iOS 编译插桩 & 代码覆盖率检测

检测代码覆盖率可以让我们得知每行代码的执行次数，从而指导测试工作。要想得知每行代码执行与否，就要在代码的入口处插入一些用于统计的代码片段。因此，插桩成为了实现代码覆盖率统计工具的通用解决方案。

编译过程

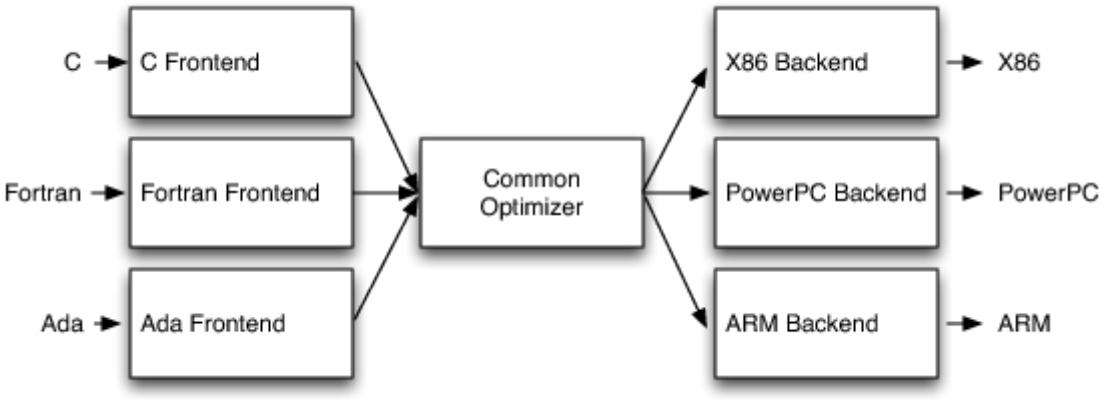
在 iOS 中，编译的过程分为以下几步：

- 预处理（preprocessing）：这步完成头文件引入、宏替换、注释处理、条件编译等操作。
- 词法分析（lexical analysis）：处理源代码，并解析出 token，比如 int, main 等。
- 语法分析（semantic analysis）：将上一步解析出的 token 组织成抽象语法树（AST）。
- 代码生成（CodeGen）：遍历 AST，生成中间代码。中间代码最终会转换为汇编代码。
- 汇编（assembling）：将汇编代码汇编为机器码，生成 object 文件（.o）。
- 链接（linking）：由连接器将多个 .o 和库链接，生成可执行程序（Mach-O）。

LLVM 简介

苹果最开始使用 GCC 作为官方编译器，后来切换为 LLVM。LLVM 最早是 Low Level Virtual Machine 的缩写，但随着项目的繁荣壮大，这个名称已经不能描述 LLVM 的作用了；因此 LLVM 现在只是一个代号，没有所谓的全称。

LLVM 与其他的编译器最为不同的一点就是广为人知的三段式结构，即前端、优化器和后端。前端接收源代码，并产生通用的 LLVM IR 代码。优化器接收 IR 代码，产生优化后的 IR 代码。后段接收 IR 代码，并产生平台相关的机器码。当我们需要支持一个新语言，只需要实现其前端。增加一种优化方案，只需要改动优化器。适配一种新的 CPU 架构，那么就只需要实现其后端就可以了。



每个人都见过的图，但我还是要放

谈到 LLVM，就不得不提到 Pass。每个 Pass 是一个小模块，可由优化器加载并执行。所谓代码优化，就是执行多个 Pass，每个 Pass 实现不同的优化方式。LLVM 也提供了一系列 API，可以让我们很容易地实现自己的 Pass。

编写 LLVM Pass

要编写一个 Pass，必须要先编译 LLVM 工程，得到开发环境。接下来就可以编写一个最简单的 Pass，它打印程序中每个函数的名称，此外什么都不做：

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/Transforms/IPO/PassManagerBuilder.h"
using namespace llvm;
namespace {
    struct MyPass : public FunctionPass { // 继承自 FunctionPass
        static char ID; // LLVM 用于识别 Pass
        MyPass() : FunctionPass(ID) {} // 构造函数
        bool runOnFunction(Function &F) override {
            errs() << "Hello: ";
            errs().write_escaped(F.getName()) << '\n'; // 函数名称
            return false;
        }
    };
}

char MyPass::ID = 0; // 初始化ID，LLVM 会设置它的值
static RegisterPass<MyPass> X("hello", "my pass", false, false);
static RegisterStandardPasses Y(
    PassManagerBuilder::EP_EarlyAsPossible,
    [](const PassManagerBuilder &Builder,
       legacy::PassManagerBase &PM) { PM.add(new MyPass()); }); // 注册Pass
```

这个 Pass 继承自 FunctionPass，这是一个以函数为单位的 Pass，即它对 IR 的作用应该只局限于函数内部，而不应该跨越多个函数。编译时每执行到一个函数，`runOnFunction` 方法就会被调用。

编译这个 Pass，会生成一个动态库 `LLVMMyPass.dylib` 。之后，使用 `load` 参数就能把它加载到优化器上：

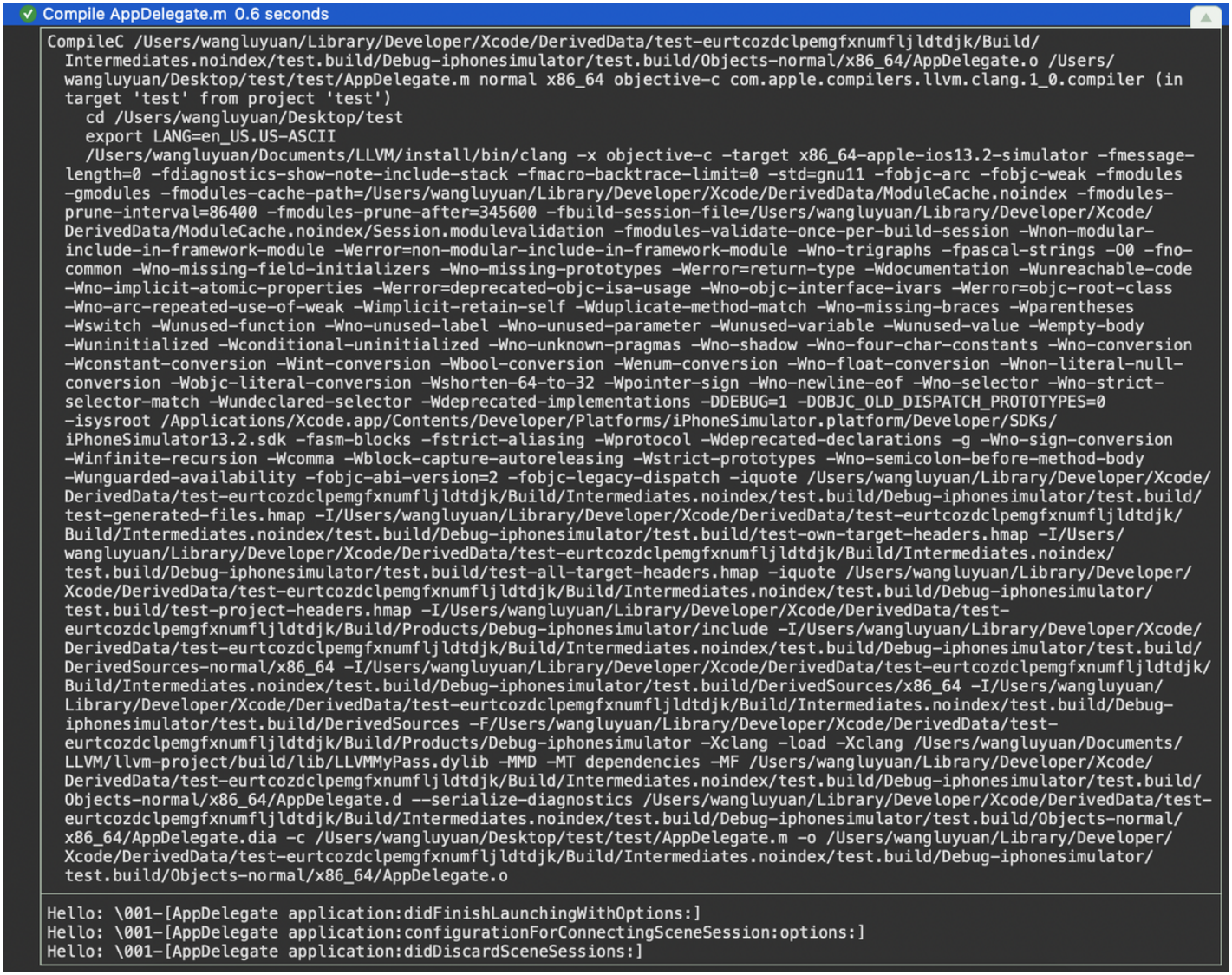
```
opt -load LLVMMyPass.dylib -hello hello.ll > /dev/null
```

控制台就会打印出 hello.ll 中的每个函数名称。

集成到 Xcode

作为 iOS Developer，只在命令行跑起来自定义的 Pass 肯定是不够的，我们需要能把它继承到 Xcode 中。

首先，要在 Build Settings 中设置 CC 和 CXX 参数，使用我们编译出的 clang 来编译程序；然后设置 Other C Flags 和 Other C++ Flags 来 load 我们自定义的 Pass。之后编译程序，就可以看到这个 Pass 的输出了：

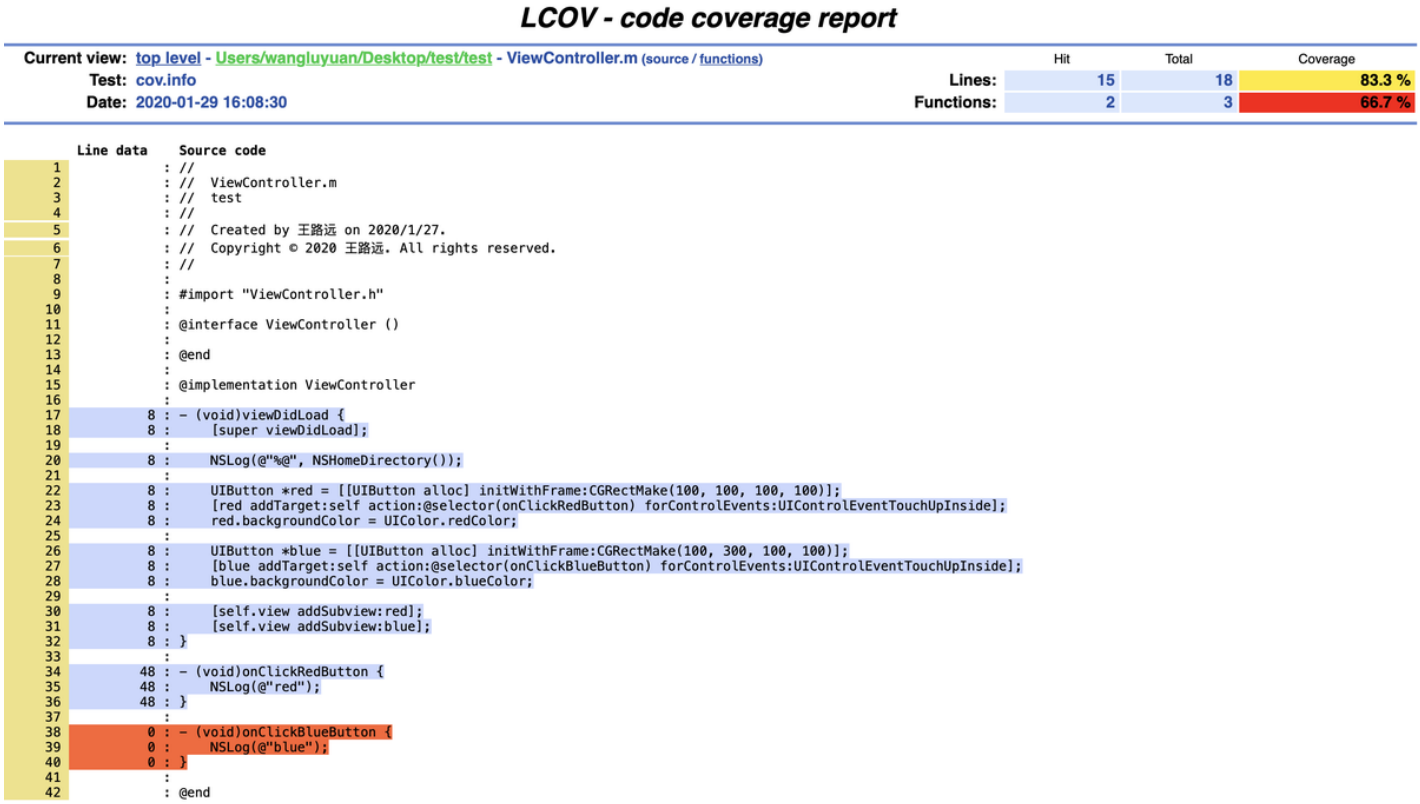


iOS 代码覆盖率检测

Xcode 中本身就集成了代码覆盖率工具 gcov。只需要在 Build Settings 中打开以下两个开关就可以使用：

```
Instrument Program Flow = Yes
Generate Legacy Test Coverage Files = Yes
```

编译时，LLVM 会运行一个 GCOVPass，生成记录代码映射关系的 notes 文件 .gcno，并进行插桩。程序运行时，会产生记录代码执行情况的 data 文件 .gcda。拿到这两类文件，就可以解析出代码覆盖率了。通过解析工具 lcov，我们甚至可以生成出非常直观的 html 报告：



LCOV - code coverage report

Current view: top level - Users/wangluyuan/Desktop/test/test - ViewController.m (source / functions)

Test: cov.info

Date: 2020-01-29 16:08:30

	Hit	Total	Coverage
Lines:	15	18	83.3 %
Functions:	2	3	66.7 %

```
      Line data    Source code
    1           : //
    2           : //  ViewController.m
    3           : //  test
    4           : //
    5           : //  Created by 王路远 on 2020/1/27.
    6           : //  Copyright © 2020 王路远. All rights reserved.
    7           : //
    8           :
    9           : #import "ViewController.h"
   10           :
   11           : @interface ViewController ()
   12           :
   13           : @end
   14           :
   15           : @implementation ViewController
   16           :
   17         8 : - (void)viewDidLoad {
   18         8 :     [super viewDidLoad];
   19           :
   20         8 :     NSLog(@"%@", NSHomeDirectory());
   21           :
   22         8 :     UIButton *red = [[UIButton alloc] initWithFrame:CGRectMake(100, 100, 100, 100)];
   23         8 :     [red addTarget:self action:@selector(onClickRedButton) forControlEvents:UIControlEventTouchUpInside];
   24         8 :     red.backgroundColor = UIColor.redColor;
   25           :
   26         8 :     UIButton *blue = [[UIButton alloc] initWithFrame:CGRectMake(100, 300, 100, 100)];
   27         8 :     [blue addTarget:self action:@selector(onClickBlueButton) forControlEvents:UIControlEventTouchUpInside];
   28         8 :     blue.backgroundColor = UIColor.blueColor;
   29           :
   30         8 :     [self.view addSubview:red];
   31         8 :     [self.view addSubview:blue];
   32         8 : }
   33           :
   34        48 : - (void)onClickRedButton {
   35        48 :     NSLog(@"red");
   36        48 : }
   37           :
   38         0 : - (void)onClickBlueButton {
   39         0 :     NSLog(@"blue");
   40         0 : }
   41           :
   42           : @end
```

在我这个示例程序中，有红色和蓝色两个按钮。在运行时，我多次点击了红色按钮，而从未点击蓝色按钮。从报告中我们可以非常清晰地看到这一点。

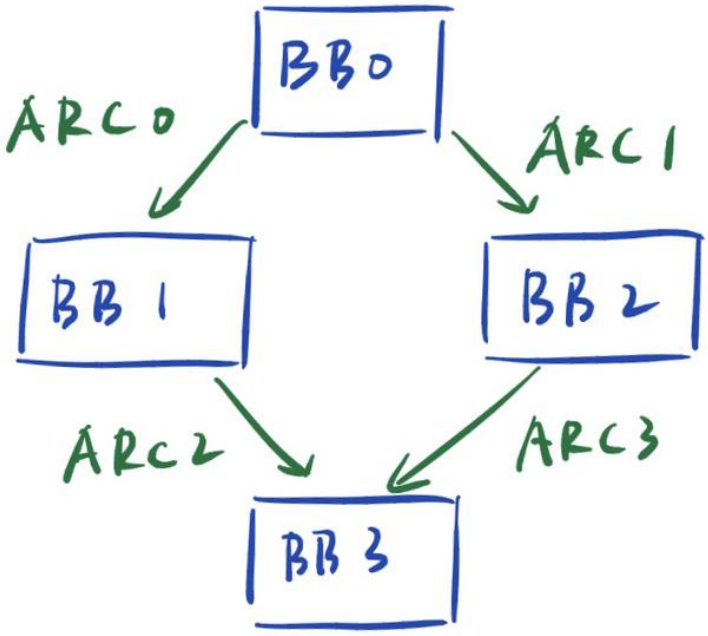
覆盖率检测原理

一行代码执行与否，需要在运行时才知道。这需要我们对源码进行改动，通过插桩来加入一些计数代码。

Basic Block（BB）

我们固然可以在每行代码前面都增加计数语句，但这并无必要，因为在顺序执行的结构中，第一行代码执行后，所有的代码就都会执行。BB 正是代码执行的基本单元。BB 的定义是，只有一个入口和一个出口，中间再无其他的 jump / branch 等语句。遇到一个有条件的跳转语句时，就会产生一个

ARC，这样一个 BB 就会有两个可能的终点。如果把 BB 看作结点，ARC 看作边，我们就会得到一个有向图，称为一个 Basic Block Graph。



如果我们知道每个 BB 的执行情况，就可以得出整个程序的代码覆盖率。
在以下这个简单的程序中：

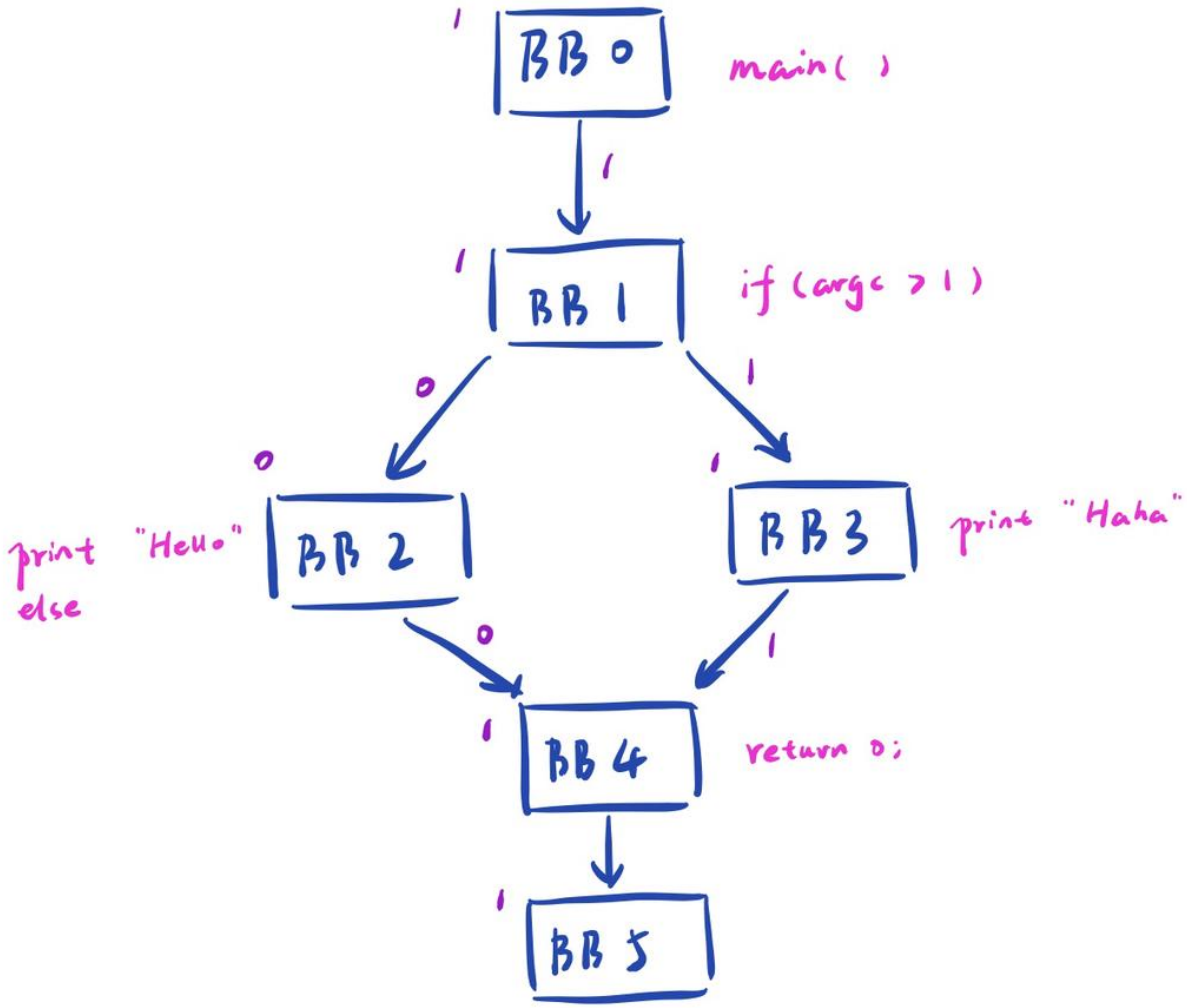
```c
#include <stdio.h>

int main(int argc, char **argv){
    if (argc > 1) {
        printf("Hello, how are you doing?\n");
    } else {
        printf("Haha, I'm doing great!\n");
    }
    return 0;
}
```

我们通过 gcov 程序可以把 .gcda 文件解析出来（gcda 和 gcno 都是二进制的，本身不可读）：

```
===== main (0) @ hello.c:3
Block : 0 Counter : 1
        Destination Edges : 1 (1),
        Lines : 3,
Block : 1 Counter : 1
        Source Edges : 0 (1),
```

```
        Destination Edges : 2 (0), 3 (1),
        Lines : 4,
Block : 2 Counter : 0
        Source Edges : 1 (0),
        Destination Edges : 4 (0),
        Lines : 5,6,
Block : 3 Counter : 1
        Source Edges : 1 (1),
        Destination Edges : 4 (1),
        Lines : 7,
Block : 4 Counter : 1
        Source Edges : 2 (0), 3 (1),
        Destination Edges : 5 (1),
        Lines : 9,
Block : 5 Counter : 1
        Source Edges : 4 (1),
File 'hello.c'
Lines executed:66.67% of 6
hello.c:creating 'hello.c.gcov'
```

根据其中的信息，我们可以画出这个 BB Graph，其中包含了代码的执行情况以及对应源码的行号：



插桩逻辑

在 GCOVPass 中，创建 .gcno 文件的逻辑如下：

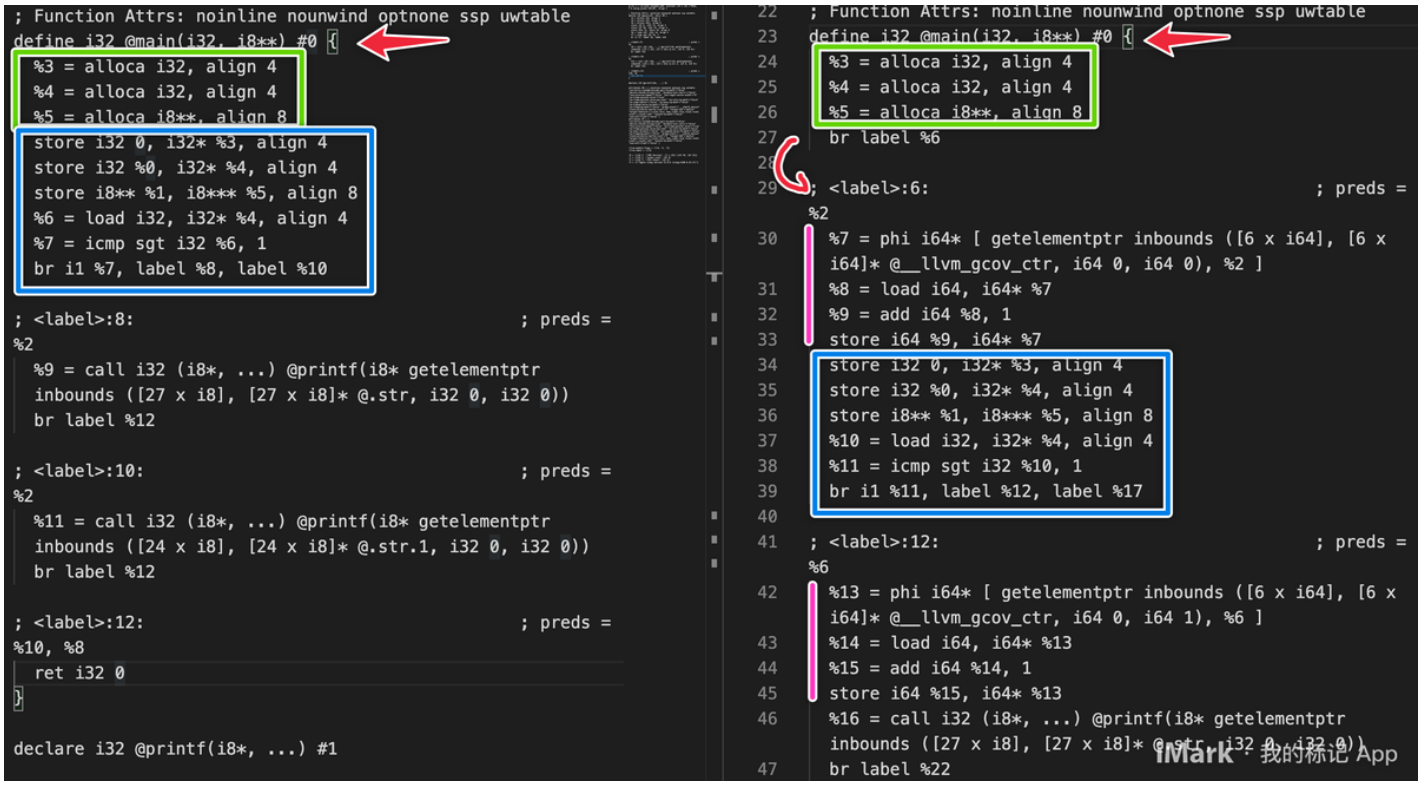
- 创建文件，写入 Magic Number
- 遍历所有函数，写入文件地址、函数名，以及函数起止位置对应的行号
- 遍历函数中的 BB，写入 BB 编号、BB 起止范围、 BB 后继结点编号，以及 BB 对应的行号信息

而生成 .gcda 文件的逻辑如下：

- 创建文件，写入 Magic Number
- 运行时写入函数信息
- 运行时写入 BB 执行信息，如果已存在 .gcda 文件，则写入合并后的执行信息

桩代码

我们可以在插桩前后分别生成 IR 代码，来查看插桩前后的区别：



图中，左边为原始代码，右边为插桩后。通过对比可以看出粉色标记的代码是插入的桩代码。可见，用于计数的代码是插入在每个 BB 最前面的。

更好的覆盖率工具

业界的解决方案

业界已有一些公司对覆盖率检测做了积极的探索，解决了 Xcode 集成的覆盖率工具操作繁琐、只能显示全量报告、难以和开发流程相结合的问题。在他们的解决方案中，已经可以做到：

- 基于 commit 给出报告 - 增量代码覆盖率检测
- 编写、调试、push 后自动产生报告 - 对开发者透明
- QA 测试后，覆盖率报告可自动回收 - 服务端可收集所有测试设备的覆盖率

引入到火山众测?

总的来说，是在 Xcode 集成的覆盖率工具上做了一些拓展。但是如果我们不仅想在公司内部分发，还想向众测用户分发时（可以作为众测任务完成度的量化体现），**可能**会存在这样的问题：根据 Technical Q&A QA1964 提到的内容，如果启用了 GCC-style coverage instrumentation 或 LLVM profiling instrumentation ，那么提交 App 的时候会被拒绝。
如果想通过 TestFlight 分发应用，可能需要我们自己实现插桩逻辑。

尝试插桩

如果要自己实现一个覆盖率检测工具，核心就是能成功的对代码进行插桩。这里我做了一个简单的尝试 - 插入打印出当前函数名称的代码：

```
extern void _mark_executed_func(char *funcName) {
    NSString *string = [NSString stringWithUTF8String:funcName];
    NSLog(@"%@", string);
}
```

要插入的代码很简单，它接收一个函数名称的字符串，并把它打印出来。我们可以想像把打印替换成写文件，那么一个函数级别的覆盖率检测小玩具就完成一半了 😊
Pass 中核心代码如下（删除了一些边边角角的代码）：

```
#include ....
using namespace llvm;
namespace {
    struct FuncCoverage : public FunctionPass {
        static char ID;
        FuncCoverage() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            if (F.getName().startswith("_mark_executed_func")) {
                return false; // 不能再给桩函数插桩了
            }
            LLVMContext &context = F.getParent()->getContext(); //拿到当前Module的Context
            BasicBlock &bb = F.getEntryBlock();
            Instruction *beginInstruction = dyn_cast<Instruction>(bb.begin());
            FunctionType *type = FunctionType::get(Type::getVoidTy(context), {Type::getInt8PtrTy(context)}, false);
            Constant *beginFun = F.getParent()->getFunction("_mark_executed_func");

            IRBuilder<> Builder(&bb);
            CallInst *inst = CallInst::Create(fun, {Builder.CreateGlobalStringPtr(F.getName())}); //构造 call instruction
            inst->insertBefore(beginInstruction);
            return true;
        }
    };
```

```
}
```

编译、运行我们上面提到的有红蓝两个按钮的程序。这两个按钮的回调函数分别为：

```
//...
- (void)onClickRedButton {
    NSLog(@"red");
}
- (void)onClickBlueButton {
    NSLog(@"blue");
}
```

这两个函数中都是没有调用打印函数名称的函数的。当我们点击按钮时，可以看到控制台有如下输出：

```
2020-02-02 20:59:31.747426+0800 test[46749:2365005]
    \^A-[ViewController viewDidLoad]
2020-02-02 20:59:31.747609+0800 test[46749:2365005]
    /Users/wangluyuan/Library/Developer/CoreSimulator/Devices/E49
    1A587-0993-4C83-A2DC-6B7B3D00EE99/data/Containers/Data/Applic
    ation/7DB30CA4-D75C-4D1C-AB64-5CF0C54C11B6
2020-02-02 20:59:31.747732+0800 test[46749:2365005] CGRectMake
2020-02-02 20:59:31.748074+0800 test[46749:2365005] CGRectMake
2020-02-02 20:59:34.712812+0800 test[46749:2365005]
    \^A-[ViewController onClickRedButton]
2020-02-02 20:59:34.713173+0800 test[46749:2365005] red
2020-02-02 20:59:39.062576+0800 test[46749:2365005]
    \^A-[ViewController onClickBlueButton]
2020-02-02 20:59:39.062878+0800 test[46749:2365005] blue
2020-02-02 20:59:43.026294+0800 test[46749:2365005]
    \^A-[ViewController onClickRedButton]
2020-02-02 20:59:43.026595+0800 test[46749:2365005] red
```

由此证明插桩成功了。

结束语

由于我没有学过编译原理，对 LLVM 也并不熟悉，实现这些 demo 的时候还是遇上了一些困难。LLVM 的官方文档很少有符合我需求的，网络上也难以找到系统的教程。直到最后的 demo 中，也存在一些严重的缺陷（但并非无法解决）。欢迎有兴趣或熟悉编译器的同学与我详细交流👋