observe的弱引用表**

## 2、简单介绍一下状态机组件，这是一个通用的有限状态机

**一是限制一个状态可以切换到其他哪些状态，避免状态切换混乱，**

**二是统一的事件消息入口，事件引起状态变化**

**两种condition控制，一种是state route的condition，一种是Event携带数据的condition.**

**1、添加route：**

machine.addRoute(.state0 => .state1)

**2、添加routeHandler，routerHandler可以和route一块添加，也可以在添加route后进行添加routeHandler**

**一起添加**：machine.addRoute(.any => .state2) { context in print("Any => 2, msg=\(context.userInfo)") }

**再此添加**：machine.addHandler(.state0 => .state1) { context in

print("0 => 1")

}

**Context:** `(event: E?, fromState: S, toState: S, userInfo: Any?)`

**3、添加事件和条件处理 event，提供enum事件、状态的associateValue处理machine.addRouteMapping**

machine.addRouteMapping { event, fromState, userInfo -> StrState? in

// no route for no-event

guard let event = event else { return nil }

switch (event, fromState) {

case (.str("gogogo"), .str("initial")):

return .str("Phase 1")

case (.str("gogogo"), .str("Phase 1")):

return .str("Phase 2")

case (.str("finish"), .str("Phase 2")):

return .str("end")

default:

return nil

}

}

}

**4、其他的ErrorHandler：**

**machine.addErrorHandler** { event, fromState, toState, userInfo in

print("[ERROR] \(fromState) => \(toState)")

}

**5、事件触发操作**

machine <-! .event0

## 3、简单介绍一下Runtime Extension组件

**非正式协议扩展：**

NSObject(**Aspect**)：面试切面

NSObject(**Association**)： 动态绑定，提供了static函数，用来加载时能够在（+load）前调用

NSObject(**Inspect**)： 反射

NSObject(**Dynamic**)：动态创建

NSObject(**Swizzle**)：isa-swizzle

NSObject(**Observer**): 观察者（有联系）将ObserverValueForKey里的逻辑进行了封装成block进行保存。

NSObject(**Pubsub**): 消息总线（无联系）, 基于EventBus的扩展

这些是Category，当然也都提供了static函数，用来加载时能够在（+load）前调用

**Swift版本：**

**使用了Proxy实现了命名空间、加入了泛型支持。**

为类添加 Ivar：运行时规定,只能在objc\_allocateClassPair与objc\_registerClassPair两个函数之间为类添加变量。

解决子类没有实现父类方法的swizzle：

1、将父类方法添加到子类，class\_addMethod会覆盖父类方法，但不会覆盖自身的方法，如果自身方法有，则添加失败，如果是继承的父类方法，则可以进行添加成功。

2、利用类似KVO的原理，生成一个内部类，将要覆盖的方法添加到内部类上，然后再把原selector的实现设置为\_objc\_msgForward,\_objc\_msgForward是runtime的消息转发环节的入口，这样当使用者调用[a selector]是，就进入了OC runtime的消息转发环节，转发到内部类，内部类进行调用相关前hook、原方法、后hook方法。

**3、Aspects如何使用原函数的参数？hook时候的参数是block，block中的参数进行逐个校验**

**Aspects:** swizzling method 主要有两部分，一个是对对象的 forwardInvocation 进行 swizzling,另一个是对传入的 selector 进行 swizzling.

**解决JSPatch兼容问题：对子类的 forwardInvocation 方法进行交换而不仅仅是替换,**

**解决只能单个remove操作：**

**解决单个对象的hook问题（一个类会产生多个对象，但hook操作的是类）**

**看Aspects: 两种结合的方式1、不覆盖父类 2、在自身实现自身消息转发便于hook方法的管理。**

public protocol Extensible {}

public extension Extensible {

var ext: Ext<Self> {

return Ext(base: self)

}

//TODO: 进行类方法扩展的限制

static var ext: Ext<Self>.Type {

return Ext<Self>.self

}

}

public struct Ext<Base> {

public let base: Base

static var type: Base.Type {

return Base.self

}

}

extension NSObject: Extensible{}

Ext结构体有个base变量，通过限定base，就可以限定Ext上相关方法的调用，进行扩展时候对Ext进行扩展。

## 4、简单介绍下动态Router组件

因为支持动态控制，所以叫动态路由

**服务控制注册**：rewrite：native->native native->web

**跳转服务注册**：可以进行链式操作

**服务匹配控制**：除了注册的服务控制，还提供了代理可以进一步进行处理。

服务访问：

服务插件：

以上的跳转服务注册、服务访问都**支持链式访问**。同时**支持参数校验**

如何实现的：见代码

**一级控制主要是在Router 代理中直接进行控制，它不提供服务的匹配跳转，只提供是否允许相关的逻辑**。

**二级控制是在Router ACL控制中 进行注册的服务控制逻辑，它既包括服务的匹配Redirect逻辑，也提供了是否允许跳转的逻辑**。

提供命名空间和默认方法实现，swift可以进行route调用。

**QueryString \_extraParams,animated,\_excavate（pop）,completion操作**

**RouteCache：提供更快的访问**

校验：requireKeys，

Routable协议，实现此协议即可使用router，参数简单，

**可以通过scheme然后返回相应的routeNode，routeNode也实现routable协议，可配置复杂参数**。

**对于RoueNode的复杂参数配置也提供了链式调用**

**通过一个Router的prepare开头的方法也暴露出RoutableBuidler的链式调用**。

**多Scheme管理**

参数类型校验、界面动画

**面向协议编程**

**特色：链式调用、ACL控制、POP换栈操作、面向协议、Body和Completion操作**

## 5、简单介绍下Module管理组件

解决问题：

Bundle包管理未收敛，多接口分别下载Web业务包、React Native业务包，同时代码分散在各个业务逻辑模块

缺乏Native模块统一管理机制，没有一个良好的协议约束

**缺乏页面跳转解耦、模块通信解耦，以及Native动态化配置跳转的运营活动支持**。

**依赖的组件有**

**Router组件**：主要是将应用页面间跳转逻辑进行解耦

**EventBus**通信组件则是将模块数据通信逻辑进行解耦

**Bundle**管理组件是管理Bundle包更新的统一入口

**Plugin插件管理组件**

**模块管理**：native、Hybrid管理

Native：解析配置注册、代码动态注册卸载（根据配置表或者时机）、section扫描注册。卸载不是真正的卸载（全局配置（有required）、加载配置）

Hybrid：bundle管理组件

**事件通信：**

router跳转服务的注册，通过router来进行

模块内和模块间事件通信：EventBus

模块间数据通信：Injector

**全局事件的统一管理和分发（分发到各module）：提供Super**

**协议约束：**

三个维度：module、Component、Plugin

**对象解耦**（依赖注入框架）:通过EventBus

Module：标识、版本、注册和取消注册的回调

Service、protocol的pair注册。

**Getsectiondata**来扫描\_\_Data 数据段section注册的服务

**1、\_\_attribute\_\_编译属性\_\_section ：**本质也是一种配置表

header： \_dyld\_get\_image\_header

**getsectiondata(header, "\_\_DATA", sectionName, &size)。**

**2、plist**

**3、动态代码ModuleManager.shared.loadModule();**  getInstance，

提供了UIWindow的扩展，使得window.rootViewController是module的rootViewCotroller。

**RCT\_EXPORT\_MODULE**则是定义了两个方法，moduleName，**定义到了全局变量**

#define RCT\_EXPORT\_MODULE(js\_name) \

RCT\_EXTERN void RCTRegisterModule(Class); \

+ (NSString \*)**moduleName** { return @#js\_name; } \

+ (void)**load** { RCTRegisterModule(self); }

最佳实践：

**Getsectiondata 必需模块**

Plist：延迟加载模块、可配置模块（可分为必需和非必需）

动态代码：非必须模块的延迟加载

**Getsectiondata与RCT\_EXPORT\_MODULE的比较：**

1、不依赖load，因为load有可能被使用了

2、可以写在模块的外部，便于代码的组织

## 6、简单介绍下Eventbus组件

支持常规的事件通信（post， on）：

public class func **post**(\_ name: String, sender: AnyObject? = nil, userInfo: [AnyHashable: Any]? = nil)

public class func **onSticky**(\_ observer: AnyObject, name: String, sender: AnyObject?, queue: DispatchQueue, handler: @escaping EventHandler)

也支持事件监听FRP编程：**publishSubject**

**支持Sticky Event传递**： dictionary，与Dictionary的不同

支持指定线程监听操作on(QUQUE), onMain(), onBackgroun()

反模式通信和安全移除。

也支持通信**web、ReactNative**，web实际上是封装了WebViewDelegate，类似WebViewBridge提供两个bridge，然后进行相互调用。所以提供了非单例的初始化方法。便于缩小事件影响范围和提高性能。

**ReactNattive是在RN容器Controller中提供一个Proxy**，尽量暴露少的方法，统一由Proxy来进行分发。初始化不同的RN容器，在Proxy上注册和注销不同的事件。

其他技术方面也使用了一些小的技术trick，如Swift NSObjectProtocol功能扩展，使用Swift Enum Association来进行传值。

**Stickty事件进行过滤后进行observable创建**

**PublishSubject事件直接进行过滤**

**特点：FRP支持，Sticky Event支持、对象传递和反模式支持（计划拆除）**

**RegisterPattern()**

**计划：对象传递和反模式支持的拆除，实际上是数据和函数的injector**

## 8、简单介绍下Bundle管理组件

它的工作流程如图所示，当一个应用启动后，会首先读取多个Bundle包的配置，同时进行上传配置检查是否需要更新，当需要更新时则进行Bundle包的下载更新，下载后会进行检查安全性校验和完整性校验，进行补丁Patch操作，然后会进行更新Bundle配置和校验信息（加载时候也会进行校验）。

加载本地Bundle包时也进行完整性检查

功能模块主要包括 多种Bundle配置下载、解压操作和文件管理，补丁Patch操作，完整性校验和安全校验功能模块。

**安全性校验和完整性校验 主要通过以下几个方面**，

1、下载使用https，减少一部分的中间人攻击，

2、文件的完整性校验使用解密后的MD5进行校验，MD5使用RSA进行加解密，**解密的公钥密钥**是使用异或混淆后的十六进制串。**第一次请求（文件地址和MD5）**

3、更新完成后的配置进行存储。

**流程：**

存在本地Bundle则加载本地、获取版本配置、检查更新、下载bundle、完整性校验、patch操作、重新加载

**RN jsbundle包**: 基于文本内容的patch（下载zip包（含jsbundle和图片资源）），相对路径进行管理图片资源

**离线Web压缩包**：给予文件的patch方式，相对路径管理图片

## 9、简单介绍下混淆组件

iOS主要的混淆有：异或、16进制、随机字符串替换、llvm

两个功能：

1、异或混淆

2、随机字符串替换：在xcode编译前进行执行shell，并将执行后对应关系存入数据库中备份，维护太繁琐，重要函数可以使用其他方式隐藏符号，则不再使用

**Llvm-Obfuscator混淆原理**：OLLVM的混淆操作就是在中间表示IR层，通过编写Pass来混淆IR，然后后端依据IR来生成的目标代码也就被混淆了，三种pass

Instructions Substitution(指令变换)

Bogus Control Flow(流程伪造)

Control Flow Flattening(流程平坦化)

**并没有字符串的替换**

## 11、简单介绍下ReactNative容器组件？

而React Native容器则是为ReactNative 逻辑模块 在Native App提供内敛的入口，使得除了调用通用基础UI及API外，在调用iOS 特有的基础UI及API和安卓基础UI及API时无差异，最大化减少FE对Native 开发的知识依赖，同时Android与iOS对React Native Proxy支持保持一致性。主要关注通信，而不是使用Native UI上

**1、提供ReactNative业务容器功能**

**2、建立通信Proxy，只对RN暴露此Proxy，****减少方法暴露。减少编译速度，在容器外进行注册**

RCTBridge类实现了一个分类：RCTBridge(RCTUIManager)。为了更好的对原生事件做桥接，为RCTBridge类实现了一个分类RCTBridge (RCTEventDispatcher)。

两个对象：

**容器Controller：载入RCTRootView，有个Bridge对象；容器并持有一个Proxy。**

**消息Proxy：使用服务注册模式，注册identifier： 处理逻辑的服务键值对。**

Native业务根据启用的RN业务模块初始化Proxy并注册该模块的服务。字典传值，根据Identifier来进行解析键值。进而进行方法上的调用。暴露消息Proxy接收消息的方法，如callNative，在callNative中进行Proxy的消息分发，该Proxy可以进行，用来在仅Proxy中进行发送消息。减少方法暴露。减少编译速度。

**RN->OC 模块名.方法名()调用方法**，传入identifier和参数字典

**OC->RN Proxy callJSFunction()**  **[bridge.eventDispatch sendAppEventWithName: body:]进行传值**

## 12、简单介绍Web容器组件是如何设计的？（文件级的patch）

这个组件针对的是UIWebView的优化。

Web容器组件功能模块主要包括**多代理插件支持、策略模式的视图控制、动静态资源分离缓存、Hybrid通信标准化**。

其中代理插件通过插件模式提供了**多代理插件支持**（weView网络状态插件、业务插件、WebView插件）

多业务下提供了不同策略模式的Native视图控制，如是否显示分享和收藏按钮，加载中和加载失败视图配置。

配合bundle管理组件、**NSURLProtocol网络拦截（缓存js, css, 图片），提供了动静态资源分离缓存，减少了部分静态资源的网络请求，提升了性能**。**使用版本号解决NSURLProtocol的升级问题**

**同时提供了标准化的与Web进行通信的Hybrid通信机制**，webViewBridge提供和Android一样的调用方法，类似webViewBridge提供Bridge（使用EventBus来进行提供）

OC ->JS: register(\_ identifier:String), { (dic, s)in }); postJSEvent(‘方法名’); 实际上调用js则是evaluateJavascript调用事件分发函数

JS-> OC: callOC(identifier, canshu) 然后进行flush，bridgeflush后进行调用identifier对应的函数

更新机制：**创建版本号文件夹，发版第一次启动则需要进行清空原文件夹**

Bundle组件根据相关bundleId、版本号、APPKey获取更新，详细见bundle管理组件

**第一，**是将整个 Web 资源目录打成一个压缩包（bundle.zip）。

**第二，**基于上一个版本，构建一个最新版本的增量压缩包（update.zip）。为了简化开发，我使用了文件级的 diff 算法——也就是说，对比两个版本的程序目录，将新增和有改动的文件连带目录结构打成压缩包。对于新版本中被删除的文件，本方案忽略，因为 Web 前端程序中多一个文件并不会有任何影响。

**第三，**整个 Web 资源目录以目录的形式存在于发布包中，目录名为 /web，当设备端本地资源不能用时，可以直接使用 Web 目录中的线上资源。另外，为了方便单步调试，调试版 App 也使用线上资源。

**第四，**版本信息文件 update.json，本次发布的版本号、上一个版本号，以及发布时间存在于这个文件中，供 App 定时下载检查。update.json 格式如下：

{"releaseTime":"160530161454","version":9601,"lastVersion":9596}

## 14、简单介绍多媒体播放器组件？

实现边缓冲边播、进度缓存、播放控制。

1、**一种localServer方案**

解码：拿到一个M3U8链接后解析出M3U8索引的具体内容，包括每一个TS的下载链接、时长等

下载：下载解析出的ts文件，

打包：合并成新的本地的m3u8文件，下载的TS数据按照播放顺序打包，供客户端播放

播放：AVPlayer。

2、**AVAssetResourceLoaderDelegate**

需要在视频播放器和服务器之间添加一层类似代理的机制，视频播放器不再直接访问服务器，而是访问代理对象，代理对象去访问服务器获得数据，之后返回给视频播放器，同时代理对象根据一定的策略缓存数据。

**AVURLAsset中的resourceLoader可以实现这个机制，resourceLoader的delegate就是上述的代理对象**。

**代理对象处理流程**

1.当视频播放器向代理请求dataRequest时，判断代理是否已经向服务器发起了请求，如果没有，则发起下载整个视频文件的请求

2.如果代理已经和服务器建立链接，则判断当前的dataRequest请求的offset是否大于当前已经缓存的文件的offset，如果大于则取消当前与服务器的请求，并从offset开始到文件尾向服务器发起请求（此时应该是由于播放器向后拖拽，并且超过了已缓存的数据时才会出现）

3.如果当前的dataRequest请求的offset小于已经缓存的文件的offset，同时大于代理向服务器请求的range的offset，说明有一部分已经缓存的数据可以传给播放器，则将这部分数据返回给播放器（此时应该是由于播放器向前拖拽，请求的数据已经缓存过才会出现）

4.如果当前的dataRequest请求的offset小于代理向服务器请求的range的offset，则取消当前与服务器的请求，并从offset开始到文件尾向服务器发起请求（此时应该是由于播放器向前拖拽，并且超过了已缓存的数据时才会出现）

5.只要代理重新向服务器发起请求，就会导致缓存的数据不连续，则加载结束后不用将缓存的数据放入本地cache

6.如果代理和服务器的链接超时，重试一次，如果还是错误则通知播放器网络错误

7.如果服务器返回其他错误，则代理通知播放器网络错误

Seek的时候dataRequest 有currentOffset

**文件缓存**：将range列表进行archive（媒体文件的配置文件）

## 15、简单介绍视频反作弊组件

**端上**: 主要是为了防止抓取视频地址，**采用了3DES对称加密获取signKey**

采用对称加密的原因是：加解密速度快，适合数据量大的情况，视频列表数据量也比较大

1、rand去请求signKey

2、使用rand+partialkey 来解密（8位）

3、存储keychain，加密的Key //不存储，**存储partialKey**

4、进行加密（requestId是时间戳），两个作用，1个是作为salt的随机部分，2是作为时间维度的判定。Sign=[signkey]+requestID+#+param.base64

5、并不是加密解密传输，仅仅进行拦截判定

**Web前端**：增加reference和token来进行判定来源

## 16、简单介绍下iOS组件化架构方案

主要解决**问题**：业务代码耦合、动态化支持不足、功能组件不健全、多项目组件不复用

首先是**分层设计**：

**业务module**

**业务SDK**：是与业务贴近的UI以及功能，可重用或简单修改个性化定制后进行复用，如聊天UI，语音识别交互UI。

与业务完全无关**的壳工程**：**也分成两种（不是层）, 公共基础组件和基础功能性组件**

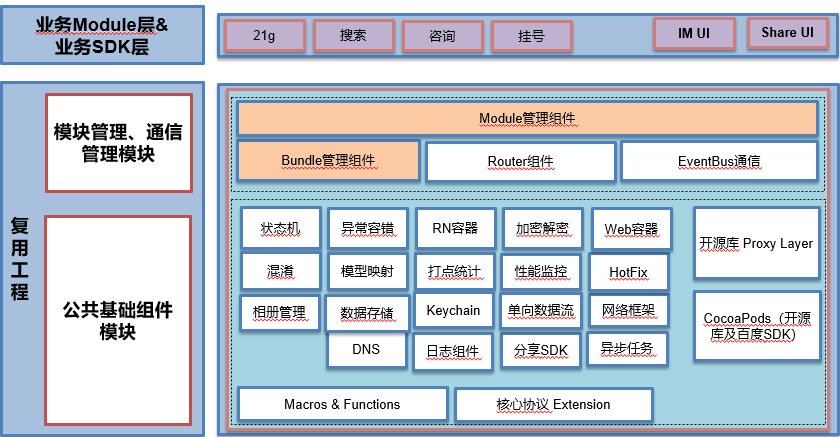
**维度划分:** 划分为了Module、Component、Plugin三个维度，Module是业务模块，Component是功能组件，Plugin是功能插件

**公共基础组件**： 基于语言特性的OC Macro、OC 运行时Extentsion、Swift默认协议实现，一般不需要修改，一般属于utils的定位

**基础功能性组件**：包括第三方的ProxyLayer层和一系列组件，异常容错、状态机、Router、Module管理、EventBus、Bundle管理、混淆、ReactNative容器、安全防护、音视频编辑组件、运行时Extention、网络框架。

**解耦与通信**：Module管理，Router界面隔离、EventBus消息通信。

**动态化方向**：ReactNative、Web容器、远程配置（RouterRewrite）



如何配置业务模块：修改依赖模块，修改入口文件

**进一步工作**：**可以进行多个工程资源文件的管理（进行统一的打包配置文件管理，而不是使用plist）,滴滴oneTool,更容易配置**

## 17、设计一个用户资料缓存系统

**用户身份和个人信息应该分开**。

**安全存储、自动登录、退出数据清理、多用户切换、业务回调**、安全传输、Token用户标识、反作弊、Token登录信息

考虑 版本升级的数据迁移（存储版本号，也可对不同用户存储版本号）、用户信息的安全性（加密、Keychain）、业务模块的交互（登录登出、用户信息）

## 18、如何设计一个多写单读的安全容器类

使用dispatch\_async\_barrier 相当于一个读写锁 dispatch\_barrier\_async、dispatch\_async **网上的一些资料是错误**的，因为后序的dispatch\_async是需要等待dispatch\_barrier\_async执行完的

## 19、如何设计一个图片缓存？

**清理时机**、线程安全、Key值设计、**清理策略**（**LRU**还是根据Size、Count，cost, 异步与同步清理）

**三级缓存**：网络缓存、内存缓存、硬盘缓存

需要主要的问题：异步加载网络、**异步解码和持久化读写**、内存对齐、数据一致性、**内存警告**、数据更新问题、**线程安全**。

**常用淘汰算法**：

1、LFU(Least Frequently Used)

根据数据的历史访问频率来淘汰数据，其核心思想是“如果数据过去被访问多次，那么将来被访问的频率也更高”。每个数据块都有一个引用计数，所有数据块按照引用计数排序，具有相同引用计数的数据块则按照时间排序

2、LRU（LeastRecently User） : 根据数据的历史访问记录来进行淘汰数据，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

一个链表保存缓存数据

新数据插入到链表头部；每当缓存命中（即缓存数据被访问），则将数据移到链表头部；当链表满的时候，将链表尾部的数据丢弃。

**当存在热点数据时**，LRU的效率很好，但偶发性的、周期性的批量操作会导致LRU命中率急剧下降，缓存污染情况比较严重。

**3、LRU-K(LeastRecently Used K)**

相比LRU，LRU-K需要多维护一个队列，用于记录所有缓存数据被访问的历史。只有当数据的访问次数达到K次的时候，才将数据放入缓存。当需要淘汰数据时，LRU-K会淘汰第K次访问时间距当前时间最大的数据（FIFO）

LRU-K具有LRU的优点，同时能够避免LRU的缺点，实际应用中LRU-2是综合各种因素后最优的选择，LRU-3或者更大的K值命中率会高，但适应性差，需要大量的数据访问才能将历史访问记录清除掉

**4、FIFO(First inFirst out)**

FIFO淘汰算法基于的思想是”最近刚访问的，将来访问的可能性比较大”

**5、2Q(Two Queues)**

类似于LRU-2，不同点在于2Q将LRU-2算法中的访问历史队列改为一个FIFO缓存队列，即：2Q算法有两个缓存队列，一个是FIFO队列，一个是LRU队列

新访问的数据插入到FIFO队列；

如果数据在FIFO队列中**一直没有被再次访问**，则最终按照FIFO规则淘汰；

如果数据在FIFO队列中被再次访问，则将数据移到LRU队列头部；

如果数据在LRU队列再次被访问，则将数据移到LRU队列头部；

LRU队列淘汰末尾的数据。

## 20、如何设计一个网络框架？

**API分散个性化配置管理**

**将Stubs作为一级公民**

**插件机制。**

**便捷无侵入的第三方的适配层**

Target -> Endpoint -> Request

endpointClosure、requestClosure、stubClosure。

**更细：**1**、失败重试 2、加解密 3、响应式编程扩展 4、优先级控制 5、线程池控制 6、安全控制**

**目前是基于Moya和RxSwift做的响应式的扩展**

**1、JSON解析，扩展对应Observable响应的map操作，提供对数组类型的扩展以及对Optional的扩展，**Moya仅提供了ObservableType当element是Response的Extension，包括status的过滤和基础类型的

**2、对于Provider的加解密、重试扩展**

**3、model类的泛型解析**

## 21、设计一个统计打点SDK

**打点类型**：页面事件、点击事件

**如何打点**：AOP打点

**打点存储**：持久化过程禁止打断，SQLite或者简单的KV存储

**上传时机**：第一次启动、进入后台

**上传流程**：先上传后删除。文件大小？

## 22、设计一个生产消费队列，考虑线程安全、优先级、失败重试。

class ProducerConsumer {

private var products: [AnyObject]?

private cond: NSCondition?

private lock: NSLock?

init() {

self.products = [];

self.cond = NSCondition();

self.lock = NSLock();

NSThread.detachNewThread(#selector(ProducerConsumer.produce));

NSThread.detachNewThread(#selector(ProducerConsumer.consume));

}

func produce() {

self.lock.lock();

self.products.addObject(NSObject());

if (self.products.count > 0) {

os\_log(‘开始消费产品’);

self.cond.signal();

}

self.lock.unlock();

}

func consume() {

self.lock.lock();

if (self.products.count <= 0) {

os\_log(‘等待产品’);

self.cond.wait();

}

self.lock.unlock();

}

}

NSCondition进行控制，多线程进行操作

使用NSCondition，实现多线程的同步，即，可实现生产者消费者问题。

基本思路是 ，首先要创建公用的NSLock实例。然后：

消费者取得锁，取产品，如果没有，则wait，这时会释放锁，直到有线程唤醒它去消费产品；

生产者制造产品，首先也是要取得锁，然后生产，再发signal，这样可唤醒wait的消费者。

**更复杂的实现：**

产品队列，两个线程池、NSOperationQueue

失败重试考虑重新建立一个队列

Dispatch\_async\_barrier、NSOperation 实现优先级和依赖和服务质量，

## 23、设计一个网络监控模块

如果指的是网络状态的化

1、**一次性监控**

2、**实时监控**，调用相关block，提供state

1、**Ponnydebugger**可以进行相关网络性能的分析。

2、使用**charles**分析模拟慢网

3、可以**Hook URLConnection CFNetwork、NSURLSession等api**

4、因为我们所使用的URLConnection、CFNetwork、NSURLSession底层都是 **BSDSocket**，所以可以尝试在socket上动手脚来实现效果，类似于通过ViewController的生命周期方法来统计页面加载时间的做法，我们Hook socket相关的方法来做，比如通过hook socket连接时的 connect方法，拿到tcp握手的起始时间，通过hook SSLHandshake方法，在SSLHandshake执行的时候拿到 SSL握手的起始时间等。

5、iOS 9 Apple 加入 **ATS** 新特性，并要求开发者使用 HTTPS，我在 iOS9、10上对 HTTPS 网络请求Hook socket方法时候，有一些方法hook 失效，猜想应该是Apple 进行了加固、加密，导致一些系统方法没办法hook

不过apple在 iOS 10 推出一个API，可以在 iOS10 版本以上进行网络信息的收集

6、- (void)**URLSession**:(NSURLSession \*)session task:(NSURLSessionTask \*)task **didFinishCollectingMetrics**:(NSURLSessionTaskMetrics \*)metrics。

## 24、设计一个方案来检测KVO的同步异步问题

1、添加两个属性监听者

2、在一个ObserverValueForKey的处理逻辑中进行阻塞，比如进行**sleep**。

3、观察另一个ObserverValueForKey的执行情况

## 25、设计一个监控线程卡顿的组件

NSRunLoop调用方法主要就是在**kCFRunLoopBeforeSources**和**kCFRunLoopBeforeWaiting**之间,还有**kCFRunLoopAfterWaiting**之后,也就是如果我们发现这两个时间内耗时太长,那么就可以判定出此时主线程卡顿

1、添加Observer在指定runloopmode下（kCFRunLoopCommonModes）

2、在observer回调中传回相关runloop activity状态（CFRunLoopActivity）

3、新起线程进行一直循环检测，如每隔50ms进行检测，检测两个状态从kCFRunLoopBeforeSources、kCFRunLoopAfterWaiting进行的时间，时间长则卡顿

## 26、设计一个进度条？

在CALayer中进行绘制。

设置引起动画的属性

在**needDisplayForKey**中返回属性

在**drawInContext**中进行绘制，或者在其代理中进行绘制**- (void)drawLayer: inContext.**进行绘制

## 27、如何设计一个冗余文件扫描组件？

通过String或者Find命令 将资源文件在工程中遍历扫描即可，无法解决动态命名的资源，需要进行检查。

Linkmap和otool工具对比SEL和class的引用

## 28、如何设计一个缓存组件？

**双向链表进行存储、NSMapTable、I/O提供线程池会更好一些、线程安全、缓存大小容量控制**、**内存警告处理、缓存命中率、性能、缓存清理时机、同步异步清理API**

## 29、谈一下插件管理组件？

**1、扩展点和扩展（插件）**

**2、扩展点可以进行配置扩展的实体类或者接口**

**3、plist配置设置**

**4、正向插件设计、非反向插件设计（有利于模块化测试）**

**让应用程序支持插件扩展的步骤：**

**1. 定义一个接口集(只有纯虚函数的类)，用来与插件交流。**

**2. 用宏Q\_DECLARE\_INTERFACE()将该接口告诉Qt元对象系统。**

**3. 应用程序中用QPluginLoader来装载插件。**

**4. 用宏qobject\_cast()来确定一个插件是否实现了接口。**

**新增<extensions>子元素。 子元素的名称必须与你想要访问的扩展点的名字匹配。**

**根据扩展点的类型：**

**如果扩展点使用interface属性声明，对于新增的子元素设置implementation属性为实现指定接口的类名。**

**如果扩展点使用beanClass属性声明，对于新增的子元素设置所有在指定bean类用@Attribute 注解的属性。**

**懒加载**

**Plugin可以使用namespace区分模块的插件**

## 30、如何组件化保证质量？或者协助测试工作？

1、持续集成、Fastlane

2、模块化测试、自动化测试

3、APM组件监控

## 31、如何设计APM组件？

应用性能问题危害最大，分别为：**网络**连接超时、闪退、**卡顿、卡顿崩溃**、黑白屏、网络劫持、交互性能差、CPU 使用率问题、**内存泄露**、不良接口。

**启动时间: 冷启动、热启动**

**卡顿: runloop、FPS**

## 32、如何进行Hybrid性能监控？

**1、View让如一个html，使用css来进行控制，而不进行html的跳转**

**2、页面进行分块编码(chunked encoding)**

**3、动静态资源分离复用**

设计模式

## 1、面向对象六大原则？

**单一职责原则：**

**开放闭合原则：**一个是对于拓展是开放的，另一个是对于修改是封闭的

**里氏替换原则：**使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象

**依赖倒置原则：**高层模块不应该依赖低层模块，两个都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖于抽象，高层模块就是调用端，低层模块就是具体实现类

**迪米特原则：**也称为最少知识原则。如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易

**接口隔离原则：**一个类对另一个类的依赖应该建立在最小的接口上

## 2、对象创建相关的设计模式有哪些？

**简单工厂**：不是23种模式中的一种，客户client通过工厂类创建具体产品

**1、原型模式（prototype）**使用类原型实例来创建对象，即通过复制原型来创建新的对象。

**2、工厂模式（factory）**:定义创建产品对象的工厂接口，将实际创建工作推迟到子类具体工厂中，一个具体工厂对应一个具体的产品，通过继承来实现。

**3、抽象工厂（abstract factory）**：提供一个创建一系列相关对象的接口，实现这个接口的具体工厂可以产生一个产品族的所有不同类型的对象。

客户端通过创建具体工厂来调用相应的创建产品的方法创建需要的产品。

**4、生成器模式（builder）**：将一个复杂对象的创建过程与它的外部表现分离，使得一个创建过程可以创建不同的外部表现，一般用在需要多步创建或者属性非常多的情况下。 4种角色

**生成器模式与抽象工厂**模式有很多相同的地方，但也有不同的地方：生成器构建复杂对象，需要多个步骤构建对象，在构建过程的最后一步返回产品；抽象工程构建简单的或复杂的对象，以单一步骤构建对象，立刻返回产品。

**5、单例模式（singleton）**：保证一个类仅有一个实例，并提供一个访问它的全局访问点

单例可能引起的问题：生存周期长，可能会导致内存的持续占用，多线程需要消息处理线程互斥问题，进行资源保护，单例在类的继承树中不利于利用，破坏继承体系，单例的全局可见性会带来一定的设计破坏。

## 3、接口适配相关的设计模式有哪些？

**1、适配器模式（adapter）**：将一个类的接口转换成client希望的另外一个接口，适配器模式使得原本不兼容的接口能够一起工作,有两种适配器模式；

**类适配器**：继承被适配的类，并实现client兼容的接口协议。在实现的方法中通过调用super来进行相关功能的引用。

**对象适配器**：包含被适配的类的实例，并实现client兼容的接口协议，在实现的方法中通过引用的被适配的类的实例进行实现。

**场景**：已有的服用类接口与需求不匹配，

**两种适配器的对比：**

类适配器只针对单一的具体的被适配的类（adaptee），易于重载adaptee的行为，因为是通过继承重载来进行实现的,只有一个adapter对象，无需额外的指针间接访问adaptee。

对象适配器可以通过引用adaptee的实例对象适配多个adaptee，难以重载adaptee的行文，需要额外的指针以间接访问adaptee并适配其行为。

委托主要是适配器模式，也是模板模式。

client需要的兼容的接口可以用两种方式来现实，一种是oc的protocol，一种是oc的block回调。

**2、桥接模式(bridge)**: 将抽象部分与它的实现部分分离，使他们都可以独立地变化。抽象和实现部分维持着自己的层次结构，父类抽象部分拥有一个实现部分的父类的一个实现。更像是一个接口适配到不同接口的一种方式。

场景：不想在抽象与实现之间形成固定的绑定关系，抽象及其实现都可以通过子类化独立进行扩展。

**3、外观模式（facade）**：系统的一组接口提供一个统一的接口来隐藏统一接口下的复杂实现，一个复杂的接口方法实现就是一个外观模式。

场景：提供一个较简单的接口来封装复杂的逻辑

## 4、抽象组合相关的设计模式有哪些？

**1、组合模式（composite）：**将对象组合成树形结构以表示”部分与整体”的层次结构，组合使得用户对单个对象和组合对象的操作使用具有一定的一致性，通常与迭代器模式一起使用。

场景：让一个树形结构具有相同的抽象接口，统一处理组合结构中的所有对象。

对于这样的结构新增的操作，可以使用访问者模式来实现，让访问者访问每一个节点进一步的处理

**2、迭代器模式（iterator）**：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。主要有两种迭代器，一种是内部迭代器，另一种是外部迭代器。

外部迭代器：外部迭代器为client提供了更多的控制，client创建并维护外部迭代器，client可以使用不同外部迭代器实现多重类型的遍历。

内部迭代器是通过集合对象的特殊接口，一次访问一个元素，

场景：需要访问组合对象内容，而不暴露内部表示，用来遍历组合对象。

## 5、对象去耦相关的设计模式有哪些？

**1、中介者模式（mediator）：**一个对象封装一系列的对象的交互方式，中介者使client和各对象之间不需要显式地相互引用，从而达到耦合松散

场景：对象引用了太多的对象

**2、观察者模式（observer）**：也叫发布-订阅模式，定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。IOS有两种实现方式

通知和KVO，两者的主要区别是通知是一个中心对象为所有观察者提供变更通知，主要从广义上关注程序事件，KVO是被观察对象直接向观察者发送通知，绑定于特定对象的特定属性的值。

## 6、行为扩展相关的设计模式有哪些？

**1、访问者模式（visitor）**：表示一个作用于某个对象接口中的各元素的操作，可以在不改变各元素类的前提下定于作用于这些元素的心操作。访问者模式有个缺点，就是访问者和目标类耦合在一起，如果访问者要支持新的类，访问者的父类和子类都需要修改。

场景：定义复杂接口的类很少做修改，但经常需要向其添加新的操作。

**2、装饰模式(decorator)**:在不改变原类文件的情况下，动态地扩展一个对象功能，它是通过创建一个包装对象，用来装饰真实的对象,比子类继承更为灵活。

可以通过类别和继承两种方式来实现装饰模式。

场景：在类定义被隐藏或无法子类化的情况下进行扩展行为。

**3、责任链模式（chain of responsibility）**：使多个对象都有机会处理请求，从而避免请求的发送者和接受者之间发生耦合，此模式将这些对象连成一个链，并沿着这个链传递请求，直到有一个对象处理它、

## 7、算法封装相关的设计模式有哪些？

**1、模板方法模式（template）**：定义一个操作算法的骨架，而将一些步骤延迟到子类中，模板方法使子类可以重定义算法的某些步骤而不改变该算法的结构

场景：一次性实现算法的不变不分，将可变不分列给子类或其他对象来访问，控制子类的阔闸门，希望定义一些钩子操作。

**2、策略模式（strategy）**：当出现条件语句讨论时，就可能需要策略模式，定义一系列算法，将他们一个一个封装起来，并且使他们可相互替换。

场景：有很多条件句的并行讨论，可以将相关的分之条件移到策略类中。

**3、命令模式(command)**:将相应的操作请求封装为一个对象，从而可用不同的请求对client记性参数化，对请求排队或支持可撤销操作。

场景：支持队列，撤销和回复，参数化对象，支持事务操作。

## 8、性能与访问对象相关的设计模式有哪些？

**1、享元模式（flyweight）**：用共享对象，尽可能减少内存使用量的模式，运用享元模式有效支持大量细粒度的对象。享元工厂管理一个享元池，并更具不同的类型返回不同的享元对象。

场景：对象的多数特有状态（外在状态）可以放在外部进行轻量化。共有状态通过享元来进行提供。

**2、代理模式（proxy）：**为替他对象提供一种代理以控制对这个对象的访问，例子，实现图片的延迟加载。

典型场景：需要远程代理；需要保护代理；需要虚拟代理（举例，图片延迟加载）

## 9、对象状态相关的设计模式是什么？

**1、备忘录模式（memonto）**：在不破坏封装的前提下，不活一个对象的内部状态，并在该对象之外保存这个状态，主要以后就可将该对象恢复到原先保存的状态。

场景：需要在某个时刻保存一个对象的状态，以便以后可以恢复到先前的状态。

## 10、代理模式和装饰模式的区别

**装饰器模式关注于在一个对象上动态的添加方法，代理模式关注于控制对对象的访问。**

当使用代理模式的时候，我们常常在一个代理类中创建一个对象的实例。当我们使用装饰器模 式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

## 11、工厂模式和策略模式的区别？

工厂模式注重的**是对象的构建过程**，**策略模式注重的是策略算法**, 不同策略算法还可以相互替换。

## 12、模板方法和委托模式的区别？

**委托模式**也是模板方法模式，委托与预先定义好的委托接口一起定义了一个特定算法，**特定算法由实现委托方法的对象提供**。

**模板方法**定义了一个特定算法，**缺少的信息由子类通过继承来提供**。

## 13、装饰模式和策略模式的区别？

装饰模式是从外表变更，每个节点不知道变更，策略模式是从内部变更，每个节点指导一组预定义的变更方式。

装饰模式侧重于**行为扩展**，策略模式侧重于某种场景下**算法封装**。

## 14、装饰模式和适配模式的区别？

功能上，装饰模式是**扩展行为**，适配器模式是**适配接口**。

## 15、三种工厂的对比

**简单工厂**的核心是一个具体工厂类，根据不同的类型

**工厂方法**类的核心是一个抽象工厂类，不同具体工厂创建不同的产品，

**抽象工厂**以**类簇**的形式出现，初始化不同的具体产品,

**抽象工厂与工厂方法的区别**：**抽象工厂**通过对象组合创建抽象产品，一个工厂可以创建多个产品，必须修改父类的接口才能支持新的产品了； **工厂方法**通过类继承创建抽象产品，一个工厂只能创建一个产品，需要子类化工厂和产品来创建新产品。

## 11、自己写过的开源组件中用到了哪些设计模式

观察者：核心协议Extension

单例：这个不用说

Proxy：命名空间功能

**Builder：网络框架**

**Visitor：插件**

Delegate：这个不用说

享元：UITableView

**策略：Web容器的视图控制**

**桥接：Module约束**

## 12、如何正确地写OC单例模式？

**要重写allocWithZone和CopyWithZone**，或者干脆使用NS\_UNAVAILABLE 把这两个方法禁用.

copyWithZone：{return self}

(id)**retain** { return self; }

(NSUInteger)**retainCount** { return 1;}

(id)**autorelease** { return self;}

扩展性面试题

## 1、Scrum的角色、工件和活动？

三个角色：Product owner， Scrum Master， Dev team。

三个工件：Product Backlog， Sprint Backlog，Tracking/Increment

四个活动：Sprint Planning，Daily Scrum， Sprint Review， Sprint Retrospective

## 2、你在项目中遇到的最大的问题是什么？怎么解决的？

医疗事业部，75%是新员工。业务复杂，耦合严重，总监层对质量要求非常高。

迅速向上反馈提出解决的两个方向，组件化、动态化，并协调资源进行相关的开发。

## 3、你是如何学习新知识的？最近在看什么书？

1、了解大概，针对性学习programming guide

2、针对性学习后的专项总结以及Coding

3、思考，面对问题该怎么解决?怎么设计和改良？

Using Swift with Cocoa and Objective-C (Swift 4) 一些导入规则更明白了

## 4、你为什么跳槽呢？

业务调整技术方向被调整，无法发挥自身技术影响力，本身秒懂是一个很成功的内部孵化产品，但却因为调整导致运营、产品、APP开发等RD的大量离开。

## 5、跳槽挺频繁的，能说明一下原因吗？

**昆仑万维**：上市前夕，海外事业部负责人，昆仑万维副VP谢强将部门独立了，独立了大约4个月，将公司解散，南下广州接着去做他的老本行游戏分发去了，这个是完全不可控的原因。

**360：**因为主要做iOS，360主要做工具类，在iOS上可做的东西实在太少，WiFi也成熟了

**蘑菇街：**当时蘑菇街合并美丽说，杭州过来的中层对美丽说和淘世界的海淘业务进行调整，很多人不太赞同一些调整而选择离开了，现在蘑菇街北研已经没有了，技术团队等都解散了。其实当时另一个原理是，女朋友在海淀这边，长期分居，自己也到了结婚的年纪，所以选择了百度这边的职位，目前和女朋友已经结婚，也算是一个不后悔的选择吧。

表明态度：**最近因业务调整的确受影响太多了，其实自己是一个特别踏实地去做事情的人，因为受业务调整影响，正因为自己经历了这么多调整和受这么多影响，自己也特别珍惜自己在一个公司的稳定性**，比如在蘑菇街海淘团队，连续1个半月都没有休息去做大改版，在秒懂百科也是连续自愿加班半年，我是一直一心要把当时所正在做的工作做好的。

## 6、自我介绍一下？

我叫李生，来自山东，毕业于\*\*，大学期间自学android开发，毕业后一直从事于移动开发工作，后来转向iOS开发，近7年的移动开发经验，5年多的管理经验，负责过语驾、360免费WiFi、淘世界App、百度医生、百度百科、秒懂百科等APP，也负责了敏捷开发流程管理工作，目前正在求职，非常期望能加入\*\*公司

## **7、参与过的最能体现你技术水平的项目是什么？主要用了哪些技术？有哪些不足？**

淘世界、百度医生、秒懂百科这三个吧，主要是电商、医疗产品的复杂性上有了更多技术架构上的思考。

不足之处是：有些预期的模块还有待完善。

## 8、如果给你一次重写这个程序的机会，你会怎么写？

**360免费WiKi，当时<SystemConfiguration/CaptiveNetwork.h>**

dlopen("System/Library/SystemConfiguration/IPConfiguration.bundle/IPConfiguration", RTLD\_LAZY);

现在获取WiFi使用NetworkExtension框架

1、向 Apple 申请开发 Network Extension 权限

2、申请包含 Network Extension 的描述文件

3、配置 Info.plist

4、配置 entitlements

5、iOS 获取 Wifi 列表代码实现

6、获取Wifi列表回调

**#import <NetworkExtension/NetworkExtension.h> 可以获取**

NEHotspotNetwork有如下相关信息

SSID：Wifi 名称

BSSID：站点的 MAC 地址

signalStrength： Wifi信号强度，该值在0.0-1.0之间

secure：网络是否安全 (不需要密码的 Wifi，该值为 false)

autoJoined： 设备是否自动连接该 Wifi，目前测试自动连接以前连过的 Wifi 的也为 false 。

justJoined：网络是否刚刚加入

chosenHelper：HotspotHelper是否为网络的所选助手。

## 9、iOS上你最喜欢什么API？为什么？

GCD，尤其是Swift重写后的GCDAPI，简洁，易用， 功能强大。

DispatchQueue.global().async

DispatchQueue.main.async

DispatchGroup：group.enter()，group.leave()

## 10、你怎么解决团队中的问题？比如某个团队组员能力不行？

1、**定位问题**，是能力不行还是问题难度评估问题？

2、**鼓励组员**，同理心，如果有些问题不容易解决，或者操守挫折，是需要进行鼓励的

3、**解决问题**，是否需要进行指导，需要则进行指导或者协调资源

## 11、什么是持续集成？什么是持续交付？

持续：不断地获取反馈，响应反馈。

持续集成：在功能模块开发、分支开发后的编译、测试、打包工作。

持续交付：更多地强调开发人员有能力进行频繁地部署，业务有能力可以随时发布。

## 12、你觉得有什么缺点？还有什么缺点？

有时候**不太确定的东西不太爱参与讨论**，所以一般要求要讨论时候能够提前通知。

有时候**目标定的太大**，不太切合实际，如读书读多少本。

## 13、如何做技术规划？

不同时期Leader也发挥不同的职能**，初期侧重从技术和项目实践方面打通设想和打造产品**，迭代和试错，

**随着业务发展，能从更合适的技术、构建、架构、业务模型等方面展开专项的工作**。

**把控团队技术发展和提升方向，服务于业务**

## 14、如何管理团队？比如有初级工程师、高级工程师、资深工程师？

**新人**可以老带新，但老带新过程一定要进行团队或者公司级别的技术积累，形成文档等，最终形成一套新人培养流程。

**明确职责**：团队应该明确关键人物的角色，公开规定好一个角色由谁来担任，职责和指标是什么，甚至可以约定任期是多久，因为角色是活的

**资深工程师**：培养小组的技术负责人，当然也有想做技术的，可从纵向、横向来进行扩展能力

## 15、多个功能类似的App如何进行灰度？

**CocoaPods虚拟target来进行引入不同文件**

**A/B Test**

**工程变量**

## 16、如何人员梯度培养？

**1、人才盘点**

**2、构建胜任力模型**

**3、人才职业生涯规划**

**4、进行人才培养、晋升体系设计，实施体系方案**

## 17、如何进行向上沟通？

**1、做出重大决定前和领导沟通**

工作进入一个决策的节点时，需要和领导沟通，提出你的决策意见、向领导询问其决策意见，双方尽快达成一致。毕竟有领导的支持，工作开展会顺利得多。另外一方面也是避免后期领导意见和你不一致，导致工作需要重新再做一次，工作效率大大降低。

**2、工作遇到风险时，及时向领导汇报**

工作中遇到风险，如：工作目标无法达成、工作会延期或者有突发情况，及时向领导汇报，注意是“及时”！比如在你处理完突发情况时，不要以为这就结束了，你的下一步应该是立马向领导汇报当前的情况是怎么样、目前是怎么处理的，让领导了解当前的情况，知道你在处理，可以让其放下心来，树立你靠谱的形象。永远不要给领导意外的“惊喜”，你只会被贴上“不靠谱”的标签。

**3、完成事情后向领导汇报情况**

不要简单地认为工作做完就可以了，在工作有阶段性结果时，主动向领导汇报一下当前的结果，讨论项目进展情况，下一步方向，让领导知道你的工作进度。不要被动地等领导问了再说，因为领导也很忙，等他问的时候可能领导已经觉得工作是不是要失控了。所以主动向领导汇报工作，不要让领导觉得你的工作有些失控，进而对你的工作能力产生怀疑。

## 18、最近读过哪些书有什么收获？

**1p理论：第三方付费**

## 19、你做了哪些优化？

**启动的优化**

**TableView优化（runloop暂时忘记）**

## 20、你还有什么问题吗？

**如果有幸加入公司，负责哪些工作呢？**

**目前团队情况是怎么样的呢？**

## 21、组件化成功的标准？

**业务快速迭代、解决业务问题、解耦通信、模块化测试**

## 22、什么才算一个好的架构？

架构不是设计出来的，而是解决业务问题，并充分考虑业务快速扩展的情况下进行的设计。

**23、怎么进行code review？**

**Peer review、组织会议进行评审好代码和坏代码**

**知乎：算法，安全，图像处理**

**http、TCP、用户密码保存、分页分段机制**