-----------多线程与编译运行猜想----------------------------------------------

网址：

http://mobile.51cto.com/iphone-386596.htm

http://blog.csdn.net/csj1987/article/details/7527590

iOS线程概述

1. iPhone中的线程应用并不是无节制的，官方给出的资料显示iPhone OS下的主线程的堆栈大小是1M，第二个 线程开始都是512KB。且该值不能通过编译器开关或线程API函数来更改。只有主线程有直接修改UI的能力。

2. 有些程序是条直线；有些程序是一个圆，不断循环，直到将它切断。圆如操作系统，一直运行直到你关机。

3. 一个运行着的程序就是一个进程或者叫做一个任务，一个进程至少包含一个线程，线程就是程序的执行流。Mac 和iOS中的程序启动，创建好一个进程的同时， 一个线程便开始运行，这个线程叫主线程。主线程在程序中 的地位和其他线程不同，主线程是其他线程最终的父线程，且所有界面的显示操作即AppKit或 UIKit的操作 必须 在主线程进行。

4. 系统中的每一个进程都有自己独立的虚拟内存空间，而同一个进程中的多个线程则共用进程的内存空间。每创建 一个新的线程，都需要一些内存(如每个线程有自己的Stack空间)和消耗一定的CPU时间

创建线程概述汇总 - DeepInComputer提及

概述：创建一个新的线程就是给进程增加了一个执行流，执行流总得有要执行的代码吧，所以新建一个线程需要提 供一个函数或者方法作为线程的入口。

单线程创建：

NSThread 的两种方式创建

NSObject 的后台运行[myObj performSelectorInBackground:@selector(doSomething) withObject:nil];

POSIX Thread：由于Mac和iOS都是基于Darwin系统，Darwin系统的XUN内核，是基于Mach和BSD 的，继承了BSD的POSIX接口，所以可以直接使用POSIX线程的相关接口来使用线程： pthread\_create

多线程进阶：

很多时候我们使用多线程，需要控制线程的并发数，毕竟线程也是消耗系统资源的，当程序中同时运行的线程 过多时，系统必然变慢。 所以很多时候我们会控制同时运行线程的数目。

NSOperation&NSOperationQueue：

NSOperation可以封装我们的操作，然后将创建好的NSOperation对象放到NSOperationQueue中， OperationQueue便开始启动新的线程去执行队列中的操作。该类可以设置并发度：

- (**void**)setMaxConcurrentOperationCount:(NSInteger)count

GCD：

GCD是Grand Central Dispatch的缩写，是一系列的BSD层面的接口，在Mac 10.6 和iOS4.0以后才 引入的，且现在NSOperation和NSOperationQueue的多线程的实现就是基于GCD的

线程间通信概述

概述：线程间通信和进程间通信从本质上讲是相似的。线程间通信就是在进程内的两个执行流之间进行数据的传递， 就像两条并行的河流之间挖出了一道单向流动长沟，使得一条河流中的水可以流入另一条河流，物质得到传递。

performSelect On The Thread：

框架为我们提供了强制在某个线程中执行方法的途径,如果两个非主线程的线程需要相互间通信，可以先将自己 的当前线程对象注册到某个全局的对象中去，这样相 互之间就可以获取对方的线程对象，然后就可以使用下面 的方法进行线程间的通信了，由于主线程比较特殊，所以框架直接提供了在出线程执行的方法。

@interface NSObject (NSThreadPerformAdditions)

- (**void**)performSelectorOnMainThread: (SEL)aSelector withObject:(id)arg waitUnti

Mach Port

其实质就是父线程创建一个NSMachPort对象，在创建子线程的时候以参数的方式将其传递给子线程，这样子 线程中就可以向这个传过来的 NSMachPort对象发送消息

如果想让父线程也可以向子线程发消息的话，那么子线程可以先向父线程发个特殊的消息，传过来的是自己创 建的另一个 NSMachPort对象，这样父线程便持有了子线程创建的port对象了，可以向这个子线程的 port对象发送消息了。

当然各自的port对象需要设置delegate以及schdule到自己所在线程的RunLoop中，这样来了消息之后， 处理port消息的delegate方法会被调用，你就可以自己处理消息了。

编译与进程运行猜想（带指正）

猜想依据：

1. 使用 Dispatch Source 调试感知时+断点暂停时，插入的代码可以修改UI，说明给予代码，线程便可执行

2. 进程是线程的容器。程序是指令、数据及其组织形式的描述，进程是程序的实体。

3. 每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region） 和堆栈（stack region）。文本区域存储处理器执行的代码；数据区域存储变量和进程执行期间使用的动态 分配的内存；堆栈区域存储着活动过程调用的指令和本地变量（调用堆栈、堆栈（用于保存运行时运数中 途产生的数据））。当进程正在运行时，状态通常储存在寄存器，其他情况在存储器。

3. iPhone OS下的主线程的堆栈大小是1M，第二个线程开始都是512KB

猜想实体：

进程是管理员，管理执行指令，持有数据资源，调配线程等工作

编译过程猜想总结

1. 原则：运行时的所有调用，都是通过确切的指针完成访问的，没有一个匹配列表的过程
2. 预处理 会过滤#开头的指令，执行相应的文本替换操作
3. 编译阶段 将遇到的每个文件半编译为易于链接的描述，此时编译结果还是独立的，但方法转码时涉及到方

法堆栈的创建和局部变量的地址替换

1. 链接阶段

遇到声明描述时：则延长 运行时方法数组 ， 匹配 index 和 方法描述 到编译时 全局声明字典 中

遇到声明描述时：则插入指令使得 运行时方法数组 对应index获得该方法的真实地址

遇到调用描述时：用匹配到的 方法数组[index] 替换方法名

总结：以上的 全局声明字典/运行时方法数组 机制使得无论是先有声明后有实现，还是反过来，方法的调

用都能及时的获得地址

1. 运行前的载入

会在分配进程资源时，会持有线程首地址和相关的堆栈首地址，然后过滤载入代码中对这些首地址的访问的代码（比如 初始化代码集的访问 ， 方法数组[index] 直接替换为目的地址等）

框架

1. 框架会随着系统的启动而被加载到内存中
2. 可以有多个程序引用到框架，但是框架代码在内存中仅有一份

预编译

1. 处理#define #if #include这类#开头的语句，这些称为预编译指令。这个过程中会把.h文件和.c/.cpp文

件组合成最终交给compile过程的原文件。这个原文件是不包含任何#开头的语句的。所有#define定义的宏也会被替换，预编译程序所完成的基本上是对源程序的“替代”工作。经过此种替代，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。

1. #include的替换规则是什么？？

替换规则就是把目的文件完整Copy粘贴到#include处，.h头文件相当于有时写在main函数前面的函数声明，函数声明使得main函数的调用能写在被调用函数前，使得代码结构美观。

1. #include会在 本地目录 / 环境变量的指向的目录 / …… 查找目标文件然后copy粘贴
2. LINE标识将被解释为当前行号(十进制数)，FILE则被解释为当前被编译的C源程序的名称。
3. 包含到c源程序中的头文件可以是系统提供的，这些头文件一般被放在 /usr/include目录下。在程序中

#include它们要使用尖括号(< >)。另外开发人员也可以定义自己的头文件，这些文件一般与c源程序放

在同一目录下，此时在#include中要用双引号("")。

编译

1. 编译程序所要作得工作就是通过词法分析和语法分析，在确认所有的指令都符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。
2. 把上面那个原文件编译成.o或者VC里是.obj文件。这个文件保存了机器码化的函数、函数的描述、

全局变量的描述、乃至段的描述等等。

1. 为了转码方法，编译阶段会为每个遇到的方法维护一个 局部变量末偏移 和 局部变量字典
2. 方法中的局部变量声明，会替换为指令：

获取当前线程首地址 + 局部变量末偏移 从此处开辟局部变量

向字典添加键值对：局部变量末偏移 — 局部变量名

局部变量末偏移 = 局部变量末偏移 + 该局部变量长度

1. 当遇到局部变量的使用时

会查询 局部变量字典 获取该变量的偏移

获取当前线程首地址 + 局部变量末偏移 的地址去替换变量名

汇编

1. 汇编实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，

都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。目标文件由段组成。

1. 通常一个目标文件中至少有两个段：

代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。

数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

3. UNIX环境下主要有三种类型的目标文件：

　　 可重定位文件（静态库文件）

其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。

共享的目标文件

这种文件存放了适合于在两种上下文里链接的代码和数据。第一种是链接程序可把它与其它可重定位

文件及共享的目标文件一起处理来创建另一个 目标文件;第二种是动态链接程序将它与另一个可执行

文件及其它的共享目标文件结合到一起，创建一个进程映象。

可执行文件

它包含了一个可以被操作系统创建一个进程来执行之的文件。汇编程序生成的实际上是第一种类型的

目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

链接

1. 链接是把目标文件、操作系统的启动代码和用到的库文件进行组织形成最终生成可执行代码的过程

静态链接

在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。

动态链接

在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。

1. 把可执行程序需要的所有的编译过程产生的.o或者.obj文件组合到一起。（这里也包括.lib文件，.lib文件

件本质上就是打包的.obj文件集合）。另外连接过程还会组合一些其他数据，比如资源、可执行文件头等

1. 编译器会维护一个 初始化指令集 ，该指令集能够在载入内存后得到自动的执行，其中包含一些零碎的

单条指令，也可以是其他指令集比如 方法数组生成指令集

1. 编译阶段会维护一个临时的 全局声明字典（该字典仅在编译阶段存在，编译完成后释放）

并会维护一个 方法数组生成指令集 ，该指令集随着编译过程不断增加，目的是在载入内存时，生成一个存放方法在 方法堆栈 中指针的数组容器

1. 遇到方法的声明，会遍历检查 全局声明字典，看是否存在该声明描述（确保ID唯一就行），若不存在，

则向 方法数组生成指令集 添加一个指令：使其多一个指针存放格，并把该格的 index 和 方法描述 配对放入 全部声明字典中

1. 编译器维护了一个 方法堆栈末偏移 这个参数（仅在编译阶段存在，编译完成后释放）

后面遇到方法的实现时，会将这个方法编译为机器码，并被打包起来（某个指令使链接时跳过这些指令集）

在打包后添加指令：系统调用查询方法堆栈首地址指令 . 返回值 + 当前 方法堆栈末偏移 处加载入上面

打包好的机器码

使用方法的描述查询 全局声明字典 ，找到匹配的index，生成一条机器指令并放入 初始化指令集 中：

系统调用查询方法堆栈首地址指令 . 返回值 + 当前 方法堆栈末偏移 并赋值给 方法数组[index]

计算打包好的机器码产生的偏移，更新 方法堆栈末偏移

1. 转码方法中对其他方法的回调时

会根据被调用方法的描述，查询 全局声明字典 ，获取匹配到的index，将 方法数组[index] 替换上去

1. 遇到标准库的调用时

虽然你代码根目录没有标准库的.lib文件，但是编译器在变意思时会自动拷贝这些标准.lib文件到根目录下

并使其参与链接

上述理论使得，标准库中无论方法是否被都用到，都会被加载到 方法堆栈 中。

1. 遇到动态库调用时

在链接结束后会单独遍历 框架配置表 ，该配置表有一个 方法描述——方法偏移量

不同于遇到方法的实现时的动作，在遍历该配置表时，若匹配到 全局声明字典 有该方法，则添加 指令：

系统调动获取XXX库地址 + 方法偏移量 添加到 方法数组[index] 中

载入 — 为了追求更高的效率设定的一个阶段，可有可无

1. 因为在这时，进程已经被开辟，进程首地址 和 方法堆栈 首地址都能够获得
2. 在执行完 初始化指令集 后，方法数组 存放了所有被调用方法的真实地址（即使所有方法还没载入）
3. 这时在载入其他方法时，“可能“会过滤里面的 方法数组[index] 为真实地址，该过程也会不断使用系统调

用获得 方法数组 的地址

1. 可能在编译时就在文件替换处打上mark了，使得载入内存的时候能够迅速找到并完成替换

运行

1. 在方法实际调用时，会通过指针找到代码块，并加载到CPU寄存器中
2. 该代码块在加载时会将全局变量以堆栈的形式从线程首地址往后初始化
3. 由于在 编译阶段 ，方法中对局部变量的访问已经替换为 系统调用获得线程首地址 + 局部变量偏移
4. 所以在加载到线程后，这些局部变量的访问会直接替换为有效的地址

其他事实

1. 新建一个.h文件并添加一个方法声明（并没有实现）

在某main函数文件中引入这个 .h 文件 ， 并在main后实现这个方法

调用该方法，编译运行，成功运行

OC的消息派发猜想（带指正）

猜想依据：

1. 类的所有缓存都存在metaclass上，所以每个对象都指向一份方法缓存，而不是每一个类的object都保存 一份。即便是从父类取到的方法，通过子类调用时，也缓存在子类的元类方法缓存里。

2. 方法通过传入struct objc\_object \*与SEL来访问到

id objc\_msgSend(id self，SEL \_cmd, ...) {}

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

其中的参数id：（struct objc\_object \*）id. isa 指向的struct objc\_class \*包含自身类的元类

元类metaclass也是objc\_class其中包含3个重要列表：

struct objc\_ivar\_list \*ivars OBJC2\_UNAVAILABLE;

struct objc\_method\_list \*\*methodLists OBJC2\_UNAVAILABLE;

struct objc\_cache \*cache OBJC2\_UNAVAILABLE;

4. 其中的方法列表结构体代码为：

struct objc\_method\_list {

struct objc\_method\_list \*obsolete OBJC2\_UNAVAILABLE;

int method\_count OBJC2\_UNAVAILABLE;

#ifdef \_\_LP64\_\_

int space OBJC2\_UNAVAILABLE;

#endif

/\* variable length structure \*/

struct objc\_method method\_list[1] OBJC2\_UNAVAILABLE;}

其中的可变数组保存的结构体为：

struct objc\_method {

SEL method\_name OBJC2\_UNAVAILABLE;

char \*method\_types OBJC2\_UNAVAILABLE;

IMP method\_imp OBJC2\_UNAVAILABLE;}

其中的关键匹配参数SEL可以由”SEL aaa=@selector(eat)“方法获得，而打印SEL将获得方法名

NSLog(@"%s",@selector(testSELFunction));

5. 其中的Cache列表结构体代码为：在《iOS原理探析》中有详解

struct objc\_cache {

uintptr\_t mask; /\* total = mask + 1 \*/

uintptr\_t occupied;

cache\_entry \*buckets[1]; };

而cache\_entry的定义如下

typedef struct {

SEL name; // same layout as struct old\_method

void \*unused;

IMP imp; // same layout as struct old\_method

} cache\_entry;

6.消息派发函数是一个多参函数：id objc\_msgSend(id self，SEL \_cmd, ...)

[object　performSelector:sel　withObject: @"test" ]

猜想实体：

0. OC先编写为C语言，在执行C的编译

1. 类中OC方法格式最终将被转义为C方法，形参直接转义并添加self，类成员的访问将变为self->成员名

然后该方法的指针会被封装到一个结构体struct objc\_method中，并被放入元类方法列表中

2. 代码中向一个对象发送消息时，遵循objc\_msgSend汇编代码顺序，使用self和SEL找到那个方法并调用

SEL就是C的字符串，查找是字符串匹配的过程，找到struct objc\_method后执行里面的方法

3. 对有参函数的访问，在objc\_msgSend中应该有类似于[object　performSelector:sel　withObject: @"test" ]的传参机制。

4. 所以 [CADisplayLink displayLinkWithTarget:self selector:@selector(bounceLinkTrick)];

可以确定一个代码块，且该代码块能够访问成功访问self中的成员

-----------Runloop源码与基本概念----------------------------------------------

CFRunloop结构体 源码

1. struct \_\_CFRunLoop {

CFRuntimeBase \_base;

pthread\_mutex\_t \_lock; /\* locked for accessing mode list \*/

\_\_CFPort \_wakeUpPort; // used for CFRunLoopWakeUp

Boolean \_unused;

volatile \_per\_run\_data \*\_perRunData; // reset for runs of the run loop

pthread\_t \_pthread;

uint32\_t \_winthread;

CFMutableSetRef \_commonModes;

CFMutableSetRef \_commonModeItems;

CFRunLoopModeRef \_currentMode;

CFMutableSetRef \_modes;

struct \_block\_item \*\_blocks\_head;

struct \_block\_item \*\_blocks\_tail;

CFAbsoluteTime \_runTime;

CFAbsoluteTime \_sleepTime;

CFTypeRef \_counterpart;

};

2. struct \_block\_item {

struct \_block\_item \*\_next;

CFTypeRef \_mode; // CFString or CFSet

void (^\_block)(void);

};

CFRunloopMode结构体 源码

1. struct \_\_CFRunLoopMode {

CFRuntimeBase \_base;

pthread\_mutex\_t \_lock; /\* must have the run loop locked before locking this \*/

CFStringRef \_name; // Mode Name, 例如 @"kCFRunLoopDefaultMode"

Boolean \_stopped;

char \_padding[3];

CFMutableSetRef \_sources0;

CFMutableSetRef \_sources1;

CFMutableArrayRef \_observers;

CFMutableArrayRef \_timers;

CFMutableDictionaryRef \_portToV1SourceMap;

\_\_CFPortSet \_portSet;

CFIndex \_observerMask;

#if USE\_DISPATCH\_SOURCE\_FOR\_TIMERS

dispatch\_source\_t \_timerSource;

dispatch\_queue\_t \_queue;

Boolean \_timerFired; // set to true by the source when a timer has fired

Boolean \_dispatchTimerArmed;

#endif

#if USE\_MK\_TIMER\_TOO

mach\_port\_t \_timerPort;

Boolean \_mkTimerArmed;

#endif

#if DEPLOYMENT\_TARGET\_WINDOWS

DWORD \_msgQMask;

void (\*\_msgPump)(void);

#endif

uint64\_t \_timerSoftDeadline; /\* TSR \*/

uint64\_t \_timerHardDeadline; /\* TSR \*/

};

Source/Timer/Observer 源码

输入源

struct \_\_CFRunLoopSource {

CFRuntimeBase \_base;

uint32\_t \_bits;

pthread\_mutex\_t \_lock;

CFIndex \_order; /\* immutable \*/

CFMutableBagRef \_runLoops;

union {

CFRunLoopSourceContext version0; /\* immutable, except invalidation \*/

CFRunLoopSourceContext1 version1; /\* immutable, except invalidation \*/

} \_context;

};

定时源

struct \_\_CFRunLoopTimer {

CFRuntimeBase \_base;

uint16\_t \_bits;

pthread\_mutex\_t \_lock;

CFRunLoopRef \_runLoop;

CFMutableSetRef \_rlModes;

CFAbsoluteTime \_nextFireDate;

CFTimeInterval \_interval; /\* immutable \*/

CFTimeInterval \_tolerance; /\* mutable \*/

uint64\_t \_fireTSR; /\* TSR units \*/

CFIndex \_order; /\* immutable \*/

CFRunLoopTimerCallBack \_callout; /\* immutable \*/

CFRunLoopTimerContext \_context; /\* immutable, except invalidation \*/

};

观察者

struct \_\_CFRunLoopObserver {

CFRuntimeBase \_base;

pthread\_mutex\_t \_lock;

CFRunLoopRef \_runLoop;

CFIndex \_rlCount;

CFOptionFlags \_activities; /\* immutable \*/

CFIndex \_order; /\* immutable \*/

CFRunLoopObserverCallBack \_callout; /\* immutable \*/

CFRunLoopObserverContext \_context; /\* immutable, except invalidation \*/

};

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

};

获取Runloop对象的方法 源码

/// 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());

}

RunloopRun方法 源码

/// 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; //协助判断是否跳出循环的参数

int retVal = 0; //是否跳出循环的判断参数

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 如果是被Timer唤醒的，回调Timer

if (msg\_is\_timer) {

\_\_CFRunLoopDoTimers(runloop, currentMode, mach\_absolute\_time())

}

/// 9.2 如果是被dispatch唤醒的，执行所有调用 dispatch\_async 等方法放入main queue 的 block

else if (msg\_is\_dispatch) {

\_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_(msg);

}

/// 9.3 如果Runloop是被 Source1 (基于port的) 的事件唤醒了，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);

///查询各种状态判定是否退出runloop

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);

}

RunloopRun运行时成员 源码

CFRunLoop {

//设定当前mode为默认mode（通过字符串）

current mode = kCFRunLoopDefaultMode

//设定 滑屏mode 和 默认mode 都接收commonItems（通过字符串）

common modes = {

UITrackingRunLoopMode

kCFRunLoopDefaultMode

}

//通用source列表

common mode items = {

// source0 (manual)

//source0不带有port端口，难道表示每次循环直接执行这些操作？

// 应用需要处理的事件列表（application 来 handle 事件列表？？）

CFRunLoopSource { order =-1, {callout = \_UIApplicationHandleEventQueue } }

// 紫色事件信号的回调

CFRunLoopSource { order =-1, {callout = PurpleEventSignalCallback } }

// FBS的SerialQueue 、 RunLoopSource 的 Handler ？？

CFRunLoopSource { order = 0, {callout = FBSSerialQueueRunLoopSourceHandler} }

// source1 (mach port)

// source都必须带port(有一个特例)，order可能表示优先级，没有callout的可能仅作为唤醒用

CFRunLoopSource {order = 0, {port = 17923}}

CFRunLoopSource {order = 0, {port = 12039}}

CFRunLoopSource {order = 0, {port = 16647}}

CFRunLoopSource {order = 0, {port = 2407,callout = \_ZL20notify\_port\_callbackP12\_\_CFMachPortPvlS1\_}}

CFRunLoopSource {order = 0, {port = 1c03,callout = \_\_IOHIDEventSystemClientAvailabilityCallback}}

CFRunLoopSource {order = 0, {port = 1b03, callout = \_\_IOHIDEventSystemClientQueueCallback}}

CFRunLoopSource {order = 1, {port = 1903,callout = \_\_IOMIGMachPortPortCallback}}

//这个可能跟performSelector有关，直接插入代码执行

CFRunLoopSource {order =-1, {callout = PurpleEventCallback}}

// Ovserver

// activities 用来表达监视什么状态，有多个监视者(回调)监视同一个状态

CFRunLoopObserver {order = -2147483647, activities = 0x1, // Entry

callout = \_wrapRunLoopWithAutoreleasePoolHandler}

CFRunLoopObserver {order = 0, activities = 0x20, // BeforeWaiting

callout = \_UIGestureRecognizerUpdateObserver}

CFRunLoopObserver {order = 1999000, activities = 0xa0, // BeforeWaiting | Exit

callout = \_afterCACommitHandler}

CFRunLoopObserver {order = 2000000, activities = 0xa0, // BeforeWaiting | Exit

callout = \_ZN2CA11Transaction17observer\_callbackEP19\_\_CFRunLoopObservermPv}

CFRunLoopObserver {order = 2147483647, activities = 0xa0, // BeforeWaiting | Exit

callout = \_wrapRunLoopWithAutoreleasePoolHandler}

// Timer

CFRunLoopTimer {firing = No, interval = 3.1536e+09, tolerance = 0,

next fire date = 453098071 (-4421.76019 @ 96223387169499),

callout = \_ZN2CAL14timer\_callbackEP16\_\_CFRunLoopTimerPv (QuartzCore.framework)}

},

//所有mode的存放处（实际的结构实体）

modes ＝ {

CFRunLoopMode {

sources0 = { /\* same as 'common mode items' \*/ },

sources1 = { /\* same as 'common mode items' \*/ },

observers = { /\* same as 'common mode items' \*/ },

timers = { /\* same as 'common mode items' \*/ },

},

CFRunLoopMode {

sources0 = { /\* same as 'common mode items' \*/ },

sources1 = { /\* same as 'common mode items' \*/ },

observers = { /\* same as 'common mode items' \*/ },

timers = { /\* same as 'common mode items' \*/ },

},

CFRunLoopMode {

sources0 = { CFRunLoopSource {order = 0, { callout = FBSSerialQueueRunLoopSourceHandler}} },

sources1 = (null),

observers = {

CFRunLoopObserver >{activities = 0xa0, order = 2000000,

callout = \_ZN2CA11Transaction17observer\_callbackEP19\_\_CFRunLoopObservermPv}

)},

timers = (null),

},

CFRunLoopMode {

sources0 = {

CFRunLoopSource {order = -1, {

callout = PurpleEventSignalCallback}}

},

sources1 = {

CFRunLoopSource {order = -1, {

callout = PurpleEventCallback}}

},

observers = (null),

timers = (null),

},

CFRunLoopMode {

sources0 = (null),

sources1 = (null),

observers = (null),

timers = (null),

}

}

}

RunloopRun方法名列表 源码

{

/// 1. 通知Observers，即将进入RunLoop

/// 此处有Observer会创建AutoreleasePool: \_objc\_autoreleasePoolPush();

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopEntry);

do {

/// 2. 通知 Observers: 即将触发 Timer 回调。

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopBeforeTimers);

/// 3. 通知 Observers: 即将触发 Source (非基于port的,Source0) 回调。

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopBeforeSources);

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_BLOCK\_\_(block);

/// 4. 触发 Source0 (非基于port的) 回调。

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_SOURCE0\_PERFORM\_FUNCTION\_\_(source0);

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_BLOCK\_\_(block);

/// 6. 通知Observers，即将进入休眠

/// 此处有Observer释放并新建AutoreleasePool: \_objc\_autoreleasePoolPop(); \_objc\_autoreleasePoolPush();

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopBeforeWaiting);

/// 7. sleep to wait msg.

mach\_msg() -> mach\_msg\_trap();

/// 8. 通知Observers，线程被唤醒

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopAfterWaiting);

/// 9. 如果是被Timer唤醒的，回调Timer

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_TIMER\_CALLBACK\_FUNCTION\_\_(timer);

/// 9. 如果是被dispatch唤醒的，执行所有调用 dispatch\_async 等方法放入main queue 的 block

\_\_CFRUNLOOP\_IS\_SERVICING\_THE\_MAIN\_DISPATCH\_QUEUE\_\_(dispatched\_block);

/// 9. 如果如果Runloop是被 Source1 (基于port的) 的事件唤醒了，处理这个事件

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_A\_SOURCE1\_PERFORM\_FUNCTION\_\_(source1);

} while (...);

/// 10. 通知Observers，即将退出RunLoop

/// 此处有Observer释放AutoreleasePool: \_objc\_autoreleasePoolPop();

\_\_CFRUNLOOP\_IS\_CALLING\_OUT\_TO\_AN\_OBSERVER\_CALLBACK\_FUNCTION\_\_(kCFRunLoopExit);

}

Mach相关 源码

消息本体(一条 Mach 消息实际上就是一个二进制数据包 (BLOB))

typedef struct {

mach\_msg\_header\_t header;

mach\_msg\_body\_t body;

} mach\_msg\_base\_t;

报头（头部定义了当前端口 local\_port 和目标端口remote\_port）

typedef struct {

mach\_msg\_bits\_t msgh\_bits;

mach\_msg\_size\_t msgh\_size;

mach\_port\_t msgh\_remote\_port;

mach\_port\_t msgh\_local\_port;

mach\_port\_name\_t msgh\_voucher\_port;

mach\_msg\_id\_t msgh\_id;

} mach\_msg\_header\_t;

发送和接受消息是通过同一个 API 进行的，其 option 标记了消息传递的方向：

mach\_msg\_return\_t mach\_msg(

mach\_msg\_header\_t \*msg,

mach\_msg\_option\_t option,

mach\_msg\_size\_t send\_size,

mach\_msg\_size\_t rcv\_size,

mach\_port\_name\_t rcv\_name,

mach\_msg\_timeout\_t timeout,

mach\_port\_name\_t notify);

mach\_msg与底层概念

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

2. 图解



分层概述：

应用层包括用户能接触到的图形应用，例如 Spotlight、Aqua、SpringBoard 等。

应用框架层即开发人员接触到的 Cocoa 等框架。

核心框架层包括各种核心框架、OpenGL 等内容。

Darwin 即操作系统的核心，包括系统内核、驱动、Shell 等内容，这一层是开源的

Darwin核心概述

在硬件层上面的三个组成部分：Mach、BSD、IOKit (还包括一些其他内容)，共同组成XNU 内核。

XNU 内核的内环被称作 Mach，其作为一个微内核，仅提供了诸如处理器调度、IPC (进程间通信)等非常 少量的基础服务。

BSD 层可以看作围绕 Mach 层的一个外环，其提供了诸如进程管理、文件系统和网络等功能。

IOKit 层是为设备驱动提供了一个面向对象(C++)的一个框架。

3. Mach — 查看源码

在 Mach 中，所有的东西都是通过自己的对象实现的，进程、线程和虚拟内存都被称为"对象"。和其他架构不 同， Mach 的对象间不能直接调用，只能通过消息传递的方式实现对象间的通信。"消息"是 Mach 中最 基础的概念，消息在两个端口 (port) 之间传递，这就是 Mach 的 IPC (进程间通信) 的核心。

1. 一条 Mach 消息实际上就是一个二进制数据包 (BLOB)，其头部定义了当前端口 local\_port 和目标端口 remote\_port

2. 发送和接受消息是通过同一个 API 进行的，其 option 标记了消息传递的方向

3. 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

4. iOS的进程间通信：https://segmentfault.com/a/1190000002400329

RunLoop知识点

获取Runloop

线程和 RunLoop 之间是一一对应的，其关系是保存在一个全局的 Dictionary 里。线程刚创建时并没有 RunLoop，如果你不主动获取，那它一直都不会有。RunLoop 的创建是发生在第一次获取时，RunLoop 的 销毁是发生在线程结束时。你只能在一个线程的内部获取其 RunLoop（主线程除外）。

RunloopMode

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。

系统默认注册了5个Mode:

1. kCFRunLoopDefaultMode: App的默认 Mode，通常主线程是在这个 Mode 下运行的。

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: 这是一个占位的 Mode，没有实际作用。

RunLoopSource

是事件产生的地方。Source有两个版本：Source0 和 Source1。

Source0 只包含了一个回调（函数指针），它并不能主动触发事件。使用时，你需要先调用 CFRunLoopSourceSignal(source)，将这个 Source 标记为待处理，然后手动调用 CFRunLoopWakeUp(runloop) 来唤醒 RunLoop，让其处理这个事件。

Source1 包含了一个 mach\_port 和一个回调（函数指针），被用于通过内核和其他线程相互发送消息。这种 Source 能主动唤醒 RunLoop 的线程，其原理在下面会讲到。

RunLoopTimer

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。

RunLoopObserver

观察者，每个 Observer 都包含了一个回调（函数指针），当 RunLoop 的状态发生变化时，观察者就能通过 回调接受到这个变化。

CommonModes

一个 Mode 可以将自己标记为"Common"属性（通过将其 ModeName 从modes中添加到 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 自动同步到具有 "Common" 标记的所有Mode里。

管理Common的接口

CFRunLoopAddCommonMode(CFRunLoopRef runloop, CFStringRef modeName);

CFRunLoopRunInMode(CFStringRef modeName, ...);

当你传入一个新的 mode name 但 RunLoop 内部没有对应 mode 时，RunLoop会自动帮你创建对 应的 CFRunLoopModeRef。对于一个 RunLoop 来说，其内部的 mode 只能增加不能删除。

管理ModeItem的接口

CFRunLoopAddSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);

CFRunLoopAddObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);

CFRunLoopAddTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);

CFRunLoopRemoveSource(CFRunLoopRef rl, CFRunLoopSourceRef source, CFStringRef modeName);

CFRunLoopRemoveObserver(CFRunLoopRef rl, CFRunLoopObserverRef observer, CFStringRef modeName);

CFRunLoopRemoveTimer(CFRunLoopRef rl, CFRunLoopTimerRef timer, CFStringRef mode);

-----------应用Runloop所实现的功能----------------------------------------------

AutoreleasePool概述

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。

事件响应概述（有网址英文拓展）

苹果在SpringBoard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 事件都是在这个发送给 UIWindow后的回调中完成的。

http://iphonedevwiki.net/index.php/IOHIDFamily

Users of IOHID always first create an IOHIDEventSystem object that interfaces with the whole HID system. An event system consists of multiple IOHIDServices and IOHIDDisplays.

IOHID defines 20 types of events, of which the SpringBoard only handles 4 of them: keyboard (for buttons), digitizer (i.e. touchscreen), accelerometer and proximity events (temperature events are handled by IOKit directly). Each event may contain sub-events.

Create example

void handle\_event (void\* target, void\* refcon, IOHIDServiceRef service, IOHIDEventRef event)

{

*// handle the events here.*

printf("Received event of type %2d from service %p.**\n**", IOHIDEventGetType(event), service);

}

int main () {

*// Create and open an event system.*

IOHIDEventSystemRef system = IOHIDEventSystemCreate(**NULL**);

IOHIDEventSystemOpen(system, handle\_event, **NULL**, **NULL**, **NULL**);

printf("HID Event system should now be running. Hit enter to quit any time.**\n**");

getchar();

IOHIDEventSystemClose(system, **NULL**);

CFRelease(system);

return 0;

}

坑爹：Since at least iOS 6.1 there is a check in the function IOHIDEventSystemCreate that compares the bundle identifier with "com.apple.springboard". If that check fails, the function will return NULL. Therefore you can no longer use the HID system in apps unless you circumvent this check.

手势识别概述

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

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

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

界面更新概述

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。

调用栈看网址，看了你也看不懂

看来是多次循环处理完所有事件才会想着去更新界面啊

定时器概述

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

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

CADisplayLink 是一个和屏幕刷新率一致的定时器（但实际实现原理更复杂，和 NSTimer 并不一样，其内部实际 是操作了一个 Source）。如果在两次屏幕刷新之间执行了一个长任务，那其中就会有一帧被跳过去（和 NSTimer 相似），造成界面卡顿的感觉。在快速滑动TableView时，即使一帧的卡顿也会让用户有所察觉。 Facebook 开源的 AsyncDisplayLink 就是为了解决界面卡顿的问题，其内部也用到了 RunLoop，这个稍后 我会再单独写一页博客来分析 —— 原理探究中 《屏幕渲染原理 与 流畅度优化》

PerformSelecter概述

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。

用到了GCD的概述

实际上 RunLoop 底层也会用到 GCD 的东西，比如 RunLoop 是用 dispatch\_source\_t 实现的 Timer（评论中 有人提醒，NSTimer 是用了 XNU 内核的 mk\_timer，我也仔细调试了一下，发现 NSTimer 确实是由 mk\_timer 驱动，而非 GCD 驱动的）。但同时 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\_\_() （见方法列表源码里）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。

关于网络请求的简单举例

1. iOS 中，关于网络请求的接口自下至上有如下几层:

CFSocket

CFNetwork       ->ASIHttpRequest

NSURLConnection ->AFNetworking

NSURLSession    ->AFNetworking2, Alamofire

• CFSocket 是最底层的接口，只负责 socket 通信。

• CFNetwork 是基于 CFSocket 等接口的上层封装，ASIHttpRequest 工作于这一层。

• NSURLConnection是基于CFNetwork 更高层的封装，提供面向对象的接口，AFNetworking 工作于这层。

• NSURLSession 是 iOS7 中新增的接口，表面上是和 NSURLConnection 并列的，但底层仍然用到了 NSURLConnection 的部分功能 (比如 com.apple.NSURLConnectionLoader 线程)，AFNetworking2 和 Alamofire 工作于这一层。

2. NSURLConnection 的工作过程。

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。

AFNetworking的应用举例

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：

下述代码使得调用networkRequestThread的线程被新建子线程、子线程命名为AFNetworking、添加port、 runWithPort

+ (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;

}

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

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

- (void)start {

[self.lock lock];

if ([self isCancelled]) {

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

} else if ([self isReady]) {

self.state = AFOperationExecutingState;

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

}

[self.lock unlock];

}

AsyncDisplayKit的应用举例

1. 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。

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。

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。

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。

2. 排版和绘制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)

3. 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

4. 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。

5. 总结：

将排版和绘图这些数值计算（还有图片解压，直接绘制成图片二进制流等）并发交给CPU去处理，然后 写入对应对象的属性中。

不能脱离主线程的比如排版好的文本，绘制好的图片，frame的更改这些就绪数值的渲染展示操作，都累积等 待最近的数值设定完毕后一次性交由GPU渲染

-----------Runloop归总猜想（待指正）----------------------------------------------

Run Loop基本猜想

猜想依据：

1. 使用 Dispatch Source 调试感知时+断点暂停时，插入的代码是UI修改代码

UI修改只能在主线程生效，而传入参数main\_queue说明可以在线程后追加执行代码

需要找到主线程插入，说明这个监听线程是异步的子线程

综上两条：说明存在API能够在其他线程中获取某个线程句柄后，向其插入代码并让其运行这个代码

2.可以通过插入信号量来阻塞当前线程：dispatch\_semaphore\_wait(semaphore, timeoutTime)

3. 可以使用pthread\_cond\_wait与cond信号控制来阻塞线程

4. Dispatch Source就是一个低级函数的 grab-bag ，能帮助你去响应或监测 Unix 信号、文件描述符、 Mach 端口、 VFS节点，以及其它晦涩的东西。

dispatch\_source\_t 、dispatch\_source\_create 、dispatch\_source\_set\_event\_handler、 dispatch\_resume(source) 说明了一个简单的监听事件触发向执行线程添加handler代码执行的过程

5. 当在当前线程通过getCurRunloop获得runloop对象时，方法内部发生了一个向一个持久字典添加 (PID)pthread\_self() - runloop对象的键值对的过程（具体代码见《Runloop详述》）

6. 《Runloop源码》中runloop对象的组成可以看看

猜想实体：

0. 目的与核心机制：

Runloop的根本目的是让当前线程不被结束和释放，并让其没有工作时休眠，有消息时被唤醒

runloop是采用mach\_port等待其他线程传递port信号来唤醒的

1. 目前参与的演员：

主线程 、runloop对象 、runloopRun 、mode/ modes 、source0 、 source1 、 commonModes/modeItems 、source控制线程 、

2. 如何让对应线程“休眠/唤醒”

当你在代码中插入runloopRun方法时，该方法就获得了当前线程的控制权，上面的隐式调用就发生在 runloopRun方法循环体执行的某个过程中

runloopRun在检测到没有需要执行的回调后，进入port信号等待阻塞中，有信号就执行对应的方法，对 应方法中去寻找回调函数的指针执行，达到主线程 唤醒 - 插入执行流 - 运行 这一机制

3. 事件源source1

runloop对象包含多个模式，每个模式包含多个源，事件源会被添加到runloop对象中的某个模式下。

source1持有需要port的ID（可能是port来源线程的地址）

若切换了模式，source线程是不受影响的，只是由于新mode不包含该源，所以不会信号不会激发回调

4. 消息处理机制

线程通信互相之间仅有一个端口，但是线程接到消息后会根据信息分发

每次循开始，source0总能被处理，开始Timer和source 的observe也总能被发出

在休眠时等待是任何种类的port，任何发向该线程唯一端口的port都能唤醒线程

每次循环仅能处理一个port消息，种类分为：timer，GCD强插，source1

处理过程中若有新port发来，会被放入先进先出的队列，且根据port发往哪个source1保持最新唯一性

处理一个port时会抽取portID来匹配对应列表的源，匹配到的回调其方法，这个过程也是串行的

处理完后会执行一些不必要的检查看是否终止下一次循环

若可以进行下一次循环，就检查队列中是否还有ready的port，有的话就处理，没有就休眠

5. 根据上述机制的拓展

由于休眠前会有回调（GPU渲染），该回调时间很长，所以期间可能会有新port信号加入队列而runloop 还是进入休眠得不到处理，只有在休眠后，又有port加入时才能唤醒，处理任何未处理的消息

定时器 & DisplayLink猜想

依据 ：

1. 定时器概述 与 界面更新概述：界面更新放在多次循环处理完所有事件进入休眠前才做

2. 原理探究中 《屏幕渲染原理 与 流畅度优化》 提及的CPU + GPU + VSign垂直同步循环

实体：

1. 定时器

定时器实际是一个永不休止的循环，不断循环检查注册在列表中的定时源是否和当前时间匹配

由于匹配列表是需要时间的，所以定时源一般不是精确匹配，都有一个精确度，所以匹配都是 >= 或 <=

匹配到后就去在当前线程回调匹配对象携带的方法指针

轮播定时器可能在每次回调中向列表注册下一个定时器，而不是都注册（也不可能），以减小列表匹配压力

轮播定时器对下一个定时源的注册可以放在回调最开始和最后，放最开始会使若任务过长错过下一个回调， 而丧失第3个任务定时源的创建，放在最后会使得定时源分隔不均匀

但一般定时器的回调是仅包含创建新定时源的代码，定时器还是专注于发信号比较好，延迟交给其他线程

2. DisplayLink界面更新

首先定时信号来自硬件的垂直同步，信号产生时说明所有渲染资源准备就绪（时钟固定，不受任务影响）

此时runloop被唤醒，或者该消息排队在其他任务后面，直到回调方法进行path计算和赋值等工作，这 一步其实很快的，但path赋值对CALayer的改变的绘图是放在所有信息处理完毕后处理的

假设多次循环后最后一个回调就是path的计算和赋值，然后即将进入休眠，runloop休眠前回调开始将 刚才提交全局容器的CALayer渲染，但不论是CPU 来渲染还是GPU，耗时都是大于1/60的（期间 Vsign的信号被不断发向Runloop的某Source，但由于已经引入入眠回调，最新信号被放入队列中。

这时间段的Vsign到达时将喷图旧的buffer的内容，displayLink也没被回调，造成“丢帧”）

然后CPU+GPU忙完后，也就是displayBuffer填满后，runloop进入休眠（注意此时喷枪还没开始工作）

render Gun可能是在Vsign前一点点时间的固定时钟频率读取当前buffer渲染的，该时间内，将锁死 buffer的交换（也就是说保证不撕裂），且交换操作不是在该时间内阻塞，而是被丢弃

所以若CPU+GPU将displayBuffer填满正好发生在renderGun喷图时间，那么其交换buffer的操作会 被丢弃。而此时主runloop已经进入waiting，在renderGun喷图完成后的Vsign将唤醒waiting 来完成新视图的绘制，刚才的buffer早晚被覆盖，renderGun在新一轮的path计算赋值的延迟中 继续渲染旧buffer，最终导致“丢帧”

但上面情况概率很低，displayBuffer的填满大多发生在renderGun工作前，假设正好在Vsign后完成 buffer交换，该buffer也得等到1/60秒后的下一次喷图（注意，此时Vsign的port被更新到runloop 消息队列中，但由于是在GPU提交前更新的，所以runloop还是进入了休眠，直到新Vsign唤醒才 开始绘图）

所以大部分的丢帧都归因于：（以下原因不存在依赖，任意组合）

串行处理消息时的其他任务的拖延 、单处理自己操作时的拖延（一般不可能） 、

休眠回调时GPU渲染的拖延（主要原因）

ASDK & displayLink优化猜想

1. 绘图计算与赋值后需要渲染的CALayer的提交这些操作，不要被消息队列中的其他任务阻塞

2. 许多不互相依赖的绘图计算可以并行化的

3. GPU可不可以并行我不知道，CPU可以分担并行化许多GPU的任务（比如圆角矩形绘制造成的离屏渲染，在迅 速滑动屏幕时，CPU仅有告知绘制圆角的任务，而GPU却要一个个重绘，明明能采用遮罩层复用来解决的， 或者使用CPU分担一半的绘制任务）

4. 为了能够不一设置frame就使得CALayer被提交到渲染队列里等待渲染（每一次frame设置啥的必定会触发提 交操作，也许为了防止重复渲染剔除队列中旧的同CALayer对象，但剔除操作本身就很耗时间），ASDK采用 node节点，在合适的时候将之前的frame的所有更改一次性set进去，减少frame的set方法的调用次数

事件传递 & 主线程工作模式猜想

依据：

1. 事件响应概述 中提及的一些信息

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

@autoreleasepool {

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

}

}

3. NULL

实体：

1. 事件传递

点击rootController下的tabFirst按钮，会触发系统的硬件发出信号，该信号由 IOKit.framework 生成 一个 IOHIDEvent 事件并传递给SpringBoard的进程 （springBorard的进程中有一个 IOHIDEventSystem来处理该事件），SpringBoard进程再通过mach port 转发给需要的App进程 App进程中的Runloop被该信息唤醒，执行\_UIApplicationHandleEventQueue()回调，该方法 会把 IOHIDEvent 处理并包装成 UIEvent，转发给APP中的各个对象

这种转发机制估计是根据触屏所在坐标，找到该坐标下最顶层的UI对象，然后依据某种优先级（沿着该 坐标的UI层栈 或 沿着顶层UI对象所在树往根方向）传递该Event对象，被捕获则不再继续传递，

2. 主线程工作模式

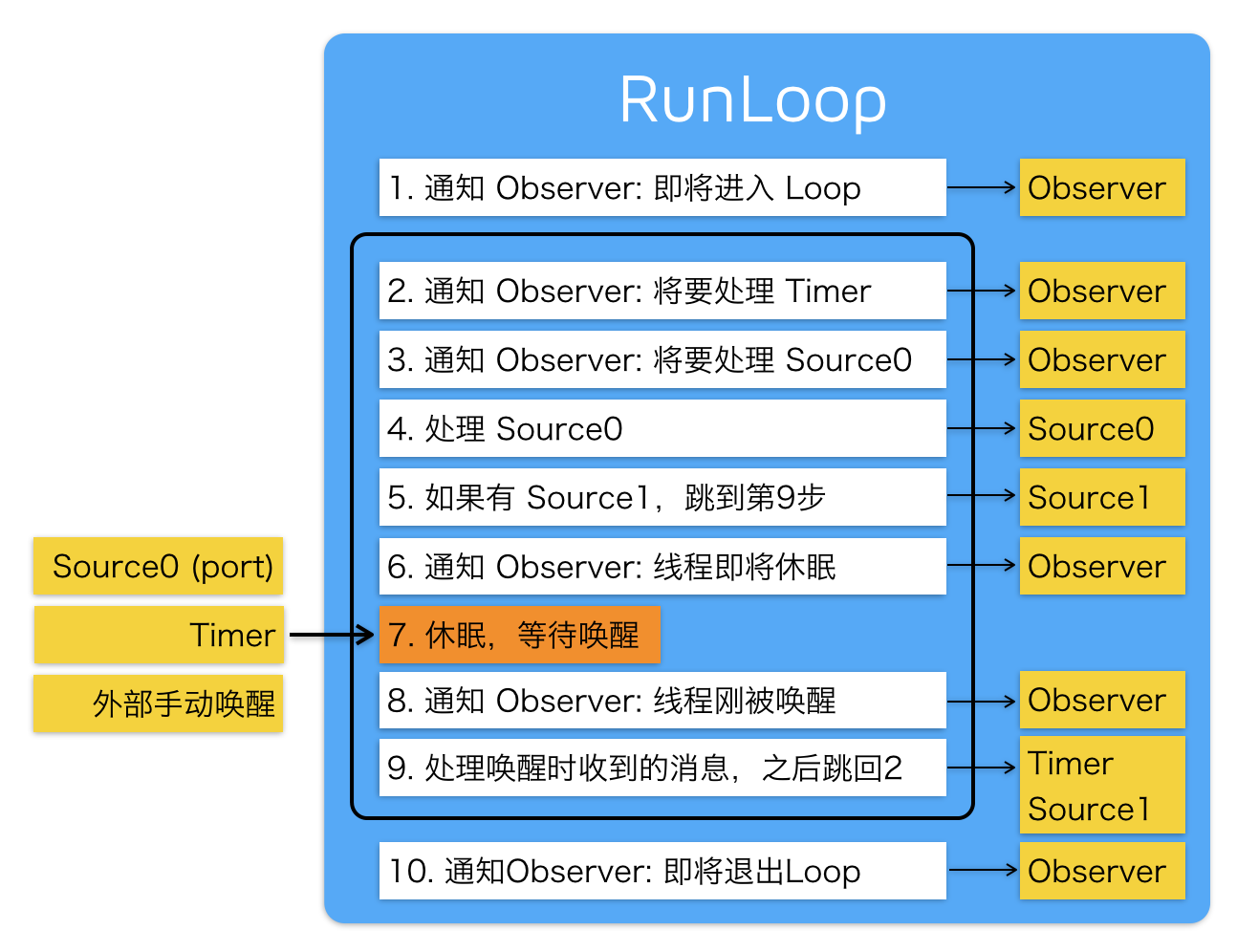
主线程一开始加载时，串行执行完UIApplicationMain中所有必要的代码后，再调用rootController的 layout逐层进行渲染配置，然后才进入runloop模式，此时runloop进入即将休眠的图形渲染方法 中，在该方法中APP的第一个界面显示完成

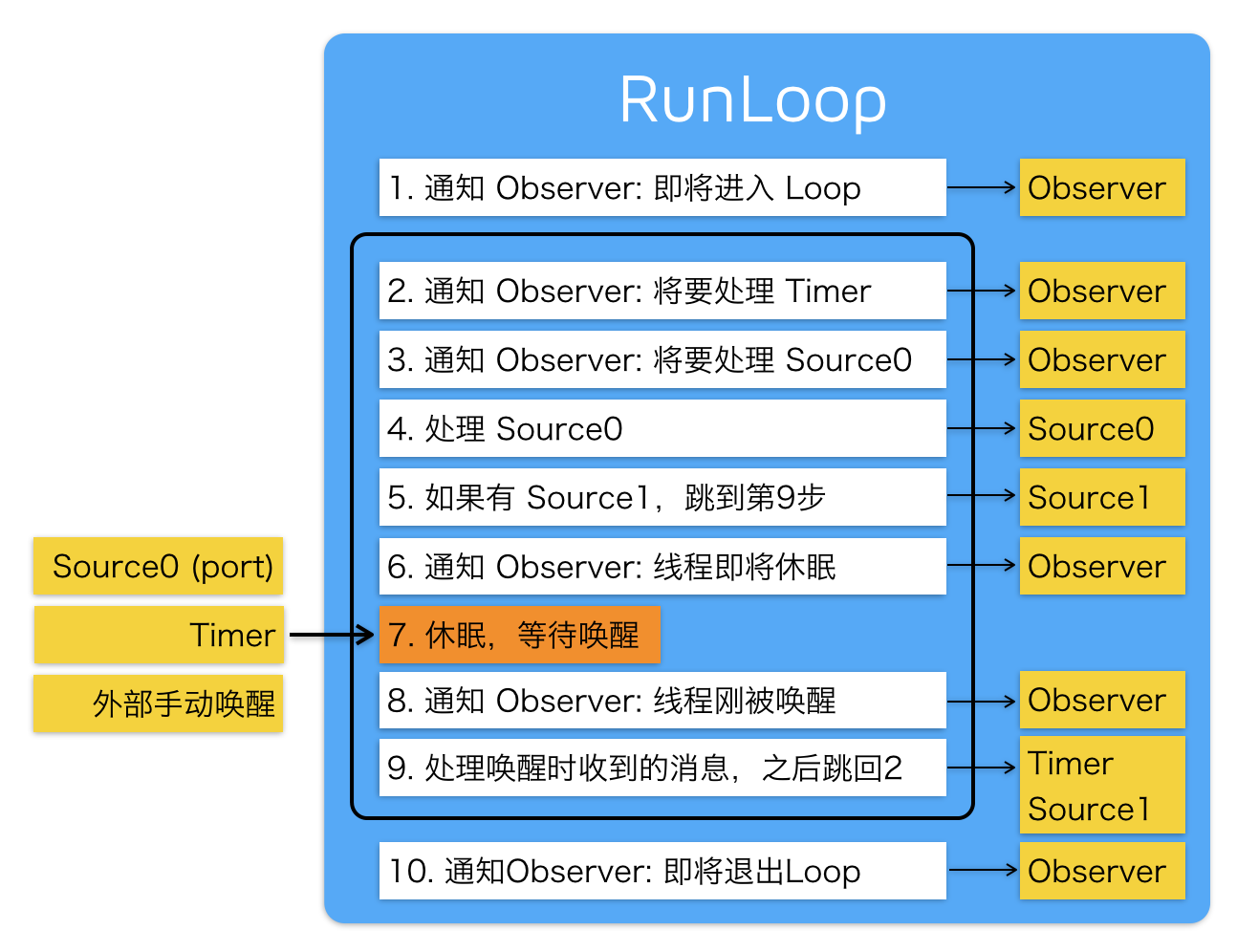
然后的controller中的代码，都是在通过各种各样的方式插入到主线程runloop中运行的

比如点rootController下First按钮，就是主runloop被springBoard的发来的port唤醒，然后通过回 调\_UIApplicationHandleEventQueue()，封装为UIEvent，沿事件链查询，最后逐渐回调到实际处 理该事件的方法的，这明显是一个 唤醒 - 插入执行流 - 运行 的过程

RunloopRun处理过程猜想

处理顺序猜想





1. RunloopRun的进入参数有：runloop，需要进入的模式，seconds？？，是否第一次处理完事件后就退出

2. 查询是否有有source/timer/observer, 没有就直接返回（违法的mode）

3. 运行的各个阶段前，会批量通知observer。某些操作后，会执行“被？？加入的block”

…… 进入循环 ……

4. 启动任何准备好的非基于端口的源

RunLoop 触发 Source0 (非port) 回调。并返回“是否源控制loop”的Boolean

5. 查看消息队列是否有需要处理的，其他线程通过唯一端口发来port消息

通过检测全局变量“\_\_Source0DidDispatchPortLastTime”和“dispatchPort”查询是否有要处理消息 若没有： 顺序执行到第6步

若有： 跳到第7步

6. 如果5为否，且并不是“源控制loop”，那么阻塞线程，等待如下方式唤醒

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。

7. 处理未处理的事件

如果新消息是被定时器唤醒，执行timer回调（多个Timer怎么办？），进入步骤 8。

如果是被dispatch唤醒的，执行所有用 dispatch\_async 等方法放入main queue 的 block，进步骤 8

如果如果Runloop是被 Source1 (基于port的) 的事件唤醒了，处理这个事件，处理完毕后进入步骤 8

8. 判断 “是否完事就退出loop”、“超时”、“刚被设定runloop来停止”、“当前mode没source了”来继续 是： 跳到4继续循环

不是： 退出runloop，移交线程在此处的控制权

…… 继续/退出循环 ……

runloopRun机制猜想

1. runloop似乎通过某种方法检查全局消息栈顶的消息种类，然后在当前循环中仅执行这一个消息的回调，期 间可能会有新的消息被放入栈中。最终栈总会被处理到底，进入休眠。而每次循环仅对一个消息处理，所 以每处理一个消息都会回调一大堆固定的方法（包括通知要处理源了，要干嘛干嘛了，检查在处理这个消 息后是否退出runloop了等等）

2.

MyRunloop猜想与设计

1. 使用wait来完成线程的阻塞与释放，将wait的信号控制器cond与mutex放在全局变量供其他线程访问和控制

该wait是一个死循环，执行完指定内容后继续进入wait

2. 配备一个全局变量 （void \*）callback（void），用来存放当前需要回调的函数，wait唤醒后执行的就是callback存 放的函数代码

3. 当一个线程需要唤醒主线程执行任务的时候，通过向callback放入方法指针，然后释放cond来完成指定方法的插入

4. 若是有多个线程需要唤醒主线程，那么只能阻塞并等待cond的释放。在这里必须让callback回调完后立刻进入wait 状态，不能让主线程和其他线程对cond的访问一同竞争。也就是说 在其中一个线程竞争到执行权后，到下一个 wait阻塞构造完前，其他线程不能和这两个线程竞争

5. 所以这里设计为：wait函数前不需要锁，wait等待构造完后释放锁(内部实现的)然后阻塞并监听，其他线程竞争这个 锁，获得锁的线程修改callback，激活cond后释放锁，wait被激活后需要锁(内部实现的)，wait返回后不释方锁(内 部实现的)，执行callback后不释放锁，顺序执行到第二次循环wait函数构造处

上述设计能够在执行callback后不放锁直接顺序执行第二次循环wait，但问题出在激活cond这一块，由于wait激 活后必须获得锁，所以之前必须被释放，所以这边出现了主线程和其他线程的竞争，其他线程竞争到后更改callback， 改完后又释放锁又进入所有线程的竞争模式，直到最后只剩主线程和一个其他线程，才能让这最后一个线程的 callback 得到执行

7. 所以我需要一种wait，在激活后释放阻塞前不需要任何锁：（要求其他线程mycond的激活需要放在最后一行）

手动在阻塞方法构造函数前放锁mymutex（在循环中仅执行一次），mywait函数构造完后，在内部释放

mymutex， 然后仅贴着检测mycond，若为未激活状态，则在此阻塞线程（释放锁后再阻塞也不惧怕竞争， 详情见 DeepInComputer — 《Linux C线程的创建与控制》）

其他线程通过获取mymutex互相竞争，竞争到的线程执行callback设定和mycond的激活，mycond的激活放 在线程最后面，且激活后不释放mymutex

mywait在检测到激活并return来放行主线程，这个动作不需要mymutex，此时mymutex依旧没有放开

主进程执行当前callback后也不释放mymutex，顺序执行第二次循环开头的mywait函数构造

mywait函数构造完后，在内部释放mymutex，检测mycond，阻塞主线程，其他线程开始竞争

8. 上述思想使得无论有多少其他线程，对cond的激活都需要竞争，没有先后。

第一个线程激活后的callback可能很长，假设有个3，4秒吧，之后的所有cond锁都要阻塞和互相竞争

主线程每次仅接受一个线程的一次激活，插入执行的方法也仅有一个（即使一个线程有多次激活也得竞争）

10. 但进阶目的是：

即使每次仅接受一个线程的一次激活，执行一个方法，也得有先来后到

若一次唤醒后的callback执行过程中有其他消息到来，也不用等到下一次的wait，能够一并执行了，直到全局的 callback列表 中没有方法了才进入下一轮的wait，节省构建wait的时间

这和实际的runloop有点相近了，实际runloop似乎通过某种方法检查全局消息栈顶的消息种类，然后在当前循环 中仅执行这一个消息的回调，期间可能会有新的消息被放入栈中。最终栈总会被处理到底，进入休眠。而每次 循环仅对一个消息处理，所以每处理一个消息都会回调一大堆固定的方法（包括通知要处理源了，要干嘛干嘛 了，检查在处理这个消息后是否退出runloop了等等）

11. 再次进阶：

上述进阶runloop的缺点就是每次只能处理一个消息，一次回调，这些执行过程都是串行的

但是每次绘图都在休眠的时候，然而你的frame变更被夹杂在一堆非绘图事务中

当这些事务不能保证在1/60秒内完成时。我的帧数就变得很低了，这是不合理的。

我希望绘图的事务能够和其他事务区别开并得到并行处理 和 及时的渲染

并且是在最近的绘图设定都完成后及时执行而不是每次设定完都渲染（原先通过休眠前调用来保证最近设定都完成）

甚至是其他不互相依赖的事务都能够得到并行优化

排版，绘制，解压图片等处理也不是一定要串行的，这些也可以并行计算

-----------屏幕渲染原理 与 ASDK优化---------------------------------------------------------

http://www.cnblogs.com/ioriwellings/p/5011993.html

《Runloop归总猜想》 — 定时器 & DisplayLink猜想 和 ASDK & displayLink优化猜想

说在前面（进阶研究）

1. 你需要查看UI类的实现方式，明白到底哪些是CPU在做，哪些是GPU在做，哪些计算重复了（比如圆角矩形 绘制造成的离屏渲染，在迅速滑动屏幕时，CPU仅有告知绘制圆角的任务，而GPU却要一个个重绘，明明能 采用遮罩层复用来解决的，原先以为新cell产生只是麻烦在cell对象的生成上，殊不知难在新cell的绘制上） 2. 因为线程不安全，所以UIView只能在主线程创建和修改

MyWavePull优化的可能

1. 尽量使每个图层的各个背景颜色，fillColor的alpha通道为1，以避免不必要的通道计算

2. 减少不必要的frame属性

硬件渲染原理

0. CPU—>总线 —RenderRequest—> GPU —> BufferOne/Two —> Controller（RenderGun） —> Monitor

1. CPU提供的RenderRequest Package总是在总线中存在来供GPU处理（假设 1/600 更新频率吧）

2. CPU+GPU对每个包的处理时间为 cgt = 1/30

3. Render Gun的喷图时间为 rt = 1/120 ，要快于处理时间

4. Render Gun会以时钟频率 t=1/60 读取Buffer中的内容刷新，若是Buffer被锁，那么画面维持最后一次刷新

一个缓存的时候：

1. 第一个包交给GPU处理，GPU锁死Buffer

2. GPU花费 cgt 的时间处理并填充Buffer，完成处理后释放Buffer的锁

3. Render Gun获得Buffer锁，同时GPU获得新RenderRequest，但因Buffer被锁，而一直处于拒绝状态

4. Render Gun在时钟频率到达时，读取内容对屏幕进行逐行渲染，用时rt

5. Render Gun释放Buffer锁，GPU持有该锁，等待CPU提交请求来进行下一次渲染

5. 由于这种同步关系， GPU开始处理包，屏幕开始刷新，都需要间隔 rt+cgt+t留白 的时间

两个缓存的时候：

1. 第一个包交给GPU处理，GPU锁死BufferOne

2. GPU花费 cgt 的时间处理并填充BufferOne，完成处理后释放BufferOne的锁

3. Render Gun获得BufferOne的锁，同时GPU获得新RenderRequest并获得BufferTwo的锁

4. Render Gun能在时钟 t 到达时不阻塞的不断渲染内容，GPU也能够不断处理新Request，各干各的

5. GPU在处理完毕后，会立刻替换Buffer，若替换时正好发生在时钟到达时，就会发生 画面撕裂 现象

垂直同步V-Sync —— 见《定时器 & DisplayLink猜想》

1. 硬件时钟定时的发送一个V-Sync信号，该信号保证Render Gun刚处于空闲就绪状态

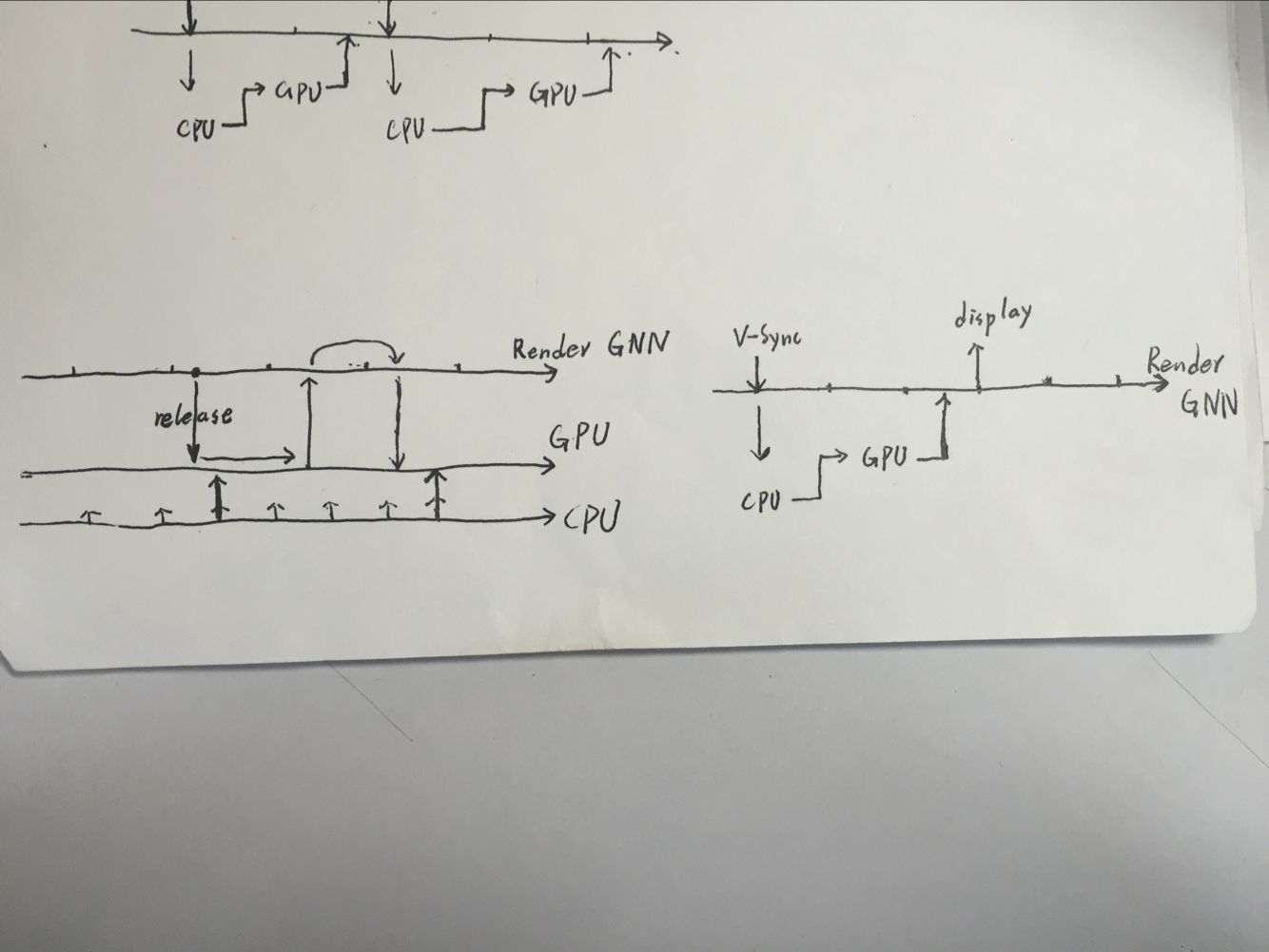
2. CPU线程会监视该信号，当出现该信号时，才计算打包并发出新Request供GPU处理

3. 和一个缓存的时候类似，特点是能捕捉信号来编程，而不是CPU空运算被GPU抛弃（CPU比GPU快很多）

4. 原文：

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。

5. 从上面的图中可以看到，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。



动画渲染原理猜想

关于图形渲染的猜想（伪镜像）：

1. 你的程序，有程序的CPU和内存 ， 而显卡有显卡的CPU和内存

2. 你的UIView只是程序内存中的对象，一堆数值而已，但程序中的方法设定了通向显卡的接口

3. 当你实例化UIWindow对象后，该对象的实例方法在显卡内存中开辟了渲染任务

4. 当你向UIWindow设定rootController后，该设定方法在对应的渲染程序中，添加了渲染树的第一个根节 点，该程序将每秒60次的从根节点读取树信息来渲染

5. 上面的rootController，后来的subview的添加方法中，不仅会在显卡中开辟内存并添加到树上，还会返 回该树节点的指针到 对应对象的程序内存中

6. 子视图的添加，UI对象属性的修改，都会触发程序携带 显卡对象指针 到显卡中执行对应的修改

关于动画的猜想（另辟蹊径）

1. 当你向UI类添加动画对象时，会触发一个并发的定时线程，但set方法本身会检查self是否存在在渲染树 中（是否持有显卡指针），若没有，就并不fork线程出来，直接结束

2. 该线程会可能存在一个runloop方法，在run到时，计算该时刻的值，然后向显卡内存输出这份更改

3. 然而这份更改并不会改变程序内存中对应的值（若你设置了forward，在线程fork之前就改变为了结束值）

4. 线程结束时，也会根据forward来决定最后一帧set到显卡内存的，到底是什么值

CPU消耗原因和解决方案（神级机制阐述）

对象创建

1. 对象的创建会分配内存、调整属性、甚至还有读取文件等操作，比较消耗 CPU 资源。

2. 尽量用轻量的对象代替重量的对象，可以对性能有所优化。比如 CALayer 比 UIView 要轻量许多，那么不 需要响应触摸事件的控件，用 CALayer 显示会更加合适。

3. 如果对象不涉及 UI 操作，则尽量放到后台线程去创建，但可惜的是包含有 CALayer 的控件，都只能在主 线程创建和操作。

4. 通过 Storyboard 创建视图对象时，其资源消耗会比直接通过代码创建对象要大非常多，在性能敏感的界 面里，Storyboard 并不是一个好的技术选择。

5. 尽量推迟对象创建的时间，并把对象的创建分散到多个任务中去。尽管这实现起来比较麻烦，并且带来的优 势并不多，但如果有能力做，还是要尽量尝试一下。

6. 如果对象可以复用，并且复用的代价比释放、创建新对象要小，那么这类对象应尽量放到一个缓存池里复用。

对象调整 （有问题：第一条可能是UIView内部没有CALayer的属性）

1. 对象的调整也经常是消耗 CPU 资源的地方。这里特别说一下 CALayer：CALayer 内部并没有属性，当调 用属性方法时，它内部是通过运行时 resolveInstanceMethod 为对象临时添加一个方法，该方法把对应 属性 值保存到内部的一个 Dictionary 里，同时还会通知 delegate、创建动画，通过displayLink添加到 runloop启动动画等等，非常消耗资源（可能是由于隐式动画生成系列的方法不应该出现在CALayer中， 所以通过类别 或 那个什么resolveInstanceMethod额外添加到frame的set方法中）

2. UIView 的关于显示相关的属性（比如 frame/bounds/transform）等实际上都是 CALayer 属性映射来的， 所以对 UIView 的这些属性进行调整时，消耗的资源要远大于一般的属性。对此你在应用中，应该尽量减 少不必要的属性修改。

3. 当视图层次调整时，UIView、CALayer 之间会出现很多方法调用与通知，所以在优化性能时，应该尽量避 免调整视图层次、添加和移除视图。

对象销毁

对象的销毁虽然消耗资源不多，但累积起来也是不容忽视的。通常当容器类持有大量对象时，其销毁时的资源 消耗就非常明显。同样的，如果对象可以放到后台线程去释放，那就挪到后台线程去。

这里有个小 Tip：把对象捕获到 block 中，然后扔到后台队列去随便发送个消息以避免编译器警告，就可以让 对象在后台线程销毁了：屌！方法结束时NSArray仅仅是release了计数器，但实际dealloc是在后台

NSArray \*tmp = self.array;

self.array = nil;

dispatch\_async(queue, ^{  [tmp class]; });

布局计算

1. 视图布局的计算是 App 中最为常见的消耗 CPU 资源的地方。如果能在后台线程提前计算好视图布局、并 且对视图布局进行缓存，那么这个地方基本就不会产生性能问题了。

2. 不论通过何种技术对视图进行布局，其最终都会落到对 UIView.frame/bounds/center 等属性的调整上。 上面也说过，对这些属性的调整非常消耗资源，所以尽量提前计算好布局，在需要时一次性调整好对应属 性，而不要多次、频繁的计算和调整这些属性。

3. 比如需要生成新界面时，该界面的初始化工作是早就弄好的，而不是显示的一瞬间布局的

Autolayout

Autolayout 是苹果本身提倡的技术，在大部分情况下也能很好的提升开发效率，但是 Autolayout 对于复杂 视图来说常常会产生严重的性能问题。随着视图数量的增长，Autolayout 带来的 CPU 消耗会呈指数级上升。 具体数据可以看这个文章：http://pilky.me/36/。 如果你不想手动调整 frame 等属性，你可以用一些工具方 法替代（比如常见的 left/right/top/bottom/width/height 快捷属性），或者使用 ComponentKit、 AsyncDisplayKit 等框架。

文本计算

1. 如果一个界面中包含大量文本（比如微博微信朋友圈等），文本的宽高计算会占用很大资源，并且不可避 免。如果你对文本显示没有特殊要求，可以参考下 UILabel 内部的实现方式：用 [NSAttributedString boundingRectWithSize:options:context:] 来计算文本宽高，用 -[NSAttributedString drawWithRect:options:context:] 来绘制文本。尽管这两个方法性能不错，但仍旧需要放到后台线程进行 以避免阻塞主线程。

2. 如果你用 CoreText 绘制文本，那就可以先生成 CoreText 排版对象，然后自己计算了，并且 CoreText 对 象还能保留以供稍后绘制使用。

文本渲染

1. 屏幕上能看到的所有文本内容控件，包括 UIWebView，在底层都是通过先 CoreText 排版、绘制为 Bitmap 显示的。

2. 常见的文本控件 （UILabel、UITextView 等），其排版和绘制都是在主线程进行的，当显示大量文本时， CPU 的压力会非常大。对此解决方案只有一个，那就是自定义文本控件，用 TextKit 或最底层的 CoreText 对文本异步绘制。尽管这实现起来非常麻烦，但其带来的优势也非常大

3. CoreText 对象创建好后，能直接获取文本的宽高等信息，避免了多次计算（调整 UILabel 大小时算一遍、 UILabel 绘制时内部再算一遍）；CoreText 对象占用内存较少，可以缓存下来以备稍后多次渲染。

图片的解码

当你用 UIImage 或 CGImageSource 的那几个方法创建图片时，图片数据并不会立刻解码。图片设置到 UIImageView 或者 CALayer.contents 中去后CALayer 被提交到 GPU 前，CGImage 中的数据才会 得到解码。这一步是发生在主线程的，并且不可避免。如果想要绕开这个机制，常见的做法是在后台线程先把 图片绘制到 CGBitmapContext 中，然后从 Bitmap 直接创建图片。目前常见的网络图片库都自带这个功能。

图像的绘制

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。

GPU 消耗原因和解决方案

[回忆图形渲染猜想](#图形渲染原理猜想)

GPU 能干的事情比较单一：接收提交的纹理（Texture）和顶点描述（三角形），应用变换（transform）、混合并 渲染，然后输出到屏幕上。通常你所看到的内容，主要也是纹理（图片）和形状（三角模拟的矢量图形）两类。

纹理的渲染

1. 所有的 Bitmap，包括图片、文本、栅格化的内容，最终都要由内存提交到显存，绑定为 GPU Texture。 不论是提交到显存的过程，还是 GPU 调整和渲染 Texture 的过程，都要消耗不少 GPU 资源。当在较 短时间显示大量图片时（比如 TableView 存在非常多的图片并且快速滑动时），CPU 占用率很低，GPU 占用非常高，界面仍然会掉帧。避免这种情况的方法只能是尽量减少在短时间内大量图片的显示，尽可能 将多张图片合成为一张进行显示。

2. 当图片过大，超过 GPU 的最大纹理尺寸时，图片需要先由 CPU 进行预处理，这对 CPU 和 GPU 都会带 来额外的资源消耗。目前来说，iPhone 4S 以上机型，纹理尺寸上限都是 4096x4096，更详细的资料可 以看这里：iosres.com。所以，尽量不要让上传的图片和视图的大小超过这个值。

视图的混合 (Composing)

当多个视图（或者说 CALayer）重叠在一起显示时，GPU 会首先把他们混合到一起。如果视图结构过于复杂， 混合的过程也会消耗很多 GPU 资源。为了减轻这种情况的 GPU 消耗，应用应当尽量减少视图数量和层次， 不透明的视图里标明 opaque 属性以避免无用的 Alpha 通道合成。当然，这也可以用上面的方法，把多个视 图预先渲染为一张图片来显示。

图形的生成

1. CALayer 的 border、圆角、阴影、遮罩（mask），CASharpLayer 的矢量图形显示，通常会触发离屏渲染 （offscreen rendering），而离屏渲染通常发生在 GPU 中。

2. 当一个列表视图中出现大量圆角的 CALayer，并且快速滑动时，可以观察到 GPU 资源已经占满，而 CPU 资源消耗很少。这时界面仍然能正常滑动，但平均帧数会降到很低。

3. 为了避免这种情况，可以尝试开启 CALayer.shouldRasterize 属性，但这会把原本离屏渲染的操作转嫁到 CPU 上去。对于只需要圆角的某些场合，也可以用一张已经绘制好的圆角图片覆盖到原本视图上面来模 拟相同的视觉效果。最彻底的解决办法，就是把需要显示的图形在后台线程绘制为图片，避免使用圆角、 阴影、遮罩等属性。

AsyncDisplayKit基本介绍

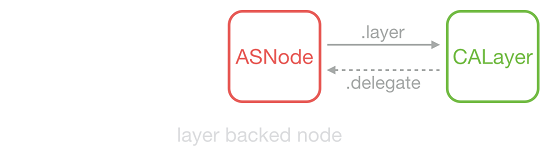
基本原理：

1. 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。

2. UIView 和 CALayer 的关系：View 持有 Layer 用于显示，View 中大部分显示属性实际是从 Layer 映射 而来；Layer 的 delegate 在这里是 View，当其属性改变、动画产生时，View 能够得到通知。UIView 和 CALayer 不是线程安全的，并且只能在主线程创建、访问和销毁。

3. 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-> CALayer这样一个关系。



4. 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 layer 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。

5. 通过模拟和封装 UIView/CALayer，开发者可以把代码中的 UIView 替换为 ASNode，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 的图层预合成 — 有时subLayer是紧贴在主Layer上没有自己的动画的

有时一个 layer 会包含很多 sub-layer，而这些 sub-layer 并不需要响应触摸事件，也不需要进行动画和位 置调整。ASDK 为此实现了一个被称为 pre-composing 的技术，可以把这些 sub-layer 合成渲染为一 张图片。

开发时，ASNode 已经替代了 UIView 和 CALayer；直接使用各种 Node 控件并设置为 layer backed 后， ASNode可以通过预合成来避免创建内部的 UIView 和 CALayer。

通过这种方式，把一个大的层级，通过一个大的绘制方法绘制到一张图上，性能会获得很大提升。CPU 避免了 创建 UIKit 对象的资源消耗，GPU 避免了多张 texture 合成和渲染的消耗，更少的 bitmap 也意味着 更少的内存占用。

ASDK 异步并发操作

ASDK 把布局计算、文本排版、图片/文本/图形渲染等操作都封装成较小的任务，并利用 GCD 异步并发执行。 如果开发者使用了 ASNode 相关的控件，那么这些并发操作会自动在后台进行，无需进行过多配置。

ASDK任务分发 — ASDK优化猜想提及

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。

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）。

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。设定CALayer和CATransaction提交等所有有操作结束 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

微博Demo技巧移步分区一UI

-----------OC Runtime（待深入研究）---------------------------------------------------------

网址：

http://mp.weixin.qq.com/s?\_\_biz=MzA5NzQ0OTY1Mw==&mid=400942983&idx=1&sn=cbd9712b73d38cd79fd407ec24c6f5d3&scene=21#wechat\_redirect

http://mp.weixin.qq.com/s?\_\_biz=MzA5NzQ0OTY1Mw==&mid=400928266&idx=1&sn=9cecb0f003b5a380af889bfedaf687ed&scene=21#wechat\_redirect

说在前面

1. id\* 、isa 、class 、objc\_class 就在这里面

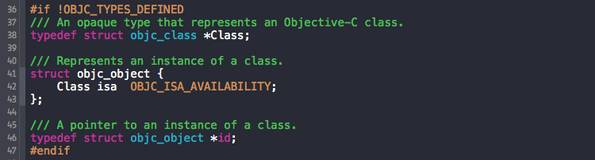
2. 关联函数（objc开头的系列C函数）就是出自于此，可以从Runtime层面操控OC对象

3. Wiki for Objective-C Objective-C Runtime Programming Guide

简介 - 什么是isa

Runtime（component）译名一般称为运行期组件，一个纯C语言写成的基础库(lib)，Objective-C编写出来的程 序必须得到Runtime的运行才能正常work，在Java、PHP或者Flash之类的编程语言当中，大家对于Runtime 并不会太陌生，Objective-C的Runtime其实也是一回事。正是Runtime实现了Objective-C许多的特性， Objective-C面向对象、消息派发、动态绑定和内存管理都与Runtime的息息相关。那么，在Objective-C当 中，对象、类、函数（方法）都是如何被构造并发挥作用的？前文提及，面向对象中的类，被看作抽象了的对 象，Runtime也是秉持这一理念。Runtime是纯C写成，用struct结构体来描述对象（实体对象）和类（抽 象对象）。

截图与解释：



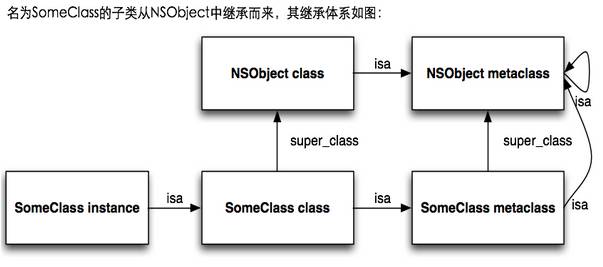


对象都是指针，指向一个结构体（id = struct objc\_object \*） —> 该结构体持有一个指针，指向另一结构体

（ (Class)objc\_object.isa = struct objc\_class \* ） —> 这个结构体由持有一个指针指向其父类/元类 （objc\_class.superClass/isa = struct objc\_class \*）

【总之 对象指针 —> 1元结构体 —> 指向Class结构体 —> 指向两个Class结构体】

截图与解释：



一个类指针，指向一个指针isa，该指针指向一个阐述类metadata的结构体；

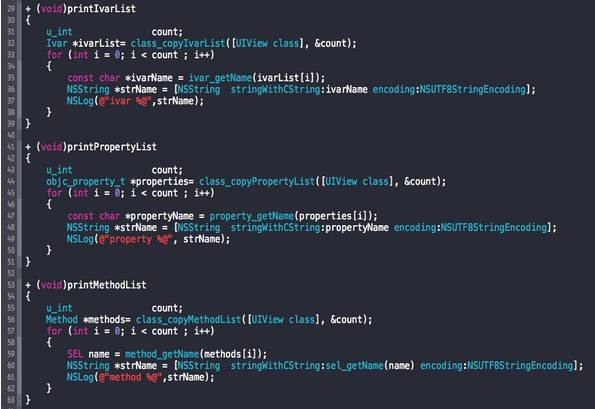
该结构体会持有一个isa指向metaclass，metaclass抽象了本类的特性，元类的isa最终指向的是它自身；

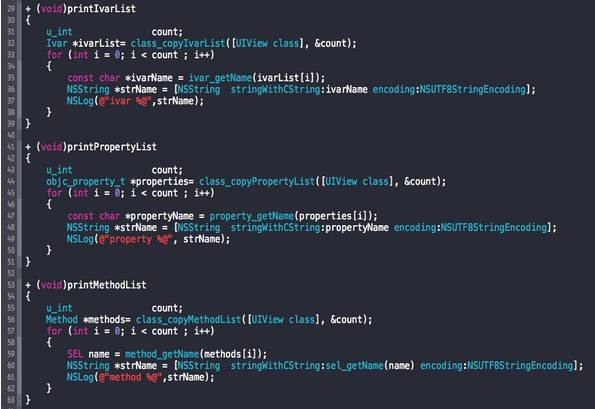
该结构体还持有superClass的阐述类metadata的结构体

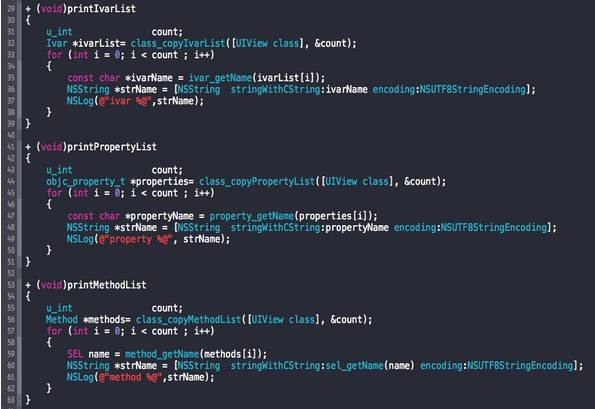
一般我们还认为阐述类metadata的结构体存放类的metadata（元数据），例如类的实例方法、类的实例变量

Runtime还允许我们通过标准的接口（C函数）对所有Objective-C类的变量、方法、属性以及协议等等作查询和 动态扩展，从而达到我们丰富项目中语言和类库特性的目标。

获取IvarList / PropertyList/ MethodList







关联objc\_getAssociatedObject —— WavePullFrash — DGElasticPullToRefreshExtensions

在类的定义之外为类增加额外的存储空间（比如想向NSObject添加额外的属性）

使用关联，我们可以不用修改类的定义而为其对象增加成员。这在我们无法访问到类的源码的时候或者是 考虑到二进制兼容性的时候是非常有用。

关联是基于关键字的，因此，我们可以为任何对象增加任意多的关联，每个都使用不同的关键字即可。关联是 可以保证被关联的对象在关联对象的整个生命周期都是可用的（垃圾自动回收环境下也不会导致资源不可回收

该模式解决了扩展（类别）NSObject对象只能扩展方法，不能扩展对象的问题，通过扩展getter和setter 方法，封装入objc关联函数，为该对象强制让这个对象持有这些指针，然后就可以通过getter和setter 访问这些指针指向的对象了

创建关联

原型：func objc\_setAssociatedObject(object: AnyObject!, \_ key: UnsafePointer<Void>, \_ value: AnyObject!, \_ policy: objc\_AssociationPolicy)

使用：

static char overviewKey;

//声明需要被扩展的对象

NSArray \* array =[[NSArray alloc] initWidthObjects:@"One", @"Two", @"Three", nil];

//声明扩展他人的对象

NSString \* overview = [[NSString alloc] initWithFormat:@"@",@"First three numbers"];

objc\_setAssociatedObject(array, &overviewKey, overview, OBJC\_ASSOCIATION\_RETAIN);

注意：这里的key不是传值，是传入指针，传入的是变量的存储位置（C语言不支持NSString）

创建关联要使用到Objective-C的运行时函数：objc\_setAssociatedObject来把一个对象与另外一个对象进行 关联。该函数需要四个参数：源对象，关键字，关联的对象和一个关联策略。

关键字是一个void类型的指针。每一个关联的关键字必须是唯一的。通常都是会采用静态变量来作为关键字。

关联策略表明了相关的对象是通过赋值，保留引用还是复制的方式进行关联的；还有这种关联是原子的还是非 原子的。这里的关联策略和声明属性时的很类似。这种关联策略是通过使用预先定义好的常量来表示的。

代码讲解：

在[overview release];处，字符串overview仍然是可用的，这是因为

OBJC\_ASSOCIATION\_RETAIN策略指明了数组要保有相关的对象。当数组array被销毁的时候，也就是 在[array release];  处overview也就会被释放，因此而被销毁

获取相关联的对象

objc\_getAssociatedObject(self, &dg\_associatedKeys.observersArray) as? [[String : NSObject]]

断开关联：set进nil即可

objc\_setAssociatedObject(array, &overviewKey, nil, OBJC\_ASSOCIATION\_ASSIGN);

使用函数objc\_removeAssociatedObjects可以断开所有关联（只有希望类恢复最初状态时才这么做）

封装为NS接口来使用

一般将set进对象，和get这种复杂的函数原型，通过扩展对象方法，来包装一下，然后直接通过该对象的实 例直接getter和setter

消息派发 - 文字概述

objc\_msgSend是消息派发最核心最基础的入口函数，除此之外还有objc\_msgsend\_stret，objc\_msgSend\_fpret， objc\_msgSendSuper等函数，然而它们的重要性和作用远不及objc\_msgSend。objc\_msgSend函数会依据 receiver和selector的来调用适当的方法。为了完成此操作，该函数需要在recevier所属的类中搜寻其“方法 列表”，如果能找到与selector字符串名称相符的方法，就跳转至该方法。

若是找不到，那就沿着继承体系继续向上查找，等找到合适的方法之后再跳转。如果最终还是找不到相符的方法， 那就执行“消息转发”操作。由此，我们可以看到，调用一个方法似乎需要相当的步骤。每一个步骤都是开销， 是否会导致Objective-C有性能问题？所幸obj\_msgSend会将匹配到得结果缓存在“快速映射表”（fast map），每个类都有这样一块缓存，若是后面还需要向该类发送和相同的selector消息，执行起来将会快许多。

这种“快速执行路径”（fast path）还是不如“静态绑定函数调用”（statically bound function call）那样快，不 过通过汇编等优化技术，映射表的查询开销已非常小，可以说，即使相比较C++的静态绑定，Objective-C的 消息派发机制已经不是性能瓶颈所在

当对象查询不到相关的方法，消息无法正确回应时，还会启动三步“消息转发”机制。

1. 先询问receiver或者说是它所属的类，看其是否能动态添加方法，以处理当前这个“未知选择子” （unkonwn selector），这叫做“动态方法解析”（dynamic method resolution），Runtime会通过回调 一个类方法来寻求动态添加方法的支持。如果Runtime完成动态添加方法的询问之后，receiver仍然无 法正常响应，

2. 则Runtime会继续向receiver询问是否有其它对象即其它receiver能处理这条消息，若返回能够处理对象， Runtime会把消息转给返回的对象，消息转发流程也就结束。

3. 若无对象返回，Runtime会把消息有关的全部细节都封装到NSInvocation对象中，再给receiver最后一 次机 会，令其设法解决当前还未处理的这条消息

步骤越往后，处理消息累计开销就越大。所以，最好能在第一步就处理完，这样的话，Runtime还可以把方法 进行缓存，在一步到位的同时进一步降低首次查询这样的开销

利用消息转发机制，我们实现一个让NSString类支持NSArray实例方法的例子，这对降低程序的Crash率有帮助

消息派发 - 方法缓存原理

http://mp.weixin.qq.com/s?\_\_biz=MzA5NzQ0OTY1Mw==&mid=400928266&idx=1&sn=9cecb0f003b5a380af889bfedaf687ed&scene=21#wechat\_redirect

从消息决议说起—objc\_msgSend汇编注释解析

1. 判断receiver是否为nil，也就是objc\_msgSend的第一个参数self，也就是要调用的那个方法所属对象
2. 从缓存里寻找，找到了则分发，否则
3. 利用objc-class.mm中\_class\_lookupMethodAndLoadCache3方法去寻找selector
   * 如果支持GC，忽略掉非GC环境的方法（retain等）
   * 从本class的method list寻找selector，如果找到，填充到缓存中，并返回selector，否则
   * 寻找父类的method list，并依次往上寻找，直到找到selector，填充到缓存，并返回selector，否则
   * 调用\_class\_resolveMethod，如果可以动态resolve为一个selector，不缓存，方法返回，否则
   * 转发这个selector，否则

4 NSInvocation报错，抛出异常

\_class\_lookupMethodAndLoadCache3这个方法

如果单纯看方法名，这个方法应该会从缓存和方法列表中查找一个方法，但是如第一节所讲，在调用这个方法 之前，我们已经是从缓存无法找到这个方法了，所以这个方法避免了再去扫描缓存查找方法的过程，而是直接 从方法列表找起。从Apple代码的注释，我们完全可以了解这一点。不顾一切地追求完美和性能，是一种品质。

追本溯源，何为方法缓存

objc\_cache的定义如下：

struct objc\_cache {

uintptr\_t mask; /\* total = mask + 1 \*/

uintptr\_t occupied;

cache\_entry \*buckets[1]; };

1)、mask：可以认为是当前能达到的最大index（从0开始的），所以缓存的size（total）是mask+1

2)、occupied：被占用的槽位，因为缓存是以散列表的形式存在的，所以会有空槽，而occupied表示当 前被占用的数目

3)、buckets：用数组表示的hash表，cache\_entry类型，每一个cache\_entry代表一个方法缓存

(buckets定义在objc\_cache的最后，说明这是一个可变长度的数组)

cache\_entry的定义如下

typedef struct {

SEL name; // same layout as struct old\_method

void \*unused;

IMP imp; // same layout as struct old\_method

} cache\_entry;

1)、name，被缓存的方法名字

2)、unused，保留字段，还没被使用。

3)、imp，方法实现

缓存和散列（不明白了）

往方法缓存里存放一个方法的代码片段：

// Scan for the first unused slot and insert there.

// There is guaranteed to be an empty slot because the

// minimum size is 4 and we resized at 3/4 full.

buckets = (cache\_entry \*\*)cache->buckets;

for (index = CACHE\_HASH(sel, cache->mask); buckets[index] != NULL; index = (index+1) & cache->mask)

{ // empty }

buckets[index] = entry;

我们可以看到sel被散列后找到一个空槽放在buckets中，而CACHE\_HASH的定义如下：

#define CACHE\_HASH(sel, mask) (((uintptr\_t)(sel)>>2) & (mask))

这段代码就是利用了sel的指针地址和mask做了一下简单计算得出的

从散列表取缓存则是利用汇编语言写成的，objc-msg-arm.mm 里面的CacheLookup方法：

代码看网址，粘贴了你也看不明白逗逼

虽然是汇编，但是注释太详尽了，理解起来并不难，还是求hash，去buckets里找，找不到按照hash 冲突的规则继续向下，直到最后。

十万个为什么

方法缓存存在什么地方：类的定义里就有cache字段，没错，类的所有缓存都存在metaclass上，所以每个对 象都指向一份方法缓存，而不是每一个类的object都保存一份。

父类方法的缓存只存在父类么，还是子类也会缓存父类的方法：即便是从父类取到的方法，也缓存在子类的 元类方法缓存里。而当用一个父类对象去调用那个方法的时候，也会在父类的metaclass里缓存一份。

类的方法缓存大小有没有限制？：只有当方法缓存第奇数次满（使用的槽位超过3/4）的时候，方法缓存的大 小才会增长（会清空缓存，否则hash值就不对了）；当第偶数次满的时候，方法缓存会被清空并重新利用。 如果\_class\_slow\_grow值为0，那么每一次方法缓存满的时候，其大小都会增长。所以单就问题而言，答 案是没有限制

为什么类的方法列表不直接做成散列表呢，做成list，还要单独缓存，多费事？：

散列表是没有顺序的，Objective-C的方法列表是一个list，是有顺序的；Objective-C在查找方法的时候 会顺着list依次寻找，并且category的方法在原始方法list的前面，需要先被找到，如果直接用hash 存方法，方法的顺序就没法保证。

List的方法还保存了除了selector和imp之外其他很多属性

散列表是有空槽的，会浪费空间