目录

数据结构

单链表

- 只有一个指向下一个节点的指针。适用于节点的增加删除。
- 单向链表增加删除节点简单。遍历时候不会死循环；
- 只能从头到尾遍历。只能找到后继，无法找到前驱，也就是只能前进。

双链表

- 有两个指针，一个指向前一个节点，一个后一个节点。适用于需要双向查找节点值的情况。
- 可以找到前驱和后继，可进可退；
- 增加删除节点复杂，需要多分配一个指针存储空间。

栈

- 栈是一种特殊的线性表，它只能在一个表的一个固定端进行数据结点的插入和删除操作。

队列

- 队列和栈类似，也是一种特殊的线性表。和栈不同的是，队列只允许在表的一端进行插入操作，而在另一端进行删除操作。

数组

- 数组是一种聚合数据类型，它是将具有相同类型的若干变量有序地组织在一起的集合。

链表

- 链表是一种数据元素按照链式存储结构进行存储的数据结构，这种存储结构具有在物理上存在非连续的特点。

树

- 树是典型的非线性结构，它是包括，2 个结点的有穷集合 。

图

- 图是另一种非线性数据结构。在图结构中，数据结点一般称为顶点，而边是顶点的有序偶对。

堆

- 堆是一种特殊的树形数据结构，一般讨论的堆都是二叉堆。

散列表

- 散列表源自于散列函数(Hash function)，其思想是如果在结构中存在关键字和T相等的记录，那么必定在F(T)的存储位置可以找到该记录，这样就可以不用进行比较操作而直接取得所查记录。

计算机基础

设计模式

- 是一套被反复使用、代码设计经验的总结
- 使用设计模式的好处是：可重用代码、让代码更容易被他人理解、保证代码可靠性
- 一般与编程语言无关，是一套比较成熟的编程思想

设计模式可以分为三大类

- 创建型模式: 对象实例化的模式，用于解耦对象的实例化过程
  - 单例模式、工厂方法模式，等等
- 结构型模式：把类或对象结合在一起形成一个更大的结构
  - 代理模式、适配器模式、组合模式、装饰模式，等等
- 行为型模式：类或对象之间如何交互，及划分责任和算法
  - 观察者模式、命令模式、责任链模式，等等

常用的设计模式

- MVP
  - 优点: view与model的解耦，交互通过presenter交互
  - 缺点: 业务增多是view与presenter接口也会越来越多
- MVC
  - 优点：View、Model可以重复利用，可以独立使用
  - 缺点：Controller的代码过于臃肿
- MVVM
  - 优点: view与数据是双向绑定，当model变化会自动同步到view
  - 缺点: 错误不好定位，双向绑定不利于view的重用

网络协议

OSI分层

- 应用层（application）：最接近终端用户的OSI层，这就意味着OSI应用层与用户之间是通过应用软件直接相互作用的。网络进程访问应用层；提供接口服务
- 表示层（presention）：数据表现形式；特定功能的实现-比如加密模式确保原始设备上加密的数据可以在目标设备上正确地解密
- 会话层（session）：主机间通信；对应用会话管理，同步
- 传输层（transport）:实现端到端传输；分可靠与不可靠传输；在传输前实现错误检测与流量控制，定义端口号（标记相应的服务）
- 网络层（network）（单位类型：报文）：数据传输；提供逻辑地址，选择路由数据包，负责在源和终点之间建立连接
- 数据链路层（date link）（单位类型：帧）：访问介质；数据在该层封装成帧；用MAC地址作为访问媒介；具有错误检测与修正功能。
- 物理层（physical）（单位类型：比特）：实现比特流的透明传输，物理接口，具有电气特性

TCP/IP分层

- 应用层：TCP/IP协议的应用层相当于OSI模型的会话层、表示层和应用层，FTP(文件传输协议)，DNS（域名系统），HTTP协议，Telnet（网络远程访问协议）
- 传输层：提供TCP(传输控制协议)，UDP（用户数据报协议）两个协议，主要功能是数据格式化、数据确认和丢失重传等
  - TCP
    - TCP提供IP环境下的数据可靠传输，它提供的服务包括数据流传送、可靠性、有效流控、全双工操作和多路复用。通过面向连接、端到端和可靠的数据包发送。通俗说，它是事先为所发送的数据开辟出连接好的通道，然后再进行数据发送
  - UDP
    - UDP则不为IP提供可靠性、流控或差错恢复功能。一般来说，TCP对应的是可靠性要求高的应用，而UDP对应的则是可靠性要求低、传输经济的应用。
- 网络层：该层负责相同或不同网络中计算机之间的通信主要处理数据包和路由。数据包是网络传输的最小数据单位。通过某条传输路线将数据包传给对方。
  - IP协议
    - 在IP层中，ARP协议用于将IP地址转换成物理地址
  - ICMP协议
    - ICMP协议用于报告差错和传送控制信息
  - IGMP协议
- 接口层：TCP/IP协议的最低一层，对实际的网络媒体的管理，包括操作系统中的设备驱动程序和计算机对应的网络接口卡

编程思想

函数式编程

- 函数式编程指的是数学意义上的函数,即映射关系（如：y = f(x),就是 y 和 x 的对应关系,可以理解为"像函数一样的编程").它的主要思想是把运算过程尽量写成一系列嵌套的函数调用

面向对象

- 面向对象的编程思想, 我们将要解决的一个个问题, 抽象成一个个类, 通过给类定义属性和方法, 让类帮助我们解决需要处理的问题.(即命令式编程, 给对象下一个个命令).

Objective-C

基础知识

- 一个NSObject对象占用多少内存?
  - 系统分配了16个字节给NSObject对象（通过malloc_size函数获得）
  - 但NSObject对象内部只使用了8个字节的空间（64bit环境下，可以通过class_getInstanceSize函数获得）
- 对象的isa指针指向哪里?
  - instance对象的isa指向class对象
  - class对象的isa指向meta-class对象
  - meta-class对象的isa指向基类的meta-class对象
- OC的类信息存放在哪里?
  - 对象方法、属性、成员变量、协议信息，存放在class对象中
  - 类方法，存放在meta-class对象中
  - 成员变量的具体值，存放在instance对象
- isa、superclass总结
  - instance的isa指向class
  - class的isa指向meta-class
  - meta-class的isa指向基类的meta-class
  - class的superclass指向父类的class
    - 如果没有父类，superclass指针为nil
  - meta-class的superclass指向父类的meta-class
    - 基类的meta-class的superclass指向基类的class
  - instance调用对象方法的轨迹
    - isa找到class，方法不存在，就通过superclass找父类
  - class调用类方法的轨迹
    - isa找meta-class，方法不存在，就通过superclass找父类

事件响应链

UIApplication 会触发 func sendEvent(_ event: UIEvent) 将一个封装好的 UIEvent 传给 UIWindow，也就是当前展示的 UIWindow，通常情况接下来会传给当前展示的 UIViewController，接下来传给 UIViewController 的根视图。这个过程是一条龙服务，没有分叉。但是在传递给当前 UIViewController 的根视图，然后在传递给controller的view,检测是否可接受事件，检测坐标是否在自己内部，遍历子视图，重复上面步骤，找到合适的控件进行响应事件。

Runtime

OC是一门动态性语言，允许很多操作在运行时进行操作

它的动态性是靠runtime进行实现，runtime是一套C语言的API,封装了很多动态性相关的函数

平时编写的OC代码，底层都是转成runtime api调用

具体应用

- 利用关联对象给分类添加属性
- 遍历类的所有成员变量（修改textfiled占位文字颜色、字典转模型、自动归档解档等）
- 交换方法（替换系统方法实现）
- 利用消息转发机制解决方法找不到的异常问题

objc_class和objc_object

```
struct objc_object {
private:
    isa_t isa;
public:
    // function here
}

struct objc_class : objc_object {
    Class superclass;
    cache_t cache;             // formerly cache pointer and vtable
    class_data_bits_t bits;
      // method here
}
```

- **objc_object** 定义在 `objc-private.h` 里面，该结构体只包含一个叫做 isa 的isa_t类型的变量
- **objc_class** 定义在 `objc-runtime-new.h` 里面，它继承自objc_object，所以除了isa成员变量之外，它还有：
    - 指向另一个objc_class结构体的指针 **super_class**；这一个objc_class包含了父类的信息
    - 一个包含函数缓存的cache_t类型变量 **cache**
    - 一个包含类的方法，属性等信息的class_data_bits_t类型的变量 **bits**
- 编译结束之后，OC的每个类都是以objc_class的结构体形式存在于内存之中，而且在内存中的位置已经固定。运行期间，创建新的对象的时候，也是创建objc_object的结构体
- 总结：OC的对象和类在内存中都是以结构体的形式存在的，类对应objc_class结构体，对象对应objc_object结构体。

  在创建这两种结构体的过程中都会初始化isa变量。isa变量存放着对象所属的类的信息，或者类所属的元类的信息。

  另外，在创建objc_class的结构体的时候还会初始化相应的cache和bits变量

class_data_bits_t

```
struct class_data_bits_t {
    uintptr_t bits;     // 包含具体信息看下面表格
    // method here
}
```

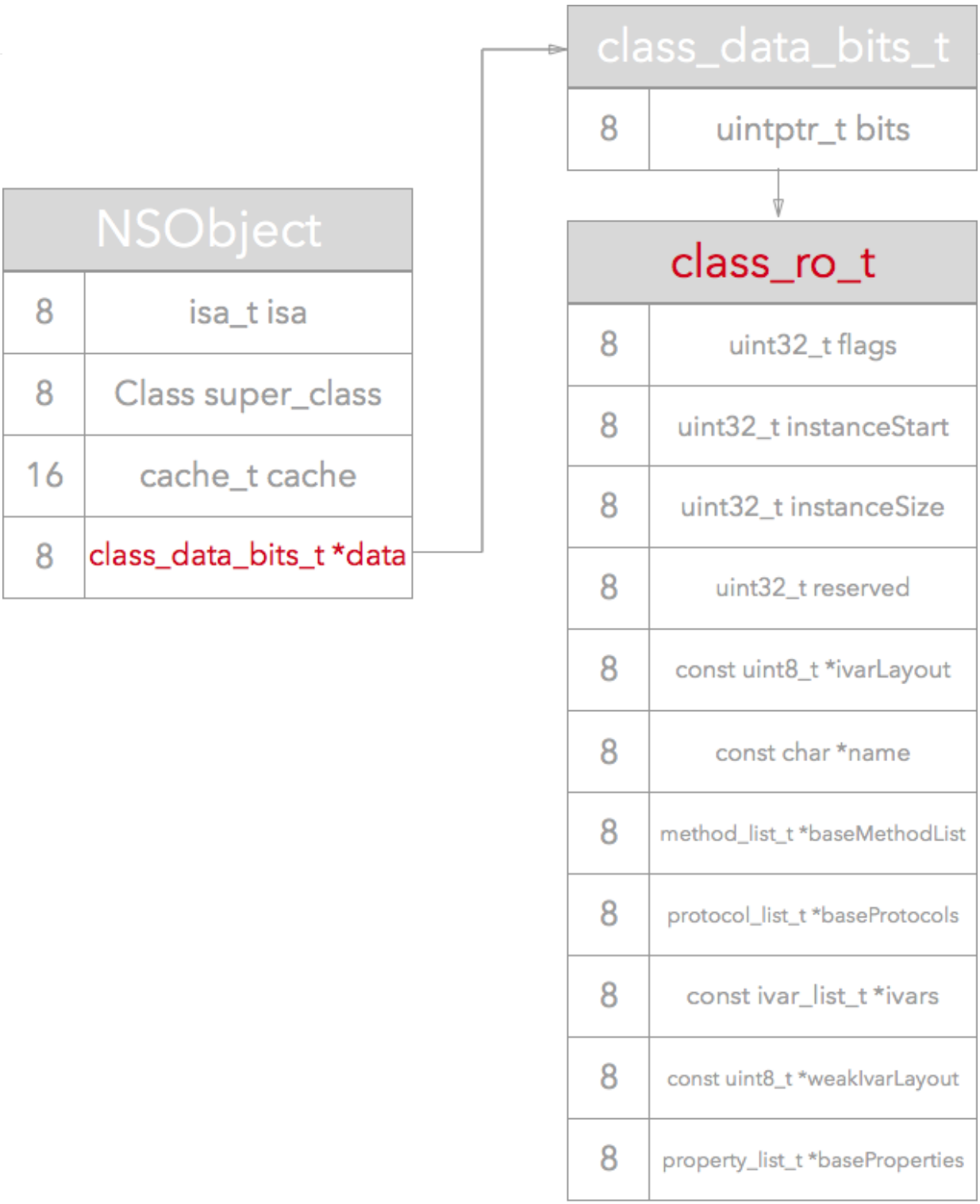
名称	介绍
is_swift	第一个bit，判断类是否是Swift类
has_default_rr	第二个bit，判断当前类或者父类含有默认的 `retain/release/autorelease/retainCount/_tryRetain/_isDeallocating/retainWeakReference/a` 方法
require_raw_isa	第三个bit， 判断当前类的实例是否需要raw_isa
data	第4-48位，存放一个指向class_rw_t结构体的指针，该结构体包含了该类的属性，方法，协议

class_rw_t 和 class_ro_t

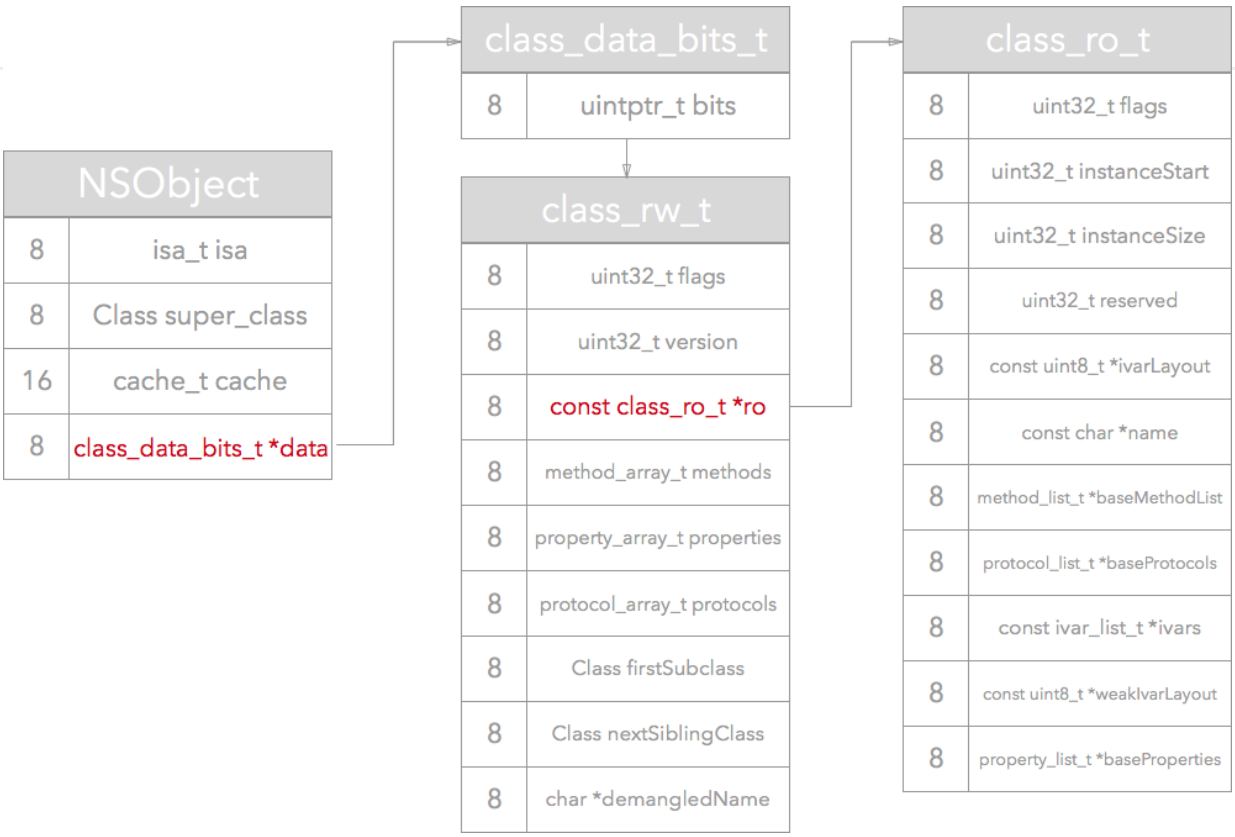
```
struct class_rw_t {
    uint32_t flags;
    uint32_t version;

    const class_ro_t *ro;

    method_array_t methods;
    property_array_t properties;
    protocol_array_t protocols;

    Class firstSubclass;
    Class nextSiblingClass;
};

struct class_ro_t {
    uint32_t flags;
    uint32_t instanceStart;
    uint32_t instanceSize;
    uint32_t reserved;

    const uint8_t * ivarLayout;

    const char * name;
    method_list_t * baseMethodList;
    protocol_list_t * baseProtocols;
    const ivar_list_t * ivars;

    const uint8_t * weakIvarLayout;
    property_list_t *baseProperties;
};
```

class_rw_t结构体内有一个指向class_ro_t结构体的指针。每个类都对应有一个class_ro_t结构体和一个class_rw_t结构体。在编译期间，class_ro_t结构体就已经确定，objc_class中的bits的data部分存放着该结构体的地址。在runtime运行之后，具体说来是在运行runtime的 `realizeClass` 方法时，会生成class_rw_t结构体，该结构体包含了class_ro_t，并且更新data部分，换成class_rw_t结构体的地址。

类的 `realizeClass` 运行之前：



类的 `realizeClass` 运行之后：



细看两个结构体的成员变量会发现很多相同的地方，他们都存放着当前类的属性、实例变量、方法、协议等等。区别在于：class_ro_t存放的是编译期间就确定的；而class_rw_t是在runtime时才确定，它会先将class_ro_t的内容拷贝过去，然后再将当前类的分类的这些属性、方法等拷贝到其中。所以可以说class_rw_t是class_ro_t的超集，当然实际访问类的方法、属性等也都是访问的class_rw_t中的内容

isa

- 在arm64架构之前，isa就是一个普通的指针，存储着Class、Meta-Class对象的内存地址
- 从arm64架构开始，对isa进行了优化，使用一个64位的共用体（union）结构存储数据，其中的33位存储内存地址，其余存储其他数据

```
union isa_t {
    Class cls;
    uintptr_t bits;
    struct {
    /*
        0，代表普通的指针，存储着Class、Meta-Class对象的内存地址
        1，代表优化过，使用位域存储更多的信息
    */
        uintptr_t nonpointer;
        // 是否有设置过关联对象，如果没有，释放时会更快
        uintptr_t has_assoc;
        // 是否有C++的析构函数（.cxx_destruct），如果没有，释放时会更快
        uintptr_t has_cxx_dtor;
        // 存储着Class、Meta-Class对象的内存地址信息
        uintptr_t shiftcls;
        // 用于在调试时分辨对象是否未完成初始化
        uintptr_t magic;
        // 是否有被弱引用指向过，如果没有，释放时会更快
        uintptr_t weakly_referenced;
        // 对象是否正在释放
        uintptr_t deallocating;
        /*
            引用计数器是否过大无法存储在isa中
            如果为1，那么引用计数会存储在一个叫SideTable的类的属性中
        */
        uintptr_t has_sidetable_rc;
        // 里面存储的值是引用计数器减1
        uintptr_t extra_rc;
    }
}
```

消息转发机制

消息发送

- 检测receiver(接收者)是否为nil,为nil退出
- receiver通过isa指针找到receiverClass
- receiverClass的cache中查找方法，找到调用方法，结束查找
- receiverClass的class_rw_t中查找方法,找到方法，调用方法并把方法缓存到receiverClass的cache中，结束查找。
- receiverClass通过superclass指针找到superClass，重复上面步骤

动态分析

- 是否曾经有动态解析，有的话直接消息转发
- 调用+resolveInstanceMethod或+resolveClassMethod来动态解析方法
- 标记为已动态解析
- 动态解析过后，会重新走“消息发送”的流程(从receiverClass的cache中查找方法)

消息转发

- 调用forwardingTargetForSelector:检测是否有备用接收者
  - 返回值不为nil,执行objc_msgSend(返回值, SEL)
  - 为nil，调用methodSignatureForSelector进行签名
    - 不为nil, 调用forwardInvocation:方法
    - 为nil，调用doesNotRecognizeSelector

weak原理

- runtime维持了一个weak表；当一个对象obj被weak指针指向时，这个weak指针会以obj的指针作为key，存储到sideTable类的weak_table这个散列表上对应的一个weak指针数组里面。 当一个对象obj的dealloc方法被调用时，Runtime会以obj的指针为key，从sideTable的weak_table散列表中，找出对应的weak指针列表，然后将里面的weak指针逐个置为nil

```
struct SideTable {
    // 保证原子操作的自旋锁
    spinlock_t slock;
    // 引用计数的 hash 表
    RefcountMap refcnts;
    // weak 引用全局 hash 表
    weak_table_t weak_table;
}
struct weak_table_t {
    // 保存了所有指向指定对象的 weak 指针
    weak_entry_t *weak_entries;
    // 存储空间
    size_t    num_entries;
    // 参与判断引用计数辅助量
    uintptr_t mask;
    // hash key 最大偏移值
    uintptr_t max_hash_displacement;
}
```

Runloop

运行循环, 在程序运行过程中循环做一些事情

应用

- 定时器（Timer）、PerformSelector
- GCD Async Main Queue
- 事件响应、手势识别、界面刷新
- 网络请求
- AutoreleasePool

作用

- 保持程序的持续运行
- 处理App中的各种事件（比如触摸事件、定时器事件等）
- 节省CPU资源，提高程序性能

具体应用

- 控制线程生命周期（线程保活）
- 解决NSTimer在滑动时停止工作的问题
- 监控应用卡顿
- 性能优化

Runloop与线程

- 每条线程都有唯一的一个与之对应的RunLoop对象
- RunLoop保存在一个全局的Dictionary里，线程作为key，RunLoop作为value
- 线程刚创建时并没有RunLoop对象，RunLoop会在第一次获取它时创建
- RunLoop会在线程结束时销毁
- 主线程的RunLoop已经自动获取（创建），子线程默认没有开启RunLoop

CFRunloopModeRef

- CFRunLoopModeRef代表RunLoop的运行模式
- 一个RunLoop包含若干个Mode，每个Mode又包含Source0/Source1/Timer/Observer
- RunLoop启动时只能选择其中一个Mode，作为currentMode
- 如果需要切换Mode，只能退出当前Loop，再重新选择一个Mode进入
  - 不同组的Source0/Source1/Timer/Observer能分隔开来，互不影响
- 如果Mode里没有任何Source0/Source1/Timer/Observer，RunLoop会立马退出

常用的Mode

- kCFRunLoopDefaultMode（NSDefaultRunLoopMode）：App的默认Mode，通常主线程是在这个Mode下运行
- 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

Mode

- source0:触摸事件处理

- source1:基于port的线程通信和系统事件捕捉
- timer:定时器事件
- observer:runloop状态监听

多线程

GCD

GCD 是苹果的低级线程接口，用于支持多核硬件上的并发代码执行。

调度组（DispatchGroup）

通常用于需要启动多个异步进程，但当所有进程完成后，我们只需要一个事件。这可以通过 DispatchGroup 来实现。

```
// 创建调度组
dispatch_group_t group = dispatch_group_create();
// 启动任务
dispatch_group_enter(group);
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
    NSLog(@"任务一");
    // 任务完成
    dispatch_group_leave(group);
});

dispatch_group_enter(group);
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, (int64_t)(3 * NSEC_PER_SEC)),
dispatch_get_main_queue(), ^{
    NSLog(@"任务二");
    dispatch_group_leave(group);
});

dispatch_queue_t queue =
dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);
// 任务完成通知，所有的 enter-leave 任务对完成后，group.notify 被调用
dispatch_group_notify(group, queue, ^{
    NSLog(@"任务结束");
});
```

调度信号量（DispatchSemaphore）

信号量，一种用来控制并发访问资源的机制，多用于多线程中，可以控制并发线程数量。

```
/*
    创建信号总量，即初始信号量允许的最大值， 最大并发任务的数量
    参数：value：信号总量的初值，数据类型为long类型。
    返回值：如果value小于0，创建的sem对象其实是NULL类型。
*/
dispatch_semaphore_t semaphore = dispatch_semaphore_create(1);
/*
    等待信号量
    参数：
        dsema：dispatch_semaphore_t对象
        timeout：DISPATCH_TIME_FOREVER或DISPATCH_TIME_NOW。
      目前一般都是选择DISPATCH_TIME_FOREVER
      DISPATCH_TIME_FOREVER 超时时间为永远，表示会一直等待信号量为正数，才会继续运行
      DISPATCH_TIME_NOW 超时时间为0，表示忽略信号量，直接运行。
*/
dispatch_semaphore_wait(semaphore, 60);
/*
    发送信号量
    参数： dsema：dispatch_semaphore_t对象
    返回值：返回值为long类型。
        当返回值为0时，表示当前并没有线程等待其处理的信号量，其处理的信号总量增加1。
        当返回值不为0时，表示其当前有一个或者多个线程等待其处理的信号量，并且该函数唤醒了一个等待的线程（当线程有优先级的时候，唤醒优先级最高的线程，否则随机唤醒）
*/
dispatch_semaphore_signal(semaphore);
```

DispatchWorkItem

`DispatchWorkItem` 是iOS8时候推出的一套API, 增加了一些对任务控制的API. 与 `Operation` 对象或者 `BlockOperation` 比较像, 主要可以主动 `cancel` , 并且增加了 `notify` 功能. 可以用于防抖和限流。

```
public class DispatchWorkItem {
    // 初始化方法 => 最关键的是 最后的 block, 实际就是 我们要执行的 TaskItem
    public init(qos: DispatchQoS = .unspecified, flags: DispatchWorkItemFlags =
[], block: @escaping @convention(block) () -> Void)
    /// sync 同步执行
    public func perform()
    /// 内置信号量 - 同步等待任务执行完成
    public func wait()
    /// 内置信号量 - 同步等待指定时间, 等待任务完成
    public func wait(timeout: DispatchTime) -> DispatchTimeoutResult
    public func wait(wallTimeout: DispatchWallTime) -> DispatchTimeoutResult
    /// 内置信号量, WorkItem 执行完成或者被调度, 信号量触发通知, 可以指定 barrier
    public func notify(qos: DispatchQoS = .unspecified, flags:
DispatchWorkItemFlags = [], queue: DispatchQueue, execute: @escaping
@convention(block) () -> Void)
    public func notify(queue: DispatchQueue, execute: DispatchWorkItem)
    /// 可以取消, 异步将 WorkItem内部的state设置层 isCancelled. 被调度立刻返回(注意, 也会
触发notify)
    public func cancel()
    /// 判断WorkItem是否被取消的状态
    public var isCancelled: Bool { get }
}
```

dispatch_once原理

dispatch_once包含了一个 predicate 参数和一个 block 参数。其中，predicate 是一个指向 dispatch_once_t 结构体的指针，用来判断该 block 是否已经被执行过。block 则是需要执行的单例创建操作。

dispatch_once 函数会执行以下两个步骤：

- 调用 dispatch_once 函数时，dispatch_once_t 结构体会被初始化为 0。
- 在 block 中执行单例创建操作，创建成功后将 dispatch_once_t 结构体的值修改为非 0 值。

完成上述两个步骤后，再次调用 dispatch_once 函数时，由于 dispatch_once_t 的值已经被修改为非 0 值，dispatch_once 函数将不会再次执行 block 中的内容，从而确保该单例只会被创建一次。

NSOperation

NSOperation 是建立在 GCD 之上的。NSOperation 的一些好处是，它有一个更友好的接口来处理 Dependencies（按特定顺序执行任务），它是可观察的（KVO 来观察属性），有暂停、取消、恢复和控制（你可以指定队列中任务的数量）。

Block

block本质上也是一个OC对象，它内部也有个isa指针

block是封装了函数调用以及函数调用环境的OC对象

```
struct Block_layout {
    void *isa;
    int flags;
    int reserved;
   // block执行时调用的函数指针，block定义时内部的执行代码都在这个函数中
    void (*invoke)(void *, ...);
    // block的详细描述
    struct Block_descriptor *descriptor;
    /* Imported variables. */
};
struct Block_descriptor {
    unsigned long int reserved;
    unsigned long int size;
    // copy/dispose，辅助拷贝/销毁函数，处理block范围外的变量时使用
    void (*copy)(void *dst, void *src);
    void (*dispose)(void *);
};
```

Block变量捕获

为了保证block内部能够正常访问外部的变量，block有个变量捕获机制

变量类型	捕获到block内部	访问方式
局部变量(auto)	☑	值传递
局部变量(static)	☑	指针传递
全局变量	☐	直接访问

auto变量被捕获时直接传入的变量的值，而static变量被捕获时传入的是变量的地址，由于auto变量出了当前作用域内存就会被销毁，所以需要将auto变量的值捕获；

static变量是一直存在内存中，但出了作用域就访问不了啦，所以只需要捕获static变量的内存地址就可以了；

全局变量是不会被捕获的，因为全局变量在哪里都可以访问，不需要进行捕获。

__block

`__block` 修饰auto变量，在block内使用，block会把修饰的变量包装成对象，所以使用 `__block` 修饰的变量可以进行修改，这是因为block内使用的指针进行修改

```
struct 包装的属性对象 {
    void *isa;
    // 指向自己
    void *forwarding;
    int flag;
    int size;
    // 修饰的变量
    ...
};
```

- block在修改NSMutableArray时需要进行 `__blcok` 吗?
  - 不需要，因为array使用的是指针修改，不存在无法修改的问题。

copy修饰

block一旦没有进行copy操作,就不会在堆上,就无法控制block的生命周期

在ARC环境下，编译器会根据情况自动将栈上的block复制到堆上，比如以下情况

- block作为函数返回值时 `return ^{ } ;`
- 将block赋值给__strong指针时 `Block bolck = ^{ };`
- block作为Cocoa API中方法名含有usingBlock的方法参数时

```
[array enumerateObjectsUsingBlock:^(id  _Nonnull obj, NSUInteger idx, BOOL *
_Nonnull stop) {
}] ;
```

- block作为GCD API的方法参数时

```
dispatch_once(&onceToken, ^{
});
```

所以ARC下使用strong/copy都可以，MRC下编译器不会自动将栈上的block复制到堆上，所以只能使用copy

类型

类型	介绍
NSGlobalBlock	没有访问auto变量的block，存放在数据段。调用copy什么都不做。
NSStackBlock	访问auto变量,存放在栈中，注意ARC中测试时会显示为NSMallockBlock,这是因为编译器对block做了处理，关闭ARC即可。会自动销毁，所以需要用copy修饰，把block存到堆中
NSMallocBlock	NSStackBlock调用copy；调用copy引用计数器加1

应用程序的内存分配

区域	block类型	存放数据类型
程序区域(.text区)		存放代码
数据区域(.data区)	_NSConcreteGlobalBlock	全局变量存储的位置
堆	_NSConcreteMallocBlock	alloc出的对象，手动释放
栈	_NSConcreteStackBlock	局部变量存放的位置，系统自动释放

Category

主要使用在方法拆分、不改变原类的基础上增加方法、属性等。

```
struct category_t{
    const char *name;    // 扩展类名
    struct _class_t *cls;    //
    const struct _method_list_t *instance_methods;  // 对象方法列表
    const struct _method_list_t *class_methods; // 类方法类别
    const struct _protoclo_list_t *protocols   // 协议列表
    const struct _prop_list_t *properties;     // 属性列表
}
```

加载过程

- 通过runtime加载某个类的所有Category数据
- 把所有的Category方法、属性、协议合并到一个新数组中
- 将合并后的分类数据插入到类原有的数据前面

注：因为分类数据是插入到类原有数据前面，所以调用属性、方法、协议等都优先调用分类中的数据。

与Extension区别

- Category是在运行时才把分类数据添加进类信息中
- Extension编译是就把数据添加进类信息中

load

- load方法会在runtime加载类、分类时调用
- 每个类、分类的+load,在程序运行过程中只调用一次

注：分类的+load不会覆盖类的+load

调用顺序

- 先调用类的+load
  - 按编译先后顺序调用（先编译先调用）
  - 调用子类的+load之前会先调用父类的+load
- 再调用分类的+load
  - 按编译先后顺序调用（先编译先调用）

initialize

- initialize方法会在类第一次接收到消息时调用

调用顺序

- 先调用父类的+initialize,再调用子类的+initialize(子类不存在时不调用)
- 先初始化父类，在初始化子类，每个类只初始化一次

load与initialize

- +initialize是通过objc_msgSend进行调用；+load是根据函数地址直接调用。
- 如果子类没有实现+initialize，会调用父类的+initialize，所以父类的+initialize可能会调用多次
- 如果分类实现的+initialize，就覆盖类本身的+initialize调用

成员变量

- 不能直接给Category添加成员变量，因为Category结构中没有属性列表，但是可以通过runtime的关联机制实现相同的效果

属性关联

- 关联对象并不是存储在被关联对象本身内存中
- 关联对象存储在全局的统一的一个AssociationsManager中
- 设置关联对象为nil，就相当于是移除关联对象

KVO&KVC

KVC

查找顺序

- 按照getKey、key、isKey、_key顺序查找方法
  - 找到直接调用方法
  - 未找到：查看accessInstanceVariablesDirectly方法返回值，默认为Yes；方法意思是是否允许直接访问成员变量。
    - Yes：按照_key、_isKey、key、isKey的顺序查找
      - 找到直接赋值
      - 未找到
    - No：调用valueForUndefinedKey:方法，抛出NSUnKnownKeyException异常

赋值顺序

- 按照setKey、_setKey顺序查找方法
  - 找到方法：传递参数，调用方法
  - 未找到：查看accessInstanceVariablesDirectly方法返回值，默认为Yes；方法意思是是否允许直接访问成员变量。
    - Yes：按照_key、_isKey、key、isKey的顺序查找
      - 找到直接赋值
      - 未找到
    - No：调用setValue:forUndefinedKey:方法，抛出NSUnKnownKeyException异常

KVO

原理

- 利用Runtime机制功态生成个子类，并且让instance对象的isa指针指向这个全新的类
- 当修改instance对象的属性时，会调用foundation的_NSSetXXXValueAndNotify函数
- 调用顺序
  - willChangeValueForKey:
  - setValue:
  - didChangeValueForKey:
- 内部会触发监听器（Oberser）的监听方法

派生类重写方法

- set方法：变化监听
- class：屏蔽方法实现
- dealloc：后续收尾
- __isKvoA：

手动触发

- 手动调用WillChangeValueForKey
- set方法赋值
- 手动调用DidChangeValueForKey

问题

- 直接修改属性不会执行属性监听方法

性能优化

App启动优化

- APP的启动可以分为2种
  - 冷启动（Cold Launch）：从零开始启动APP
  - 热启动（Warm Launch）：APP已经在内存中，在后台存活着，再次点击图标启动APP
- 通过添加环境变量可以打印出APP的启动时间分析（Edit scheme -> Run -> Arguments）
  - DYLD_PRINT_STATISTICS设置为1
  - 如果需要更详细的信息，那就将DYLD_PRINT_STATISTICS_DETAILS设置为1
- dyld
  - 减少动态库、合并一些动态库（定期清理不必要的动态库）
  - 减少Objc类、分类的数量、减少Selector数量（定期清理不必要的类、分类）
  - 减少C++虚函数数量
  - Swift尽量使用struct
- runtime
  - 用+initialize方法和dispatch*once取代所有的__attribute*((constructor))、C++静态构造器、ObjC的+load
- main
  - 在不影响用户体验的前提下，尽可能将一些操作延迟，不要全部都放在finishLaunching方法中
  - 按需加载

安装包瘦身

资源（图片、音频、视频等）

- 采取无损压缩
- 去除没有用到的资源

可执行文件瘦身

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

CPU和GPU

- CPU: 对象的创建和销毁、对象属性的调整、布局计算、文本的计算和排版、图片的格式转换和解码、图像的绘制（Core Graphics)
  - 主要思路：尽可能减少CPU、GPU资源消耗
  - 尽量用轻量级的对象，比如用不到事件处理的地方，可以考虑使用CALayer取代UIView
  - 不要频繁地调用UIView的相关属性，比如frame、bounds、transform等属性，尽量减少不必要的修改
  - 尽量提前计算好布局，在有需要时一次性调整对应的属性，不要多次修改属性
  - Autolayout会比直接设置frame消耗更多的CPU资源
  - 图片的size最好刚好跟UIImageView的size保持一致
  - 控制一下线程的最大并发数量
  - 尽量把耗时的操作放到子线程
    - 文本处理（尺寸计算、绘制）

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

离屏渲染

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

卡顿检测

- 可以添加Observer到主线程RunLoop中，通过监听RunLoop状态切换的耗时，以达到监控卡顿的目的

耗电优化

- CPU优化
    - 尽可能降低CPU、GPU功耗
    - 少用定时器
    - 优化I/O操作
        - 尽量不要频繁写入小数据，最好批量一次性写入
        - 读写大量重要数据时，考虑用dispatch_io，其提供了基于GCD的异步操作文件I/O的API。用dispatch_io系统会优化磁盘访问
        - 数据量比较大的，建议使用数据库（比如SQLite、CoreData）
    - 网络优化
        - 减少、压缩网络数据
        - 如果多次请求的结果是相同的，尽量使用缓存
        - 使用断点续传，否则网络不稳定时可能多次传输相同的内容
        - 网络不可用时，不要尝试执行网络请求
        - 让用户可以取消长时间运行或者速度很慢的网络操作，设置合适的超时时间
        - 批量传输，比如，下载视频流时，不要传输很小的数据包，直接下载整个文件或者一大块一大块地下载。如果下载广告，一次性多下载一些，然后再慢慢展示。如果下载电子邮件，一次下载多封，不要一封一封地下载
- 定位优化

    - 如果只是需要快速确定用户位置，最好用CLLocationManager的requestLocation方法。定位完成后，会自动让定位硬件断电
    - 如果不是导航应用，尽量不要实时更新位置，定位完毕就关掉定位服务
    - 尽量降低定位精度，比如尽量不要使用精度最高的kCLLocationAccuracyBest
    - 需要后台定位时，尽量设置pausesLocationUpdatesAutomatically为YES，如果用户不太可能移动的时候系统会自动暂停位置更新
    - 尽量不要使用startMonitoringSignificantLocationChanges，优先考虑startMonitoringForRegion:
- 硬件检测优化
    - 用户移动、摇晃、倾斜设备时，会产生动作(motion)事件，这些事件由加速度计、陀螺仪、磁力计等硬件检测。在不需要检测的场合，应该及时关闭这些硬件

图片的处理

一张图片从磁盘中显示到屏幕上过程大致如下：从磁盘加载图片信息、解码二进制图片数据为位图、通过 CoreAnimation 框架处理最终绘制到屏幕上。实际上图片的绘制过程往往不是性能瓶颈，最耗时的操作是解码过程，若图片文件过大，从磁盘读取的过程也有可观的耗时。

加载和解压

一般使用imageNamed:或者imageWithData:从内存中加载图片生成UIImage的实例，此刻图片并不会解压，当 RunLoop 准备处理图片显示的事务（CATransaction）时，才进行解压，而这个解压过程是在主线程中的，这是导致卡顿的重要因素。

imageNamed: 方法

使用imageNamed:方法加载图片信息的同时（生成UIImage实例），还会将图片信息缓存起来，所以当使用该方法第一次加载某张图片时，会消耗较多的时间，而之后再次加载该图片速度就会非常快（注意此时该图片是未绘制到屏幕上的，也就是说还未解压）。

在绘制到屏幕之前，第一次解压成功后，系统会将解压信息缓存到内存。值得注意的是，这些缓存都是全局的，并不会因为当前UIImage实例的释放而清除，在收到内存警告或者 APP 第一次进入后台才有可能会清除，而这个清除的时机和内容是系统决定的，我们无法干涉。

imageWithData: 方法

使用imageWithData:方式加载图片时，不管是加载过程还是解压过程，都不会像imageNamed:缓存到全局，当该UIImage实例释放时，相关的图片信息和解压信息就会销毁。

两种加载方式的区别

从上面的分析可知，imageNamed:使用时会产生全局的内存占用，但是第二次使用同一张图片时性能很好；imageWithData:不会有全局的内存占用，但对于同一张图片每次加载和解压都会“从头开始”。

由此可见，imageNamed:适合“小”且“使用频繁”的图片，imageWithData:适合“大”且“低频使用”的图片。

加载和解压的优化

加载优化

对于加载过程，若文件过大或加载频繁影响了帧率（比如列表展示大图），可以使用异步方式加载图片，减少主线程的压力,代码大致如下：

```
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
      UIImage *image = [UIImage imageWithContentsOfFile:[[NSBundle mainBundle]
pathForResource:@"testImage" ofType:@"jpeg"]];
      dispatch_async(dispatch_get_main_queue(), ^{
          //业务
      });
});
```

解压优化

解压是耗时的，而系统默认是在主线程执行，所以业界通常有一种做法是，异步强制解压，也就是在异步线程主动将二进制图片数据解压成位图数据，使用CGBitmapContextCreate(...)系列方法就能实现。

锁

互斥锁

防止两条线程同时对同一公共资源(比如全局变量)进行读写的机制。当获取锁操作失败时，线程会进入睡眠，等待锁释放时被唤醒

- 递归锁:可重入锁，同一个线程在锁释放前可再次获取锁，即可以递归调用
- 非递归锁:不可重入，必须等锁释放后才能再次获取锁

自旋锁

线程反复检查锁变量是否可用。由于线程在这一过程中保持执行， 因此是一种忙等待。一旦获取了自旋锁，线程会一直保持该锁，直至显式释 放自旋锁。自旋锁避免了进程上下文的调度开销，因此对于线程只会阻塞很短时间的场合是有效的

区别

- 互斥锁在线程获取锁但没有获取到时，线程会进入休眠状态，等锁被释放时线程会被唤醒
- 自旋锁的线程则会一直处于等待状态（忙等待）不会进入休眠——因此效率高

三方库

SDWebImage

原理解析

- 入口 setImageWithURL:placeholderImage:options:会先把 placeholderImage显示，然后 SDWebImageManager 根据 URL 开始处理图片。（以URL的MD5值作为key）
- 进入SDImageCache从内存缓存查找SDImageCacheDelegate回调给SDWebImageManager，然后通过NSDWebImageManagerDelegate回调展示
- 如果内存缓存中没有，生成 ` NSOperation ` 添加到队列，开始从硬盘（Disk）查找图片是否已经下载
  - 有： 回主线程进行结果回调 NotifyDelegate，将图片添加到内存缓存中SDImageCache，再回调展示
  - 无： 共享或重新生成一个SDWebImageDownloader下载图片，由 NSURLSession实现相关 delegate，来判断图片下载中、下载完成和下载失败。
- 下载完后，放入硬盘，加入缓存，再回调展示

SDWebImage缓存为什么使用MapTable

- NSMaptable是可变的，没有不可变的类
- 可以持有键和值的弱引用，当键值当中的一个被释放时，整个这一项都会移除掉
- 可以对成员进行copy操作
- 可以存储任意的指针，通过指针来进行相等性和散列检查

AFNetworking

框架核心

NSURLSession

- AFURLSessionManager
- AFHTTPSessionManager

序列化/反序列化

- AFURLRequestSerialization上传的数据转换成JSON格式
- AFJSONResponseSerializer JSON解析器

安全协议

- AFSecurityPolicy 是针对 HTTPS的 服务

网络管理器

- AFNetworkReachabilityManager，网络状态检测

UIKit

- 提供了网络请求过程中与UI界面显示相关的操作接口 ActivityIndicator、UIAlertView、UIButton、UIImageView、UIprogressView、UIWebView

请求过程

- 初始化会话管理类：AFURLSessionManager
- 配置会话模式类型：NSURLSessionConfig

- 创建任务Task对象，启动任务
- 通过KVO监听download进度和upload进度
- 由任务代理回调处理：AFURLSessionmanagerTaskDelegate，数据响应，错误响应

MJExtension

- NSString、NSData 转化成JSON对象：(NSDictionary本身就是json对象) [keyValuesArray mj_JSONObject]
- 遍历属性，返回属性列表，映射成对象MJProperty。 在Block回调中可以获取到每一个MJProperty（封装的属性） 通过单例做属性缓存

Swift

Swift和Objective-C有什么区别?

- Swift是强类型（静态）语言，有类型推断，Objective-C弱类型（动态）语言
- Swift面向协议编程，Objective-C面向对象编程
- Swift注重值类型，Objective-C注重引用类型
- Swift支持泛型，Objective-C只支持轻量泛型（给集合添加泛型）
- Swift支持静态派发（效率高）、动态派发（函数表派发、消息派发）方式，Objective-C支持动态派发（消息派发）方式
- Swift支持函数式编程（高阶函数）
- Swift的协议不仅可以被类实现，也可以被Struct和Enum实现
- Swift有元组类型、支持运算符重载
- Swift支持命名空间
- Swift支持默认参数
- Swift比Objective-C代码更简洁

派发机制

静态派发(直接派发)

- final、static关键字的函数
- Struct、Enum
- 协议的Extensions、 Class的Extensions

函数表派发

- Class默认、协议声明函数

消息派发

- dynamic修饰的函数(包含Extension)

Struct和Class的区别?

- Struct不支持继承，Class支持继承
- Struct是值类型，Class是引用类型
- Struct无法修改自身属性值，函数需要添加mutating关键字
- Struct不需要deinit方法，因为值类型不关系引用计数，Class需要deinit方法
- Struct初始化方法是基于属性的

?与??的区别

- ？用来声明可选值，如果变量未初始化则自动初始化nil；在操作可选值时，如果可选值时nil则不响应后续的操作；使用as?进行向下转型操作；
- ?? 用来判断左侧可选值非空（not nil）时返回左侧值可选值，左侧可选值为空（nil）则返回右侧的值。

mutating的作用

- Swift中协议是可以被Struct和Enum实现的，mutating关键字是为了能在被修饰的函数中修改Struct或Enum的变量值，对Class完全透明。

Set(集合类型)的使用场景

- Set存储值类型相同、无序、去重

final关键词的用法

- final关键词的作用：它修饰的类、方法、变量是不能被继承或重写的，编译器会报错。另外，通过它可以显示的指定函数的派发机制。

lazy关键词的用法

- lazy关键词的作用：指定延时加载（懒加载），懒加载存储属性只会在首次使用时才会计算初始值属性。懒加载属性必须声明（var）为变量，因为常量属性（let）初始化之前会有值。 lazy修饰的属性非线程安全的。

闭包是引用类型吗?

- 闭包是引用类型。如果一个闭包被分配给一个变量，这个变量复制给另一个变量，那么他们引用的是同一个闭包，他们的捕捉列表也会被复制。

static和class的区别

- static可以修饰属性和方法
  - 所修饰的属性和方法不能够被重写。
  - static修饰的类方法和属性包含了final关键字的特性，重写会报错
- class修饰方法和计算属性
  - 我们同样可以使用class修饰方法和计算属性，但是不能够修饰存储属性
  - 类方法和计算属性是可以被重写的，可以使用class关键字也可以static

closure与block的区别

- closure是匿名函数、block是一个结构体对象
- closure通过逃逸闭包来在内部修改变量，block 通过 __block 修饰符

闭包

- 闭包会强引用它捕获的所有变量，非逃逸闭包不会造成循环应用，并且非逃逸闭包它的上下文的内存可以保存在栈上而不是堆上

逃逸闭包

- 在函数作用域以外被调用的闭包

非逃逸闭包

- 在函数作用域以内被调用的闭包

Swift为什么将String,Array,Dictionary设计成值类型?

- 值类型相比引用类型,最大的优势在于内存使用的高效.
- 值类型在栈上操作,引用类型在堆上操作.栈上的操作仅仅是单个指针的上下移动,而堆上的操作则牵涉到合并、移位、重新链接等.也就是说Swift这样设计,大幅减少了堆上的内存分配和回收的次数.同时写时复制又将值传递和复制的开销降到了最低.
- String,Array,Dictionary设计成值类型,也是为了线程安全考虑.通过Swift的let设置,使得这些数据达到了真正意义上的“不变”,它也从根本上解决了多线程中内存访问和操作顺序的问题.
- 设计成值类型还可以提升API的灵活度.

值类型的写时复制

- 只有当一个结构体发生了写入行为时才会有复制行为。
- 在结构体内部用一个引用类型来存储实际的数据，在不进行写入操作的普通传递过程中，都是将内部的reference的引用计数+1，在进行写入操作时，对内部的reference做一次copy操作用来存储新的数据，防止和之前的reference产生意外的数据共享。
- swift中提供该[isKnownUniquelyReferenced]函数，他能检查一个类的实例是不是唯一的引用，如果是，我们就不需要对结构体实例进行复制，如果不是，说明对象被不同的结构体共享，这时对它进行更改就需要进行复制。

值类型与引用类型

- 类是引用类型, 结构体为值类型
- 结构体不可以继承
- 值类型被赋予给一个变量、常量或者被传递给一个函数的时候，其值会被拷贝
- 引用类型在被赋予到一个变量、常量或者被传递到一个函数时，其值不会被拷贝。因此，引用的是已存在的实例本身而不是其拷贝
- 值类型是在栈中处理，引用类型是在堆中处理，栈中只牵扯到一个指针的上下移动，堆中要牵扯到合并、移动、重新链接等情况，所以值类型比引用类型更高效

存储属性和计算属性

存储属性

- 存储在特定类或结构体实例里的一个常量或变量。存储属性可以是变量存储属性（用关键字 var 定义），也可以是常量存储属性（用关键字 let 定义）。

计算属性

- 除存储属性外，类、结构体和枚举可以定义计算属性。计算属性不直接存储值，而是提供一个getter和一个可选的setter，来间接获取和设置其他属性或变量的值。

SwiftUI

描述 SwiftUI 中的视图

- `SwiftUI` 中的视图是一个轻量级瞬态对象，旨在在更改其源时被扔掉。
- 这就是SwiftUI具有数据驱动的性质， `SwiftUI` 通过在必要时以高性能的方式重新计算视图主体来处理更新。开发人员不会直接处理视图刷新，而是系统处理此操作。
- SwiftUI视图是引擎盖下的 `struct` ，符合 `SwiftUI` 的 `View` 协议，以描述用户可查看的元素。

描述 `@State` 、 `@Binding` 、 `@ObservedObject` 、 `@Published` 和 `@EnvironmentObject` 之间的区别

- @State
  - 属性包装器用于为应用程序中的数据创建单一真相源，该数据会随着时间的推移而变异（并适当更新视图）。
  - `@State` 中的包裹值是任何东西（通常是值类型）。 `wrappedValue` 存储在堆上，当它更改无效时，视点和视点此值的视图将被标记为更新。
- @Binding
  - `@Binding` 属性包装器允许存储数据的属性与显示和突变该数据的视图之间的双向连接。
  - `wappedValue` 是绑定到其他东西的值（任何东西）。它从其他来源获取/设置包裹值的值。当绑定值更改时，它使标记为更新的视图无效。
- @ObservedObject 和 Published
  - 为了使通过 `@Published` 观察到的属性到包含类之外的类，该包含类需要继承 `ObservableObject` ，相关属性应标记为 `@Published` ——该属性合成 `objectWillChange` 发布者以宣布该值将发生变化。
- @EnvironmentObject
  - 一个属性包装器，允许视图访问并响应自定义定义的设置和条件。

描述 `Spacer` 组件的作用

`Spacer` 组件可以通过占用尽可能多的空间来撑开相邻的区域，它有以下特点：

- 灵活的间距：
  - 可以使用 `minLength` 参数来控制 `Spacer` 试图占用的空间量。
  - 如果有多个 `Spacers` ，它们可以在彼此之间平均分配可用空间。
- 自适应布局：
  - `Spacer` 通过根据可用空间自动调整间距，帮助创建适应不同屏幕尺寸和方向的布局。
- 可访问性：
  - `Spacer` 不仅有助于布局设计；它还可以通过在UI元素之间创建清晰的视觉分离来促进应用程序的可访问性，这有助于用户更好地理解界面。

描述 SwiftUI 生命周期

- 解释 `SwiftUI` 视图与 `UIViewController` 有何不同?
  - 概括的来讲，所有的 `SwiftUI` 视图都是声明性的，而 `UIKit` 是命令式的。
- 如何在 SwiftUI 中处理视图初始化和取消初始化?
  - 在 `UIKit` 中， `deinit` 当视图控制器被释放时会调用该方法。在 `SwiftUI` 中，因为结构体是轻量级的，当它们超出范围时，它们会自动释放。
- SwiftUI 初始化和 UIKit 有什么区别?
  - `SwiftUI` 视图是轻量级的，可以频繁创建和销毁。 `SwiftUI` 视图的方法 `init` 可用于执行初始设置。
  - 当视图添加到屏幕时，会调用 `onAppear` 修饰符，当视图从屏幕上删除时，会调用 `onDisappear` 修饰符，这些修饰符可以用在所有视图上。
  - 这和 UIKit 中的 `viewWillAppear` 和 `viewWillDisappear` 效果接近，示例代码如下：

```
Text("SwiftUI 2023 面试题 by Codeun.com")
    .onAppear { print("视图出现") }
    .onDisappear { print("视图消失") }
```

- SwiftUI 如何更新视图?
  - 每当视图依赖的某些状态发生变化时，SwiftUI 视图的主体就会重新计算和绘制，使用 `@Bindings` 、 `@State` 、 `@ObservedObjects` 、 `@EnvironmentObjects` 、 `@AppStorage` 和 `@SceneStorage` 这些属性包装器都能以不同方式影响到页面视图的更新。
- SwiftUI 如何获知销毁视图?
  - `SwiftUI` 视图是值类型，因此没有直接方法来直接检测它们什么时候被释放。但是可以使用的对象（引用类型）（如 `ObservableObject` 实例）析构器 (`deinit`)，这样在释放它们时会得到该析构器调用。

描述 iOS 14 及更高版本中的 SwiftUI 应用程序生命周期

- 随着 `iOS 14` 以及更高版本中引入 `@main` ，您可以使用 SwiftUI 创建应用程序，而无需传统的 AppDelegate。现在可以直接使用 `App` 协议来定义应用程序的入口点。
- 生命周期方法（例如处理后台或前台事件）可以使用新的应用程序生命周期方法进行处理；
- 列举与 `App` 协议相关联的生命周期事件
  - onContinueUserActivity 当应用程序继续从另一台设备移交的用户活动时调用
  - onOpenURL 当应用程序通过URL打开时调用
- 总而言之，处理 `SwiftUI` 生命周期的工作围绕着了解视图何时被重新绘制，对特定的生命周期事件使用 `onAppear` 和 `onDisappear` 等修饰符，以及以让 `SwiftUI` 知道何时需要更新的方式管理状态。与 `SwiftUI` 一样，关键是陈述性地思考：根据 UI 的状态定义 UI 的外观，系统管理实际的渲染和更新。

Flutter

Dart语法中dynamic，var，object三者的区别

- var定义的类型是不可变的
- dynamic和object类型是可以变的，而dynamic 与object 的最大的区别是在静态类型检查上

const和final的区别

相同点

- 必须初始化
- 只能赋值一次

不同点

- final可修饰实例变量、const不可以修饰实例变量
- 访问类中const修饰的变量需要static修饰
- const修饰的List集合任意索引不可修改，final修饰的可以修改
- const 用来修饰变量 只能被赋值一次，在编译时赋值 final 用来修饰变量 只能被赋值一次，在运行时赋值
- final 只可用来修饰变量， const 关键字即可修饰变量也可用来修饰 常量构造函数 当const修饰类的构造函数时，它要求该类的所有成员都必须是final的。

?? 与 ??= 的区别

??

- 左边如果为空返回右边的值，否则不处理。

??=

- 左边如果为空把B的值赋值给A

什么是flutter里的key? 有什么用?

- key是Widgets，Elements和SemanticsNodes的标识符。
- key有LocalKey 和 GlobalKey两种。
  - LocalKey
    - 如果要修改集合中的控件的顺序或数量。
  - GlobalKey
    - 允许 Widget 在应用中的 任何位置更改父级而不会丢失 State。

Widget

StatelessWidget

- 不会重新构建

StatefulWidget

- 可以重新构建

生命周期

StatelessWidget

- 构造函数
- build方法

StatefulWidget

- widget的构造方法
- createState
- state的构造方法
- state的initState方法(重写该方法时，必须要先调用super. initState())
- didChangeDependencies方法
  - 调用initState方法后，会调用该方法
  - 从其他widget中依赖一些数据发生改变时，比如用InheritedWidget，provider来监听数据的改变
- state的build方法（当调用setState方法，会重新调用build进行渲染）
- state的deactivate方法（当state被暂时从视图移除的时候会调用，页面push走、pop回来的时候都会调用。因为push、pop会改变widget在视图树位置，需要先移除再添加。重写该方法时，必须要先调用super.deactivate()）
- state的dispose方法。页面被销毁的时候调用，如：pop操作。通常情况下，自己的释放逻辑放在super.dispose()之前，先操作子类在操作父类。

async和await

- await的出现会把await之前和之后的代码分为两部分，await并不像字面意思所表示的程序运行到这里就阻塞了，而是立刻结束当前函数的执行并返回一个Future，函数内剩余代码通过调度异步执行。
- async是和await搭配使用的，await只在async函数中出现。在async 函数里可以没有await或者有多个await。

future和steam有什么不一样?

- Future 用于处理单个异步操作
- Stream 用来处理连续的异步操作

Widget、Element、RenderObject、Layer都有什么关系?

- Widget会被inflate（填充）到Element，并由Element管理底层渲染树。Widget并不会直接管理状态及渲染,而是通过State这个对象来管理状态。Flutter创建Element的可见树，相对于Widget来说，是可变的，通常界面开发中，我们不用直接操作Element,而是由框架层实现内部逻辑。就如一个UI视图树中，可能包含有多个TextWidget(Widget被使用多次)，但是放在内部视图树的视角，这些TextWidget都是填充到一个个独立的Element中。Element会持有renderObject和widget的实例。记住，Widget 只是一个配置，RenderObject 负责管理布局、绘制等操作。 在第一次创建 Widget 的时候，会对应创建一个 Element， 然后将该元素插入树中。如果之后 Widget 发生了变化，则将其与旧的 Widget 进行比较，并且相应地更新 Element。重要的是，Element 不会被重建，只是更新而已。

Widget

- 仅用于存储渲染所需要的信息。

RenderObject

- 负责管理布局、绘制等操作。

Element

- 控件树上的实体

dart是弱引用还是强引用

- 强引用

dart是值传递还是引用传递

- 值类型

热重载

- Flutter 的热重载是基于 JIT 编译模式的代码增量同步。由于 JIT 属于动态编译，能够将 Dart 代码编译成生成中间代码，让 Dart VM 在运行时解释执行，因此可以通过动态更新中间代码实现增量同步。
- 热重载的流程可以分为 5 步，包括：扫描工程改动、增量编译、推送更新、代码合并、Widget 重建。Flutter 在接收到代码变更后，并不会让 App 重新启动执行，而只会触发 Widget 树的重新绘制，因此可以保持改动前的状态，大大缩短了从代码修改到看到修改产生的变化之间所需要的时间。
- 另一方面，由于涉及到状态的保存与恢复，涉及状态兼容与状态初始化的场景，热重载是无法支持的，如改动前后 Widget 状态无法兼容、全局变量与静态属性的更改、main 方法里的更改、initState 方法里的更改、枚举和泛型的更改等。 可以发现，热重载提高了调试 UI 的效率，非常适合写界面样式这样需要反复查看修改效果的场景。但由于其状态保存的机制所限，热重载本身也有一些无法支持的边界。

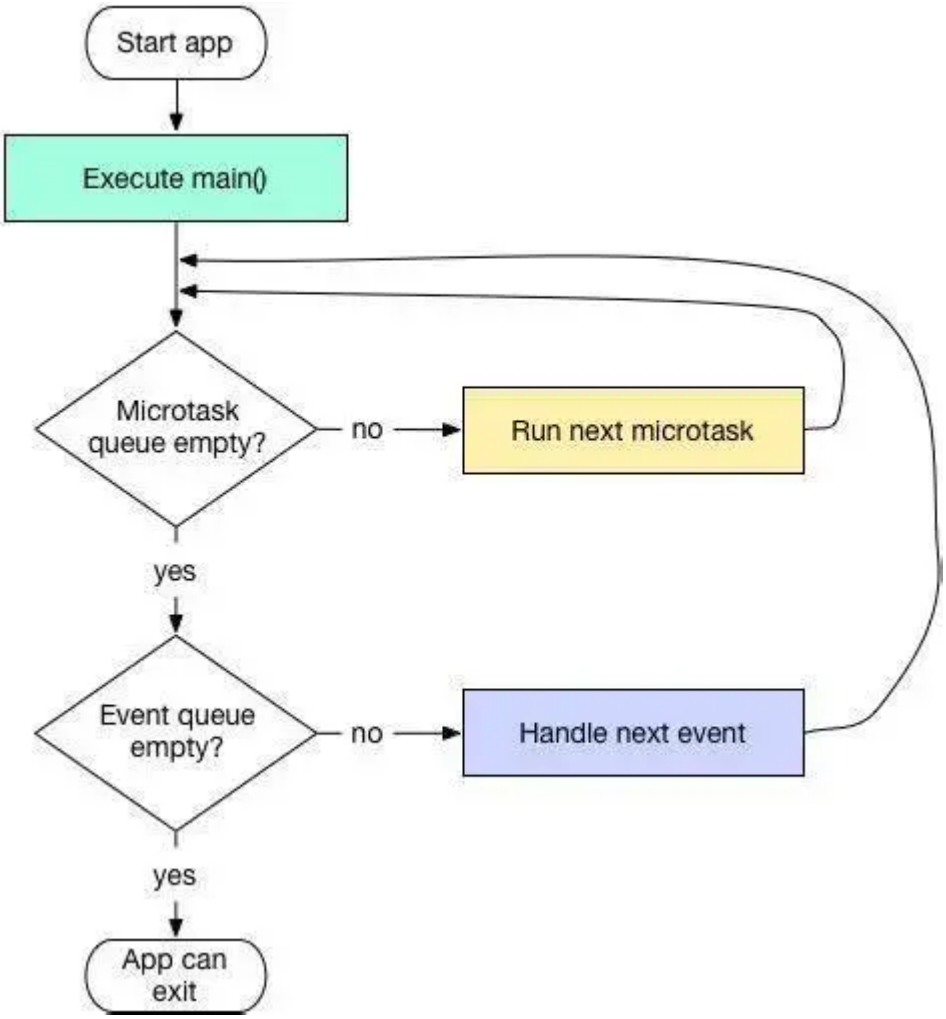
Dart 的作用域

- Dart 没有 「public」「private」等关键字，默认就是公开的，私有变量使用下划线 _开头。

Dart 当中的 **「 .. 」表示什么意思?

- Dart 当中的 「..」意思是 「级联操作符」，为了方便配置而使用。「..」和「.」不同的是 调用「..」后返回的相当于是 this，而「.」返回的则是该方法返回的值 。

Dart 是不是单线程模型？是如何运行的?



- Dart 在单线程中是以消息循环机制来运行的，包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。
- 当Flutter应用启动后，消息循环机制便启动了。首先会按照先进先出的顺序逐个执行微任务队列中的任务，当所有微任务队列执行完后便开始执行事件队列中的任务，事件任务执行完毕后再去执行微任务，如此循环往复，生生不息。

Dart 是如何实现多任务并行的?

- Dart 是单线程的，不存在多线程，那如何进行多任务并行的呢？其实，Dart的多线程和前端的多线程有很多的相似之处。Flutter的多线程主要依赖Dart的并发编程、异步和事件驱动机制。
- 在Dart中，一个Isolate对象其实就是一个isolate执行环境的引用，一般来说我们都是通过当前的isolate去控制其他的isolate完成彼此之间的交互，而当我们想要创建一个新的Isolate可以使用Isolate.spawn方法获取返回的一个新的isolate对象，两个isolate之间使用SendPort相互发送消息，而isolate中也存在了一个与之对应的ReceivePort接受消息用来处理，但是我们需要注意的是，ReceivePort和SendPort在每个isolate都有一对，只有同一个isolate中的ReceivePort才能接受到当前类的SendPort发送的消息并且处理。

Dart异步编程中的 Future关键字?

- Dart 在单线程中是以消息循环机制来运行的，其中包含两个任务队列，一个是“微任务队列” microtask queue，另一个叫做“事件队列” event queue。 在Java并发编程开发中，经常会使用Future来处理异步或者延迟处理任务等操作。而在Dart中，执行一个异步任务同样也可以使用Future来处理。在 Dart 的每一个 Isolate 当中，执行的优先级为： Main > MicroTask > EventQueue。

mixin机制

- mixin 是Dart 2.1 加入的特性，以前版本通常使用abstract class代替。简单来说，mixin是为了解决继承方面的问题而引入的机制，Dart为了支持多重继承，引入了mixin关键字，它最大的特殊处在于： mixin定义的类不能有构造方法，这样可以避免继承多个类而产生的父类构造方法冲突。 mixins的对象是类，mixins绝不是继承，也不是接口，而是一种全新的特性，可以mixins多个类，mixins的使用需要满足一定条件。

介绍下Flutter的FrameWork层和Engine层，以及它们的作用

- Flutter的FrameWork层是用Dart编写的框架（SDK），它实现了一套基础库，包含Material（Android风格UI）和Cupertino（iOS风格）的UI界面，下面是通用的Widgets（组件），之后是一些动画、绘制、渲染、手势库等。这个纯 Dart实现的 SDK被封装为了一个叫作 dart:ui的 Dart库。我们在使用 Flutter写 App的时候，直接导入这个库即可使用组件等功能。
- Flutter的Engine层是Skia 2D的绘图引擎库，其前身是一个向量绘图软件，Chrome和 Android均采用 Skia作为绘图引擎。Skia提供了非常友好的 API，并且在图形转换、文字渲染、位图渲染方面都提供了友好、高效的表现。Skia是跨平台的，所以可以被嵌入到 Flutter的 iOS SDK中，而不用去研究 iOS闭源的 Core Graphics / Core Animation。Android自带了 Skia，所以 Flutter Android SDK要比 iOS SDK小很多。

简述Flutter的线程管理模型

- Flutter Engine层会创建一个Isolate，并且Dart代码默认就运行在这个主Isolate上。必要时可以使用spawnUri和spawn两种方式来创建新的Isolate，在Flutter中，新创建的Isolate由Flutter进行统一的管理。 事实上，Flutter Engine自己不创建和管理线程，Flutter Engine线程的创建和管理是Embeder负责的，Embeder指的是将引擎移植到平台的中间层代码，Flutter Engine层的架构示意图如下图所示。 在Flutter的架构中，Embeder提供四个Task Runner，分别是Platform Task Runner、UI Task Runner Thread、GPU Task Runner和IO Task Runner，每个Task Runner负责不同的任务，Flutter Engine不在乎Task Runner运行在哪个线程，但是它需要线程在整个生命周期里面保持稳定。

Future和Isolate有什么区别?

- future是异步编程，调用本身立即返回，并在稍后的某个时候执行完成时再获得返回结果。在普通代码中可以使用await 等待一个异步调用结束。 isolate是并发编程，Dartm有并发时的共享状态，所有Dart代码都在isolate中运行，包括最初的main()。每个isolate都有它自己的堆内存，意味着其中所有内存数据，包括全局数据，都仅对该isolate可见，它们之间的通信只能通过传递消息的机制完成，消息则通过端口(port)收发。isolate只是一个概念，具体取决于如何实现，比如在Dart VM中一个isolate可能会是一个线程，在Web中可能会是一个Web Worker。

Navigator? MaterialApp做了什么?

- Navigator是在Flutter中负责管理维护页面堆栈的导航器

- MaterialApp在需要的时候，会自动为我们创建Navigator。Navigator.of(context)，会使用context来向上遍历Element树，找到MaterialApp提供的_NavigatorState再调用其push/pop方法完成导航操作。

Flutter的理念架构

- Flutter框架自下而上分为Embedder、Engine和Framework三层。其中，Embedder是操作系统适配层，实现了渲染 Surface设置，线程设置，以及平台插件等平台相关特性的适配；Engine层负责图形绘制、文字排版和提供Dart运行时，Engine层具有独立虚拟机，正是由于它的存在，Flutter程序才能运行在不同的平台上，实现跨平台运行；Framework层则是使用Dart编写的一套基础视图库，包含了动画、图形绘制和手势识别等功能，是使用频率最高的一层。

React Native

- React Native是一个JavaScript框架，由Facebook开发，以满足日益增长的移动应用开发的需求。它是开源的，基于JavaScript的。它被设计为用可重复使用的组件构建本地移动应用程序。它使用了大量的ReactJS组件，但在不同的设备上以原生方式实现它们。它调用Objective-C（用于iOS）和Java（用于Android）中的本地渲染API。
- ReactJS也是由Facebook开发的。它是一个开源的JavaScript库，用于为移动和网络应用开发响应式的用户界面。它有一个可重复使用的组件库，旨在帮助开发者为他们的应用程序建立基础。

React Native与ReactJS有什么不同?

- 主要区别。 - 语法。React Native和ReactJS都使用JSX，但ReactJS使用HTML标签，而React Native不使用。 - 导航。React Native使用自己的内置导航库，而ReactJS使用react-router。 - 动画。ReactJS使用CSS动画。React Native使用其动画API。 - DOM。ReactJS使用部分刷新的虚拟DOM。React Native在渲染UI组件时需要使用其本地API。 - 用法。ReactJS主要用于Web应用开发，而React Native则专注于移动应用。

什么是JSX?

- JavaScript XML，或JSX，是React使用的一种XML/HTML模板语法。它扩展了ECMAScript，允许XML/HTML类文本与JavaScript和React代码重合。它允许我们把HTML放到JavaScript中。

什么是核心React组件，它们做什么?

- Props。你可以使用props来传递数据给不同的React组件。Props是不可变的，这意味着props不能改变它们的值。
- ScrollView。ScrollView是一个滚动的容器，用来承载多个视图。你可以用它来渲染大型列表或内容。
- 状态。你使用状态来控制组件。在React中，状态是可变的，这意味着它可以在任何时候改变值。
- 风格。React Native不需要任何特殊的语法来进行样式设计。它使用JavaScript对象。
- 文本。文本组件在你的应用程序中显示文本。它使用textInput来接受用户的输入。
- 视图。视图用于构建移动应用程序的用户界面。它是一个你可以显示你的内容的地方。

什么是Redux，什么时候应该使用它?

- Redux是一个JavaScript应用程序的状态管理工具。它可以帮助你编写一致的应用程序，可以在不同环境下运行的应用程序，以及易于测试的应用程序。
- 不是所有的应用程序都需要Redux。它的设计是为了帮助你确定何时出现状态变化。根据Redux的官方文档，以下是一些你想使用Redux的例子。
  - 你的应用状态经常更新
  - 你有大量的应用状态，并且在应用的许多地方都需要它
  - 更新你的应用状态的逻辑很复杂
  - 你希望看到状态是如何随时间更新的
  - 你的应用程序有一个中等或较大的代码库，并将由多个人员进行操作

什么是状态，如何使用它?

- 在React Native中，状态处理的是可改变的数据。状态是可变的，意味着它可以在任何时候改变值。你应该在构造函数中初始化它，然后在你想改变它时调用setState。

如何调试React应用程序，你可以使用哪些工具?

开发者菜单

- 重新加载：重新加载应用程序
- Debug JS Remotely：打开一个JavaScript调试器
- 启用实时重载：导致应用程序在选择 "保存 "后自动重新加载
- 启用热重新加载：观察变化
- 切换检查器：切换检查器界面，以便我们可以检查UI元素和它们的属性
- 显示Perf Monitor：监控性能

Chrome开发工具

- 你可以使用这些DevTools来调试React Native应用程序。你需要确保它连接到同一个WiFi。如果你使用的是Windows或Linux，按 Ctrl + M+，如果你使用的是macOS，按 命令+R.在开发者菜单中，你选择 "Debug JS Remotely"，它将打开默认调试器。

React开发工具

- 要使用React的开发者工具，你必须使用桌面应用程序。这些工具允许你调试React组件和样式。

React本地调试器

- 如果你在你的React应用中使用Redux，这对你来说是一个好的调试器。它是一个桌面应用，在一个应用中整合了Redux的和React的开发者工具。

React Native CLI

- 使用React Native命令行界面来进行调试。

当你调用SetState时会发生什么?

- 当你在React中调用SetState时，你传递给它的对象将被合并到组件的当前状态中。这就触发了一种叫做*调和的*东西。调和的目的是以最有效的方式更新用户界面。 React通过构建一个React元素树，并将其与之前的元素树进行比较来实现这一目的。这向React显示了所发生的确切变化，因此React可以在必要的地方进行更新。

描述一下虚拟DOM是如何工作的。

- 在React Native中，虚拟DOM是真实DOM的一个副本。它是一个节点树，列出了元素以及它们的属性、内容、和属性。每当我们的底层数据发生变化时，虚拟DOM会重新渲染用户界面。之后，其他DOM表现和虚拟DOM表现之间的差异将被计算出来，而真实DOM将被更新。

描述Flexbox以及它最常用的属性。

- Flexbox是一种布局模式，使元素能够在容器内协调和分配空间。它在不同的屏幕尺寸上提供了一个一致的布局。
- flexDirection：用于指定元素的对齐方式（垂直或水平）。
- justifyContent：用于决定元素在一个给定的容器内应该如何分布
- alignItems：用于指定一个给定的容器内的元素沿次轴的分布。

功能性组件和类组件的区别是什么?

- 功能性组件也被称为无状态组件。功能性组件接受道具并返回HTML。它们在不使用状态的情况下给出解决方案，它们可以定义为有或没有箭头函数。
- 类组件也被称为有状态组件。它们是ES6类，扩展了React库中的组件类。它们实现了逻辑和状态。类组件在返回HTML时需要有一个render()方法。你可以向它们传递道具，并通过this.props访问它们。

列出一些你可以优化应用程序的方法。

- 压缩或转换我们的原始JSON数据，而不是仅仅存储它
- 为CPU架构制作缩小尺寸的APK文件
- 优化本地库和状态操作的数量
- 在列表项上使用关键属性
- 压缩图片和其他图形元素
- 使用Proguard来最小化应用程序的大小，并剥离我们的字节码及其依赖的部分。

在iOS和Android中，内存泄露的一些原因是什么，如何检测它们?

- 如果在componentDidMount中添加了未发布的定时器或监听器，或者关闭范围泄漏，就会发生内存泄漏。
- 要检测iOS的内存泄漏，你可以进入Xcode，产品，然后是配置文件。
- 要检测Android的内存泄漏，你可以使用性能监视器。