一、对象内存模型

- isa指针的作用
- 对象的isa指向类对象，类对象的isa指向元类，元类的isa指向根元类，根元类的isa指向自己。
- 类对象的 `superClass` 指针指向父类对象，直到指向根类对象，根类对象的 `superClass` 指向 `nil` ，元类也如此，直到根元类，根元类的 `superClass` 指向根类对象
- 对象的内存分布
  - 对象的变量表
  - 对象大小
  - 对象方法表
  - cache?
  - 协议
  - char *name?

```
struct objc_class {
    Class isa  OBJC_ISA_AVAILABILITY;

#if !__OBJC2__
    Class super_class                                        OBJC2_UNAVAILABLE;
    const char *name                                         OBJC2_UNAVAILABLE;
    long version                                             OBJC2_UNAVAILABLE;
    long info                                                OBJC2_UNAVAILABLE;
    long instance_size                                       OBJC2_UNAVAILABLE;
    struct objc_ivar_list *ivars                             OBJC2_UNAVAILABLE;
    struct objc_method_list **methodLists                    OBJC2_UNAVAILABLE;
    struct objc_cache *cache                                 OBJC2_UNAVAILABLE;
    struct objc_protocol_list *protocols                     OBJC2_UNAVAILABLE;
#endif

} OBJC2_UNAVAILABLE;
/* Use `Class` instead of `struct objc_class *` */
```

Runtime

reference:objc runtime

1. 消息接收
   - 编译时间确定接受到的消息，运行时间通过 `@selector` 找到对应的方法。
   - 消息接受者如果能直接找到 `@selector` 则直接执行方法，否则转发消息。若最终找不到，则运行时崩溃。
2. 术语
   - `SEL`
     - 方法名的C字符串，用于辨别不同的方法。
     - 用于传递给 `objc_msgSend(self, SEL)` 函数
   - `Class`
     - `Class` 是指向类对象的指针，继承于 `objc_object`
   - `categoty`
     - `categoty` 是结构体` `categoty` *t 的指针：* *typedef struct* `categoty` t * `categoty` ;`
     - `categoty` 是在 `app` 启动时加载镜像文件，通过 `read_imgs` 函数向类对象中的 `class_rw_t` 中的某些分组中添加指针，完成 `categoty` 的属性添加
   - `Method`
     - 方法包含以下
       - `IMP` :函数指针，方法的具体实现
       - `types` : `char*` ，函数的参数类型，返回值等信息
       - `SEL` :函数名
3. 消息转发
   - `objc_msgSend()` 函数并不返回数据，而是它转发消息后，调用了相关的方法返回了数据。
   - 整个流程:
     1. 检测这个 `selector` 是不是要忽略的。比如 `Mac OS X` 开发，有了垃圾回收就不理会 `retain` , `release` 这些函数了。
     2. 检测这个 `target` 是不是 `nil` 对象。 `ObjC` 的特性是允许对一个 `nil` 对象执行任何一个方法不会 Crash，因为会被忽略掉。
     3. 如果上面两个都过了，那就开始查找这个类的 `IMP` ，先从 `cache` 里面找，完了找得到就跳到对应的函数去执行。

4. 如果 `cache` 找不到就找一下方法分发表。
5. 如果分发表找不到就到超类的分发表去找，一直找，直到找到 `NSObject` 类为止。
6. 进入动态解析: `resolveInstanceMethod:` 和 `resolveClassMethod:` 方法
7. 若上一步返回 `NO` ,进入重定向： `- (id)forwardingTargetForSelector:(SEL)aSelector` 和 `+ (id)forwardingTargetForSelector:(SEL)aSelector`
8. 若上一步返回的对象或者类对象仍然没能处理消息或者返回 `NO` ，进入消息转发流程： `forwardInvocation`

- 重定向和消息转发都可以用于实现**多继承**

4. `Objective-C Associated Objects` 关联对象
   - 在 `OS X 10.6` 之后， `Runtime` 系统让 `Objc` 支持向对象动态添加变量。涉及到的函数有以下三个：

```
void objc_setAssociatedObject ( id object, const void *key, id value, objc_AssociationPolicy policy );
id objc_getAssociatedObject ( id object, const void *key );
void objc_removeAssociatedObjects ( id object );
```

   - 这些方法以键值对的形式动态地向对象添加、获取或删除关联值。其中关联政策是一组枚举常量：

```
enum {
   OBJC_ASSOCIATION_ASSIGN  = 0,
   OBJC_ASSOCIATION_RETAIN_NONATOMIC  = 1,
   OBJC_ASSOCIATION_COPY_NONATOMIC  = 3,
   OBJC_ASSOCIATION_RETAIN  = 01401,
   OBJC_ASSOCIATION_COPY  = 01403
};
```

5. `Method Swizzling` 方法混淆
   - 作用是修改 `SEL` 对应的 `IMP` 指针
   - 用于 `debug` ，避免数组越界等问题.

Categoty

reference：美团技术博客：深入理解Objective-C：Category

categoty 简介

- `categoty` 开头动态地给类添加属性及方法，通过这个特性，使得 `objc` 可以模拟多继承。也可以把一个类拆分到各个文件中，使用方可以按需加载。也可以多人协作共同完成一个类。

categoty 类比 extension

- `extension` 看起来很像一个匿名的 `categoty` ，但是 `extension` 和有名字的 `categoty` 几乎完全是两个东西。 `extension` 在编译期决议，它就是类的一部分，在编译期和头文件里的 `@interface` 以及实现文件里的 `@implement` 一起形成一个完整的类，它伴随类的产生而产生，亦随之一起消亡。 `extension` 一般用来隐藏类的私有信息，你必须有一个类的源码才能为一个类添加 `extension` ，**所以你无法为系统的类比如 `NSString` 添加 `extension`** 。
- 但是 `categoty` 则完全不一样，**它是在运行期决议的**。就 `categoty` 和 `extension` 的区别来看，我们可以推导出一个明显的事实， `extension` 可以添加实例变量，而 `categoty` 是无法添加实例变量的（因为在运行期，对象的内存布局已经确定，如果添加实例变量就会破坏类的内部布局，这对编译型语言来说是灾难性的）。

categoty 内存结构

```
typedef struct `categoty`_t {
    const char *name;
    classref_t cls;
    struct method_list_t *instanceMethods;
    struct method_list_t *classMethods;
    struct protocol_list_t *protocols;
    struct property_list_t *instanceProperties;
} `categoty`_t;
```

我们知道，所有的 `OC` 类和对象，在 `runtime` 层都是用 `struct` 表示的， `categoty` 也不例外，在 `runtime` 层， `categoty` 用结构体 `categoty_t` （在 `objc-runtime-new.h` 中可以找到此定义），它包含了

1. 类的名字（name）
2. 类（cls）
3. `category` 中所有给类添加的实例方法的列表（ `instanceMethods` ）
4. `category` 中所有添加的类方法的列表（ `classMethods` ）
5. `category` 实现的所有协议的列表（ `protocols` ）
6. `category` 中添加的所有属性（ `instanceProperties` ）

category 的加载

`category` 是在 `runtime` 进入入口时被添加进类的， `runtime` 通过遍历编译后产生的 `DATA` 端上的 `category_t` 数组动态地将内容添加到类上。

需要注意的时， `category` 中的方法其实并没有覆盖原类中的同名方法，只是单纯地添加到方法列表中而已。但是由于 `objc` 在方法调用中查找到类对象中的方法列表时，只要找到了第一个对应消息中的选择子的方法，就认为是找到了对应的方法并调用。所以形成了 `category` 中的方法会覆盖原类中的同名方法的假象。

根据这个原理，我们可以得知，如果有多个类别中同时都复写了类中的某个方法，那么最终调用的是最后被加载的 `category` ，也就是最后被编译的 `category` 文件。

category 中 Load 方法

`category` 中也可以复写原类中的 `+Load` 方法，因为类的 `Load` 方法永远发生在 `category` 的 `Load` 方法之前。如果多个 `category` 都复写了 `Load` 方法，则 `category` 的 `Load` 执行顺序取决于对应 `category` 文件的编译顺序。

category 与关联对象

由于 `category` 中无法动态添加实例变量，所以可以通过关联对象的手段为对象增加实例变量:

```
#import "MyClass+Category1.h"
#import <objc/runtime.h>

@implementation MyClass (Category1)

+ (void)load
{
    NSLog(@"%@",@"load in Category1");
}

- (void)setName:(NSString *)name
{
    objc_setAssociatedObject(self,
                             "name",
                             name,
                             OBJC_ASSOCIATION_COPY);
}

- (NSString*)name
{
    NSString *nameObject = objc_getAssociatedObject(self, "name");
    return nameObject;
}

@end
```

关联对象存储在一张全局的 `map` 里，其中 `map` 的 `key` 是被关联的对象的指针地址，该 `map` 的 `value` 存储的是另外一张 `map` ，此处称为 `AssociationsHashMap` ，该 `AssociationsHashMap` 的kv分别是设置关联对象时的kv。

KVO

- 原理:
  1. `isa swizzling` 方法，通过 `runtime` 动态创建一个中间类，继承自被监听的类。
  2. 使原有类的 `isa` 指针指向这个中间类，同时重写改类的 `Class` 方法，使得该类的 `Class` 方法返回自己而不是 `isa` 的指向。
  3. 复写中间类的对应被监听属性的 `setter` 方法，调用添加进来的方法，然后给当前中间类的父类也就是原类的发送 `setter` 消息。
- 自定义KVO:

  1. 通过给 `NSObject` 添加分类的方法，添加新的对象方法:

```
#import <Foundation/Foundation.h>
@interface NSObject (ACoolCustom_KVO)
/**
 *    自定义添加观察者
 *
 *    @param oberserver 观察者
 *    @param keyPath    要观察的属性
 *    @param block      自定义回调
 */
- (void)zl_addOberver:(id)oberserver
          forKeyPath:(NSString *)keyPath
               block:(void(^)(id oberveredObject,NSString *keyPath,id newValue,id oldValue))block;
```

  1. 动态创建中间类， 更改原有类的 `isa` 指向，同时重写中间类的 `Class` 方法:

```
Class originalClass = object_getClass(self);
//创建中间类 并使其继承被监听的类
Class kvoClass = objc_allocateClassPair(originalClass, kvoClassName.UTF8String, 0);
//向runtime动态注册类
objc_registerClassPair(kvoClass);
object_setClass(self, kvoClass);
//替换被监听对象的class方法...
//原始类的class方法的实现
//原始类的class方法的参数等信息
Method clazzMethod = class_getInstanceMethod(originalClass, @selector(class));
const char *types = method_getTypeEncoding(clazzMethod);
class_addMethod(kvoClass, @selector(class), (IMP)new_class, types);
```

1. 利用 `runtime` 给对象动态增加关联属性保存外部传进来的回调 `block`：

```
objc_setAssociatedObject(self, kObjectPropertyKey ,block, OBJC_ASSOCIATION_RETAIN_NONATOMIC);
```

1. 利用 `runtime` 替换中间类的 `setter` 方法，在新的方法中，调用 `block` 后再向父类发送原消息:

```
// 利用函数指针强制转换
void (*objc_msgSendSuperCasted)(void *, SEL, id) = (void *)objc_msgSendSuper;
// 给父类 发送原消息
objc_msgSendSuperCasted(&superclass, _cmd, newValue);
// 调用block
ZLObservingBlock block = objc_getAssociatedObject(self, kObjectPropertyKey);
dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0), ^{
    block(self, nil, newValue);
});
```

App 启动

Load 与 Initialize

Load

`+Load` 方法的含义是将类对象或者 `category` 加载到内存中，这个过程发生在App启动时，在所有类都被注册后。在Category详解中也有所提及。

`+Load` 方法是自顶而下的，也就是说会从类的根类由上而下执行。所以如果重写 `+Load` 方法，是不需要调用 `super` 的。

值得第一提的是， `Category` 中的 `+Load` 方法会在类的 `+Load` 方法之后被调用，所以 `category` 的 `+Load` 方法不会覆盖本类的 `+Load` 方法。而且由于 `Category` 的加载顺序取决编译顺序，所以一个类下不同的 `Category` 的 `+Load` 方法都会被调用，而且调用顺序和编译顺序是一致的。

Initialize

与 `Load` 不同的是， `Initialize` 方法不会在一开始就会被调用，而是在类收到第一条消息时才会被调用。

值得注意的点是：类初始化的时候每个类只会调用一次 `+initialize` ，如果子类没有实现 `initialize` ，那么将会调用父类的 `+initialize` ，也就是意味着父类的 `+initialize` 可能会被多次调用。这与 `Load` 方法截然不同。

无论是重写 `+Load` 还是 `+Inilialize` 方法，都不需要调用 `superClass` 。 通过查看这两个方法的源码，还可以得知 `+Load` 方法是直接取函数指针调用，不走消息流程，而 `+inilialize` 是走消息流程的。也就是说 `Category` 里的 `Initialize` 是会覆盖类的 `initialize` 方法。

Block

Block 基本原理

```
int main() {
    void (^blk)(void) = ^{
        (printf("hello world!"));
    };
    blk();
    return 0;
}
```

如上，通过 `clang` 转换为 `cpp` 源码，截取关键部分:

```
// block 的真面目
struct __block_impl {
  void *isa;
  int Flags;
  int Reserved;
  void *FuncPtr;
};

// ..... 一大坨无关代码

// 通过这个结构体包装block，用于快速构造block
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int flags=0) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {

        (printf("hello world!"));
    }

static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
int main() {
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA));
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}
```

比较核心的是以下内容: `__block_impl` 结构体， `__main_block_impl_0` 结构体， `__main_block_func_0` 函数， `&_NSConcreteStackBlock` ， `__main_block_desc_0` 结构体。

- `__block_impl` 结构体:

  该结构体定义在源文件上方，其中 `isa` 指针指向当前 `block` 对象类对象， `FuncPtr` 指向 `block` 保存的函数指针。

- `__main_block_desc_0` 结构体:
  - 用于保存 `__main_block_impl_0` 结构体大小等信息。
- `__main_block_func_0` 静态函数:
  - 用于存储当前 `block` 的代码块。
- `__main_block_impl_0` 结构体:

  该结构体包装了 `__block_impl` 结构体，同时包含 `__main_block_desc_0` 结构体。对外提供一个构造函数，构造函数需要传递函数指针(`__main_block_func_0` 静态函数)、 `__main_block_desc_0` 实例。

由此我们可知，上述一个简单的 `block` 定义及调用过程被转换为了:

1. 定义 `block` 变量相当于调用 `__main_block_impl_0` 构造函数，通过函数指针传递代码块进 `__main_block_impl_0` 实例。
2. 构造函数内部，将外部传递进来的 `__main_block_func_0` 函数指针，设置内部实际的 `block` 变量(`__block_impl` 类型的结构体)的函数指针。
3. 调用 `block` 时，取出 `__main_block_impl_0` 类型结构体中的 `__block_impl` 类型的结构体的函数指针(`__main_block_func_0`)并调用。

至此，一个简单的 `block` 原理描述完毕。

__block 捕获原理

`block` 内部可以直接使用外部变量，但是在不加 `__block` 修饰符的情况下，是无法修改的。比如下面这段代码:

```
int main() {
    int a = 1;
    void (^blk)(void) = ^{
        printf("%d", a);
    };
    blk();
    return 0;
}
```

经过 `cpp` 重写后:

```
struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  int a;
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, int _a, int flags=0) : a(_a) {
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
  int a = __cself->a; // bound by copy

        printf("%d", a);
    }

static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0)};
int main() {
    int a = 1;
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, a));
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}
```

经过上一节的讨论，我们知道代码块是由 `__main_block_impl_0` 构造函数传通过函数指针传递进来的。在这节的例子中，可以发现构造函数中多了一个 `int _a` 参数。同时 `__main_block_impl_0` 结构体也多了一个 `int a` 属性用于保存 `block` 内部使用的变量 `a` 。

由于构造函数的参数为整形，在 `c++` 中，函数的形参为值拷贝，也就是说 `__main_block_impl_0` 结构体中的属性 `a` ，是外部 `a` 变量的拷贝。在代码块内部(也就是 `__main_block_func_0` 函数)我们通过 `__cself` 指针拿到 `a` 变量。故我们在代码块中是无法修改 `a` 变量的，同时如果外部 `a` 变量被修改了，那么 `block` 内部也是无法得知的。

如果想要在内部修改 `a` 变量，可以通过 `__block` 关键字:

```
int main() {
    __block int number = 1;
    void (^blk)(void) = ^{
        number = 3;
        printf("%d", number);
    };
    blk();
    return 0;
}
```

```
struct __Block_byref_number_0 {
  void *__isa;
__Block_byref_number_0 *__forwarding;
 int __flags;
 int __size;
 int number;
};

struct __main_block_impl_0 {
  struct __block_impl impl;
  struct __main_block_desc_0* Desc;
  __Block_byref_number_0 *number; // by ref
  __main_block_impl_0(void *fp, struct __main_block_desc_0 *desc, __Block_byref_number_0 *_number, int flags=0) : number(_number->__forwardin
    impl.isa = &_NSConcreteStackBlock;
    impl.Flags = flags;
    impl.FuncPtr = fp;
    Desc = desc;
  }
};
static void __main_block_func_0(struct __main_block_impl_0 *__cself) {
  __Block_byref_number_0 *number = __cself->number; // bound by ref

        (number->__forwarding->number) = 3;
        printf("%d", (number->__forwarding->number));
    }
static void __main_block_copy_0(struct __main_block_impl_0*dst, struct __main_block_impl_0*src) {_Block_object_assign((void*)&dst->number,
(void*)src->number, 8/*BLOCK_FIELD_IS_BYREF*/);}

static void __main_block_dispose_0(struct __main_block_impl_0*src) {_Block_object_dispose((void*)src->number, 8/*BLOCK_FIELD_IS_BYREF*/);}

static struct __main_block_desc_0 {
  size_t reserved;
  size_t Block_size;
  void (*copy)(struct __main_block_impl_0*, struct __main_block_impl_0*);
  void (*dispose)(struct __main_block_impl_0*);
} __main_block_desc_0_DATA = { 0, sizeof(struct __main_block_impl_0), __main_block_copy_0, __main_block_dispose_0};
int main() {
    __attribute__((__blocks__(byref))) __Block_byref_number_0 number = {(void*)0,(__Block_byref_number_0 *)&number, 0, sizeof(__Block_byref_n
    void (*blk)(void) = ((void (*)())&__main_block_impl_0((void *)__main_block_func_0, &__main_block_desc_0_DATA, (__Block_byref_number_0 *)&
    ((void (*)(__block_impl *))((__block_impl *)blk)->FuncPtr)((__block_impl *)blk);
    return 0;
}
```

可以看到整形 `number` 变量变成了 `__Block_byref_number_0` 结构体实例，在 `__main_block_impl_0` 构造函数中，将该实例的指针传递进来。 `__Block_byref_number_0` 结构体中，通过 `__forwarding` 指向自己，整形 `number` 存储具体的值。

上节提到，构造函数中的参数是值拷贝，故此处代码块内部拿到的指针拷贝一样可以操作外部的 `__Block_byref_number_0` 结构体中的值，通过这种方式实现了 `block` 内部修改外部值。

block 引起的循环引用原理

RunLoop

reference:深入理解Runloop

Runloop 基本原理

一般来讲，一个线程一次只能执行一个任务，执行完成后线程就会退出。如果我们需要一个机制，让线程能随时处理事件但并不退出，通常的代码逻辑是这样的：

```
function loop() {
    initialize();
    do {
        var message = get_next_message();
        process_message(message);
    } while (message != quit);
}
```

在 `iOS` 中， `Runloop` 与线程是一一对应的关系，这种关系存在一张全局字典里：

```
/// 全局的Dictionary，key 是 pthread_t， value 是 CFRunLoopRef
static CFMutableDictionaryRef loopsDic;
/// 访问 loopsDic 时的锁
static CFSpinLock_t loopsLock;

/// 获取一个 pthread 对应的 RunLoop。
CFRunLoopRef _CFRunLoopGet(pthread_t thread) {
    OSSpinLockLock(&loopsLock);

    if (!loopsDic) {
        // 第一次进入时，初始化全局Dic，并先为主线程创建一个 RunLoop。
        loopsDic = CFDictionaryCreateMutable();
        CFRunLoopRef mainLoop = _CFRunLoopCreate();
        CFDictionarySetValue(loopsDic, pthread_main_thread_np(), mainLoop);
    }

    /// 直接从 Dictionary 里获取。
    CFRunLoopRef loop = CFDictionaryGetValue(loopsDic, thread));

    if (!loop) {
        /// 取不到时，创建一个
        loop = _CFRunLoopCreate();
        CFDictionarySetValue(loopsDic, thread, loop);
        /// 注册一个回调，当线程销毁时，顺便也销毁其对应的 RunLoop。
        _CFSetTSD(..., thread, loop, __CFFinalizeRunLoop);
    }

    OSSpinLockUnLock(&loopsLock);
    return loop;
}

CFRunLoopRef CFRunLoopGetMain() {
    return _CFRunLoopGet(pthread_main_thread_np());
}

CFRunLoopRef CFRunLoopGetCurrent() {
    return _CFRunLoopGet(pthread_self());
}
```

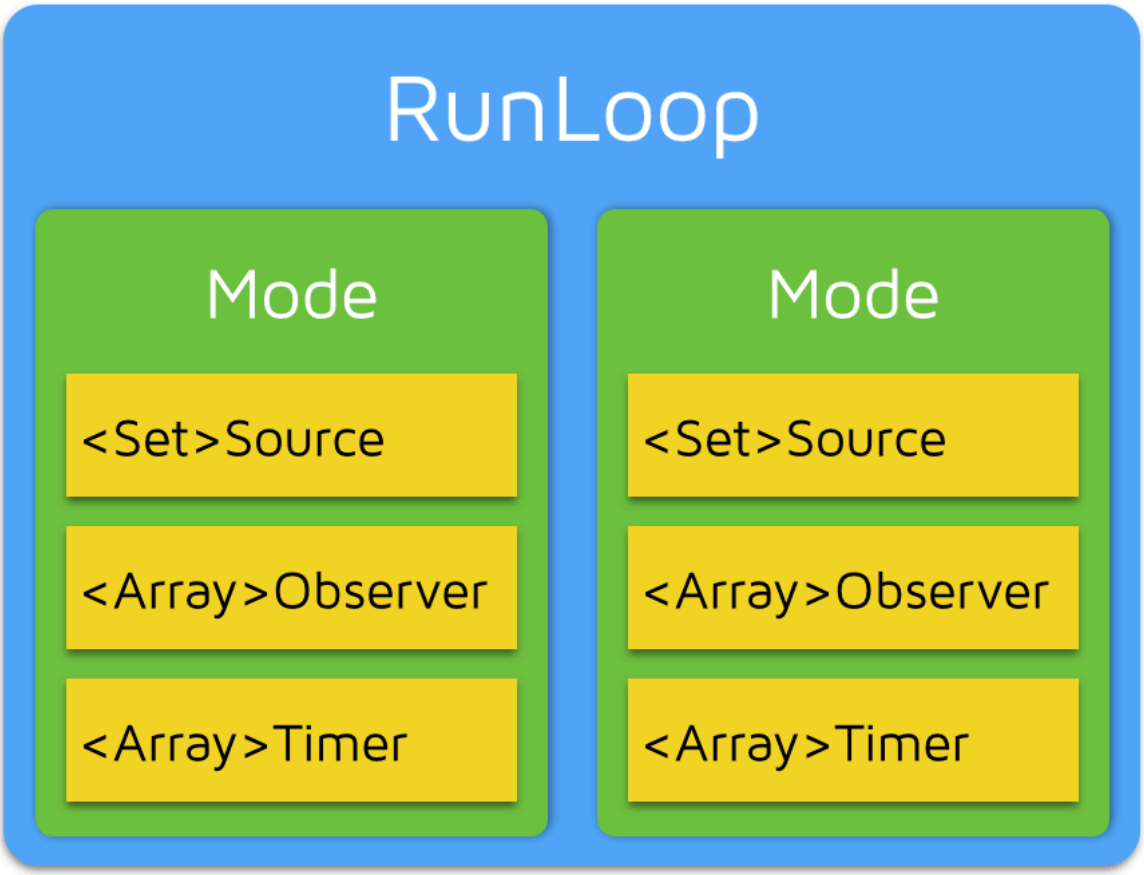
苹果是不允许直接创建 `Runloop` 的，只能通过:

```
CFRunLoopRef CFRunLoopGetMain() {
    return _CFRunLoopGet(pthread_main_thread_np());
}

CFRunLoopRef CFRunLoopGetCurrent() {
    return _CFRunLoopGet(pthread_self());
}
```

这两个函数来获得当前或者某个线程的 `Runloop` 。根据上面的源码可以得出结论：

> `Runloop` 创建是发生在第一次获取之前，若从未获取，则永远不会创建。 `Runloop` 销毁是在线程结束前，除主线程外，所有 `Runloop` 只能在当前线程中获取，主线程的 `Runloop` 可以在任意线程中获取。

Models in Runloop



一个 `RunLoop` 包含若干个 `Mode`，每个 `Mode` 又包含若干个 `Source/Timer/Observer`。每次调用 `RunLoop` 的主函数时，只能指定其中一个 `Mode`，这个 `Mode` 被称作 `CurrentMode`。如果需要切换 `Mode`，只能退出 `Loop`，再重新指定一个 `Mode` 进入。这样做主要是为了分隔开不同组的 `Source/Timer/Observer`，让其互不影响。

1. Source:
    - `Souce` 是事件发生的地方。
    - `Souce` 分为 `Source0` 与 `Source1`， `Source0` 只包含一个函数指针做为回调，且不能主动唤醒 `Runloop`，使用时需要先手动将其标为待处理，然后手动唤醒 `Runloop`； `Source1` 也包含函数指针，同时还包含一个 `mach_por`。它可以主动唤醒 `Runloop`，用于内核与其他线程发送消息。
2. Timer
    - `Timer` 是基于时间的触发器，它和 `NSTimer` 是 `toll-free bridged` 的，可以混用。其包含一个时间长度和一个回调（函数指针）。当其加入到 `RunLoop` 时， `RunLoop` 会注册对应的时间点，当时间点到时， `RunLoop` 会被唤醒以执行那个回调。
3. Observer
    - `Obserer` 是观察者，用于观察 `Runloop` 各种状态。它也包含函数指针，用于状态改变时回调通知外部。

```
typedef CF_OPTIONS(CFOptionFlags, CFRunLoopActivity) {
    kCFRunLoopEntry         = (1UL << 0), // 即将进入Loop
    kCFRunLoopBeforeTimers  = (1UL << 1), // 即将处理 Timer
    kCFRunLoopBeforeSources = (1UL << 2), // 即将处理 Source
    kCFRunLoopBeforeWaiting = (1UL << 5), // 即将进入休眠
    kCFRunLoopAfterWaiting  = (1UL << 6), // 刚从休眠中唤醒
    kCFRunLoopExit          = (1UL << 7), // 即将退出Loop
};
```

上面的 `Source/Timer/Observer` 被统称为 `mode item`，一个 `item` 可以被同时加入多个 `mode`。但一个 `item` 被重复加入同一个 `mode` 时是不会有效果的。如果一个 `mode` 中一个 `item` 都没有，则 `RunLoop` 会直接退出，不进入循环。

`CFRunLoopMode` 和 `CFRunLoop` 的结构大致如下：

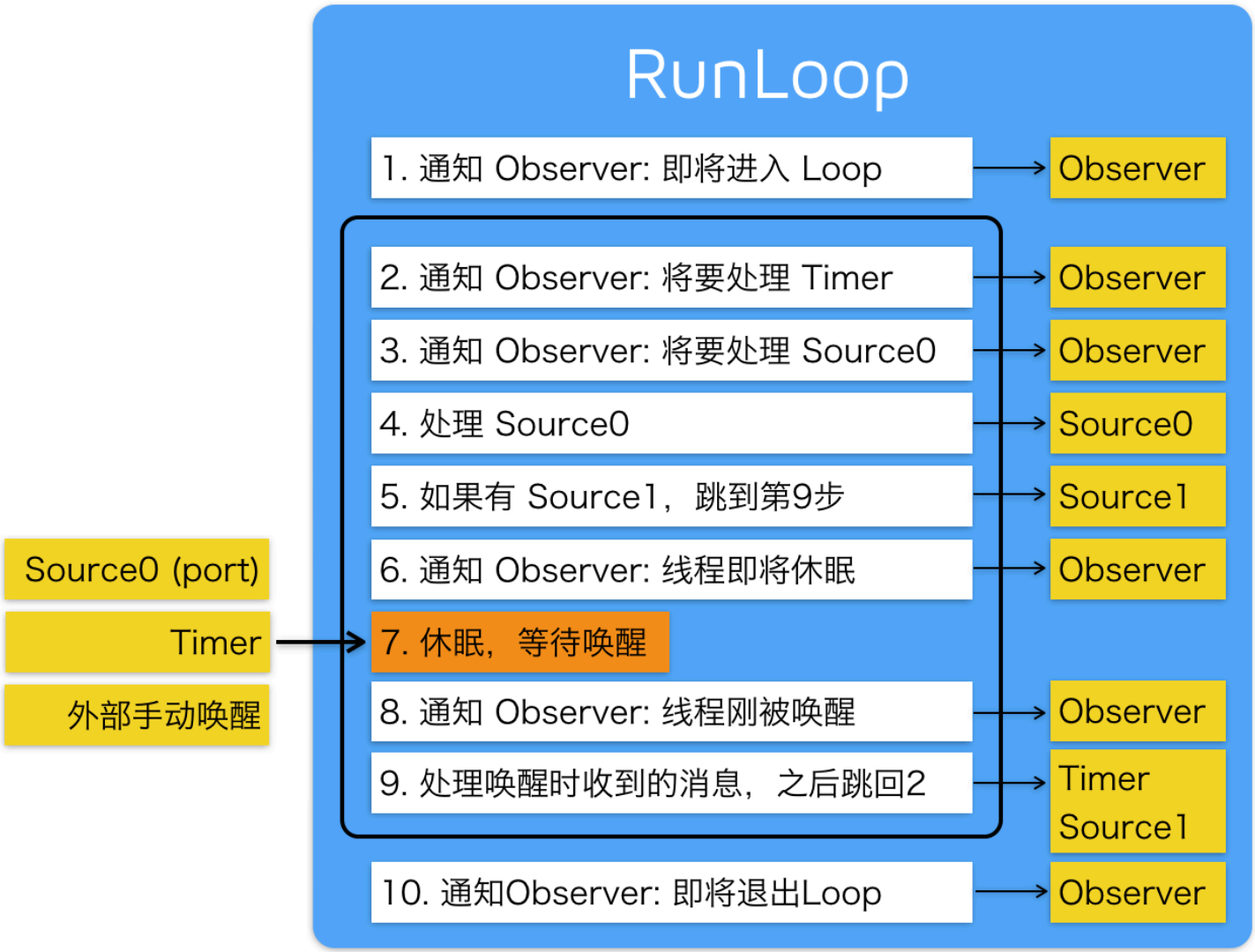
```
struct __CFRunLoopMode {
    CFStringRef _name;            // Mode Name, 例如 @"kCFRunLoopDefaultMode"
    CFMutableSetRef _sources0;    // Set
    CFMutableSetRef _sources1;    // Set
    CFMutableArrayRef _observers; // Array
    CFMutableArrayRef _timers;    // Array
    ...
};

struct __CFRunLoop {
    CFMutableSetRef _commonModes;     // Set
    CFMutableSetRef _commonModeItems; // Set<Source/Observer/Timer>
    CFRunLoopModeRef _currentMode;    // Current Runloop Mode
    CFMutableSetRef _modes;           // Set
    ...
};
```

这里有个概念叫 “CommonModes”：一个 `Mode` 可以将自己标记为”Common”属性（通过将其 `ModeName` 添加到 `RunLoop` 的 “commonModes” 中）。每当 `RunLoop` 的内容发生变化时，`RunLoop` 都会自动将 `_commonModeItems` 里的 `Source/Observer/Timer` 同步到具有 “Common” 标记的所有Mode里。

应用场景举例：主线程的 `RunLoop` 里有两个预置的 `Mode`：`kCFRunLoopDefaultMode` 和 `UITrackingRunLoopMode`。这两个 `Mode` 都已经被标记为”Common”属性。`DefaultMode` 是 App 平时所处的状态，`TrackingRunLoopMode` 是追踪 `ScrollView` 滑动时的状态。当你创建一个 `Timer` 并加到 `DefaultMode` 时，`Timer` 会得到重复回调，但此时滑动一个 `TableView` 时，`RunLoop` 会将 `mode` 切换为 `TrackingRunLoopMode`，这时 `Timer` 就不会被回调，并且也不会影响到滑动操作。

有时你需要一个 `Timer`，在两个 `Mode` 中都能得到回调，一种办法就是将这个 `Timer` 分别加入这两个 `Mode`。还有一种方式，就是将 `Timer` 加入到顶层的 `RunLoop` 的 “commonModeItems” 中。”commonModeItems” 被 `RunLoop` 自动更新到所有具有”Common”属性的 `Mode` 里去

Runloop 处理事件的逻辑



```
/// 用DefaultMode启动
void CFRunLoopRun(void) {
    CFRunLoopRunSpecific(CFRunLoopGetCurrent(), kCFRunLoopDefaultMode, 1.0e10, false);
}

/// 用指定的Mode启动，允许设置RunLoop超时时间
int CFRunLoopRunInMode(CFStringRef modeName, CFTimeInterval seconds, Boolean stopAfterHandle) {
    return CFRunLoopRunSpecific(CFRunLoopGetCurrent(), modeName, seconds, returnAfterSourceHandled);
}

/// RunLoop的实现
int CFRunLoopRunSpecific(runloop, modeName, seconds, stopAfterHandle) {

    /// 首先根据modeName找到对应mode
    CFRunLoopModeRef currentMode = __CFRunLoopFindMode(runloop, modeName, false);
    /// 如果mode里没有source/timer/observer, 直接返回。
    if (__CFRunLoopModeIsEmpty(currentMode)) return;

    /// 1. 通知 Observers: RunLoop 即将进入 loop。
    __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopEntry);

    /// 内部函数，进入loop
    __CFRunLoopRun(runloop, currentMode, seconds, returnAfterSourceHandled) {

        Boolean sourceHandledThisLoop = NO;
        int retVal = 0;
        do {

            /// 2. 通知 Observers: RunLoop 即将触发 Timer 回调。
            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeTimers);
            /// 3. 通知 Observers: RunLoop 即将触发 Source0 (非port) 回调。
            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeSources);
            /// 执行被加入的block
            __CFRunLoopDoBlocks(runloop, currentMode);

            /// 4. RunLoop 触发 Source0 (非port) 回调。
            sourceHandledThisLoop = __CFRunLoopDoSources0(runloop, currentMode, stopAfterHandle);
            /// 执行被加入的block
            __CFRunLoopDoBlocks(runloop, currentMode);

            /// 5. 如果有 Source1 (基于port) 处于 ready 状态，直接处理这个 Source1 然后跳转去处理消息。
            if (__Source0DidDispatchPortLastTime) {
                Boolean hasMsg = __CFRunLoopServiceMachPort(dispatchPort, &msg)
                if (hasMsg) goto handle_msg;
            }

            /// 通知 Observers: RunLoop 的线程即将进入休眠(sleep)。
            if (!sourceHandledThisLoop) {
                __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopBeforeWaiting);
            }

            /// 7. 调用 mach_msg 等待接受 mach_port 的消息。线程将进入休眠, 直到被下面某一个事件唤醒。
            /// • 一个基于 port 的Source 的事件。
            /// • 一个 Timer 到时间了
            /// • RunLoop 自身的超时时间到了
            /// • 被其他什么调用者手动唤醒
            __CFRunLoopServiceMachPort(waitSet, &msg, sizeof(msg_buffer), &livePort) {
                mach_msg(msg, MACH_RCV_MSG, port); // thread wait for receive msg
            }

            /// 8. 通知 Observers: RunLoop 的线程刚刚被唤醒了。
            __CFRunLoopDoObservers(runloop, currentMode, kCFRunLoopAfterWaiting);

            /// 收到消息，处理消息。
            handle_msg:

            /// 9.1 如果一个 Timer 到时间了，触发这个Timer的回调。
            if (msg_is_timer) {
                __CFRunLoopDoTimers(runloop, currentMode, mach_absolute_time())
            }

            /// 9.2 如果有dispatch到main_queue的block，执行block。
            else if (msg_is_dispatch) {
                __CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__(msg);
            }

            /// 9.3 如果一个 Source1 (基于port) 发出事件了，处理这个事件
```

```
            else {
                CFRunLoopSourceRef source1 = __CFRunLoopModeFindSourceForMachPort(runloop, currentMode, livePort);
                sourceHandledThisLoop = __CFRunLoopDoSource1(runloop, currentMode, source1, msg);
                if (sourceHandledThisLoop) {
                    mach_msg(reply, MACH_SEND_MSG, reply);
                }
            }

            /// 执行加入到Loop的block
            __CFRunLoopDoBlocks(runloop, currentMode);


            if (sourceHandledThisLoop && stopAfterHandle) {
                /// 进入loop时参数说处理完事件就返回。
                retVal = kCFRunLoopRunHandledSource;
            } else if (timeout) {
                /// 超出传入参数标记的超时时间了
                retVal = kCFRunLoopRunTimedOut;
            } else if (__CFRunLoopIsStopped(runloop)) {
                /// 被外部调用者强制停止了
                retVal = kCFRunLoopRunStopped;
            } else if (__CFRunLoopModeIsEmpty(runloop, currentMode)) {
                /// source/timer/observer一个都没有了
                retVal = kCFRunLoopRunFinished;
            }

            /// 如果没超时，mode里没空，loop也没被停止，那继续loop。
        } while (retVal == 0);
    }

    /// 10. 通知 Observers：RunLoop 即将退出。
    __CFRunLoopDoObservers(rl, currentMode, kCFRunLoopExit);
}
```

Runloop 常见用法

Autorelease Pool

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

第一个 `Observer` 监视的事件是 `Entry(即将进入Loop)`，其回调内会调用 `_objc_autoreleasePoolPush()` 创建自动释放池。其 order 是-2147483647，优先级最高，保证创建释放池发生在其他所有回调之前。

第二个 `Observer` 监视了两个事件： `BeforeWaiting(准备进入休眠)` 时调用`_objc_autoreleasePoolPop()` 和 `_objc_autoreleasePoolPush()` 释放旧的池并创建新池； `Exit(即将退出Loop)` 时调用 `_objc_autoreleasePoolPop()` 来释放自动释放池。这个 Observer 的 order 是 2147483647，优先级最低，保证其释放池子发生在其他所有回调之后。

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

事件响应

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

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

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

界面更新

当在操作 UI 时，比如改变了 `Frame`、更新了 `UIView/CALayer` 的层次时，或者手动调用了 `UIView/CALayer 的 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。

苹果注册了一个 `Observer` 监听 `BeforeWaiting` (即将进入休眠) 和 `Exit` (即将退出Loop) 事件，回调去执行一个很长的函数：
`_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv()`。这个函数里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。

定时器

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

当时间点到后，容许有多少最大误差。

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

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

PerformSelecter

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

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

GCD

`GCD` 提供的某些接口也用到了 `RunLoop` ， 例如 `dispatch_async()` 。

当调用 `dispatch_async(dispatch_get_main_queue(), block)` 时， `libDispatch` 会向主线程的 `RunLoop` 发送消息， `RunLoop` 会被唤醒，并从消息中取得这个 `block` ，并在回调 `__CFRUNLOOP_IS_SERVICING_THE_MAIN_DISPATCH_QUEUE__()` 里执行这个 `block` 。但这个逻辑仅限于 `dispatch` 到主线程， `dispatch` 到其他线程仍然是由 `libDispatch` 处理的。

AFN 保活

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

```
+ (void)networkRequestThreadEntryPoint:(id)__unused object {
    @autoreleasepool {
        [[NSThread currentThread] setName:@"AFNetworking"];
        NSRunLoop *runLoop = [NSRunLoop currentRunLoop];
        [runLoop addPort:[NSMachPort port] forMode:NSDefaultRunLoopMode];
        [runLoop run];
    }
}

+ (NSThread *)networkRequestThread {
    static NSThread *_networkRequestThread = nil;
    static dispatch_once_t oncePredicate;
    dispatch_once(&oncePredicate, ^{
        _networkRequestThread = [[NSThread alloc] initWithTarget:self selector:@selector(networkRequestThreadEntryPoint:) object:nil];
        [_networkRequestThread start];
    });
    return _networkRequestThread;
}
```

AsyncDisplayKit

`AsyncDisplayKit` 是 `Facebook` 推出的用于保持界面流畅性的框架，其原理大致如下：

`UI` 线程中一旦出现繁重的任务就会导致界面卡顿，这类任务通常分为3类：排版，绘制，UI对象操作。

1. 排版通常包括计算视图大小、计算文本高度、重新计算子式图的排版等操作。
2. 绘制一般有文本绘制 (例如 `CoreText`)、图片绘制 (例如预先解压)、元素绘制 (`Quartz`)等操作。
3. UI对象操作通常包括 `UIView/CALayer` 等 `UI` 对象的创建、设置属性和销毁。

其中前两类操作可以通过各种方法扔到后台线程执行，而最后一类操作只能在主线程完成，并且有时后面的操作需要依赖前面操作的结果 （例如 `TextView` 创建时可能需要提前计算出文本的大小）。 `ASDK` 所做的，就是尽量将能放入后台的任务放入后台，不能的则尽量推迟 (例如视图的创建、属性的调整)。

为此， `ASDK` 创建了一个名为 `ASDisplayNode` 的对象，并在内部封装了 `UIView/CALayer` ，它具有和 `UIView/CALayer` 相似的属性，例如 `frame` 、 `backgroundColor` 等。所有这些属性都可以在后台线程更改，开发者可以只通过 `Node` 来操作其内部的 `UIView/CALayer` ，这样就可以将排版和绘制放入了后台线程。但是无论怎么操作，这些属性总需要在某个时刻同步到主线程的 `UIView/CALayer` 去。

`ASDK` 仿照 `QuartzCore/UIKit` 框架的模式，实现了一套类似的界面更新的机制：即在主线程的 `RunLoop` 中添加一个 `Observer` ，监听了 `kCFRunLoopBeforeWaiting` 和 `kCFRunLoopExit` 事件，在收到回调时，遍历所有之前放入队列的待处理的任务，然后一一执行。

ARC

reference：深入理解Objective C的ARC机制、Objective-C 引用计数原理、黑幕背后的Autorelease

ARC 基本原理

使用ARC，开发者不再需要手动的 `retain/release/autorelease` . 编译器会自动插入对应的代码，再结合 `Objective-C` 的 `runtime` ，实现自动引用计数。

如:

```
NSObject * obj;
{
    obj = [[NSObject alloc] init]; //引用计数为1
}
NSLog(@"%@",obj);
```

```
NSObject * obj;
{
    obj = [[NSObject alloc] init]; //引用计数为1
    [obj relrease]
}
NSLog(@"%@",obj);
```

其中， `retain` , `release` 的方法实现原理:

```
- (id)retain {
    return ((id)self)->rootRetain();
}
inline id objc_object::rootRetain()
{
    if (isTaggedPointer()) return (id)this;
    return sidetable_retain();
}

#if SUPPORT_MSB_TAGGED_POINTERS
#   define TAG_MASK (1ULL<<63)
#else
#   define TAG_MASK 1

inline bool
objc_object::isTaggedPointer()
{
#if SUPPORT_TAGGED_POINTERS
    return ((uintptr_t)this & TAG_MASK);
#else
    return false;
#endif
}
```

由 `retain` 方法的实现可知，有些对象如果支持使用 TaggedPointer，苹果会直接将其指针值作为引用计数返回；如果当前设备是 64 位环境并且使用 `Objective-C 2.0` ，那么“一些”对象会使用其 `isa` 指针的一部分空间来存储它的引用计数；否则 `Runtime` 会使用一张散列表来管理引用计数。

```
id objc_object::sidetable_retain()
{
    //获取table
    SideTable& table = SideTables()[this];
    //加锁
    table.lock();
    //获取引用计数
    size_t& refcntStorage = table.refcnts[this];
    if (! (refcntStorage & SIDE_TABLE_RC_PINNED)) {
         //增加引用计数
        refcntStorage += SIDE_TABLE_RC_ONE;
    }
    //解锁
    table.unlock();
    return (id)this;
}
```

与 `retain` 类似， `release` 方法也是如此的逻辑，只是增加了若引用计数为0，则销毁对象:

```
SideTable& table = SideTables()[this];
bool do_dealloc = false;
table.lock();
//找到对应地址的
RefcountMap::iterator it = table.refcnts.find(this);
if (it == table.refcnts.end()) { //找不到的话，执行dellloc
    do_dealloc = true;
    table.refcnts[this] = SIDE_TABLE_DEALLOCATING;
} else if (it->second < SIDE_TABLE_DEALLOCATING) {//引用计数小于阈值，dealloc
    do_dealloc = true;
    it->second |= SIDE_TABLE_DEALLOCATING;
} else if (! (it->second & SIDE_TABLE_RC_PINNED)) {
//引用计数减去1
    it->second -= SIDE_TABLE_RC_ONE;
}
table.unlock();
if (do_dealloc  &&  performDealloc) {
    //执行dealloc
    ((void(*)(objc_object *, SEL))objc_msgSend)(this, SEL_dealloc);
}
return do_dealloc;
```

再介绍下 `SideTable` 这个类，它用于管理引用计数表和 `weak` 表，并使用 `spinlock_lock` 自旋锁来防止操作表结构时可能的竞态条件。它用一个 64*128 大小的 `uint8_t` 静态数组作为 `buffer` 来保存所有的 `SideTable` 实例。并提供三个公有属性。

```
spinlock_t slock;//保证原子操作的自选锁
RefcountMap refcnts;//保存引用计数的散列表
weak_table_t weak_table;//保存 weak 引用的全局散列表
```

`weak` 表的作用是在对象执行 `dealloc` 的时候将所有指向该对象的 `weak` 指针的值设为 `nil`，避免悬空指针。这是 `weak` 表的结构：

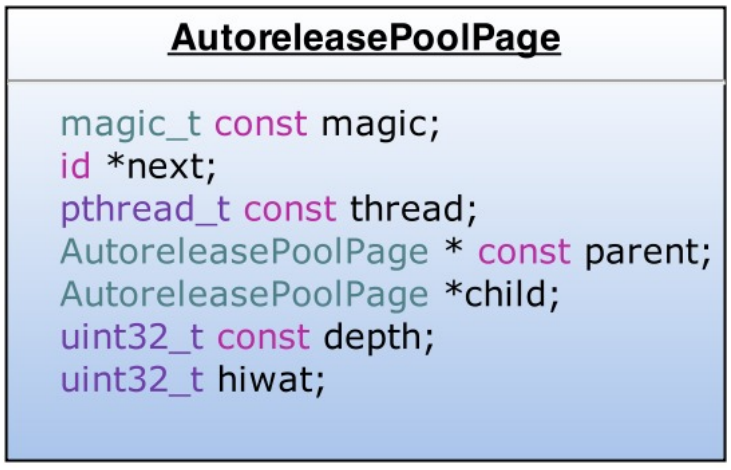
```
struct weak_table_t {
    weak_entry_t *weak_entries;
    size_t    num_entries;
    uintptr_t mask;
    uintptr_t max_hash_displacement;
};
```

苹果使用一个全局的 `weak` 表来保存所有的 `weak` 引用。并将对象作为键， `weak_entry_t` 作为值。 `weak_entry_t` 中保存了所有指向该对象的 `weak` 指针。

autoreleasepool 原理

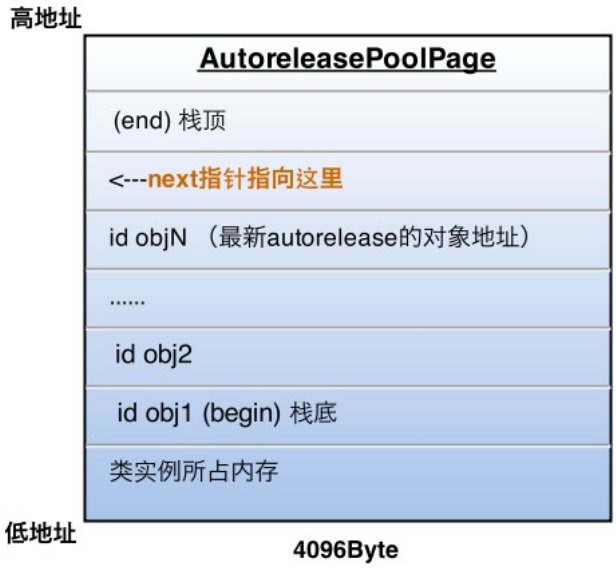
`Autorelease` 对象是在当前的 `runloop` 迭代结束时释放的，而它能够释放的原因是系统在每个 `runloop` 迭代中都加入了自动释放池 `Push` 和 `Pop`

`Autorelease` 对象的各种方法都是对类 `AutoreleasePoolPage` 的封装，其中 `AutoreleasePoolPage` 类结构如下:



- `AutoreleasePool` 并没有单独的结构，而是由若干个 `AutoreleasePoolPage` 以双向链表的形式组合而成（分别对应结构中的parent指针和child指针）
- `AutoreleasePool` 是按线程一一对应的（结构中的 `thread` 指针指向当前线程）
- `AutoreleasePoolPage` 每个对象会开辟4096字节内存（也就是虚拟内存一页的大小），除了上面的实例变量所占空间，剩下的空间全部用来储存 `autorelease` 对象的地址
- 上面的 `id *next` 指针作为游标指向栈顶最新add进来的 `autorelease` 对象的下一个位置
- 一个 `AutoreleasePoolPage` 的空间被占满时，会新建一个 `AutoreleasePoolPage` 对象，连接链表，后来的 `autorelease` 对象在新的page加入

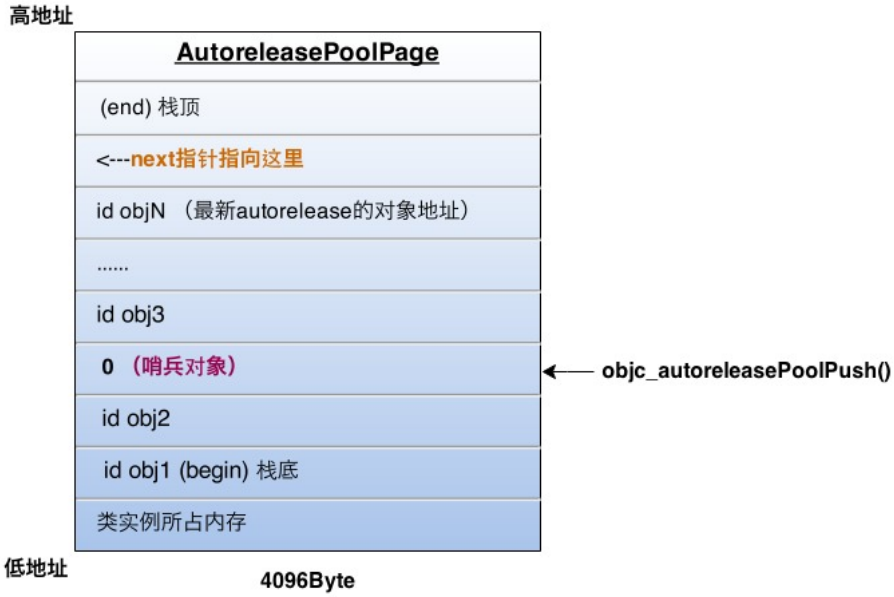
所以，若当前线程中只有一个 `AutoreleasePoolPage` 对象，并记录了很多 `autorelease` 对象地址时内存如下图：



图中的情况，这一页再加入一个 `autorelease` 对象就要满了（也就是 `next` 指针马上指向栈顶），这时就要执行上面说的操作，建立下一页 `page` 对象，与这一页链表连接完成后，新page的next指针被初始化在栈底（ `begin` 的位置），然后继续向栈顶添加新对象。

所以，向一个对象发送 `- autorelease` 消息，就是将这个对象加入到当前 `AutoreleasePoolPage` 的栈顶 `next` 指针指向的位置

每当进行一次 `objc_autoreleasePoolPush` 调用时， `runtime` 向当前的 `AutoreleasePoolPage` 中 `add` 进一个哨兵对象，值为0（也就是个 `nil` ），那么这一个 `page` 就变成了下面的样子：



`objc_autoreleasePoolPush` 的返回值正是这个哨兵对象的地址，被 `objc_autoreleasePoolPop` (哨兵对象)作为入参，于是：

- 根据传入的哨兵对象地址找到哨兵对象所处的 `page`
- 在当前 `page` 中，将晚于哨兵对象插入的所有 `autorelease` 对象都发送一次 `- release` 消息，并向回移动 `next` 指针到正确位置
- 从最新加入的对象一直向前清理，可以向前跨越若干个 `page` ，直到哨兵所在的 `page`

Tableview 优化

引用自保持界面流畅的技巧，YYAsyncLayer源码分析

预排版

当获取到 `API JSON` 数据后，把每条 `Cell` 需要的数据都在后台线程计算并封装为一个布局对象 `CellLayout` 。 `CellLayout` 包含所有文本的 `CoreText` 排版结果、 `Cell` 内部每个控件的高度、 `Cell` 的整体高度。每个 `CellLayout` 的内存占用并不多，所以当生成后，可以全部缓存到内存，以供稍后使用。这样， `TableView` 在请求各个高度函数时，不会消耗任何多余计算量；当把 `CellLayout` 设置到 `Cell` 内部时， `Cell` 内部也不用再计算布局了。

避免离屏渲染

有关离屏渲染的部分在第十三章:iOS中的渲染已经有所描述，不在赘述。

异步绘制

异步绘制的基本概念

图像的绘制通常是指用那些以 `CG` 开头的方法把图像绘制到画布中，然后从画布创建图片并显示这样一个过程。这个最常见的地方就是 `[UIView drawRect:]` 里面了。由于 `CoreGraphic` 方法通常都是线程安全的，所以图像的绘制可以很容易的放到后台线程进行。一个简单异步绘制的过程大致如下（实际情况会比这个复杂得多，但原理基本一致):

```
- (void)display {
    dispatch_async(backgroundQueue, ^{
        CGContextRef ctx = CGBitmapContextCreate(...);
        // draw in context...
        CGImageRef img = CGBitmapContextCreateImage(ctx);
        CFRelease(ctx);
        dispatch_async(mainQueue, ^{
            layer.contents = img;
        });
    });
}
```

在此也要注意， `CPU` 渲染也不是万能的，毕竟 `GPU` 在浮点数运算上还是胜于 `CPU` 。某些情况下，可能 `CPU` 渲染的效果还不如离屏渲染，究竟采用哪种技术，还需要根据具体的业务场景来抉择。

异步绘制时如何控制并发数

十一. 开源库

SDWebImg

引用自sdwebimg源码解析

简介

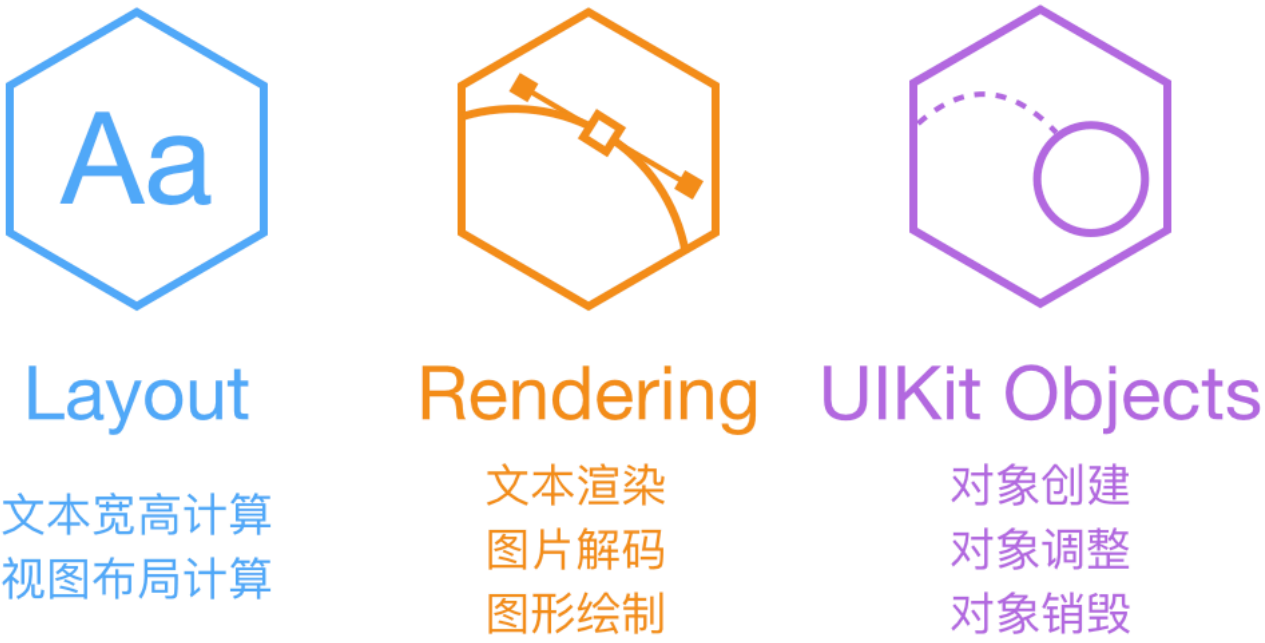
基本原理

YYKit

ASDK

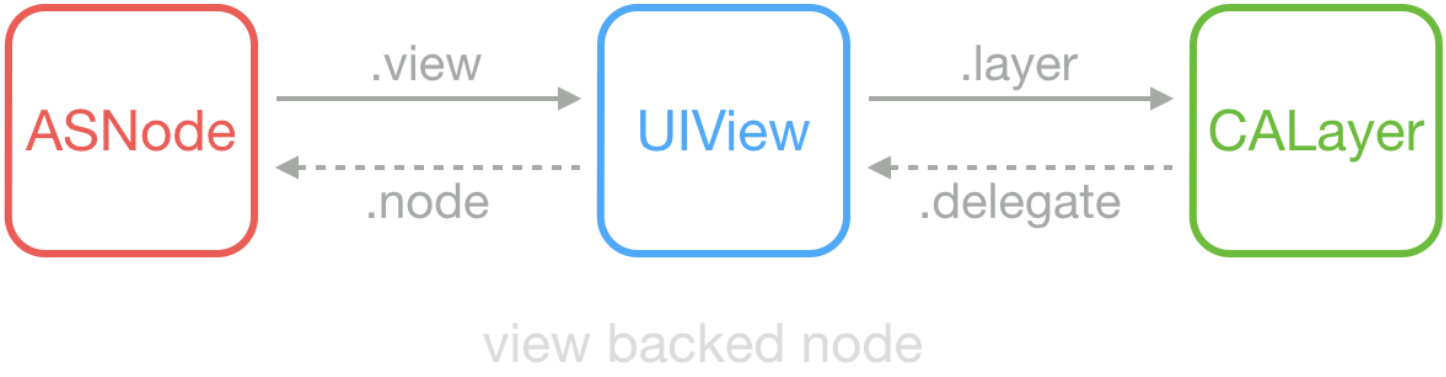
在runloop章节有所简介。

asdk 基本原理

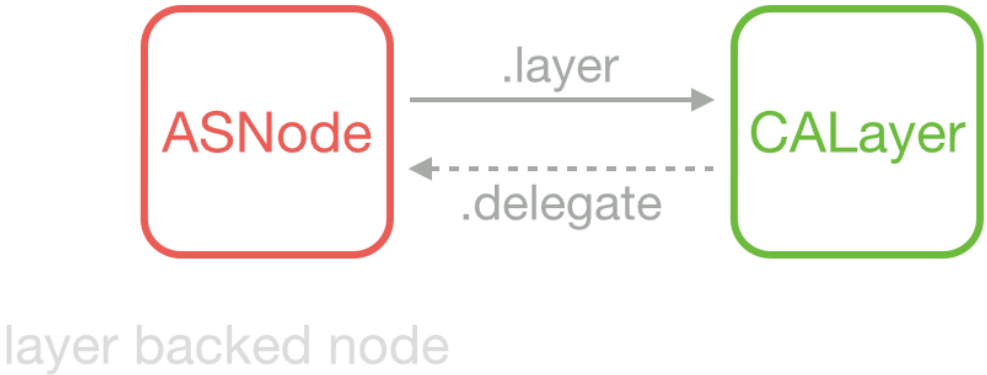


`ASDK` 认为，阻塞主线程的任务，主要分为上面这三大类。文本和布局的计算、渲染、解码、绘制都可以通过各种方式异步执行，但 `UIKit` 和 `Core Animation` 相关操作必需在主线程进行。 `ASDK` 的目标，就是尽量把这些任务从主线程挪走，而挪不走的，就尽量优化性能。

`ASDK` 为此创建了 `ASDisplayNode` 类，包装了常见的视图属性（比如 `frame/bounds/alpha/transform/backgroundColor/superNode/subNodes` 等），然后它用 `UIView->CALayer` 相同的方式，实现了 `ASNode->UIView` 这样一个关系:



当不需要响应触摸事件时， `ASDisplayNode` 可以被设置为 `layer backed` ，即 `ASDisplayNode` 充当了原来 `UIView` 的功能，节省了更多资源:



与 `UIView` 和 `CALayer` 不同， `ASDisplayNode` 是线程安全的，它可以在后台线程创建和修改。 `Node` 刚创建时，并不会在内部新建 `UIView` 和 `CALayer` ，直到第一次在主线程访问 `view` 或 `layer` 属性时，它才会在内部生成对应的对象。当它的属性（比如 `frame/transform` ）改变后，它并不会立刻同步到其持有的 `view` 或 `laye` r 去，而是把被改变的属性保存到内部的一个中间变量，稍后在需要时，再通过某个机制一次性设置到内部的 `view` 或 `layer` 。

通过模拟和封装 `UIView/CALayer` ，开发者可以把代码中的 `UIView` 替换为 `ASNode` ，很大的降低了开发和学习成本，同时能获得 `ASDK` 底层大量的性能优化。为了方便使用， `ASDK` 把大量常用控件都封装成了 `ASNode` 的子类，比如 `Button、Control、Cell、Image、ImageView、Text、TableView、CollectionView` 等。利用这些控件，开发者可以尽量避免直接使用 `UIKit` 相关控件，以获得更完整的性能提升。

asdk 中 runloop的应用

`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（这个值取决设备硬件，比如 `iPhone` 真机上通常是 `59.97`）。 `iOS` 图形服务接收到 `VSync` 信号后，会通过 `IPC` 通知到 `App` 内。 `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，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。这个 `Observer` 的优先级是 `2000000`，低于常见的其他 `Observer`。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` 捕获，并通过 `CATransaction` 提交到一个中间状态去（ `CATransaction` 的文档略有提到这些内容，但并不完整）。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 `CA` 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， CA 会通过 DisplayLink` 等机制多次触发相关流程。

`ASDK` 在此处模拟了 `Core Animation` 的这个机制：所有针对 `ASNode` 的修改和提交，总有些任务是必需放入主线程执行的。当出现这种任务时， `ASNode` 会把任务用 `ASAsyncTransaction(Group)` 封装并提交到一个全局的容器去。 `ASDK` 也在 `RunLoop` 中注册了一个 `Observer`，监视的事件和 `CA` 一样，但优先级比 `CA` 要低。当 `RunLoop` 进入休眠前、 `CA` 处理完事件后， `ASDK` 就会执行该 `loop` 内提交的所有任务。具体代码见这个文件：ASAsyncTransactionGroup。

通过这种机制， `ASDK` 可以在合适的机会把异步、并发的操作同步到主线程去，并且能获得不错的性能。

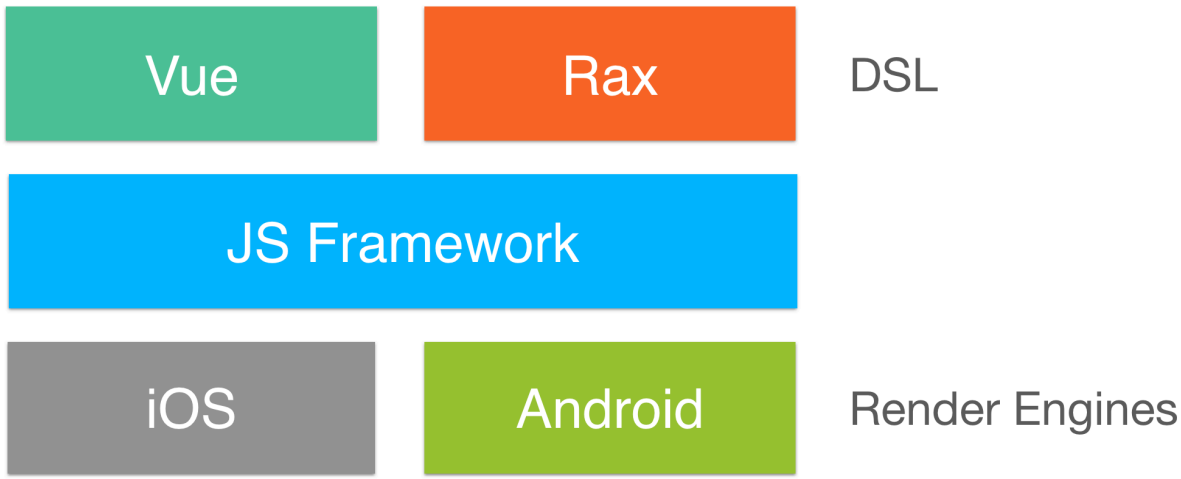
十二. Weex

本节不讲解如何使用weex,只关注weex iOS部分原理。如有兴趣，可自行去官网学习基本知识

Weex 基本结构

`Weex` 目前最新版本支持 `Rax` 与 `VueJS`，其中 `Rax` 即为 `React` 语法，目的使前端开发者更易上手。

`Weex SDK` 结构主要有三部分组成:



1. 顶层 `framework`
    - 此层的作用是编译目标文件至js，比如如果使用的 `.vue` 文件，就将此文件通过 `framework` 编译成对应的 `jsBundle`。此层其实是 `weex` 集成了 `vuejs` 或者 `react`。
2. 中间层 `js runtime`
    - 此层的作用是承接上层的 `jsbundle` 文件，在 `iOS` 上， `weex sdk` 通过使用系统 `JSCore` 来跑 `jsbundle` 文件。通过理解js文件，开始执行各种指令。比如在 `vuejs` 中的 `creatElement` 指令，会被转换为 `creatNative` 指令发送给下一层。通过这种方式，隔离了前端开发者与客端户开发工作，使得前端开发者也可以通过常用的js语法生成 `native` 组件，调用原生的各种方法。
3. 底层 `Render Engine`
    - 此层承接 `weex runtime` 传来的各种指令，开始执行 `weex sdk` 中"预置"的 `native` 代码。

Weex JS Framework 详解

`Weex JS Framework` 主要分为以下几个功能:

1. 适配前端层
2. 构建渲染指令树

3. JS-Native 通信
4. JS Service
5. 准备环境接口

适配前端层

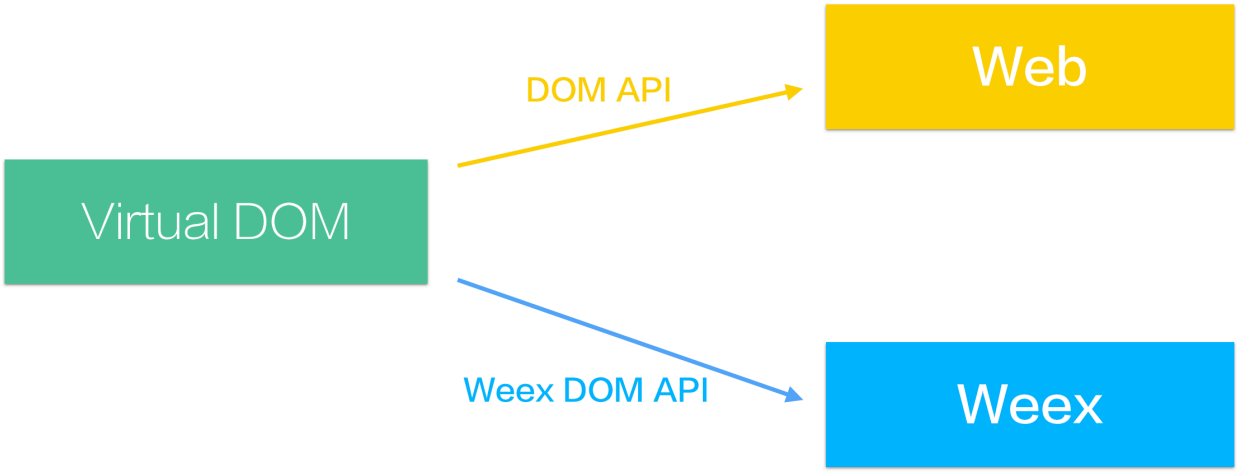
前端框架在 `Weex` 和浏览器中的执行过程不一样，这个应该不难理解。如何让一个前端框架运行在 `Weex` 平台上，是 `JS Framework` 的一个关键功能。

以 `Vue.js` 为例，在浏览器上运行一个页面大概分这么几个步骤：首先要准备好页面容器，可以是浏览器或者是 `WebView`，容器里提供了标准的 `Web API`。然后给页面容器传入一个地址，通过这个地址最终获取到一个 `HTML` 文件，然后解析这个 `HTML` 文件，加载并执行其中的脚本。想要正确的渲染，应该首先加载执行 `Vue.js` 框架的代码，向浏览器环境中添加 `Vue` 这个变量，然后创建好挂载点的 `DOM` 元素，最后执行页面代码，从入口组件开始，层层渲染好再挂载到配置的挂载点上去。

在 `Weex` 里的执行过程也比较类似，不过 `Weex` 页面对应的是一个 `js` 文件，不是 `HTML` 文件，而且不需要自行引入 `Vue.js` 框架的代码，也不需要设置挂载点。过程大概是这样的：首先初始化好 `Weex` 容器，这个过程中会初始化 `JS Framework`，`Vue.js` 的代码也包含在了其中。然后给 `Weex` 容器传入页面地址，通过这个地址最终获取到一个 `js` 文件，客户端会调用 `createInstance` 来创建页面。

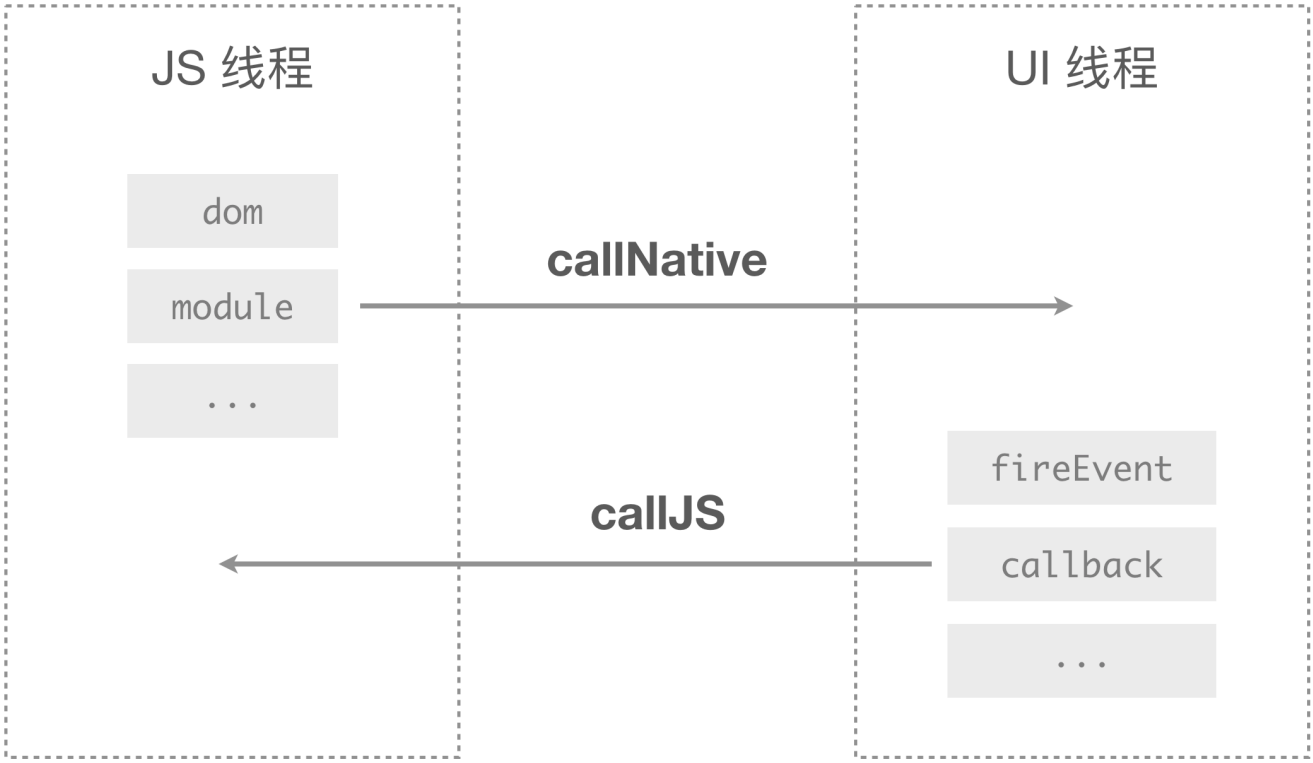
构建渲染指令树

不同的前端框架里 `Virtual DOM` 的结构、`patch` 的方式都是不同的，这也反应了它们开发理念和优化策略的不同，但是最终，在浏览器上它们都使用一致的 `DOM API` 把 `Virtual DOM` 转换成真实的 `HTMLElement`。在 `Weex` 里的逻辑也是类似的，只是在最后一步生成真实元素的过程中，不使用原生 `DOM API`，而是使用 `JS Framework` 里定义的一套 `Weex DOM API` 将操作转化成渲染指令发给客户端。



`JS Framework` 提供的 `Weex DOM API` 和浏览器提供的 `DOM API` 功能基本一致，在 `Vue` 和 `Rax` 内部对这些接口都做了适配，针对 `Weex` 和浏览器平台调用不同的接口就可以实现跨平台渲染。

JS-Native 通信原理



首先，页面的 `js` 代码是运行在 `js` 线程上的，然而原生组件的绘制、事件的捕获都发生在 `UI` 线程。在这两个线程之间的通信用的是 `callNative` 和 `callJS` 这两个底层接口（现在已经扩展到了很多个），它们默认都是异步的，在 `JS Framework` 和原生渲染器内部都基于这两个方法做了各种封装。

`callNative` 是由客户端向 `JS` 执行环境中注入的接口，提供给 `JS Framework` 调用，界面的节点（上文提到的渲染指令树）、模块调用的方法和参数都是通过这个接口发送给客户端的。为了减少调用接口时的开销，其实现在已经开了更多更直接的通信接口，其中有些接口还支持同步调用（支持返回值），它们在原理上都和 `callNative` 是一样的。

`callJS` 是由 `JS Framework` 实现的，并且也注入到了执行环境中，提供给客户端调用。事件的派发、模块的回调函数都是通过这个接口通知到 `JS Framework`，然后再将其传递给上层前端框架。

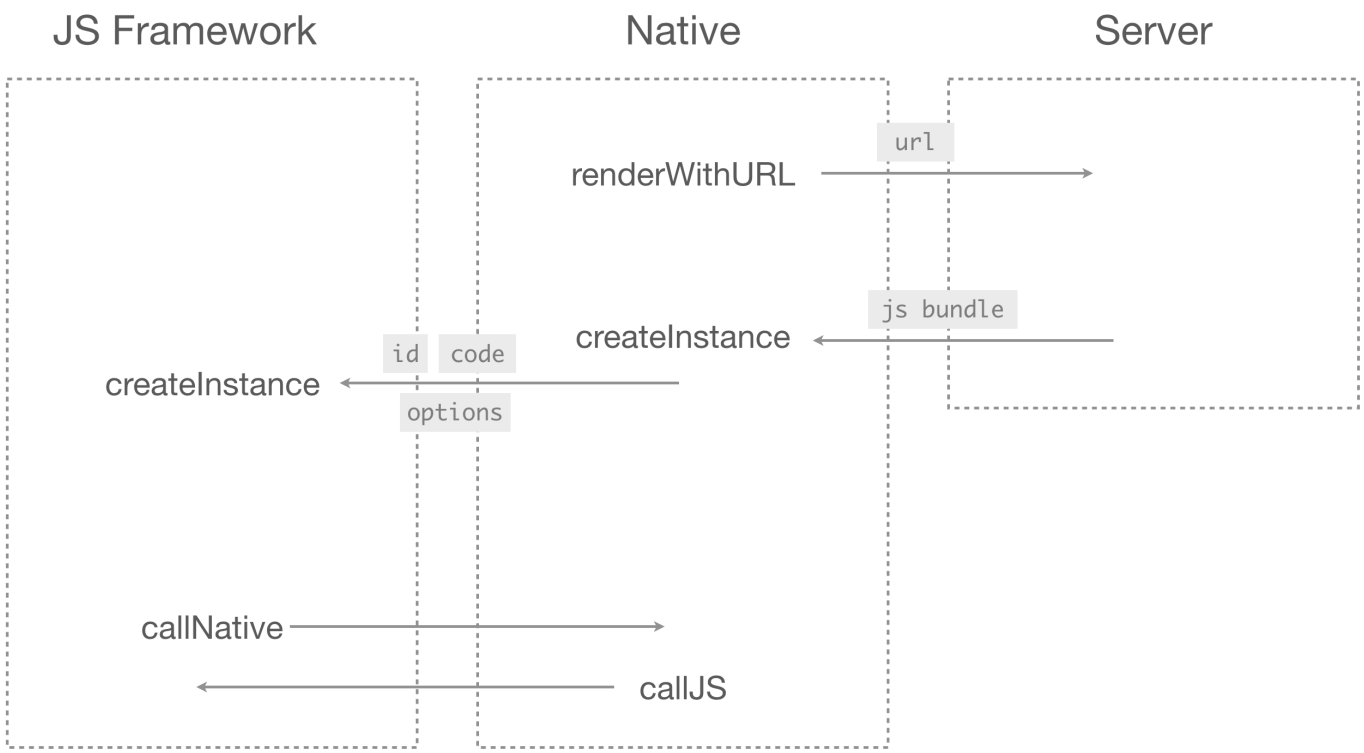
JS Service

`Weex` 是一个多页面的框架，每个页面的 `js bundle` 都在一个独立的环境里运行，不同的 `Weex` 页面对应到浏览器上就相当于不同的“标签页”，普通的 `js` 库没办法实现在多个页面之间实现状态共享，也很难实现跨页通信。

在 `JS Framework` 中实现了 `JS Service` 的功能，主要就是用来解决跨页面复用和状态共享的问题的，例如 BroadcastChannel 就是基于 `JS Service` 实现的，它可以在多个 `Weex` 页面之间通信。

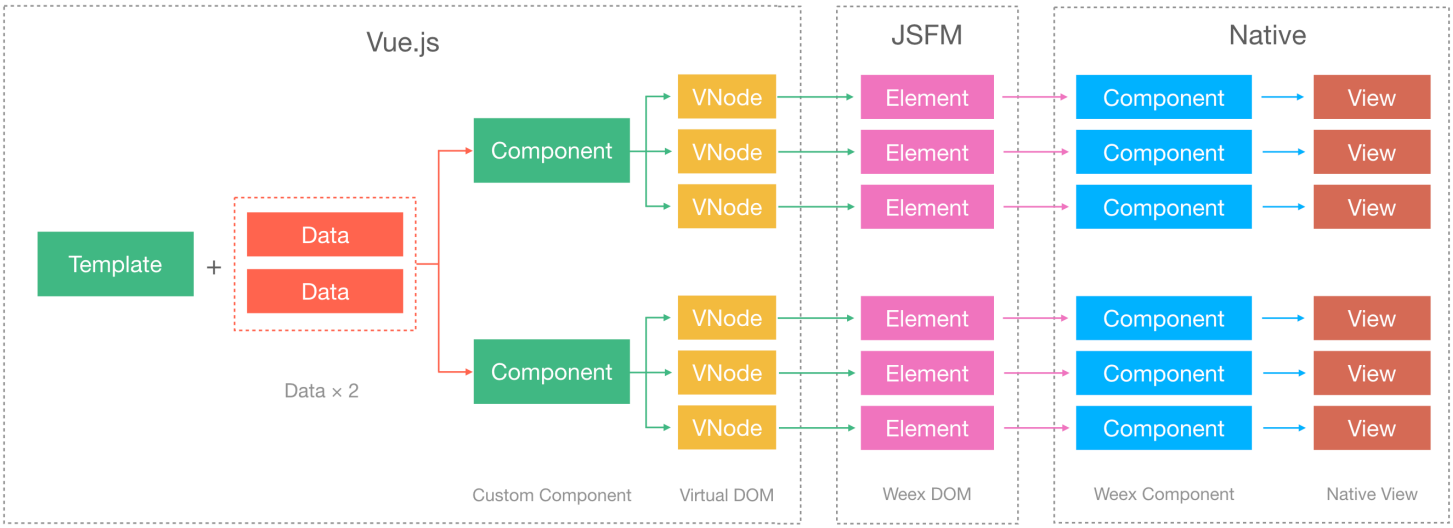
Weex 渲染过程及事件响应

Weex 的页面渲染

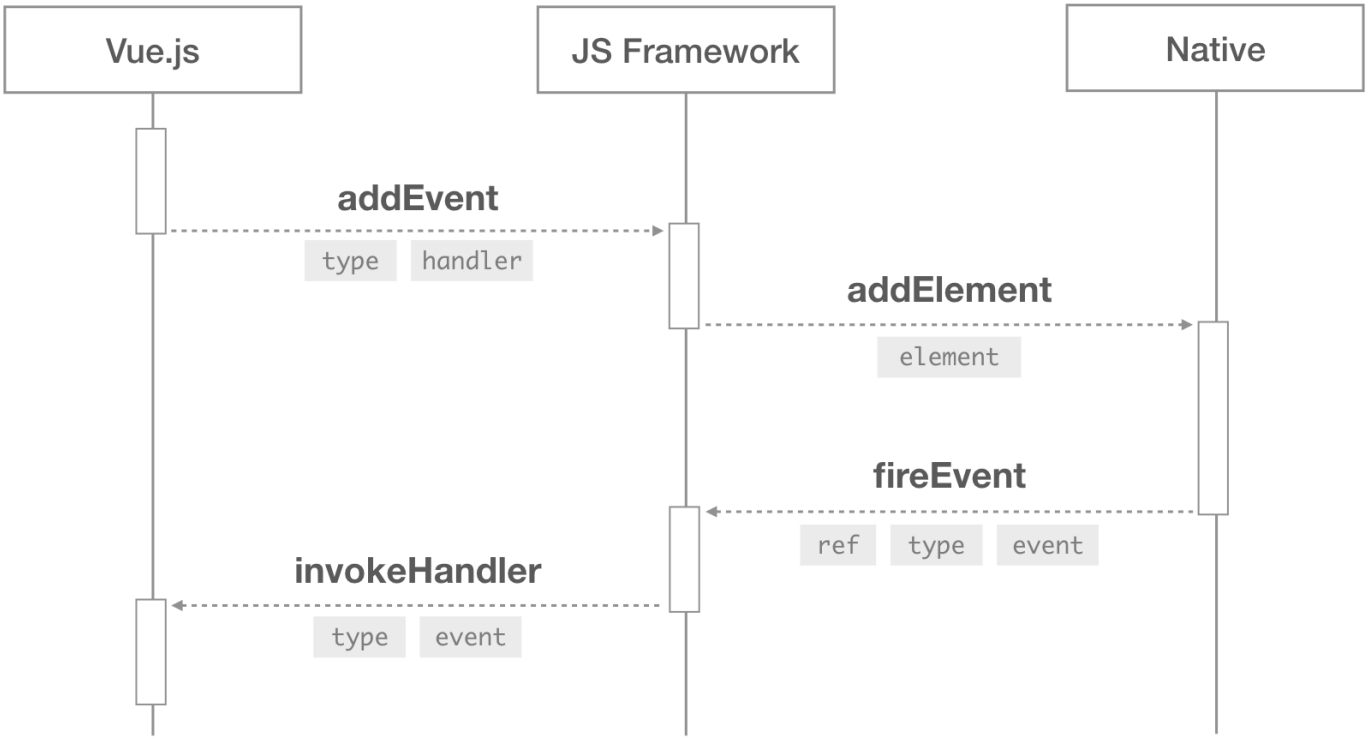


先描述渲染前的流程： `Weex` 会先拉取一个 `JSBundle` ，这个 `JSBundle` 就是之前提到的 `VueJS` 编译后的 `js` 打包文件。再收到 `JSBunlde` 后， `Weex SDK` 会通过 `native` 代码创建 `intance` ，之后再将一些配置及环境变量、 `JSBundle` 一起传给 `JS Framework` ，其目的是通过 `JS Framwork` 生成之前提到过的**渲染指令树**。

在此之后就可以开始正式的渲染页面流程:



Weex 的事件响应



如上图所示，如果在 `Vue.js` 里某个标签上绑定了事件，会在内部执行 `addEventListener` 给节点绑定事件，这个接口在 `Weex` 平台下调用的是 `JS Framework` 提供的 `addEvent` 方法向元素上添加事件，传递了事件类型和处理函数。 `JS Framework` 不会立即向客户端发送添加事件的指令，而是把事件类型和处理函数记录下来，节点构建好以后再一起发给客户端，发送的节点中只包含了事件类型，不含事件处理函数。客户端在渲染节点时，如果发现节点上包含事件，就监听原生 UI 上的指定事件。

当原生 UI 监听到用户触发的事件以后，会派发 `fireEvent` 命令把节点的 `ref` 、事件类型以及事件对象发给 `JS Framework` 。 `JS Framework` 根据 `ref` 和事件类型找到相应的事件处理函数，并且以事件对象 `event` 为参数执行事件处理函数。目前 `Weex` 里的事件模型相对比较简单，并不区分捕获阶段和冒泡阶段，而是只派发给触发了事件的节点，并不向上冒泡，类似 `DOM` 模型里 `level 0` 级别的事件。

Weex 中的 iOS: JavaScriptCore

此节简单介绍下 `Weex` 中的 `JavaScriptCore` 。

OC、JS互相调用

```
 var addElement = function(){
    return 'addElement';
 }
```

比如上述js文件

```
//1.加载上面js文件到JSContext
JSContext *context = [[JSContext alloc]init];
[context evaluateScript:jsFilePath];

// 2.调用addElement方法
JSValue *jsFunction = context[@"addElement"];
JSValue *addElement = [jsFunction callWithArguments:nil];
```

把 `js` 文件加载进 `objc` 的 `JSContext` 后，就可以调用 `js` 文件里定义的方法了，这个就是 `objc` 调用 `js` 的原理。

```
//注册native方法给js调用
- (void)regiestNativeFunction {
    //注册一个callCreateBody方法给js调用
    self.jsContext[@"callCreateBody"] = ^(NSString *msg){
        NSLog(@"js:msg:%@",msg);
    };
    //注册一个callAddEvent方法给js调用
    self.jsContext[@"callAddEvent"] = ^(NSString *msg){
        NSLog(@"js:msg:%@",msg);
    };
     //注册一个callAddElement方法给js调用
   self.jsContext[@"callAddElement"] = ^(NSString *msg){
        NSLog(@"js:msg:%@",msg);
    };
    //使用js调用objc
    [self.jsContext evaluateScript:@"callAddEvent('hello,i am js side')"];
}
```

weex封装的全局方法

上面的代码 `native` 端注册了三个方法给 `js` 端调用， `weex` 基于这种能力封装了一系列的方法，下面列出的就是 `weex` 提供的操作UI的一些的全局方法。

`global.callCreateBody, global.callAddElement, global.callRemoveElement, global.callMoveElement, global.callUpdateAttrs, global.callUpdat`

通信数据格式

weex有了JS-Native相互通信的能力后，再按照一定格式发送数据给native端就可以渲染UI了。这个格式要与native端协商好，目前weex接受的格式类似下面的json格式

```
{
    ref: "", //js端随机生成的数字id
    type: "", //native端支持的component类型
              //比如text div list等
    attr: {}, //component支持的属性
              //比如text的value
    style: {} //元素的样式，native端基于Facebook的yoga
              //框架的前身cssLayout来生成native UI

}
```

iOS中的渲染与绘制

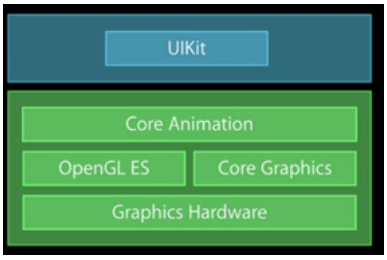
本章尝试讲解 `iOS` 中渲染与绘制的部分，是第十章Tableview优化的基础内容。

引用自:

- Core Animation
- iOS 事件处理机制与图像渲染过程

理解 iOS 渲染框架运作机理

`UIKit` 是常用的框架，显示、动画都通过 `CoreAnimation` 。 `CoreAnimation` 是核心动画，依赖于 `OpenGL ES` 做 `GPU` 渲染， `CoreGraphics` 做 `CPU` 渲染；最底层的 `GraphicsHardWare` 是图形硬件。



理解 CALayer

在 `iOS` 当中，所有的视图都从一个叫做 `UIView` 的基类派生而来， `UIView` 可以处理触摸事件，可以支持基于 `Core Graphics` 绘图，可以做仿射变换（例如旋转或者缩放），或者简单的类似于滑动或者渐变的动画。

`CALayer` 类在概念上和 `UIView` 类似，同样也是一些被层级关系树管理的矩形块，同样也可以包含一些内容（像图片，文本或者背景色），管理子图层的位置。它们有一些方法和属性用来做动画和变换。和 `UIView` 最大的不同是 `CALayer` 不处理用户的交互。 `CALayer` 并不清楚具体的响应链。

`UIView` 和 `CALayer` 是一个平行的层级关系，每一个 `UIView` 都有一个 `CALayer` 实例的图层属性，也就是所谓的 `backing layer` ，视图的职责就是创建并管理这个图层，以确保当子视图在层级关系中添加或者被移除的时候，他们关联的图层也同样对应在层级关系树当中有相同的操作。实际上这些背后关联的 `Layer` 图层才是真正用来在屏幕上显示和做动画， `UIView` 仅仅是对它的一个封装，提供了一些 `iOS` 类似于处理触摸的具体功能，以及 `Core Animation` 底层方法的高级接口。

`UIView` 的 `Layer` 在系统内部，被维护着三份同样的树形数据结构，分别是：

- 图层树（这里是代码可以操纵的，设置属性的最终值会立刻在这里更新）；
- 呈现树（是一个中间层，系统就在这一层上更改属性，进行各种渲染操作。比如一个动画是更改alpha值从0到1，那么在逻辑树上此属性会被立刻更新为最终属性1，而在动画树上会根据设置的动画时间从0逐步变化到1）；
- 渲染树（其属性值就是当前正被显示在屏幕上的属性值）；

渲染驱动器：CADisplayLink

此处与Runloop中的定时器重叠。

`NSTimer` 其实就是 `CFRunLoopTimerRef` 。一个 `NSTimer` 注册到 `RunLoop` 后， `RunLoop` 会为其重复的时间点注册好事件。

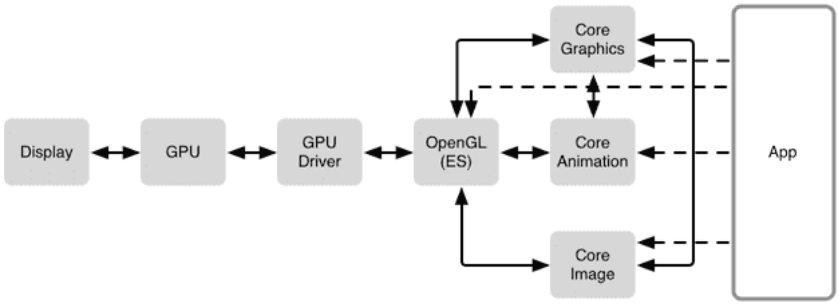
`RunLoop` 为了节省资源，并不会在非常准确的时间点回调这个 `Timer` 。 `Timer` 有个属性叫做 `Tolerance` (宽容度)，标示了当时间点到后，容许有多少最大误差。如果某个时间点被错过了，例如执行了一个很长的任务，则那个时间点的回调也会跳过去，不会延后执行。

`RunLoop` 是用GCD的 `dispatch_source_t` 实现的 `Timer` 。当调用 `NSObject` 的 `performSelecter:afterDelay:` 后，实际上其内部会创建一个 `Timer` 并添加到当前线程的 `RunLoop` 中。所以如果当前线程没有 `RunLoop` ，则这个方法会失效。当调用 `performSelector:onThread:` 时，实际上其会创建一个 `Timer` 加到对应的线程去，同样的，如果对应线程没有 `RunLoop` 该方法也会失效。

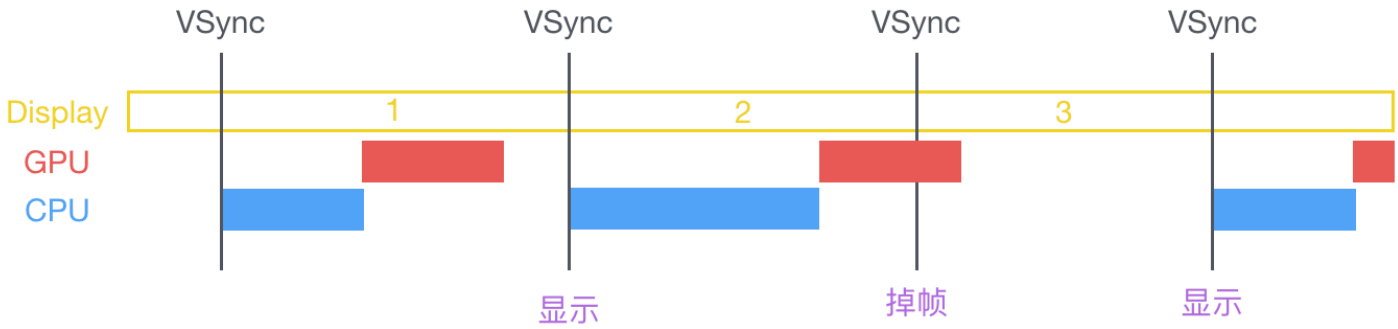
`CADisplayLink` 是一个和屏幕刷新率（每秒刷新60次）一致的定时器（但实际实现原理更复杂，和 `NSTimer` 并不一样，其内部实际是操作了一个 `Source` ）。如果在两次屏幕刷新之间执行了一个长任务，那其中就会有一帧被跳过去，造成界面卡顿的感觉。

渲染流程

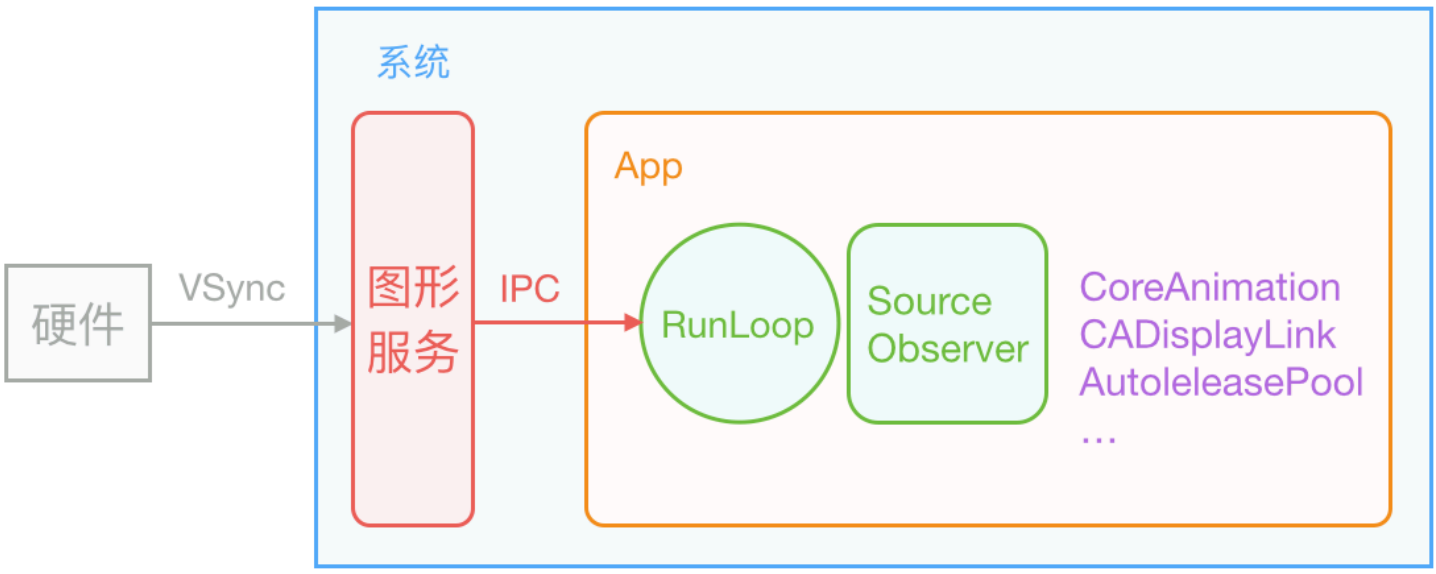
此处在Runloop应用中均有提及。



通常来说，计算机系统中 `CPU` 、 `GPU` 、显示器是以上面这种方式协同工作的。 `CPU` 计算好显示内容提交到 `GPU` ， `GPU` 渲染完成后将渲染结果放入帧缓冲区，随后视频控制器会按照 `VSync` 信号如下图所示，逐行读取帧缓冲区的数据，经过可能的数模转换传递给显示器显示。



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。从上图中可以看到， `CPU` 和 `GPU` 不论哪个阻碍了显示流程，都会造成掉帧现象。所以开发时，也需要分别对 `CPU` 和 `GPU` 压力进行评估和优化。



`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（这个值取决设备硬件，比如 `iPhone` 真机上通常是 59.97）。iOS 图形服务接收到 `VSync` 信号后，会通过 `IPC` 通知到 `App` 内。 `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，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，从而根据时间来不断更新这个动画的属性值并绘制出来。

为了不阻塞主线程， `Core Animation` 的核心是 `OpenGL ES` 的一个抽象物，所以大部分的渲染是直接提交给 `GPU` 来处理。 而 `Core Graphics/Quartz 2D` 的大部分绘制操作都是在主线程和 `CPU` 上同步完成的，比如自定义 `UIView` 的 `drawRect` 里用 `CGContext` 来画图。

渲染时机

上面已经提到过： `Core Animation` 在 `RunLoop` 中注册了一个 `Observer` 监听 `BeforeWaiting` (即将进入休眠) 和 `Exit` (即将退出 `Loop`) 事件 。当在操作 UI 时，比如改变了 `Frame` 、更新了 `UIView/CALayer` 的层次时，或者手动调用了 `UIView/CALayer` 的 `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。

这个函数内部的调用栈大概是这样的：

```
_ZN2CA11Transaction17observer_callbackEP19__CFRunLoopObservermPv()
    QuartzCore:CA::Transaction::observer_callback:
        CA::Transaction::commit();
            CA::Context::commit_transaction();
                CA::Layer::layout_and_display_if_needed();
                    CA::Layer::layout_if_needed();
                          [CALayer layoutSublayers];
                          [UIView layoutSubviews];
                    CA::Layer::display_if_needed();
                          [CALayer display];
                          [UIView drawRect];
```

GPU渲染 Vs. CPU渲染

`OpenGL中` ， `GPU` 屏幕渲染有以下两种方式：

1. `On-Screen Rendering` :意为当前屏幕渲染，指的是 `GPU` 的渲染操作是在当前用于显示的屏幕缓冲区中进行。
2. `Off-Screen Rendering` :意为离屏渲染，指的是 `GPU` 在当前屏幕缓冲区以外新开辟一个缓冲区进行渲染操作。

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

相比于当前屏幕渲染，离屏渲染的代价是很高的，主要体现在两个方面：

1. 创建新缓冲区

要想进行离屏渲染，首先要创建一个新的缓冲区。

2. 上下文切换

   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有需要将上下文环境从离屏切换到当前屏幕。而上下文环境的切换是要付出很大代价的。

设置了以下属性时，都会触发离屏绘制：

`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。

而其它属性如果是开启的，就不会有缓存，离屏绘制会在每一帧都发生。

在开发时需要根据实际情况来选择最优的实现方式，尽量使用`On-Screen Rendering`。简单的`Off-Screen Rendering`可以考虑使用`Core Graphics`让`CPU`来渲染。

多线程

这章并不会详细介绍一些基本的如**线程**、**进程**、**GCD**、**NSOperationQueue**等概念，更多的是对第七章:CGD中的内容做一些补充。如果有看不懂的地方，推荐阅读并发编程：API及挑战及相关内容。

为什么要使用GCD 而不用 p_thread 或者 NSThread

`GCD`队列的内部使用的是线程。`GCD`管理这些线程，并且使用`GCD`的时候，你不需要自己创建线程。但是重要的外在部分`GCD`会呈现给你，也就是用户`API`，一个很大不同的抽象层级。当使用`GCD`来完成并发的工作时，你不必考虑线程方面的问题，取而代之的，只需考虑队列和功能点（提交给队列的`block`）。虽然往下深究，依然都是线程，但是`GCD`的抽象层级为你惯用的编码提供了更好的方式。

队列和功能点同时解决了一个连续不断的扇出的问题：如果我们直接使用线程，并且想要做一些并发的事情，我们很可能将我们的工作分成`100`个小的功能点，然后基于可用的`CPU`内核数量来创建线程，假设是`8`。我们把这些功能点送到这`8`个线程中。当我们处理这些功能点时，可能会调用一些函数作为功能的一部分。写那个函数的人也想要使用并发，因此当你调用这个函数的时候，这个函数也会创建`8`个线程。现在，你有了`8 × 8 = 64`个线程，尽管你只有`8`个CPU内核——也就是说任何时候只有12%的线程实际在运行而另外88%的线程什么事情都没做。使用`GCD`你就不会遇到这种问题，**当系统关闭`CPU`内核以省电时，`GCD`甚至能够相应地调整线程数量。**

`GCD`通过创建所谓的线程池来大致匹配`CPU`内核数量。要记住，线程的创建并不是无代价的。每个线程都需要占用内存和内核资源。这里也有一个问题：如果你提交了一个`block`给 GCD，但是这段代码阻塞了这个线程，那么这个线程在这段时间内就不能用来完成其他工作——它被阻塞了。为了确保功能点在队列上一直是执行的，`GCD`不得不创建一个新的线程，并把它添加到线程池。

如果你的代码阻塞了许多线程，这会带来很大的问题。首先，线程消耗资源，此外，创建线程会变得代价高昂。创建过程需要一些时间。并且在这段时间中，`GCD`无法以全速来完成功能点。有不少能够导致线程阻塞的情况，但是最常见的情况与`I/O`有关，也就是从文件或者网络中读写数据。正是因为这些原因，你不应该在`GCD`队列中以阻塞的方式来做这些操作。