19-热点问题答疑（二）：基础模块问题答疑

你好，我是戴铭。

这是我们《iOS开发高手课》专栏的第二期答疑文章，我将继续和你分享大家在学习前面文章时遇到的最普遍的问题。

首先呢，我要感谢你这段时间对专栏的关注，让我感觉写专栏这件事儿格外有意义。通过这段时间对大家留言问题的观察，我也发现还有很多同学对 RunLoop 原理的一些基础概念不是很了解。这就导致在出现了比如卡顿或者线程问题时找不到好的解决方案，所以我今天就先和你分享一下学习RunLoop的方法和参考资料。

其实，**目前关于RunLoop 原理的高质量资料非常多，那我们究竟应该怎么利用这些资料，来高效地掌握RunLoop的原理呢？**

我建议你按照下面的顺序来学习RunLoop 原理，坚持下来你就会对RunLoop的基础概念掌握得八九不离十了。

- 首先，你可以看一下孙源的一个线下分享《RunLoop》，对 RunLoop 的整体有个了解。
- 然后，你可以再看官方文档，全面详细地了解苹果公司设计的 RunLoop 机制，以及如何运用 RunLoop来解决问题。
- 最后，了解了RunLoop的机制和运用后，你需要深入了解 RunLoop 的实现，掌握 RunLoop 原理中的基础概念。ibireme 的一篇文章 《深入理解RunLoop》，结合着底层 CFRunLoop 的源码，对RunLoop 机制进行了深入分析。

好了，关于RunLoop原理学习的内容，我就先说到这里。接下来，**我再跟你说说最近被问到的，我认为比较重要的两个问题：**

- 一个是，使用 dlopen()，App能不能审核通过？
- 另一个是， matrix-iOS 里的卡顿监控系统，与我在第13篇文章里提到的卡顿监控系统有什么区别？

其实，我知道大家还都比较关注课后作业的解题思路，但是考虑到有很多同学还没有静下心来去思考、去完成，所以我准备过一段时间再和你分享这部分内容。这里，我还是想再和你分享一下我在开篇词中提出的观点：

> 对于咱们手艺人来说，不动手都是空谈，就像绘画教程，光看不练，是不会有进步的。这就如同九阴真经的口诀，铭记于心后还需要常年累月的修炼才能精进。动手就会碰到问题，就会思考，这个主动过程会加深你的记忆，这样后面再碰到问题时，你会更容易将相关知识串联起来，形成创新式的思考。

这些作业确实有难度，也确实需要你投入很多精力，如果你在动手解决这些问题的过程中，具体有哪里卡住了，欢迎给我留言。我可以针对你遇到的问题给出有针对性的解答，或许这样对你的帮助会更大。

现在，我们就从第一个问题说起吧。

使用 dlopen() 能不能审核通过？

@Ant同学在第6篇文章“App 如何通过注入动态库的方式实现极速编译调试？”中留言问到：

> Injection 使用了 dlopen() 方法，审核能通过吗？ 是不是调试的时候用，提交App Store时候移除呢？

苹果公司关于App审核的规定，你可以点击这个链接查看。其中2.5.2规定如下：

> App 应自包含在自己的套装中，不得在指定容器范围外读取或写入数据，也不得下载、安装或执行会引入或更改 App 特性或功能的代码，包括其他 App。仅在特殊情况下，用于教授、开发或允许学生测试可执行代码的教育类 App 可以下载所提供的代码，但这类代码不得用于其他用途。这类 App 必须开放 App 提供的源代码，让客户可以完全查看和编辑这些源代码。

2018年11月，苹果公司集中下线了718个 App，主要原因就是它们违反了 2.5.2 这个条款，下面是苹果公司对于违反了 2.5.2条款的回复：

> Your app, extension, and/or linked framework appears to contain code designed explicitly with the capability to change your app’s behavior or functionality after App Review approval, which is not in compliance with App Store Review Guideline 2.5.2 and section 3.3.2 of the Apple Developer Program License Agreement.

> This code, combined with a remote resource, can facilitate significant changes to your app’s behavior compared to when it was initially reviewed for the App Store. While you may not be using this functionality currently, it has the potential to load private frameworks, private methods, and enable future feature changes. This includes any code which passes arbitrary parameters to dynamic methods such as dlopen(), dlsym(), respondsToSelector:, performSelector:, method_exchangeImplementations(), and running remote scripts in order to change app behavior and/or call SPI, based on the contents of the downloaded script. Even if the remote resource is not intentionally malicious, it could easily be hijacked via a Man In The Middle (MiTM) attack, which can pose a serious security vulnerability to users of your app.

苹果公司在这段回复中，提到了使用 dlopen()、dlsym()、respondsToSelector:、performSelector:、method_exchangeImplementations() 这些方法去执行远程脚本，是不被允许的。因为这些方法和远程资源相结合，可能加载私有框架和私有方法，可能使 App 的行为发生重大变化。这就会和审核时的情况不一样，即使使用的远程资源本身不是恶意的，但是它们也很容易被劫持，给用户带来不可预计的伤害，使得应用程序有安全漏洞。

其实，我在第11篇答疑文章里就提到，**苹果公司不允许通过运行时加载远程动态库的 App 上线 App Store。**

那么现在，我们回到 Ant同学提的问题本身，App 带着 Injection 上线后，如果使用 dlopen() 去读取远程动态库，就会被拒绝。另外，在我看来，Injection 本来就是用于线下调试的，为什么还要带着它上 App Store 呢。

下面我来说下第二个问题，matrix-iOS 里卡顿监控系统，与我在第13篇文章里提到的卡顿监控系统有什么区别？

matrix-iOS

第13篇文章上线后，有很多朋友和我反馈说，微信最近开源了一个卡顿监控系统 matrix-iOS，并询问我它和我在这篇文章里提到的卡顿监控系统，有什么区别。

因为matrix-iOS 对性能的优化考虑得非常全面，这些优化不仅能够应用在卡顿监控上，对于其他监控都有很好的借鉴作用，所以非常值得我们深入了解一下。接下来，我就这个话题和你展开一下。

记得在2015年8月的时候，微信团队的一位同学做了一次关于iOS卡顿监控方案的分享。这次分享让我受益匪浅，而且这也是我第一次听说 iOS 卡顿监控方案。次月，微信团队就放出了一篇文章专门介绍卡顿监控方案“微信iOS卡顿监控系统”。之后，很多团队参照这篇文章开发了自己的卡顿监控系统。我在第13篇文章中设计的卡顿监控系统，也是按照这个思路写的。

在今年的4月3号，微信团队将他们的卡顿监控系统matrix开源出来了，包括Matrix for iOS/macOS和Android系统的监控方案。关于matrix-iOS的卡顿监控原理，你可以点击这个链接查看。

如果你的 App 现在还没有卡顿监控系统，可以考虑直接集成 matrix-iOS，直接在 Podfile 里添加 pod ‘matrix-wechat’ 就可以了。如果已经有了卡顿监控系统，我建议你阅读下 matrix-iOS 的代码，里面有很多细节值得我们学习。比如：

- 子线程监控检测时间间隔：matrix-iOS 监控卡顿的子线程是通过 NSThread 创建的，检测时间间隔正常情况是1秒，在出现卡顿情况下，间隔时间会受检测线程退火算法影响，按照斐波那契数列递增，直到没有卡顿时恢复为1秒。
- 子线程监控退火算法：避免一个卡顿会写入多个文件的情况。
- RunLoop 卡顿时间阈值设置：对于 RunLoop 超时阈值的设置，我在第13篇文章里建议设置为3秒，微信设置的是2秒。
- CPU 使用率阈值设置：当单核 CPU 使用率超过 80%，就判定 CPU 占用过高。CPU 使用率过高，可能导致 App 卡顿。

在我看来，这四点是能够让卡顿监控系统在对 App 性能损耗很小的情况下，更好地监控到线上 App 卡顿情况的四个细节，也是和我们第13篇文章中的卡顿方案有所不同的地方。

那接下来，我就跟你说下 matrix-iOS 的这四处细节具体是如何实现的吧。matrix-iOS 卡顿监控系统的主要代码在 WCBlockMonitorMgr.mm文件中。

子线程监控检测时间间隔

matrix-iOS 是在 addMonitorThread 方法里，通过 NSThread 添加一个子线程来进行监控的。addMonitorThread 方法代码如下：

```
- (void)addMonitorThread
{
    m_bStop = NO;
    m_monitorThread = [[NSThread alloc] initWithTarget:self selector:@selector(threadProc) object:nil];
    [m_monitorThread start];
}
```

这段代码中创建的 NSThread 子线程，会去执行 threadProc 方法。这个方法包括了子线程监控卡顿的所有逻辑。它的代码比较多，我先和你分析与检测时间间隔相关的代码，然后再和你分析其他的主要代码：

```
while (YES) {
    @autoreleasepool {
        if (g_bMonitor) {
            // 检查是否卡顿，以及卡顿原因
            ...
            // 针对不同卡顿原因进行不同的处理
            ...
        }

        // 时间间隔处理，检测时间间隔正常情况是1秒，间隔时间会受检测线程退火算法影响，按照斐波那契数列递增，直到没有卡顿时恢复
        for (int nCnt = 0; nCnt < m_nIntervalTime && !m_bStop; nCnt++) {
            if (g_MainThreadHandle && g_bMonitor) {
                int intervalCount = g_CheckPeriodTime / g_PerStackInterval;
                if (intervalCount <= 0) {
                    usleep(g_CheckPeriodTime);
                } else {
                    ...
                }
            } else {
                usleep(g_CheckPeriodTime);
            }
        }

        if (m_bStop) {
            break;
        }
    }
```

可以看出，创建的子线程通过 while 使其成为常驻线程，直到主动执行 stop 方法才会被销毁。其中，使用 usleep 方法进行时间间隔操作， g_CheckPeriodTime就是正常情况的时间间隔的值，退火算法影响的是 m_nIntervalTime，递增后检测卡顿的时间间隔就会不断变长。直到判定卡顿已结束，m_nIntervalTime 的值会恢复成1。

接下来，跟踪 g_CheckPeriodTime 的定义就能够找到正常情况下子线程卡顿监控的时间间隔。
g_CheckPeriodTime 的定义如下：

```
static useconds_t g_CheckPeriodTime = g_defaultCheckPeriodTime;
```

其中 g_defaultCheckPeriodTime 的定义是：

```
#define BM_MicroFormat_Second 1000000
const static useconds_t g_defaultCheckPeriodTime = 1 * BM_MicroFormat_Second;
```

可以看出，子线程监控检测时间间隔g_CheckPeriodTime，被设置的值就是1秒。

子线程监控退火算法

子线程监控检测时间间隔设置为1秒，在没有卡顿问题，不需要获取主线程堆栈信息的情况下性能消耗几乎可以忽略不计。但是，当遇到卡顿问题时，而且一个卡顿持续好几秒的话，就会持续获取主线程堆栈信息，增加性能损耗。更重要的是，持续获取的这些堆栈信息都是重复的，完全没有必要。

所以，matrix-iOS 采用了退火算法递增时间间隔，来避免因为同一个卡顿问题，不断去获取主线程堆栈信息的情况，从而提升了算法性能。

同时，一个卡顿问题只获取一个主线程堆栈信息，也就是一个卡顿问题 matrix-iOS 只会进行一次磁盘存储，减少了存储 I/O 也就减少了性能消耗。

所以，这种策略能够有效减少由于获取主线程堆栈信息带来的性能消耗。

那么，**matrix-iOS 是如何实现退火算法的呢？**

因为触发退火算法的条件是卡顿，所以我们先回头来看看子线程监控卡顿主方法 threadProc 里和发现卡顿后处理相关的代码：

```
while (YES) {
    @autoreleasepool {
        if (g_bMonitor) {
            // 检查是否卡顿，以及卡顿原因
            EDumpType dumpType = [self check];
            if (m_bStop) {
                break;
            }

            // 针对不同卡顿原因进行不同的处理
            ...
            if (dumpType != EDumpType_Unlag) {
                if (EDumpType_BackgroundMainThreadBlock == dumpType ||
                    EDumpType_MainThreadBlock == dumpType) {
                    if (g_CurrentThreadCount > 64) {
                        // 线程数超过64个，认为线程过多造成卡顿，不用记录主线程堆栈
                        dumpType = EDumpType_BlockThreadTooMuch;
                        [self dumpFileWithType:dumpType];
                    } else {
                        EFilterType filterType = [self needFilter];
                        if (filterType == EFilterType_None) {
                            if (g_MainThreadHandle) {
                                if (g_PointMainThreadArray != NULL) {
                                    free(g_PointMainThreadArray);
                                    g_PointMainThreadArray = NULL;
                                }
                                g_PointMainThreadArray = [m_pointMainThreadHandler getPointStackCursor];
                                // 函数主线程堆栈写文件记录
                                m_potenHandledLagFile = [self dumpFileWithType:dumpType];
                                // 回调处理主线程堆栈文件
                                ...

                            } else {
                                // 主线程堆栈写文件记录
```

```
                                m_potenHandledLagFile = [self dumpFileWithType:dumpType];
                                ...
                            }
                        } else {
                            // 对于 filterType 满足退火算法、主线程堆栈数太少、一天内记录主线程堆栈过多这些情况不用进行
                            ...
                        }
                    }
                } else {
                    m_potenHandledLagFile = [self dumpFileWithType:dumpType];
                }
            } else {
                [self resetStatus];
            }
        }

        // 时间间隔处理，检测时间间隔正常情况是1秒，间隔时间会受检测线程退火算法影响，按照斐波那契数列递增，直到没有卡顿时恢复
        ...

    }
}
```

可以看出，当检测出主线程卡顿后，matrix-iOS 会先看线程数是否过多。为什么会先检查线程数呢？

我在17篇文章“远超你想象的多线程的那些坑”里提到线程过多时 CPU 在切换线程上下文时，还会更新寄存器，更新寄存器时需要寻址，而寻址的过程还会有较大的 CPU 消耗。你可以借此机会再回顾下这篇文章的相关内容。

按照微信团队的经验，线程数超出64个时会导致主线程卡顿，如果卡顿是由于线程多造成的，那么就没必要通过获取主线程堆栈去找卡顿原因了。根据 matrix-iOS 的实测，每隔 50 毫秒获取主线程堆栈会增加 3% 的 CPU 占用，所以当检测到主线程卡顿以后，我们需要先判断是否是因为线程数过多导致的，而不是一有卡顿问题就去获取主线程堆栈。

如果不是线程过多造成的卡顿问题，matrix-iOS 会通过 needFilter 方法去对比前后两次获取的主线程堆栈，如果两次堆栈是一样的，那就表示卡顿还没结束，满足退火算法条件，needFilter 方法会返回 EFilterType。EFilterType 为 EFilterType_Annealing，表示类型为退火算法。满足退火算法后，主线程堆栈就不会立刻进行写文件操作。

在 needFilter 方法里，needFilter 通过 [m_pointMainThreadHandler getLastMainThreadStack] 获取当前主线程堆栈，然后记录在 m_vecLastMainThreadCallStack 里。下次卡顿时，再获取主线程堆栈，新获取的堆栈和上次记录的 m_vecLastMainThreadCallStack 堆栈进行对比：

- 如果两个堆栈不同，表示这是一个新的卡顿，就会退出退火算法；
- 如果两个堆栈相同，就用斐波那契数列递增子线程检查时间间隔。

递增时间的代码如下：

```
if (bIsSame) {
    NSUInteger lastTimeInterval = m_nIntervalTime;
    // 递增 m_nIntervalTime
```

```
    m_nIntervalTime = m_nLastTimeInterval + m_nIntervalTime;
    m_nLastTimeInterval = lastTimeInterval;
    MatrixInfo(@"call stack same timeinterval = %lu", (unsigned long) m_nIntervalTime);
    return EFilterType_Annealing;
}
```

可以看出，将子线程检查主线程时间间隔增加后，needFilter 就直接返回 EFilterType_Annealing 类型表示当前情况满足退火算法。使用退火算法，可以有效降低没有必要地获取主线程堆栈的频率。这样的话，我们就能够在准确获取卡顿的前提下，还能保障 App 性能不会受卡顿监控系统的影响。

RunLoop 卡顿时间阈值设置

RunLoop 超时检查的相关逻辑代码都在 check 方法里。check 方法和 RunLoop 超时相关代码如下：

```
- (EDumpType)check
{
    // 1. RunLoop 超时判断
    // RunLoop 是不是处在执行方法状态中
    BOOL tmp_g_bRun = g_bRun;
    // 执行了多长时间
    struct timeval tmp_g_tvRun = g_tvRun;

    struct timeval tvCur;
    gettimeofday(&tvCur, NULL);
    unsigned long long diff = [WCBlockMonitorMgr diffTime:&tmp_g_tvRun endTime:&tvCur];

    ...

    m_blockDiffTime = 0;
    // 判断执行时长是否超时
    if (tmp_g_bRun && tmp_g_tvRun.tv_sec && tmp_g_tvRun.tv_usec && __timercmp(&tmp_g_tvRun, &tvCur, <) && d
        m_blockDiffTime = tvCur.tv_sec - tmp_g_tvRun.tv_sec;

        ...

        return EDumpType_MainThreadBlock;
    }

    ...

    // 2. CPU 使用率

    ...

    // 3. 没问题
    return EDumpType
```

可以看出，在判断执行时长是否超时代码中的 g_RunLoopTimeOut 就是超时的阈值。通过这个阈值，我们就可以知道 matrix-iOS 设置的 RunLoop 卡顿时间阈值是多少了。g_RunLoopTimeOut 的定义如下：

```
static useconds_t g_RunLoopTimeOut = g_defaultRunLoopTimeOut;
const static useconds_t g_defaultRunLoopTimeOut = 2 * BM_MicroFormat_Second;
```

可以看出，matrix-iOS 设置的 RunLoop 卡顿时间阈值是2秒。我在第13篇文章里设置的卡顿时间阈值是3秒，@80后空巢老肥狗在评论区留言到：

> 这个3秒是不是太长了，1秒60帧，每帧16.67ms。RunLoop 会在每次sleep之前去刷新UI，这样的话如果掉了30帧，就是500ms左右，用户的体验就已经下去了，能感觉到卡顿了。

关于卡顿时间阈值设置的这个问题，其实我和 matrix-iOS 的想法是一致的。你在实际使用时，如果把这个阈值设置为2秒后发现的线上卡顿问题比较多，短期内无法全部修复的话，可以选择把这个值设置为3秒。

还有一点我需要再说明一下，**我们所说的卡顿监控方案，主要是针对那些在一段时间内用户无法点击，通过日志也很难复现问题的情况而做的。这样的卡顿问题属于头部问题，对用户的伤害是最大的，是需要优先解决的。**这种方案，是不适合短时间掉帧的情况的。短时间掉帧问题对用户体验也有影响，但是属于优化问题。

除了 RunLoop 超时会造成卡顿问题外，在 check 方法里还有对于 CPU 使用率的判断处理，那么我再带你来看看 matrix-iOS 是如何通过 CPU 使用率来判断卡顿的。

CPU 使用率阈值设置

我在第18篇文章“怎么减少 App 电量消耗？”中，设置的 CPU 使用率阈值是 90%。那么，matrix-iOS 是如何设置这个 CPU 使用率阈值的呢？check 方法里的相关代码如下：

```
if (m_bTrackCPU) {
    unsigned long long checkPeriod = [WCBlockMonitorMgr diffTime:&g_lastCheckTime endTime:&tvCur];
    gettimeofday(&g_lastCheckTime, NULL);
    // 检查是否超过 CPU 使用率阈值限制，报 CPU 使用率一段时间过高
    if ([m_cpuHandler cultivateCpuUsage:cpuUsage periodTime:(float)checkPeriod / 1000000]) {
        MatrixInfo(@"exceed cpu average usage");
        BM_SAFE_CALL_SELECTOR_NO_RETURN(_delegate, @selector(onBlockMonitorIntervalCPUTooHigh:), onBlockMon
        if ([_monitorConfigHandler getShouldGetCPUIntervalHighLog]) {
            return EDumpType_CPUIntervalHigh;
        }
    }
    // 针对 CPU 满负荷情况，直接报 CPU 使用率过高引起卡顿
    if (cpuUsage > g_CPUUsagePercent) {
        MatrixInfo(@"check cpu over usage dump %f", cpuUsage);
        BM_SAFE_CALL_SELECTOR_NO_RETURN(_delegate, @selector(onBlockMonitorCurrentCPUTooHigh:), onBlockMoni
        if ([_monitorConfigHandler getShouldGetCPUHighLog]) {
            return EDumpType_CPUBlock;
        }
    }
}
```

通过上面代码，你会发现 matrix-iOS 使用了两个阈值，分别返回两种类型的问题，对应两种导致卡顿的情况：

- 一个是， CPU 已经满负荷，直接返回 CPU 使用率过高引起卡顿；
- 另一个是，持续时间内 CPU 使用率一直超过某个阈值，就返回 CPU 使用率造成了卡顿。

如上面代码所示，CPU 使用率阈值就在 cultivateCpuUsage:cpuUsage periodTime:periodSec 方法里。阈值相关逻辑代码如下：

```
if (cpuUsage > 80. && m_tickTok == 0 && m_bLastOverEighty == NO) {
    MatrixInfo(@"start track cpu usage");
    m_foregroundOverEightyTotalSec = 0;
    m_backgroundOverEightyTotalSec = 0;
    m_bLastOverEighty = YES;
}
```

可以看到，matrix-iOS 设置的 CPU 使用率阈值是80%。

到这里，我就已经把 matrix-iOS 的卡顿监控系统4个非常值得我们学习的细节说完了。而matrix-iOS 如何利用 RunLoop 原理去获取卡顿时长的原理，我已经在第13篇文章里跟你说过，这里就不再赘述了。

总结

在今天这篇文章中，我和你分享了下最近这段时间大家对专栏文章的一些问题。

首先，是关于对RunLoop原理的学习。我发现有很多同学在这方面的基础比较薄弱，所以特意梳理了这方面的学习方法和资料，希望可以帮到你。

然后，我针对大家比较关注的苹果公司审核动态化的相关规定，通过Injection里面带dlopen()方法能否审核通过和你做了说明，希望可以帮助你了解类似 dlopen()这样的技术应该怎样使用。

最后，我针对第13篇文章的监控系统，分析了最近微信团队新开源的matrix-iOS监控系统，为你详细分析了其中与卡顿监控相关的实现细节，也希望对你完善自己的监控系统有所帮助。

感谢你的收听，欢迎你在评论区给我留言分享你的观点，也欢迎把它分享给更多的朋友一起阅读。

精选留言：

- data 2019-04-23 08:53:01
  要咬牙去阅读这些优秀的源码才能提高自己

- 夏了南城 2019-04-23 00:43:45
  四个实现细节还是比较深刻的