

目录

前言	1.1
Block概览	1.2
Block详解	1.3
Block正向开发	1.3.1
Block定义	1.3.2
Block类型	1.3.2.1
flags	1.3.2.2
invoke	1.3.2.3
signature	1.3.2.3.1
descriptor	1.3.2.4
imported variables	1.3.2.5
Block分析实例	1.4
YouTube逆向	1.4.1
Block心得	1.5
动态调试Block	1.5.1
相关objc函数	1.5.2
附录	1.6
参考资料	1.6.1

iOS逆向开发：Block匿名函数

- 最新版本： v0.6
- 更新时间： 20221027

简介

介绍iOS逆向期间会涉及到的Block匿名回调函数。先解释为何要研究Block；再详细介绍Block，包括正向开发时的如何写Block代码，然后是逆向时Block的结构定义，包括定义的代码和定义的结构图，再分别介绍每一部分的含义，包括Block类型、flags、invoke，以及其中的signature函数签名，以及descriptor、imported variables引用的量；再介绍iOS逆向期间所涉及到的Block最核心的内容是哪些，然后举例详细阐述如何通过Block搞懂函数调用逻辑，以及导入变量的参数传递，顺带介绍如何优化IDA伪代码。然后再整理Block相关的心得，比如动态调试期间，如何查看Block信息、和Block相关的objc函数。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- [crifan/ios_re_objc_block: iOS逆向开发：Block匿名函数](#)

如何使用此HonKit源码去生成发布为电子书

详见：[crifan/honkit_template: demo how to use crifan honkit template and demo](#)

在线浏览

- [iOS逆向开发：Block匿名函数 book.crifan.org](#)
- [iOS逆向开发：Block匿名函数 crifan.github.io](#)

离线下载阅读

- [iOS逆向开发：Block匿名函数 PDF](#)
- [iOS逆向开发：Block匿名函数 ePub](#)
- [iOS逆向开发：Block匿名函数 Mobi](#)

版权和用途说明

此电子书教程的全部内容，如无特别说明，均为本人原创。其中部分内容参考自网络，均已备注了出处。
如有版权，请通过邮箱联系我 `admin 艾特 crifan.com`，我会尽快删除。谢谢合作。

各种技术类教程，仅作为学习和研究使用。请勿用于任何非法用途。如有非法用途，均与本人无关。

鸣谢

感谢我的老婆陈雪的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

更多其他电子书

本人 `crifan` 还写了其他 100+ 本电子书教程，感兴趣可移步至：

[crifan/crifan_ebook_readme: Crifan的电子书的使用说明](#)

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-10-27 22:33:16

Block概览

- Block
 - 名称
 - 官方标准叫法其实是: Blocks
 - 大家常简称为: Block = 代码块
 - 是什么:
 - Objc的对象
 - iOS 的 ObjC 中的一个将 数据 与相关 行为 相结合的对象
 - (ObjC对) C 语言扩展
 - Blocks 是添加到 C 、 objective-C 和 C++ 的语言级功能, 它允许您创建不同的代码段, 这些代码段可以像值一样传递给方法或函数。
 - 它们还具有从封闭范围捕获值的能力
 - 类似: 其他语言中的 闭包 = closure 、 lambda
 - 功能和用途: 用于创建匿名函数, 实现函数的异步 (或同步) 调用
 - 相关底层机制
 - Block可以从局部变量中获取值; 发生这种情况时, 必须动态分配内存。
 - 初始分配是在堆栈 Stack 上完成的, 但是运行时提供了一个 Block_copy 函数, 给定一个块指针, 该函数要么将底层块对象复制到堆 Heap , 将其引用计数设置为 1 并返回新的块指针, 要么 (如果 Block 对象已经在堆 Heap 上) 将其引用计数增加 1
 - 配对函数是 Block_release , 它将引用计数减少 1, 如果计数达到零并且在堆 Heap 上, 则销毁该对象
 - 官网
 - [Working with Blocks](#)
 - [Objective-C Automatic Reference Counting \(ARC\) — Clang 16.0.0git documentation](#)

为何要研究ObjC中的Block?

iOS逆向期间, 最初是要找到某函数被调用的地方, 通过函数调用堆栈, 找到最后, 发现找不下去了

之后发现其实是 Block 的机制实现的: Block = 实现了 函数 (同步或) 异步的回调

TODO:

- 找到Block的函数如何被调用的 = Block函数的上层调用函数
 - 【整理】iOS逆向调试心得: 如何找到Block的(pthread_wqthread和_dispatch_call_block_and_release的函数调用最初来源

crifan.org, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by Gitbook最后更新: 2022-10-25 15:16:04

Block详解

此处详细介绍Block的相关内容。

主要包括：

- iOS的ObjC的正向开发时：Block代码是什么样的
- iOS的ObjC的逆向时：Block的详细定义，以及每个字段的含义是什么

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 16:36:50

Block正向开发

Block函数定义：

```
double (^multiplyTwoValues)(double, double);
```

Block函数写法：

```
^ (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

显示指定返回值类型：

```
^ double (double firstValue, double secondValue) {
    return firstValue * secondValue;
}
```

调用：

```
double (^multiplyTwoValues)(double, double) =
    ^(double firstValue, double secondValue) {
        return firstValue * secondValue;
};

double result = multiplyTwoValues(2,4);

 NSLog(@"%@", result);
```

可以从作用域内，直接引用值：

```
- (void)testMethod {
    int anInteger = 42;

    void (^testBlock)(void) = ^{
        NSLog(@"Integer is: %i", anInteger);
    };

    testBlock();
}
```

TODO:

- 继续整理 * 【整理】iOS的ObjC的基础知识：正向开发时Block相关知识

Block定义

TODO:

- 【记录】iOS中的Block的定义画出结构图

Block结构体定义

- 来源=官网源码
 - Block_private.h
 - 旧: https://opensource.apple.com/source/libclosure-38/Block_private.h
 - 新: https://opensource.apple.com/source/libclosure-63/Block_private.h
 - TODO: 更新版, 记得还有额外的属性, 抽空也去整理过来

-> 合并后的最新的理解:

Block定义的代码

```
struct Block_layout {
    void *isa;
    volatile int32_t flags; // contains ref count
    int32_t reserved;
    void (*invoke)(void *, ...);
    struct Block_descriptor_1 *descriptor;
    // imported variables
};

// merged new layout
struct Block_descriptor {
    // Block_descriptor_1
    uintptr_t reserved;
    uintptr_t size;

    // Block_descriptor_2
    // requires BLOCK_HAS_COPY_DISPOSE
    void (*copy)(void *dst, const void *src);
    void (*dispose)(const void *);

    // Block_descriptor_3
    // requires BLOCK_HAS_SIGNATURE
    const char *signature;
    const char *layout; // contents depend on BLOCK_HAS_EXTENDED_LAYOUT
};
```

Block定义的结构图

- 自己的详细图

- - 在线浏览
 - [iOS的Block定义的结构图 | ProcessOn免费在线作图,在线流程图,在线思维导图](#)
 - 别人的简单的图

◦

TODO:

- 【整理】iOS逆向心得：ObjC中Block的定义

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 16:40:35

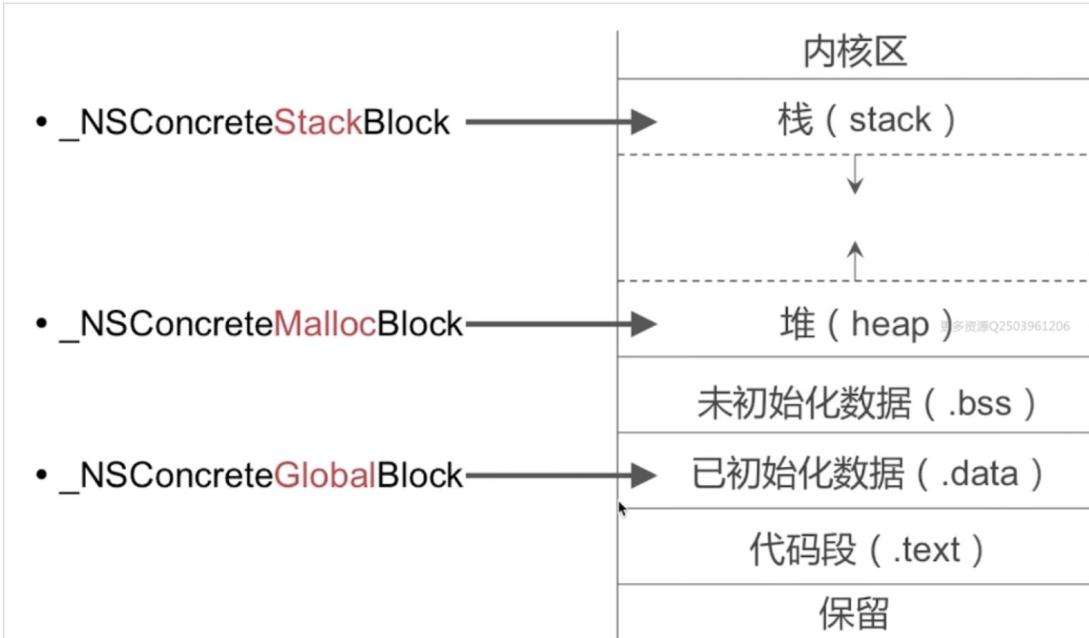
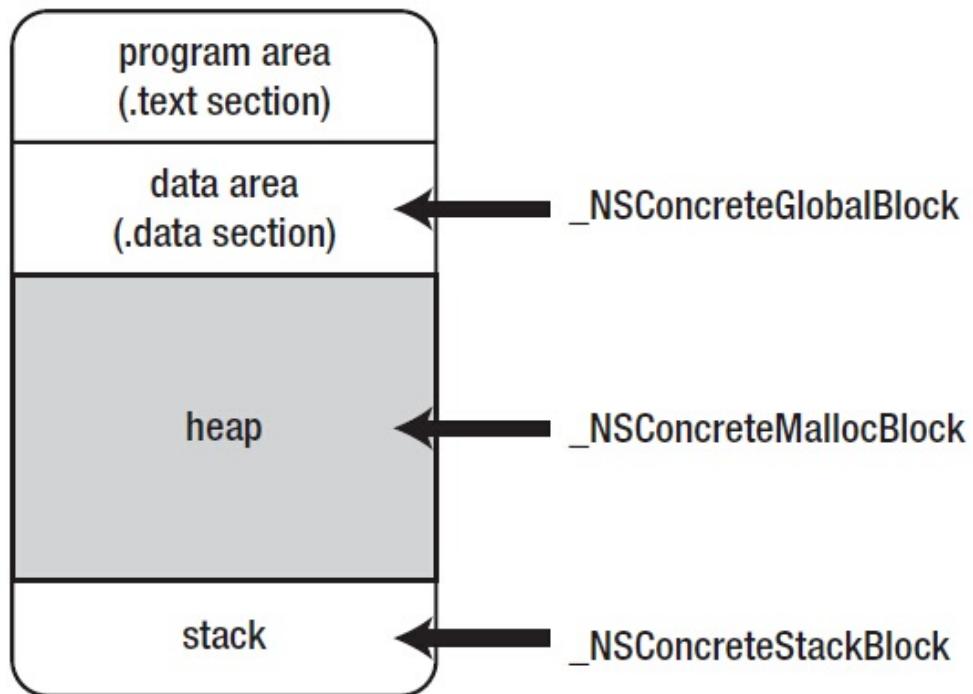
Block类型

Block有多种类型：

- Block类型
 - 大类：3类
 - global
 - malloc
 - stack
 - 具体：很多种
 - _NSConcreteGlobalBlock
 - _NSConcreteMallocBlock
 - _NSConcreteStackBlock
 - _NSConcreteAutoBlock
 - _NSConcreteFinalizingBlock
 - _NSConcreteWeakBlockVariable
- 最常见的3种Block类型
 - 文字
 - _NSConcreteGlobalBlock
 - _NSConcreteMallocBlock
 - _NSConcreteStackBlock
 - 图

Block Class	Copied From	How “Copy” Works
_NSConcreteStackBlock	Stack	Copy from the stack to the heap
_NSConcreteGlobalBlock	.data section of the program	Do nothing
_NSConcreteMallocBlock	Heap	Increment the reference count of the object

Application's memory arrangement



- Block类型识别和转换
 - 访问了auto变量的block是 `_NSStackBlock` 类型
 - 没有访问auto变量的block是 `_NSGlobalBlock` 类型
 - 而对 `_NSStackBlock` 类型进行copy操作就会变为 `_NSMallocBlock` 类型

iOS逆向中，IDA伪代码中最常看到的是：`_NSConcreteStackBlock`。

flags

Block中的 flags 是个 标签，一个int型数值，不同的位 bit 表示不同的含义。

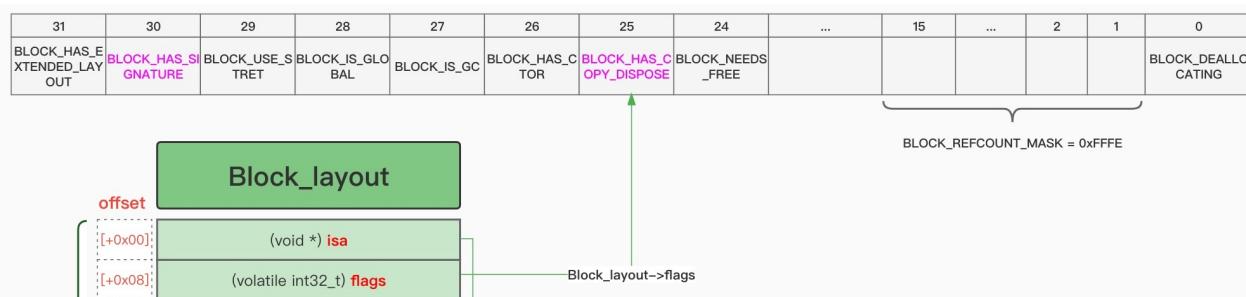
具体定义：

[Block_private.h](#)

```
// Values for Block_layout->flags to describe block objects
enum {
    BLOCK DEALLOCATING =      (0x0001), // runtime
    BLOCK REFCOUNT_MASK =     (0xffff), // runtime
    BLOCK NEEDS_FREE =        (1 << 24), // runtime
    BLOCK HAS COPY DISPOSE =  (1 << 25), // compiler
    BLOCK HASCTOR =          (1 << 26), // compiler: helpers have C++ code
    BLOCK ISGC =              (1 << 27), // runtime
    BLOCK ISGLOBAL =          (1 << 28), // compiler
    BLOCK USE_STRET =         (1 << 29), // compiler: undefined if !BLOCK HAS SIGNATURE
    BLOCK HAS SIGNATURE =    (1 << 30), // compiler
    BLOCK HAS EXTENDED LAYOUT = (1 << 31) // compiler
};

struct Block_layout {
    ...
    volatile int32_t flags; // contains ref count
    ...
};
```

详见整理的[Block结构图](#)中的定义：



reference引用

```
// 3
if (aBlock->flags & BLOCK_NEEDS_FREE) {
    // latches on high
    latching_incr_int(&aBlock->flags);
    return aBlock;
}
```

If the block's flags includes BLOCK_NEEDS_FREE then the block is a heap block (you'll see why shortly). In this case, all that needs doing is the reference count needs incrementing and then the same block returned.

如果flags中有BLOCK_NEEDS_FREE，则意味着：是个heap的block，即类型是 _NSConcreteMallocBlock

```
// 8  
result isa = _NSConcreteMallocBlock;
```

Here the block's isa pointer is set to be _NSConcreteMallocBlock, which means it's a heap block

最后设置为： _NSConcreteMallocBlock

crifan.org, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by Gitbook 最后更新：
2022-10-25 17:07:04

invoke

TODO:

- 【已解决】iOS逆向心得：Block的invoke函数调用时的函数参数
 - 【整理】iOS逆向调试心得：Block的invoke函数调用
-

Block中的 `invoke`，是核心的内容，决定了：具体要 调用 的（子） 函数 。

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：

2022-10-25 15:57:57

signature

TODO:

- 【整理】iOS逆向心得：Block的invoke函数的签名signature的含义
-

- signature

- v8@?0
 - 函数定义： void ^();
- v32@?0@"NSError"8@16@"TTHttpResponse"24
 - 函数定义： void ^(NSError *, id, TTHttpResponse *);
- v12@0:4@8
 - 函数定义： - (void) setSomething:(id) anObject;

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：

2022-10-25 15:48:06

descriptor

Block中的 `descriptor` 描述符，决定了Block有哪些额外的属性和具体定义。

`Block_private.h`

中的，3种 `Block_descriptor` : `Block_descriptor_1`、`Block_descriptor_2`、`Block_descriptor_3`

```
#define BLOCK_DESCRIPTOR_1 1
struct Block_descriptor_1 {
    uintptr_t reserved;
    uintptr_t size;
};

#define BLOCK_DESCRIPTOR_2 1
struct Block_descriptor_2 {
    // requires BLOCK_HAS_COPY_DISPOSE
    void (copy)(void *dst, const void *src);
    void (dispose)(const void *);
};

#define BLOCK_DESCRIPTOR_3 1
struct Block_descriptor_3 {
    // requires BLOCK_HAS_SIGNATURE
    const char *signature;
    const char *layout;      // contents depend on BLOCK_HAS_EXTENDED_LAYOUT
};

struct Block_layout {
    void *isa;
    volatile int32_t flags; // contains ref count
    int32_t reserved;
    void (invoke)(void *, ...);
    struct Block_descriptor_1 *descriptor;
    // imported variables
};
```

合并后，就是前面[Block详细定义](#)所整理的：

```
// merged new layout
struct Block_descriptor {
    // Block_descriptor_1
    uintptr_t reserved;
    uintptr_t size;

    // Block_descriptor_2
    // requires BLOCK_HAS_COPY_DISPOSE
    void (copy)(void *dst, const void *src);
    void (dispose)(const void *);

    // Block_descriptor_3
    // requires BLOCK_HAS_SIGNATURE
```

```
    const char signature;
    const char layout; // contents depend on BLOCK_HAS_EXTENDED_LAYOUT
};
```

TODO:

- Block_descriptor_1、Block_descriptor_2、Block_descriptor_3
 - 【整理】iOS逆向心得：Block中Block_descriptor_1、Block_descriptor_2、Block_descriptor_3的计算逻辑和位置

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 17:06:56

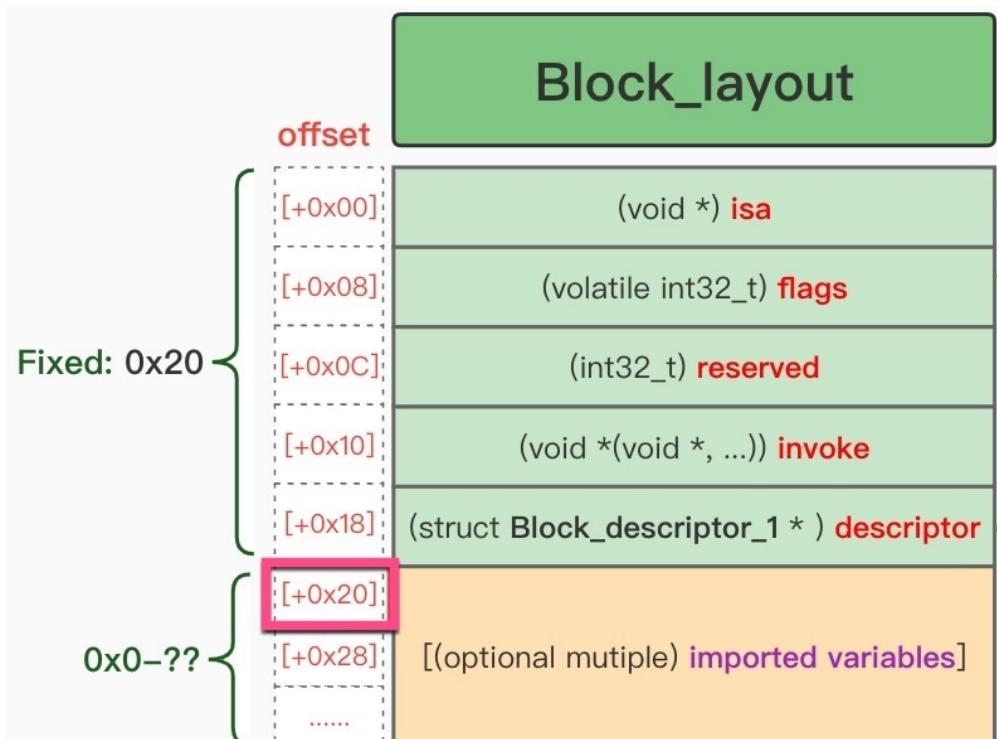
imported variables

TODO:

- 【已解决】iOS逆向心得：Block的invoke函数调用时的函数参数
- 【整理】iOS逆向心得：Block被调用时的函数参数
- 【已解决】研究抖音关注逻辑：Block函数调用时的参数传递逻辑之sub_93413B8调用sub_9341898
 - 举例
 - 【未解决】研究抖音关注逻辑：sub_93413B8调用__lldb_unnamed_symbol583654即sub_9341898的调用逻辑
- 计算（引用的变量）参数的个数
 - 【已解决】iOS逆向心得：Block中能否和如何计算invoke函数的参数的个数
-

Block是通过 `imported variables = 引用变量`，去实现参数传递的。

- 最新的理解：parameters vs imported variables
 - parameters
 - Block（的invoke函数）被调用时，传入的各个参数
 - x0=Block本身
 - x1、x2、x3等等：才是invoke函数的参数
 - imported variables
 - Block中的额外引用的变量的叫法：
 - Block中0x20之后额外引用的变量



- 标准叫法=Block定义中的注释的说法：`imported variables = 导入的变量`

- =Block引用的变量
- =Block额外引用的变量
- =Block额外的参数
- Block所引用的变量

◦

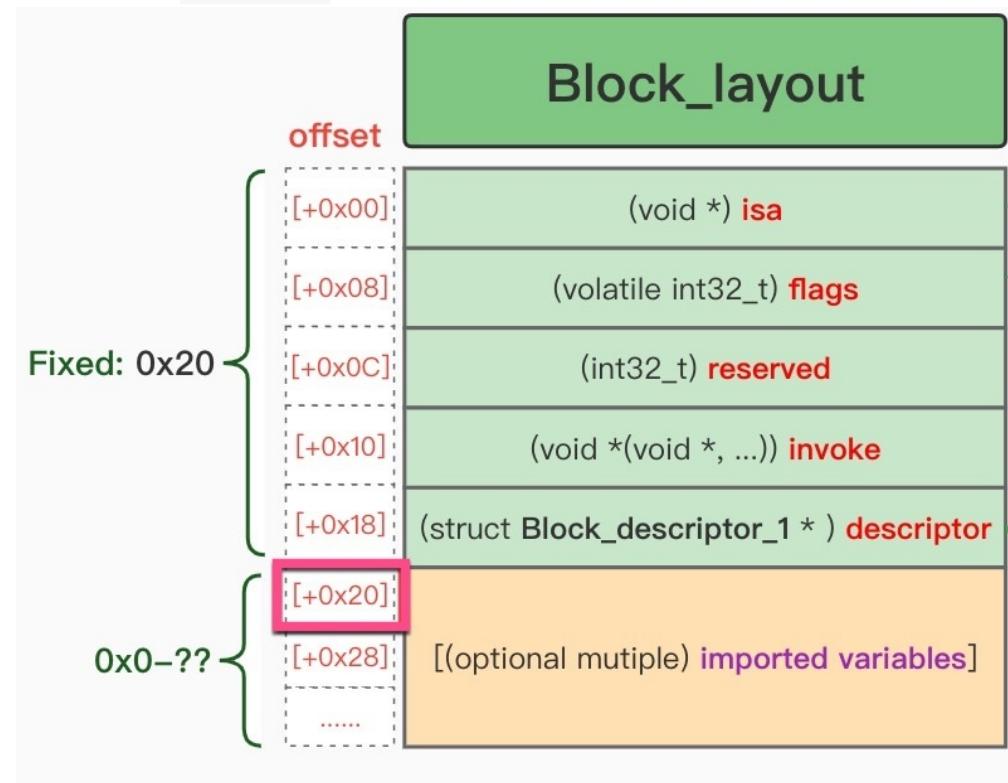
crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新:
2022-10-25 16:41:28

Block分析实例

Block最核心的部分

iOS逆向期间，和搞懂代码逻辑相关的，所涉及到的Block机制最核心的部分是

- `invoke` = 调用了什么函数
 - 知道A函数调用了B函数
 - 明白后续会调用到B函数 = B函数后续会同步或异步执行
- `imported variables` = 额外传入了什么参数
 - `imported variables` = 引用的变量 = 导入的变量 = 额外传入的参数
 - A函数调用B函数，往往有些额外的参数，由于是匿名函数，无法像普通函数调用传递参数，所以底层实现机制是，在 `descriptor` 之后，按照顺序去摆放所引用的参数 = 导入的变量 = `imported variables`
 - 具体的说就是，在 `block+0x20` 之后，可能有0或多个，引用的参数



crifan.org, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 16:27:42

YouTube逆向

此处通过具体实例来说明：

iOS逆向 YouTube 期间，如何搞懂 Block 异步函数调用，包括如何传递参数 引用变量 = imported variables。

YouTube的函数调用逻辑是：

```
当去掉广告过滤时，调用到 MLServerABRLoader onesieDataLoader:didCompleteChunksWithError: 的顺序：
:
- [MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]
    sub_10380F8 = -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:
latencyLogger:]_block
    sub_10381F8 = -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:qu
ue:latencyLogger:]_block_block + 344 =
        -[HAMBaseChunkLoadTask didCompleteWithError:] + 412
        -[MLServerABROnesieDataLoader chunks:didCompleteLoadingWithError:] + 92
        MLServerABRLoader onesieDataLoader:didCompleteChunksWithError:
参数举例：
[case 1]:
    arg1 0x0000000282b42000,
    arg2 1 element,
    arg3 domain: "com.google.ios.hamplayer" - code: 10000
[case 2]:
    arg1 <-> MLServerABROnesieDataLoader: 0x282bd5e80 ,
    arg2 <-> __NSSingleObjectArrayI 0x2807d1510 (
        <HAMMP4InitializationChunk: 0x282bc3b80>
    )
    arg3 NULL
```

其中的：

- sub_10380F8 = -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block

就是函数：

- -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]
的 Block 回调函数
 - 后续会异步执行

而后续继续调用了其他的Block回调函数：

- sub_10381F8 = -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block_block

下面就从对应的IDA伪代码中，去分析和解释具体逻辑和过程：

Block通过invoke调用子函数

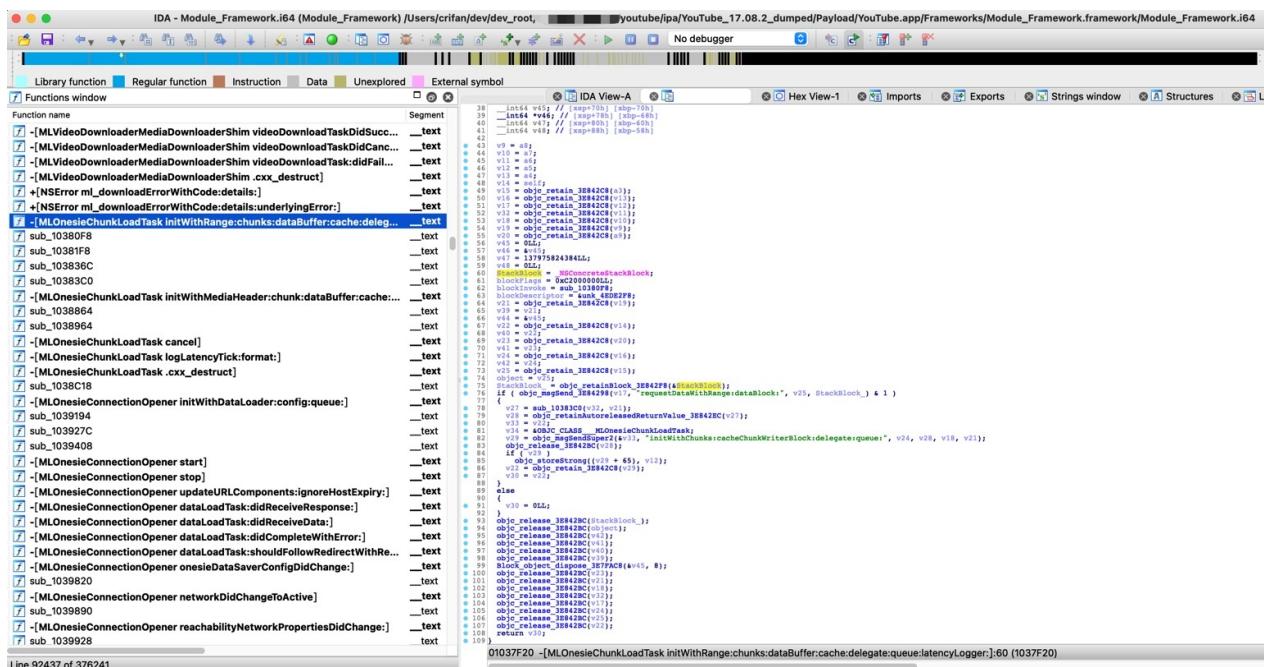
```
-[MLOnesieChunkLoadTask
initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]
```

第一层: -[MLOnesieChunkLoadTask
initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]

IDA伪代码:

```
MLOnesieChunkLoadTask * __cdecl [MLOnesieChunkLoadTask initWithRange chunks dataBuffer cache de
legate queue latencyLogger ](MLOnesieChunkLoadTask * self, SEL a2, id a3, id a4, id a5, id a6, i
d a7, id a8, id a9)
{
    ...
    id StackBlock; // [xsp+20h] [xbp-C0h]
    _int64 blockFlags; // [xsp+28h] [xbp-B8h]
    _int64 (__fastcall blockInvoke)(_QWORD *, _int64, char, char); // [xsp+30h] [xbp-B0h]
    void * blockDescriptor; // [xsp+38h] [xbp-A8h]

    ...
    StackBlock = _NSConcreteStackBlock;
    blockFlags = 0xC2000000LL;
    blockInvoke = sub_10380F8;
    blockDescriptor = Bunk_4EDE2F8;
    v21 = objc_retain_3E842C8(v19);
    ...
    StackBlock_ = objc_retainBlock_3E842F8(&StackBlock);
    if (objc_msgSend_3E84298(v17, "requestDataWithRange:dataBlock:", v25, StackBlock_) & 1 )
    ...
}
```



而其中的Block中被调用的，后续会异步执行的函数是：

- sub_10380F8

就是此处的，iOS逆向期间，调试找出的，函数调用堆栈中的：

- -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block

对应的真正触发调用子函数的代码是：

```
objc_msgSend_3E84298(v17, "requestDataWithRange:dataBlock:", v25, StackBlock_)
```

对应的 v17 requestDataWithRange:dataBlock: 的内部，会有 dispatch_async 之类的真正触发调用此处的 Block 的 invoke 函数

```
-[MLOnesieChunkLoadTask
initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block
```

第二层： -[MLOnesieChunkLoadTask
initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block

IDA伪代码是：

```
void __fastcall sub_10380F8(_QWORD a1, __int64 a2, char a3, char a4)
{
    ...
    struct objc_object v7; // x23
    struct objc_object v8; // x19
    id stackBlock; // [xsp+8h] [xbp-88h]
    __int64 blockFlags; // [xsp+10h] [xbp-80h]
    void *(__fastcall *blockInvoke)(__int64); // [xsp+18h] [xbp-78h]
    void *blockDescriptor; // [xsp+20h] [xbp-70h]

    ...
    stackBlock = _NSConcreteStackBlock;
    blockFlags = 0xC2000000LL;
    blockInvoke = sub_10381F8;
    blockDescriptor =unk_4EDE2C8;

    ...
    v7 = v6[4];
    ...
    dispatch_async_3E82B64(v7, &stackBlock);
    ...
}
```

```

void __fastcall sub_10380F8(_QWORD *a1, _int64 a2, char a3, char a4)
{
    char v5; // w19
    char v6; // x20
    struct objc_object *v7; // x23
    struct objc_object *v8; // x19
    id stackBlock; // [xsp+8h] [xbp-88h]
    int blockFlags; // [xsp+10h] [xbp-80h]
    void *blockDescriptor; // [xsp+20h] [xbp-70h]
    id v13; // [xsp+28h] [xbp-68h]
    id v14; // [xsp+30h] [xbp-60h]
    id v15; // [xsp+32h] [xbp-58h]
    id v16; // [xsp+40h] [xbp-50h]
    id object; // [xsp+48h] [xbp-48h]
    _int64 v18; // [xsp+50h] [xbp-40h]
    char v19; // [xsp+58h] [xbp-38h]
    char v20; // [xsp+59h] [xbp-37h]

    v4 = a4;
    v5 = a3;
    v6 = a1;
    stackBlock = NSConcreteStackBlock;
    blockFlags = 0x2000000000000000L;
    blockDescriptor = unk_4ED2E2C8;
    v13 = objc_retain_3E842C8(a2);
    v14 = v6[9];
    v7 = v5[4];
    v8 = objc_retain_3E842C8(v6[5]);
    v15 = objc_retain_3E842C8(v6[6]);
    v19 = v6[7];
    v16 = objc_retain_3E842C8(v6[7]);
    v20 = v5;
    object = objc_retain_3E842C8(v6[8]);
    v9 = objc_retain_3E842C8(v13);
    dispatch_async_3E82B64(v9, &stackBlock);
    objc_release_3E842BC(object);
    objc_release_3E842BC(v16);
    objc_release_3E842BC(v15);
    objc_release_3E842BC(v14);
    objc_release_3E842BC(v13);
    objc_release_3E842BC(v8);
}

```

01038130 sub_10380F8:24 (1038130)

而此Block中被调用的函数是：

- sub_10381F8

也正好对应着，调试时通过函数调用堆栈找到的：

- -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block_block

而真正触发子函数调用的代码是：

- dispatch_async_3E82B64(v7, &stackBlock);

```

-[MLOnesieChunkLoadTask
initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block_block

```

第三层： -[MLOnesieChunkLoadTask

```

initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block_block

```

IDA伪代码是：

```

void __fastcall sub_10381F8(_int64 a1)
{
    _int64 v1; // x19
}

```

```

void *v2; // x0
__int64 v3; // x2
struct objc_object v4; // x0
id v5; // x0
struct objc_object v6; // x20
struct objc_object v7; // x0
struct objc_object v8; // x21
struct objc_object v9; // x20
struct objc_object v10; // x0
__int64 v11; // [xsp+0h] [xbp-40h]
__objc2_class *v12; // [xsp+8h] [xbp-38h]
__int64 v13; // [xsp+10h] [xbp-30h]
__objc2_class *v14; // [xsp+18h] [xbp-28h]

v1 = a1;
v2 = *(a1 + 32);
if ( v2 )
{
    *((*(v1 + 72) + 8LL) + 24LL) += objc_msgSend_3E84298(v2, "length");
    v3 = *(v1 + 32);
    v13 = *(v1 + 40);
    v14 = __OBJC_CLASS__MLOnesieChunkLoadTask;
    objc_msgSendSuper2(v13, "didReceiveData:error:", v3, 0LL);
    if ( *(v1 + 48) )
    {
        if ( *(v1 + 80) )
        {
            v4 = objc_msgSend_3E84298( (v1 + 56), "firstObject");
            v5 = objc_retainAutoreleasedReturnValue_3E842EC(v4);
            v6 = v5;
            v7 = objc_msgSend_3E84298(v5, "format");
            v8 = objc_retainAutoreleasedReturnValue_3E842EC(v7);
            objc_release_3E842BC(v6);
            objc_msgSend_3E84298(*(v1 + 40), "logLatencyTick:format:", *(v1 + 48), v8);
            objc_release_3E842BC(v8);
        }
    }
    if ( *(v1 + 81) )
    {
        if ( objc_msgSend_3E84298(*(v1 + 64), "isBounded")
            && objc_msgSend_3E84298(*(v1 + 64), "length") == *((*(v1 + 72) + 8LL) + 24LL) )
        {
            v9 = 0LL;
        }
        else
        {
            v10 = objc_msgSend_3E84298( __OBJC_CLASS__NSError, "HAMErrorWithCode:", 10000LL);
            v9 = objc_retainAutoreleasedReturnValue_3E842EC(v10);
        }
        v11 = *(v1 + 40);
        v12 = __OBJC_CLASS__MLOnesieChunkLoadTask;
        objc_msgSendSuper2( v11, "didCompleteWithError:", v9, v11, __OBJC_CLASS__MLOnesieChunkLoadT
ask);
        objc_release_3E842BC(v9);
    }
}

```

```

void __fastcall sub_10381F8(_int64 a1)
{
    int64 v1; // x19
    void *v2; // x0
    int64 v3; // x2
    struct objc_object *v4; // x0
    id v5; // x0
    struct objc_object *v6; // x20
    struct objc_object *v7; // x0
    struct objc_object *v8; // x21
    struct objc_object *v9; // x20
    struct objc_object *v10; // x20
    _int64 v11; // [xsp+0h] [xbp-40h]
    _objc2_class *v12; // [xsp+8h] [xbp-38h]
    _int64 v13; // [xsp+10h] [xbp-30h]
    _objc2_class *v14; // [xsp+18h] [xbp-28h]
    ...
    v1 = a1;
    v2 = v1 + 32;
    if ( v2 )
    {
        *((*(v1 + 72) + 8LL) + 24LL) += objc_msgSend_3E84298(v2, "length");
        v3 = *(v1 + 32);
        v3 = *(v1 + 40);
        v3 = objc_release_3E842BC(v3);
        objc_msgSendSuper2(v11, "didReceiveData:error:", v3, 0LL);
        if ( !(v1 + 48) )
        {
            if ( !(v1 + 80) )
            {
                v4 = objc_msgSend_3E84298(*(v1 + 56), "firstObject");
                v5 = objc_retainAutoreleasedReturnValue_3E842BC(v4);
                v6 = v5;
                v7 = objc_msgSend_3E84298(v5, "format");
                v8 = objc_release_3E842BC(v7);
                objc_release_3E84298(v5);
               objc_msgSend_3E84298(*(v1 + 40), "logLatencyTick:format:", *(v1 + 48), v8);
               objc_release_3E842BC(v8);
            }
            else
            {
                if ( !(v1 + 81) )
                    if ( objc_msgSend_3E84298(*(v1 + 64), "isBounded") && objc_msgSend_3E84298(*(v1 + 64), "length") == *((*(v1 + 72) + 8LL) + 24LL) )
                        v9 = 0LL;
                    else
                    {
                        v10 = objc_msgSend_3E84298(*(v1 + 64), "NSErrorWithCode:", 10000LL);
                        v9 = *(v1 + 40);
                        v11 = objc_release_3E842BC(v10);
                        v11 = *(v1 + 40);
                        v11 = objc_CLASS__MLOnesieChunkLoadTask;
                        objc_msgSendSuper2(v11, "didCompleteWithError:", v9, v11, &OBJC_CLASS__MLOnesieChunkLoadTask);
                       objc_release_3E842BC(v11);
                    }
            }
        }
    }
}

```

进一步的，从伪代码中，可以看出此函数的大概逻辑是：

- 判断收到的数据？是否出错，如果出错，则报错。

-》由此实现，iOS逆向期间，一点点找到被调用的函数，期间可能会涉及到Block的异步调用，搞懂函数的代码逻辑。

Block通过imported variables传递参数

接下来继续分析：Block的引用变量的传递过程

而如何搞懂此处Block的函数异步调用时，传递了哪些额外参数= imported variables=引用变量？

核心思路是：

- 从 block+0x20 之后，如果有连续的变量赋值，那基本上就是对应所引用的参数
 - 同时，也去被调用的函数中，互相对照，才能确定
 - 被调用函数中，如果从 block+0x20 之后位置获取值，也就是，获取对应的 imported variables，就的确表示对应的值是引用的变量了

对应此处具体的过程就是：

MLOnesieChunkLoadTask initWithRange传递给sub_10380F8

先确认 block+0x20 的位置

先从 Block 的起始位置去算起：

```
MLOnesieChunkLoadTask * __cdecl __MLOnesieChunkLoadTask initWithRange chunks dataBuffer cache de
legate queue latencyLogger ](MLOnesieChunkLoadTask * self, SEL a2, id a3, id a4, id a5, id a6, i
d a7, id a8, id a9)
{
    ...
    id StackBlock; // [xsp+20h] [xbp-C0h]
    _int64 blockFlags; // [xsp+28h] [xbp-B8h]
    void (*blockInvoke)(_QWORD *, _int64, char, char); // [xsp+30h] [xbp-B0h]
    void *blockDescriptor; // [xsp+38h] [xbp-A8h]

    StackBlock = _NSConcreteStackBlock;
    blockFlags = 0xC2000000LL;
    blockInvoke = sub_10380F8;
    blockDescriptor = unk_4EDE2F8;
    StackBlock = _NSConcreteStackBlock;
```

可以算出：

- [block+0x0] = Block 自身 = _NSConcreteStackBlock
 - 此处： [xsp+20h] == sp+0x20
- [block+0x8] = flags = 0xC2000000LL
 - 此处： [xsp+28h] == sp+0x28
- [block+0x10] = invoke = sub_10380F8
 - 此处： [xsp+30h] == sp+0x30
- [block+0x18] = descriptor = &unk_4EDE2F8
 - 此处： [xsp+38h] == sp+0x38

继续计算出 imported variable :

- [block+0x20] = 第1个引用变量
 - 此处： [xsp+40h] = sp+0x40 的 v39
- [block+0x28] = 第2个引用变量
 - 此处： [xsp+48h] = sp+0x48 的 v40
- [block+0x30] = 第3个引用变量
 - 此处： [xsp+50h] = sp+0x50 的 v41
- [block+0x38] = 第4个引用变量
 - 此处： [xsp+58h] = sp+0x58 的 v42
- [block+0x40] = 第5个引用变量
 - 此处： [xsp+60h] = sp+0x60 的 object
 - 注：此处变量名之所以叫 object，而不是叫做 v43，其实是反编译器尝试理解代码逻辑，并给变量命名，结果由于（本身就很难）完全猜出代码逻辑，所以变量名命名的不是很合适，此种情况属于正常现象
- [block+0x48] = 第6个引用变量
 - 此处： [xsp+68h] = sp+0x68 的 v44
- [block+0x50] = 第7个引用变量
 - 此处： [xsp+70h] = sp+0x70 的 v45

- [block+0x58] = 第8个引用变量
 - 此处: [xsp+78h] = sp+0x78 的 v46
- [block+0x60] = 第9个引用变量
 - 此处: [xsp+80h] = sp+0x80 的 v47
- [block+0x68] = 第10个引用变量
 - 此处: [xsp+88h] = sp+0x88 的 v48

需要说明的是，一般Block的引用变量的个数，也就2, 3个左右，此处看起来引用变量多达10个

而目前暂时不是完全确定，后续变量的确是引用这么多变量

->需要后续看Block中被调用函数的实际使用情况，才能确定，到底引用了几个变量

继续分析Block的引用变量：

代码稍微优化（改名）后是：

```
void __fastcall sub_10380F8(_QWORD inputBlock, __int64 a2, char a3, char a4)
{
    char v4; // w19
    char v5; // w20
    _QWORD *curBlock; // x21
    struct objc_object v7; // x23
    struct objc_object v8; // x19
    id stackBlock; // [xsp+8h] [xbp-88h]
    __int64 blockFlags; // [xsp+10h] [xbp-80h]
    void (*blockInvoke)(__int64); // [xsp+18h] [xbp-78h]
    void *blockDescriptor; // [xsp+20h] [xbp-70h]
    id v13; // [xsp+28h] [xbp-68h]
    id v14; // [xsp+30h] [xbp-60h]
    id v15; // [xsp+38h] [xbp-58h]
    id v16; // [xsp+40h] [xbp-50h]
    id object; // [xsp+48h] [xbp-48h]
    __int64 v18; // [xsp+50h] [xbp-40h]
    char v19; // [xsp+58h] [xbp-38h]
    char v20; // [xsp+59h] [xbp-37h]

    v4 = a4;
    v5 = a3;
    curBlock = inputBlock;
    stackBlock = _NSConcreteStackBlock;
    blockFlags = 0xC2000000LL;
    blockInvoke = sub_10381F8;
    blockDescriptor = &unk_4EDE2C8;
    v13 = objc_retain_3E842C8(a2);
    v18 = curBlock[9];
    v7 = curBlock[4];
    v14 = objc_retain_3E842C8(curBlock[5]);
    v15 = objc_retain_3E842C8(curBlock[6]);
    v19 = v4;
    v16 = objc_retain_3E842C8(curBlock[7]);
    v20 = v5;
    object = objc_retain_3E842C8(curBlock[8]);
    v8 = objc_retain_3E842C8(v13);
    dispatch_async_3E82B64(v7, &stackBlock);
}
```

```
...  
}
```

此处可以看出其中的：

- curBlock[4]
- curBlock[5]
- curBlock[6]
- curBlock[7]
- curBlock[8]
- curBlock[9]

就是：

- 来自上层的Block所传入的引用变量= `imported variables`

同时也要清楚，为何是从 `curBlock[4]` 的4开始，而不是从0开始，则是因为：

- 从 0 开始的 0 到 3，都是 Block 相关的属性
 - `curBlock[0] = block + 0x0 = Block`
 - `curBlock[1] = block + 0x8 = flags`
 - `curBlock[2] = block + 0x10 = invoke`
 - `curBlock[3] = block + 0x18 = descriptor`

所以后续的 `imported variables` 才是：

- `curBlock[4] = block + 0x20`
- `curBlock[5] = block + 0x28`
- `curBlock[6] = block + 0x30`
- `curBlock[7] = block + 0x38`
- `curBlock[8] = block + 0x40`
- `curBlock[9] = block + 0x48`

至此，就清楚了，从：

- `-[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]`

调用了Block，其中：

- 被调用的函数是： `sub_10380F8`
 - == `-[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block`
- 传入的参数=引用的变量
 - `[block+0x20]` = 第1个引用变量：当时的 `[xsp+40h] = sp +0x40` 的 v39
 - 被调用函数 `sub_10380F8` 中所引用的： `curBlock[4]`
 - `[block+0x28]` = 第2个引用变量：当时的 `[xsp+48h] = sp +0x48` 的 v40
 - 被调用函数 `sub_10380F8` 中所引用的： `curBlock[5]`
 - `[block+0x30]` = 第3个引用变量：当时的 `[xsp+50h] = sp +0x50` 的 v41
 - 被调用函数 `sub_10380F8` 中所引用的： `curBlock[6]`
 - `[block+0x38]` = 第4个引用变量：当时的 `[xsp+58h] = sp +0x58` 的 v42
 - 被调用函数 `sub_10380F8` 中所引用的： `curBlock[7]`

- [block+0x40] = 第5个引用变量：当时的 [xsp+60h] = sp +0x60 的 object
 - 被调用函数sub_10380F8中所引用的: curBlock[8]
- [block+0x48] = 第6个引用变量：当时的 [xsp+68h] = sp +0x68 的 v44
 - 被调用函数sub_10380F8中所引用的: curBlock[9]

至此，其中函数调用的大体逻辑，就清楚了：

- -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]
- 其中用Block回调了函数： sub_10380F8 (= -[MLOnesieChunkLoadTask initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block)
 - 且通过 imported variables 传入了 6 个参数变量

优化IDA伪代码

接着，也就可以继续去优化代码了：

给对应的变量改名字-》变成人类易读的代码-》让看代码的你自己更容易读懂代码逻辑

MLOnesieChunkLoadTask initWithRange

```

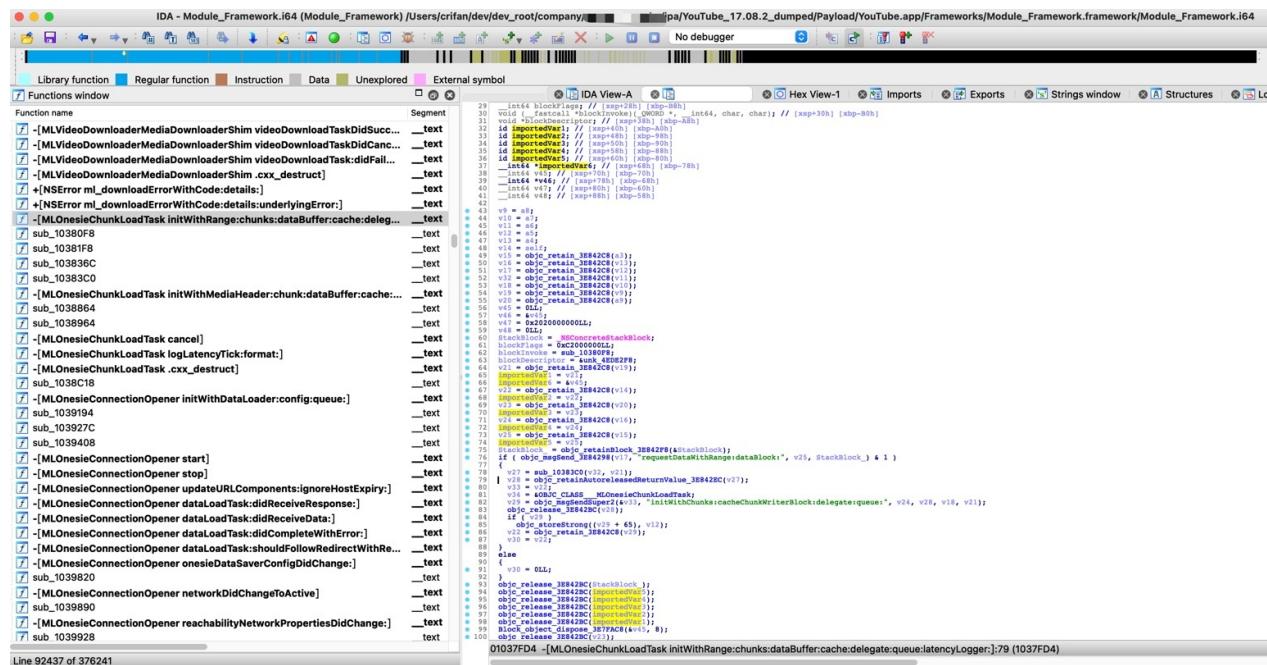
MLOnesieChunkLoadTask * __cdecl -[MLOnesieChunkLoadTask initWithRange chunks dataBuffer cache de
legate queue latencyLogger ](MLOnesieChunkLoadTask * self, SEL a2, id a3, id a4, id a5, id a6, i
d a7, id a8, id a9)
{
  ...
  id StackBlock; // [xsp+20h] [xbp-C0h]
  __int64 blockFlags; // [xsp+28h] [xbp-B8h]
  void (*blockInvoke)(__QWORD *, __int64, char, char); // [xsp+30h] [xbp-B0h]
  void *blockDescriptor; // [xsp+38h] [xbp-A8h]
  id importedVar1; // [xsp+40h] [xbp-A0h]
  id importedVar2; // [xsp+48h] [xbp-98h]
  id importedVar3; // [xsp+50h] [xbp-90h]
  id importedVar4; // [xsp+58h] [xbp-88h]
  id importedVar5; // [xsp+60h] [xbp-80h]
  __int64 importedVar6; // [xsp+68h] [xbp-78h]
  ...

  StackBlock = _NSConcreteStackBlock;
  blockFlags = 0xC2000000LL;
  blockInvoke = sub_10380F8;
  blockDescriptor = unk_4EDE2F8;
  v21 = objc_retain_BE842C8(v19);
  importedVar1 = v21;
  importedVar6 = &v45;
  v22 = objc_retain_BE842C8(v14);
  importedVar2 = v22;
  v23 = objc_retain_BE842C8(v20);
  importedVar3 = v23;
  v24 = objc_retain_BE842C8(v16);
  importedVar4 = v24;
  v25 = objc_retain_BE842C8(v15);
  importedVar5 = v25;
}

```

StackBlock_ = **objc_retainBlock_3E842F8**(**StackBlock**)

1



sub 10380F8

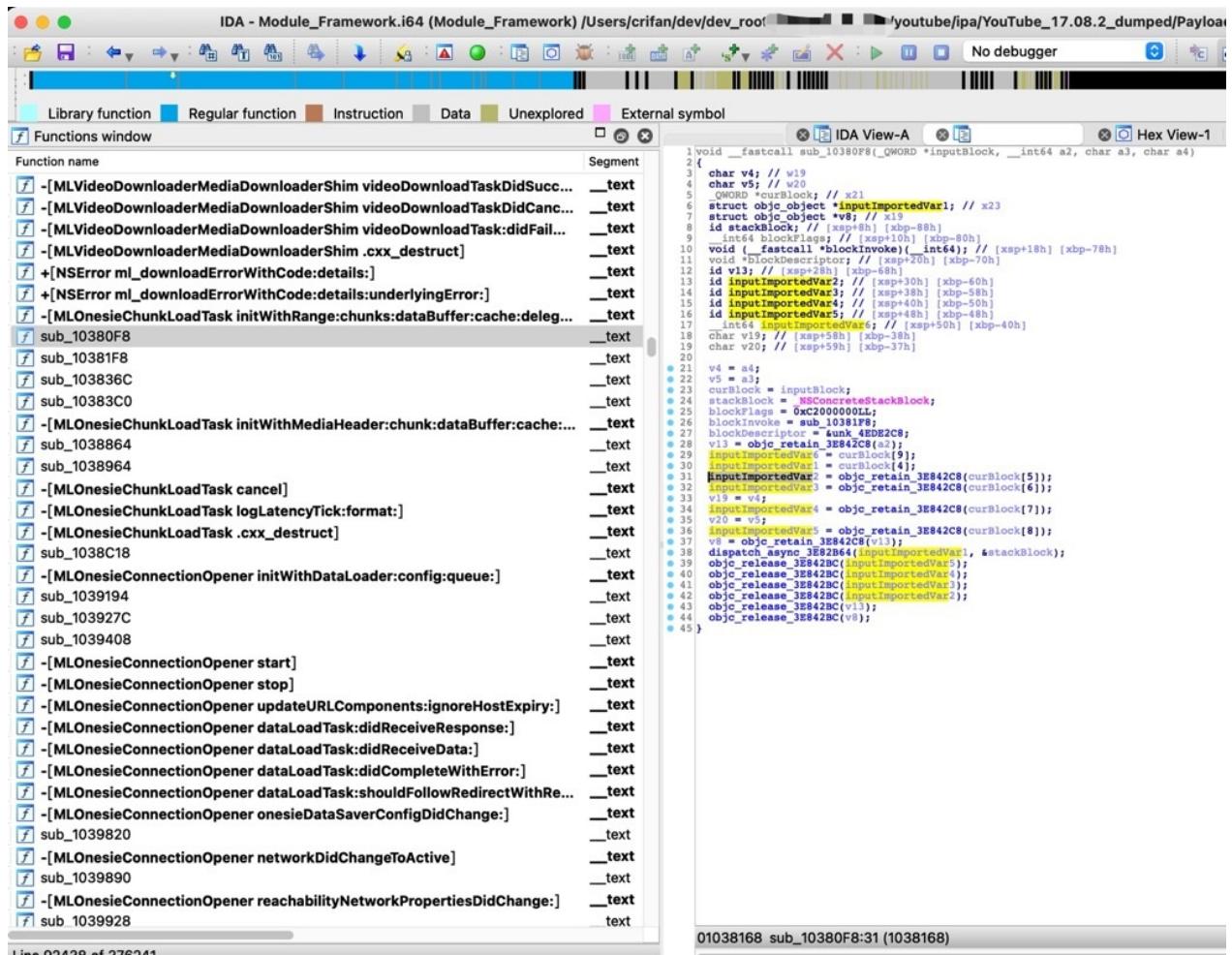
```
void __fastcall sub_10380F8(_QWORD inputBlock, __int64 a2, char a3, char a4)
{
    ...
    _QWORD curBlock; // x21
    struct objc_object inputImportedVar1; // x23
    struct objc_object v8; // x19
    id stackBlock; // [xsp+8h] [xbp-88h]
    __int64 blockFlags; // [xsp+10h] [xbp-80h]
    void (*blockInvoke)(__int64); // [xsp+18h] [xbp-78h]
    void *blockDescriptor; // [xsp+20h] [xbp-70h]
    id v13; // [xsp+28h] [xbp-68h]
    id inputImportedVar2; // [xsp+30h] [xbp-60h]
    id inputImportedVar3; // [xsp+38h] [xbp-58h]
    id inputImportedVar4; // [xsp+40h] [xbp-50h]
    id inputImportedVar5; // [xsp+48h] [xbp-48h]
    __int64 inputImportedVar6; // [xsp+50h] [xbp-40h]
    ...

    curBlock = inputBlock;
    stackBlock = _NSConcreteStackBlock;
    blockFlags = 0xC2000000LL;
    blockInvoke = sub_10381F8;
    blockDescriptor = unk_4EDE2C8;
    v13 = objc_retain_3E842C8(a2);
    inputImportedVar6 = curBlock[9];
    inputImportedVar1 = curBlock[4];
    inputImportedVar2 = objc_retain_3E842C8(curBlock[5]);
    inputImportedVar3 = objc_retain_3E842C8(curBlock[6]);
}
```

```

v19 = v4;
objc_retain(inputImportedVar4 = objc_retain(curBlock[7]));
v20 = v5;
objc_retain(inputImportedVar5 = objc_retain(curBlock[8]));
v8 = objc_retain(v13);
dispatch_async(dispatch_get_main_queue(), ^{
    ...
})
}

```



以及，用上述同样的方法去分析，之后的

- sub_10380F8
 - = -[MLOnesieChunkLoadTask

initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block

所调用的Block函数：

- sub_10381F8
 - = -[MLOnesieChunkLoadTask

initWithRange:chunks:dataBuffer:cache:delegate:queue:latencyLogger:]_block_block

中的Block引用变量，然后分析完毕后，再去优化代码，效果是这样的：

```

void __fastcall sub_10381F8(__int64 inputBlock)
{
    __int64 curBlock; // x19
    void * importedVar1_; // x0
    __int64 importedVar1; // x2
    struct objc_object v4; // x0
    id v5; // x0
    struct objc_object v6; // x20
    struct objc_object v7; // x0
    struct objc_object v8; // x21
    struct objc_object v9; // x20
    struct objc_object v10; // x0
    __int64 importedVar2_; // [xsp+0h] [xbp-40h]
    __objc2_class v12; // [xsp+8h] [xbp-38h]
    __int64 importedVar2; // [xsp+10h] [xbp-30h]
    __objc2_class v14; // [xsp+18h] [xbp-28h]

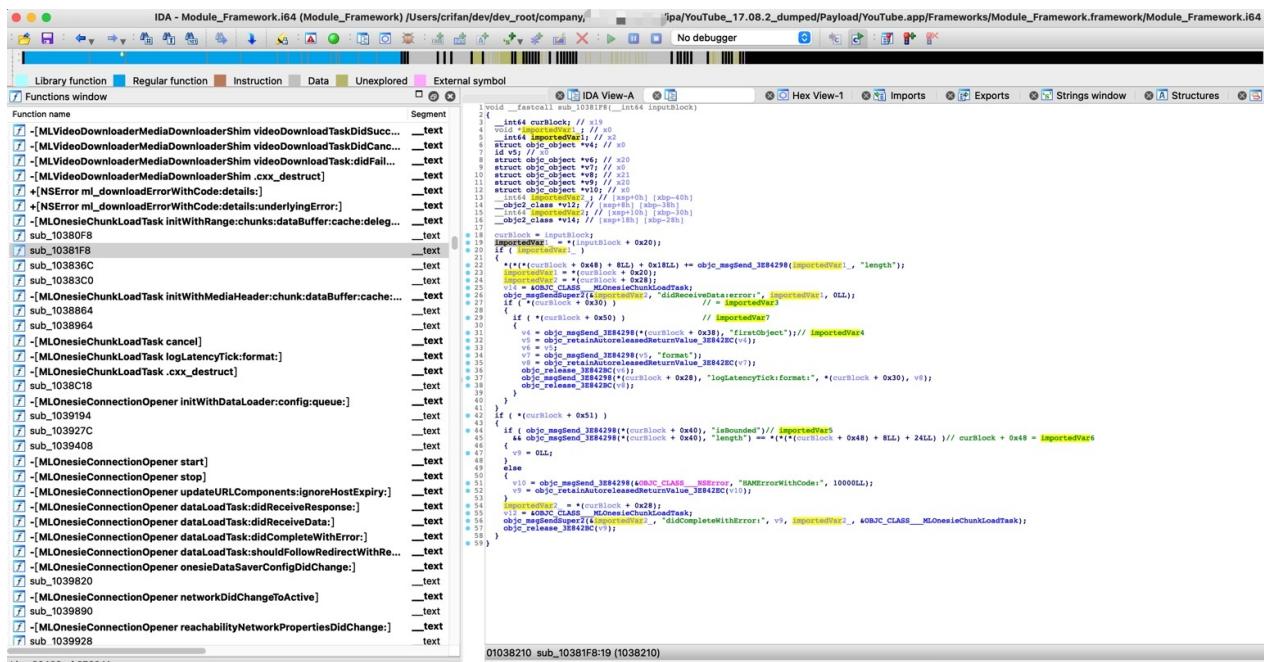
    curBlock = inputBlock;
    importedVar1_ = *(inputBlock + 0x20);
    if ( importedVar1_ )
    {
        *((*(curBlock + 0x48) + 8LL) + 0x18LL) += objc_msgSend_3E84298(importedVar1_, "length");
        importedVar1 = *(curBlock + 0x20);
        importedVar2 = *(curBlock + 0x28);
        v14 = .OBJC_CLASS__MLOnesieChunkLoadTask;
        objc_msgSendSuper2(importedVar2, "didReceiveData:error:", importedVar1, 0LL);
        if ( *(curBlock + 0x30) ) // = importedVar3
        {
            if ( *(curBlock + 0x50) ) // importedVar7
            {
                v4 = objc_msgSend_3E84298( (curBlock + 0x38), "firstObject");// importedVar4
                v5 = objc_retainAutoreleasedReturnValue_3E842EC(v4);
                v6 = v5;
                v7 = objc_msgSend_3E84298(v5, "format");
                v8 = objc_retainAutoreleasedReturnValue_3E842EC(v7);
                objc_release_3E842BC(v6);
                objc_msgSend_3E84298(*(curBlock + 0x28), "logLatencyTick:format:", *(curBlock + 0x30),
v8);
                objc_release_3E842BC(v8);
            }
        }
        if ( *(curBlock + 0x51) )
        {
            if ( objc_msgSend_3E84298(*(curBlock + 0x40), "isBounded")// importedVar5
                && objc_msgSend_3E84298(*(curBlock + 0x40), "length") == *((((curBlock + 0x48) + 8LL) +
24LL) )// curBlock + 0x48 = importedVar6
            {
                v9 = 0LL;
            }
            else
            {
                v10 = objc_msgSend_3E84298(.OBJC_CLASS__NSError, "HAMErrorWithCode:", 10000LL);
                v9 = objc_retainAutoreleasedReturnValue_3E842EC(v10);
            }
        }
    }
}

```

```
    }

    importedVar2_ = *(curBlock + 0x28);
    v12 = objc_CLASS__MLOnesieChunkLoadTask;
    objc_msgSendSuper2(importedVar2_, "didCompleteWithError:", v9, importedVar2_, objc_CLASS__MLOnesieChunkLoadTask);
    objc_release_3E842BC(v9);

}
}
```



sub 10381F8进一步优化

以及，在满足条件：

- 对于objc_msgSend等逻辑一定的了解
 - 加上此处YouTube的头文件的配合：查找相关函数，可以确定是哪个类
 - 其他相关知识

可以进一步优化为：

```
void __fastcall sub_10381F8(__int64 inputBlock)
{
    __int64 curBlock; // x19
    void receivedData; // x0
    __int64 receivedData_; // x2
    struct objc_object *v4; // x0
    id v5; // x0
    struct objc_object *v6; // x20
    struct objc_object *v7; // x0
    struct objc_object *v8; // x21
    struct objc_object *newHamError_; // x20
    struct objc_object newHamError; // x0
    __int64 curMLOnesieChunkLoadTask_; // [xsp+0h]
    __objc2_class *v12; // [xsp+8h] [xbp-38h]
    __int64 curMLOnesieChunkLoadTask; // [xsp+10h]
    __objc2_class *v14; // [xsp+18h] [xbp-28h]
```

```

curBlock = inputBlock;
receivedData = *(inputBlock + 0x20);
if ( receivedData )
{
    *((*(curBlock + 0x48) + 8LL) + 0x18LL) += objc_msgSend_3E84298(receivedData, "length");
    receivedData_ = *(curBlock + 0x20);
    curMLOnesieChunkLoadTask = *(curBlock + 0x28);
    v14 =_OBJC_CLASS__MLOnesieChunkLoadTask;
    objc_msgSendSuper2(curMLOnesieChunkLoadTask, "didReceiveData:error:", receivedData_, 0LL);
    if ( *(curBlock + 0x30) ) // = importedVar3 = LatencyTick
    {
        if ( *(curBlock + 0x50) ) // importedVar7
        {
            v4 = objc_msgSend_3E84298( (curBlock + 0x38), "firstObject");// importedVar4
            v5 = objc_retainAutoreleasedReturnValue_3E842EC(v4);
            v6 = v5;
            v7 = objc_msgSend_3E84298(v5, "format");
            v8 = objc_retainAutoreleasedReturnValue_3E842EC(v7);
           objc_release_3E842BC(v6);
            objc_msgSend_3E84298(*(curBlock + 0x28), "logLatencyTick:format:", *(curBlock + 0x30),
v8);
            objc_release_3E842BC(v8);
        }
    }
    if ( *(curBlock + 0x51) ) // importedVar8
    {
        if ( objc_msgSend_3E84298(*(curBlock + 0x40), "isBounded")// importedVar5 = HAMRange
            && objc_msgSend_3E84298(*(curBlock + 0x40), "length") == *((*(curBlock + 0x48) + 8LL) +
0x18LL) )// curBlock + 0x48 = importedVar6
        {
            newHamError_ = 0LL;
        }
        else
        {
            newHamError = objc_msgSend_3E84298(_OBJC_CLASS__NSError, "HAMErrorWithCode:", 10000LL);
            newHamError_ = objc_retainAutoreleasedReturnValue_3E842EC(newHamError);
        }
        curMLOnesieChunkLoadTask_ = *(curBlock + 0x28);
        v12 = _OBJC_CLASS__MLOnesieChunkLoadTask;
        objc_msgSendSuper2(curMLOnesieChunkLoadTask_, "didCompleteWithError:", newHamError_);
        objc_release_3E842BC(newHamError_);
    }
}
}

```

```

void fastcall sub_10381F8(_int64 inputBlock)
{
    _int64 curBlock; // x19
    _int64 receivedData; // x2
    struct objc_object *v4; // x20
    struct objc_object *v5; // x21
    struct objc_object *v6; // x22
    struct objc_object *v7; // x23
    struct objc_object *v8; // x24
    struct objc_object *v9; // x25
    struct objc_object *v10; // x26
    struct objc_object *v11; // x27
    struct objc_object *v12; // x28
    struct objc_object *v13; // x29
    struct objc_object *v14; // x30
    _int64 curlBlock = inputBlock;
    _int64 receivedData = *(curlBlock + 0x20);
    if (*curlBlock + 0x51) // ImportedVar#0
    {
        if (*curlBlock + 0x48) // ImportedVar#1
        {
            if (*curlBlock + 0x40, "length") == *(curlBlock + 0x28); // ImportedVar#2
            v1 = 4030C_CLASS_MLOnesieChunkLoadTask;
            v1->didReceiveDataError_(curlBlock, receivedData, &receivedData_error_, 0LL);
            if (*curlBlock + 0x30) // ImportedVar#3
            {
                if (*curlBlock + 0x50) // ImportedVar#4
                {
                    v1 = objc_msgSend_3B84298(*curlBlock + 0x30), "firstObject"); // ImportedVar#5
                    v1 = objc_retainAutoreleasedReturnValue_3B8429C(v1);
                    v1 = objc_msgSend_3B84298(v1, "format");
                    v1 = objc_release_3B8429C(v1);
                    objc_release_3B8429C(curlBlock + 0x28), "logLatencyTick:format:", *(curlBlock + 0x30), v9);
                    objc_release_3B8429C(v1);
                }
            }
            if (*curlBlock + 0x51) // ImportedVar#6
            {
                if (objc_msgSend_3B84298(*curlBlock + 0x40), "isBounded") // ImportedVar#7 = HAMRange
                {
                    if (objc_msgSend_3B84298(*curlBlock + 0x40), "length") == *(curlBlock + 0x48) + 0xLL) // curlBlock + 0x48 = ImportedVar#8
                    {
                        newHamError_ = 0LL;
                    }
                    else
                    {
                        newHamError_ = objc_msgSend_3B84298(4030C_CLASS_NSSError, "HAMErrorWithCode:", 10000LL);
                    }
                    curlBlock->curlTaskDidFinishTask_(curlBlock, receivedData, newHamError_);
                }
                v1 = 4030C_CLASS_MLOnesieChunkLoadTask;
                v1->curlTaskDidFinishTask_(curlBlock, receivedData, newHamError_);
                objc_release_3B8429C(newHamError_);
            }
        }
    }
}

```

相对来说，更加明确了代码大概的逻辑：

- 判断接受到的 data 数据，是否为空
 - 如果不为空，继续处理
 - 如果有错误，则log记录延迟? latency的tick
- 如果 curlBlock + 0x51 不为空
 - 处理HAMRange相关的，判断是isBounded 且length长度符合预期
 - 则表示error为空
 - 否则表示有error错误
 - 则继续去生成错误，错误码是 10000
 - 并报错： MLOnesieChunkLoadTask_ didCompleteWithError:

-》如果加上前后文和相关代码逻辑，就会越加的，逐渐的，彻底搞懂：代码逻辑。

如此，iOS逆向中，研究函数调用时，常会用到Block：（匿名的）异步函数调用

加上imported variables，搞清楚参数的传递

和上下文代码的逻辑和类、函数等内容

才能真正的，一点点的，搞懂代码逻辑调用，和具体函数参数等详细的代码逻辑。

crifan.org, 使用署名4.0国际(CC BY 4.0)协议发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 16:27:09

Block心得

TODO:

- 【已解决】 Xcode中lldb调试遇到Block类型变量 `_NSMallocBlock_`
- 【整理】 iOS的ObjC的Block函数: `_Block_object_dispose`
- 【整理】 IDA使用心得: 根据函数列表可以找到iOS的ObjC的Block的回调函数
- 【整理】 iOS逆向心得: Block被调用时的函数参数
- 特殊情况
 - 嵌套调用
 - 【整理】 iOS逆向心得: Block嵌套调用举例
 - Block中调用其他Block
 - 【整理】 iOS逆向心得: Block函数中调用引用变量中的其他Block函数

有些Block只有`invoke`, 没有`copy`和`dispose`

举例:

```
(lldb) po 0x0000000104c68028
<__NSGlobalBlock__: 0x104c68028>
  signature: "v8@?0"
  invoke    : 0x104c65f70 (/private/var/containers/Bu
(lldb) x/4gx 0x0000000104c68028
0x104c68028: 0x00000001e18b7370 0x0000000050000000
0x104c68038: 0x0000000104c65f70 0x0000000104c68008
```

是正常的, 因为此时:

```
BLOCK_HAS_COPY_DISPOSE = 0
```

表示没有 `copy` 和 `dispose`

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新:
2022-10-25 16:52:15

动态调试Block

TODO:

- 【整理】iOS逆向心得：Block的invoke函数的签名signature的含义
 - 把如何动态调试查看函数签名的内容整理过来
- 查看Block详情
 - 【已解决】研究抖音关注逻辑：用Block的调试函数去查看Block详情
- 把自己写的，打印Block各个属性详情的代码，整理过来

举例：

```
(lldb) reg r x0
x0 = 0x000000014726ef70
(lldb) po 0x000000014726ef70
<__NSMallocBlock__: 0x14726ef70>
signature: "v32@?0@NSError"8@16@"TTHttpResponse"24"
invoke : 0x111735758 (<private var containers Bundle Application 1FFDC079 CC8A 4219 955A E01
C73207969 Aweme.app/Frameworks/AwemeCore.framework AwemeCore`+[MKMapView(AWEMap) awe_screenScope
])
copy : 0x108c97674 (<private var containers Bundle Application 1FFDC079 CC8A 4219 955A E01
C73207969 Aweme.app/Frameworks/AwemeCore.framework AwemeCore`+[AWELaunchMainPlaceholder _genera
teBootLoaderLogs])
dispose : 0x108c9767c (<private var containers Bundle Application 1FFDC079 CC8A 4219 955A E01
C73207969 Aweme.app/Frameworks/AwemeCore.framework AwemeCore`+[AWELaunchMainPlaceholder _genera
teBootLoaderLogs])
```

继续查看Block详情：

```
(lldb) po Block_size(0x14726ef70)
0x0000000000000030

(lldb) po _Block_has_signature(0x14726ef70)
0x0000000000000001

(lldb) po _Block_use_stret(0x14726ef70)
nil
(lldb) po _Block_signature(0x14726ef70)
0x0000000107067dd6

(lldb) po _Block_layout(0x14726ef70)
nil
(lldb) po _Block_extended_layout(0x14726ef70)
0x0000000000000100

(lldb) po _Block_tryRetain(0x14726ef70)
0x0000000000000001

(lldb) po _Block_isDeallocating(0x14726ef70)
```

n11

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新:
2022-10-25 16:03:44

相关objc函数

objc_retainBlock

TODO:

- 【已解决】iOS基础知识：objc_retainBlock

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-10-25 16:04:14

附录

下面列出相关参考资料。

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新：
2022-03-17 20:39:28

参考资料

- [Working with Blocks](#)
- [Objective-C Automatic Reference Counting \(ARC\) — Clang 16.0.0git documentation](#)
- [Block 的 Block_descriptor_1 | SeanChense](#)
- [Block 实现中的 flags | SeanChense](#)
- [Block Implementation Specification — Clang 15.0.0git documentation \(llvm.org\)](#)
- [iOS底层原理篇\(十八\) ---- Block底层原理_@Block_Smile的博客-CSDN博客](#)
- [一种查看Block中引用的所有外部对象的实现方法 - K码农 \(kmanong.top\)](#)
- [block那些事——block 内部结构\(1/5\) | 雪峰的blog \(zxfcumtcs.github.io\)](#)
- [block那些事——block copy\(2/5\) | 雪峰的blog \(zxfcumtcs.github.io\)](#)
- [iOS Block 总结 | Edgar's Blog \(tbfungeek.github.io\)](#)
- [【译】《A look inside blocks Episode 3 \(Block_copy\)》 - 掘金](#)
- [A look inside blocks: Episode 3 \(Block_copy\) - Matt Galloway](#)
- [A look inside blocks: Episode 1 - Matt Galloway](#)
- [A look inside blocks: Episode 2 - Matt Galloway](#)
- [mikeash.com: Friday Q&A 2010-01-22: Toll Free Bridging Internals](#)
- [objective c - Is there a way to wrap an ObjectiveC block into function pointer? - OGeek|极客中国-技术改变生活,极客改变未来](#)
- [mikeash.com: Friday Q&A 2010-02-12: Trampolining Blocks with Mutable Code](#)
- [iOS的Block定义的结构图 | ProcessOn免费在线作图,在线流程图,在线思维导图](#)
-

crifan.org, 使用[署名4.0国际\(CC BY 4.0\)协议](#)发布 all right reserved, powered by Gitbook最后更新:
2022-10-25 16:47:24