抖音研发实践：基于二进制文件重排的解决方案 APP启动速度提升超15%

作者: Leo (字节跳动技术团队开发者)

技术背景

启动是App给用户的第一印象，对用户体验至关重要。抖音的业务迭代迅速，如果放任不管，启动速度会一点点劣化。为此抖音iOS客户端团队做了大量优化工作，除了传统的修改业务代码方式，我们还做了些开拓性的探索，发现修改代码在二进制文件的布局可以提高启动性能，方案落地后在抖音上启动速度提高了约15%。

本文从原理出发，介绍了我们是如何通过静态扫描和运行时trace找到启动时候调用的函数，然后修改编译参数完成二进制文件的重新排布。

原理

Page Fault

进程如果能直接访问物理内存无疑是很不安全的，所以操作系统在物理内存的上又建立了一层虚拟内存。为了提高效率和方便管理，又对虚拟内存和物理内存又进行分页（Page）。当进程访问一个虚拟内存Page而对应的物理内存却不存在时，会触发一次缺页中断（Page Fault），分配物理内存，有需要的话会从磁盘mmap读人数据。

通过App Store渠道分发的App，Page Fault还会进行签名验证，所以一次Page Fault的耗时比想象的要多：

重排

编译器在生成二进制代码的时候，默认按照链接的Object File(.o)顺序写文件，按照Object File内部的函数顺序写函数。

> 静态库文件.a就是一组.o文件的ar包，可以用ar -t查看.a包含的所有.o。

简化问题：假设我们只有两个page：page1/page2，其中绿色的method1和method3启动时候需要调用，为了执行对应的代码，系统必须进行两个Page Fault。

但如果我们把method1和method3排布到一起，那么只需要一个Page Fault即可，这就是二进制文件重排的核心原理。

我们的经验是优化一个Page Fault，启动速度提升0.6~0.8ms。

核心问题

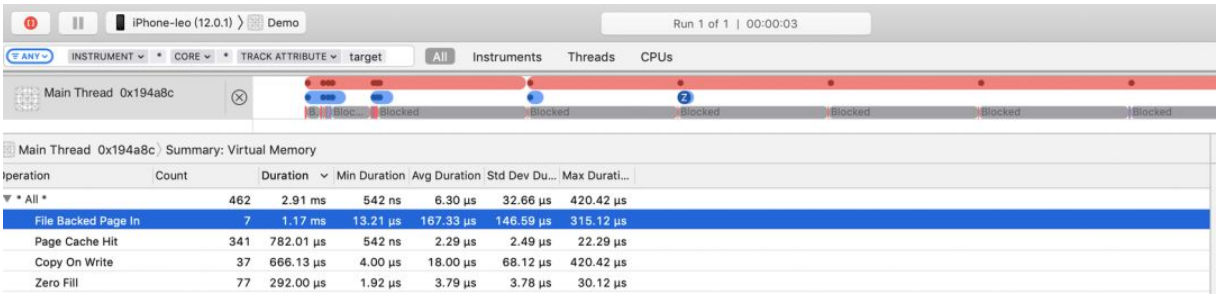
为了完成重排，有以下几个问题要解决：

- 重排效果怎么样 - 获取启动阶段的page fault次数
- 重排成功了没 - 拿到当前二进制的函数布局
- 如何重排 - 让链接器按照指定顺序生成Mach-O
- 重排的内容 - 获取启动时候用到的函数

System Trace

日常开发中性能分析是用最多的工具无疑是Time Profiler，但Time Profiler是基于采样的，并且只能统计线程实际在运行的时间，而发生Page Fault的时候线程是被blocked，所以我们需要用一个不常用但功能却很强大的工具：System Trace。

选中主线程，在VM Activity中的File Backed Page In次数就是Page Fault次数，并且双击还能按时序看到引起Page Fault的堆栈：



signpost

现在我们在Instrument中已经能拿到某个时间段的Page In次数，那么如何和启动映射起来呢?

我们的答案是：os_signpost。

os_signpost是iOS 12开始引入的一组API，可以在Instruments绘制一个时间段，代码也很简单：

```
1os_log_t logger = os_log_create("com.bytedance.tiktok", "performanc
e");
2os_signpost_id_t signPostId = os_signpost_id_make_with_pointer(log
ger,sign);
3//标记时间段开始
4os_signpost_interval_begin(logger, signPostId, "Launch","%{public}s"
, "");
5//标记结束
6os_signpost_interval_end(logger, signPostId, "Launch");
```

通常可以把启动分为四个阶段处理：

有多少个Mach-O，就会有多少个Load和C++静态初始化阶段，用signpost相关API对对应阶段打点，方便跟踪每个阶段的优化效果。

Linkmap

Linkmap是iOS编译过程的中间产物，记录了二进制文件的布局，需要在Xcode的Build Settings里开启Write Link Map File：

比如以下是一个单页面Demo项目的linkmap。

```
# Object files:
[  0] linker synthesized
[  1] ~/Library/Developer/Xcode/DerivedData/Demo/Build/Intermediates.noindex/Demo.build/Debug-iphoneos/Demo.build/Objects-normal/arm64/DemoLogger.o
[  2] !/Library/Developer/Xcode/DerivedData/Demo/Build/Intermediates.noindex/Demo.build/Debug-iphoneos/Demo.build/Objects-normal/arm64/ViewController.o
...
# Sections:
# Address   Size        Segment Section
0x10000680C 0x00000358  __TEXT  __text
0x100006B64 0x0000006C  __TEXT  __stubs
0x100006BD0 0x00000084  __TEXT  __stub_helper
...
0x100008010 0x00000048  __DATA  __la_symbol_ptr
....
# Symbols:
# Address   Size        File  Name
0x10000680C 0x0000004C  [  2] -[ViewController viewDidLoad]
0x100006858 0x000000A4  [  3] _main
0x1000068FC 0x0000006C  [  4] -[AppDelegate application:didFinishLaunchingWithOptions:]
0x100006968 0x00000040  [  4] -[AppDelegate applicationWillResignActive:]
0x1000069A8 0x00000040  [  4] -[AppDelegate applicationDidEnterBackground:]
0x1000069E8 0x00000040  [  4] -[AppDelegate applicationWillEnterForeground:]
0x100006A28 0x00000040  [  4] -[AppDelegate applicationDidBecomeActive:]
```

linkmap主要包括三大部分：

- Object Files 生成二进制用到的link单元的路径和文件编号
- Sections 记录Mach-O每个Segment/section的地址范围
- Symbols 按顺序记录每个符号的地址范围

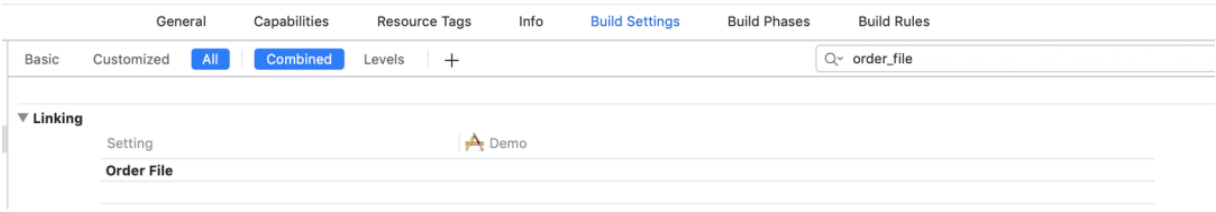
ld

Xcode使用的链接器件是ld，ld有一个不常用的参数-order_file，通过man ld可以看到详细文档：

> Alters the order in which functions and data are laid out. For each section in the output file, any symbol in that section that are specified in the order file file is moved to the start of its section and laid out in the same order as in the order file file.

可以看到，order_file中的符号会按照顺序排列在对应section的开始，完美的满足了我们的需求。

Xcode的GUI也提供了order_file选项：



如果order_file中的符号实际不存在会怎么样呢?

ld会忽略这些符号，如果提供了link选项-orderfilestatistics，会以warning的形式把这些没找到的符号打印在日志里。

获得符号

还剩下最后一个，也是最核心的一个问题，获取启动时候用到的函数符号。

我们首先排除了解析Instruments(Time Profiler/System Trace) trace文件方案，因为他们都是基于特定场景采样的，大多数符号获取不到。最后选择了静态扫描+运行时Trace结合的解决方案。

Load

Objective C的符号名是`+-[Class_name(category_name)` method:name:]，其中+表示类方法，-表示实例方法。

刚刚提到`linkmap`里记录了所有的符号名，所以只要扫一遍`linkmap`的`__TEXT,__text`，正则匹配`("^\+\[.*\ load\]$")`既可以拿到所有的`load`方法符号。

C++静态初始化

C++并不像Objective C方法那样，大部分方法调用编译后都是objc_msgSend，也就没有一个入口函数去运行时hook

但是可以用`-finstrument-functions`在编译期插桩“`hook`”，但由于抖音的很多依赖由其他团队提供静态库，这套方案需要修改依赖的构建过程。二进制文件重排在没有业界经验可供参考，不确定收益的情况下，选择了并不完美但成本最低的静态扫描方案。

- 扫描linkmap的**DATA，**modinitfunc，这个section存储了包含C++静态初始化方法的文件，获得文件号[5]。

```
1 //__mod_init_func
2 0x100008060    0x00000008  [  5] l tmp7
3 //[  5]对应的文件
4 [  5] .../Build/Products/Debug-iphoneos/libStaticLibrary.a(StaticLi
brary.o)
```

- 通过文件号，解压出.o。

```
1➜  lipo libStaticLibrary.a -thin arm64 -output arm64.a
2➜  ar -x arm64.a StaticLibrary.o
```

- 通过.o，获得静态初始化的符号名`_demo_constructor`。

```
➜  objdump -r -section=__mod_init_func StaticLibrary.o
2
3 StaticLibrary.o:    file format Mach-O arm64
4
5 RELOCATION RECORDS FOR [__mod_init_func]:
6 0000000000000000 ARM64_RELOC_UNSIGNED _demo_constructor
```

- 通过符号名，文件号，在linkmap中找到符号在二进制中的范围：

```
10x100004A30    0x0000001C  [  5] _demo_constructor
```

- 通过起始地址，对代码进行反汇编：

```
 1 ➜  objdump -d --start-address=0x100004A30 --stop-address=0x100004A
4B demo_arm64
 2
 3 _demo_constructor:
 4 100004a30:    fd 7b bf a9     stp x29, x30, [sp, #-16]!
 5 100004a34:    fd 03 00 91     mov x29, sp
 6 100004a38:    20 0c 80 52     mov w0, #97
 7 100004a3c:    da 06 00 94     bl  #7016
 8 100004a40:    40 0c 80 52     mov w0, #98
 9 100004a44:    fd 7b c1 a8     ldp x29, x30, [sp], #16
10 100004a48:    d7 06 00 14     b   #7004
```

- 通过扫描bl指令扫描子程序调用，子程序在二进制的开始地址为：

100004a3c +1b68（对应十进制的7016）。

```
11 00004a3c:    da 06 00 94     bl  #7016
```

- 通过开始地址，可以找到符号名和结束地址，然后重复5～7，递归的找到所有的子程序调用的函数符号。

合理避坑

STL里会针对string生成初始化函数，这样会导致多个.o里存在同名的符号，例如：

```
1 __ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEEC1I
DnEEPKc
```

类似这样的重复符号的情况在C++里有很多，所以C/C++符号在order_file里要带着所在的.o信息：

```
1 //order_file.txt
2 libDemoLibrary.a(object.o):__GLOBAL__sub_I_demo_file.cpp
```

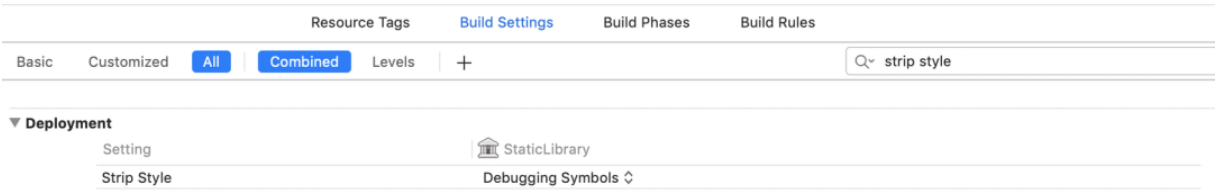
局限性

branch系列汇编指令除了bl/b，还有br/blr，即通过寄存器的间接子程序调用，静态扫描无法覆盖到这种情况。

Local符号

在做C++静态初始化扫描的时候，发现扫描出了很多类似l002的符号。经过一番调研，发现是依赖方输出静态库的时候裁剪了local符号。导致`__GLOBAL__sub_I_demo_file.cpp`变成了l002。

需要静态库出包的时候保留local符号，CI脚本不要执行strip -x，同时Xcode对应target的Strip Style修改为Debugging symbol：



> 静态库保留的local符号会在宿主App生成IPA之前裁剪掉，所以不会对最后的IPA包大小有影响。宿主App的Strip Style要选择All Symbols，宿主动态库选择Non-Global Symbols。

Objective C方法

绝大部分`Objective C`的方法在编译后会走`objc_msgSend`，所以通过`fishhook(https://github.com/facebook/fishhook)` hook这一个C函数即可获得`Objective C`符号。由于`objc_msgSend`是变长参数，所以`hook`代码需要用汇编来实现：

```
//代码参考InspectiveC
__attribute__((__naked__))
static void hook_Objc_msgSend() {
    save()
    __asm volatile ("mov x2, lr\n");
    __asm volatile ("mov x3, x4\n");
    call(blr, &before_objc_msgSend)
    load()
    call(blr, orig_objc_msgSend)
    save()
    call(blr, &after_objc_msgSend)
    __asm volatile ("mov lr, x0\n");
    load()
    ret()
}
```

子程序调用时候要保存和恢复参数寄存器，所以save和load分别对x0~x9, q0~q9入栈/出栈。call则通过寄存器来间接调用函数：

```
#define save() \
__asm volatile ( \
"stp q6, q7, [sp, #-32]!\n"\
...

#define load() \
```

```
 7__asm volatile ( \
 8"ldp x0, x1, [sp], #16\n" \
 9...
10
11#define call(b, value) \
12__asm volatile ("stp x8, x9, [sp, #-16]!\n"); \
13__asm volatile ("mov x12, %0\n" :: "r"(value)); \
14__asm volatile ("ldp x8, x9, [sp], #16\n"); \
15__asm volatile (#b " x12\n");
```

在`before_objc_msgSend`中用栈保存`lr`，在`after_objc_msgSend`恢复lr。由于要生成`trace`文件，为了降低文件的大小，直接写入的是函数地址，且只有当前可执行文件的`Mach-O`(app和动态库)代码段才会写入：

iOS中，由于ALSR`(https://en.wikipedia.org/wiki/Address_space_layout_randomization)`的存在，在写入之前需要先减去偏移量slide：

```
1 IMP imp = (IMP)class_getMethodImplementation(object_getClass(self),
 _cmd);
2 unsigned long imppos = (unsigned long)imp;
3 unsigned long addr = immpos - macho_slide
```

获取一个二进制的`__text`段地址范围：

```
1 unsigned long size = 0;
2 unsigned long start = (unsigned long)getsectiondata(mhp,  "__TEX
T", "__text", &size);
3 unsigned long end = start + size;
```

获取到函数地址后，反查`linkmap`既可找到方法的符号名。

Block

`block`是一种特殊的单元，`block`在编译后的函数体是一个C函数，在调用的时候直接通过指针调用，并不走`objc_msgSend`，所以需要单独`hook`。

通过`Block`的源码可以看到`block`的内存布局如下：

```
struct Block_layout {
    void *isa;
    int32_t flags; // contains ref count
    int32_t reserved;
    void  *invoke;
    struct Block_descriptor1 *descriptor;
};
struct Block_descriptor1 {
    uintptr_t reserved;
    uintptr_t size;
};
```

其中invoke就是函数的指针，hook思路是将invoke替换为自定义实现，然后在reserved保存为原始实现。

```
//参考 https://github.com/youngsoft/YSBlockHook
if (layout->descriptor != NULL && layout->descriptor->reserved == NULL)
{
    if (layout->invoke != (void *)hook_block_envoke)
    {
        layout->descriptor->reserved = layout->invoke;
        layout->invoke = (void *)hook_block_envoke;
    }
}
```

由于block对应的函数签名不一样，所以这里仍然采用汇编来实现 `hook_block_envoke`：

```
__attribute__((__naked__))
static void hook_block_envoke() {
    save()
    __asm volatile ("mov x1, lr\n");
    call(blr, &before_block_hook);
    __asm volatile ("mov lr, x0\n");
    load()
    //调用原始的invoke，即resvered存储的地址
    __asm volatile ("ldr x12, [x0, #24]\n");
    __asm volatile ("ldr x12, [x12]\n");
    __asm volatile ("br x12\n");
}
```

在`before_block_hook`中获得函数地址（同样要减去slide）。

```
intptr_t before_block_hook(id block,intptr_t lr)
{
    Block_layout * layout = (Block_layout *)block;
    //layout->descriptor->reserved即block的函数地址
    return lr;
}
```

同样，通过函数地址反查`linkmap`既可找到`block`符号。

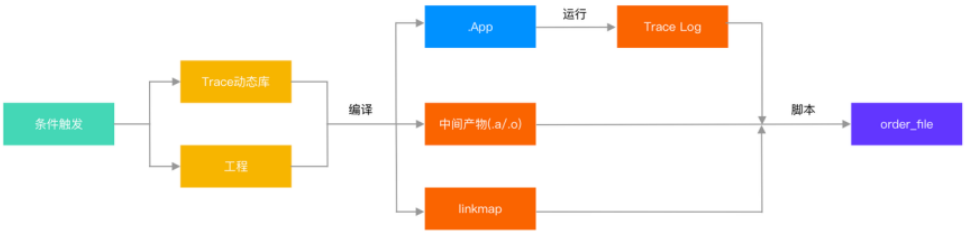
瓶颈

基于静态扫描+运行时trace的方案仍然存在少量瓶颈：

- initialize hook不到
- 部分block hook不到
- C++通过寄存器的间接函数调用静态扫描不出来

目前的重排方案能够覆盖到80%～90%的符号，未来我们会尝试编译期插桩等方案来进行100%的符号覆盖，让重排达到最优效果。

整体流程



1. 设置条件触发流程
2. 工程注入Trace动态库，选择release模式编译出.app/linkmap/中间产物
3. 运行一次App到启动结束，Trace动态库会在沙盒生成Trace log
4. 以Trace Log，中间产物和linkmap作为输入，运行脚本解析出order_file

总结

成功验证了二进制文件重排方案在iOS APP开发中的可行性和稳定性。基于二进制文件重排，我们在针对抖音的iOS客户端上的优化工作中，获得了约15%的启动速度提升。

抽象来看，APP开发中大家会遇到这样一个通用的问题，即在某些情况下，APP运行需要进行大量的Page Fault，这会影响代码执行速度。而二进制文件重排方案，目前看来是解决这一通用问题比较好的方案。

未来我们会进行更多的尝试，让二进制文件重排在更多的业务场景落地。