LLVM

LLVM概述

LLVM是构架编译器(compiler)的框架系统，以C++编写而成，用于优化以任意程序语言编写的程序的编译时间(compile-time)、链接时间(link-time)、运行时间(run-time)以及空闲时间(idle-time)，对开发者保持开放，并兼容已有脚本。
LLVM计划启动于2000年，最初由美国UIUC大学的Chris Lattner博士主持开展。2006年Chris Lattner加盟Apple Inc.并致力于LLVM在Apple开发体系中的应用。Apple也是LLVM计划的主要资助者。
目前LLVM已经被苹果IOS开发工具、Xilinx Vivado、Facebook、Google等各大公司采用。

传统编译器设计

编译器前端(Frontend)

编译器前端的任务是**解析源代码**。它会进行：词法分析，语法分析，语义分析，检查源代码是否存在错误，然后构建**抽象语法树**（Abstract Syntax Tree，AST）,LLVM的前端还会生成**中间代码**(intermediate representation ，**IR**)。

优化器（Optimizer）

优化器负责进行各种优化。改善代码的运行时间，例如消除冗余计算等。

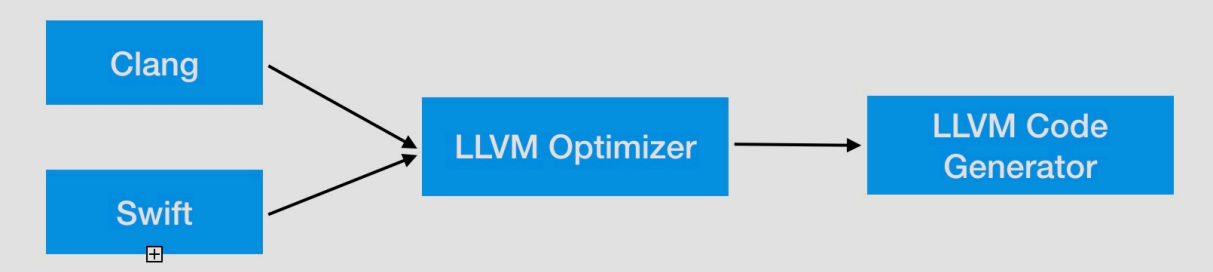
后端（Backend）/代码生成器（CodeGenerator）

将代码映射到目标指令集。生成机器语言，并且进行机器相关的代码优化。

iOS的编译器架构

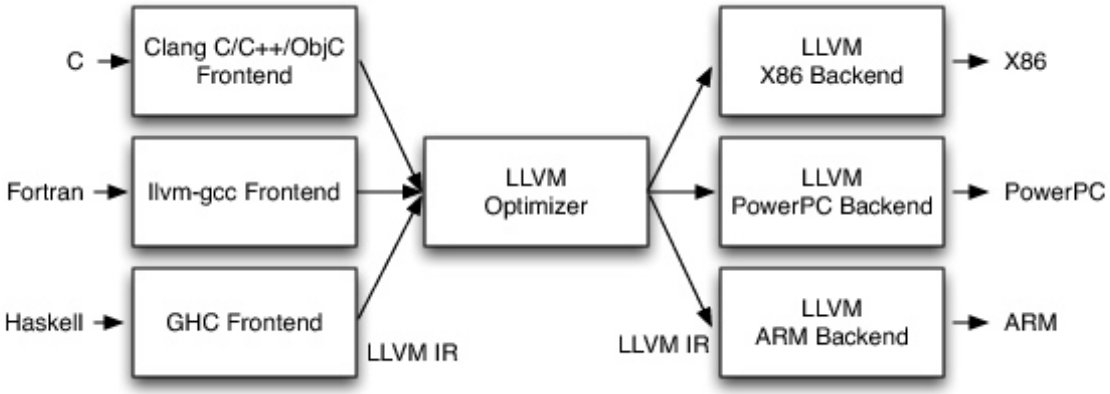
Objective C/C/C++使用的编译器前端是**Clang**，Swift是**Swift**，后端都是

LLVM。



LLVM的设计

当编译器决定支持多种源语言或多种硬件架构时，LLVM最重要的地方就来了。其他的编译器如GCC，它方法非常成功，但由于它是作为整体应用程序设计的，因此它们的用途受到了很大的限制。
LLVM设计的最重要方面是，使用通用的代码表示形式（IR），它是用来在编译器中表示代码的形式。所以LLVM可以为任何编程语言独立编写前端，并且可以为任意硬件架构独立编写后端。



Clang

Clang是LLVM项目中的一个子项目。它是基于LLVM架构的轻量级编译器，诞生之初是为了替代GCC，提供更快的编译速度。它是负责编译C、C++、Objecte-C语言的编译器，它属于整个LLVM架构中的，编译器**前端**。对于开发者来说，研究Clang可以给我们带来很多好处。

编译流程

通过命令可以打印源码的编译阶段

```
clang -ccc-print-phases main.m
```

```
```

0: input, "main.m", objective-c
1: preprocessor, {0}, objective-c-cpp-output
2: compiler, {1}, ir
3: backend, {2}, assembler
4: assembler, {3}, object
5: linker, {4}, image
6: bind-arch, "x86_64", {5}, image

0：输入文件：找到源文件。
1：预处理阶段：这个过程处理包括宏的替换，头文件的导入。
2：编译阶段：进行词法分析、语法分析、检测语法是否正确，最终生成IR。
3：后端：这里 LLVM 会通过一个一个的Pass去优化，每个Pass做一些事情，最终生成汇编代码。
4：生成目标文件。
5：链接：链接需要的动态库和静态库，生成可执行文件。
6：通过不同的架构，生成对应的可执行文件。

预处理阶段

执行如下命令

```
clang -E main.m
```

执行完毕可以看到头文件的导入和宏的替换。

编译阶段

词法分析

预处理完成后就会进行**词法分析**.这里会把代码切成一个个 **Token**，比如大小括号，等于号还有字符串等。

```
clang -fmodules -fsyntax-only -Xclang -dump-tokens main.m
```

```
annot_module_include '#import <stdio.h>
#define C 30

typedef int HK_INT_64;

int main(int argc, const char * argv[]) {
    @autoreleasepool {
   '            Loc=<main.m:9:1>
typedef 'typedef'        [StartOfLine]  Loc=<main.m:12:1>
int 'int'        [LeadingSpace] Loc=<main.m:12:9>
identifier 'HK_INT_64'   [LeadingSpace] Loc=<main.m:12:13>
semi ';'                 Loc=<main.m:12:22>
int 'int'        [StartOfLine]  Loc=<main.m:14:1>
identifier 'main'        [LeadingSpace] Loc=<main.m:14:5>
l_paren '('              Loc=<main.m:14:9>
int 'int'                Loc=<main.m:14:10>
```

语法分析

词法分析完成之后就是**语法分析**，它的任务是验证语法是否正确。在词法分析的基础上将单词序列组合成各类语法短语，如“程序”，“语句”，“表达式”等等，然后将所有节点组成抽象**语法树**（Abstract Syntax Tree，AST） 。语法分析程序判断源程序在结构上是否正确。

```
clang -fmodules -fsyntax-only -Xclang -ast-dump main.m
```

```
|-FunctionDecl 0x7f957018e6e8 <line:14:1, line:22:1> line:14:5 main 'int (int, const char **)'
| |-ParmVarDecl 0x7f9570064478 <col:10, col:14> col:14 argc 'int'
| |-ParmVarDecl 0x7f9570064590 <col:20, col:38> col:33 argv 'const char **':'const char **'
| `-CompoundStmt 0x7f957018f048 <col:41, line:22:1>
|   |-ObjCAutoreleasePoolStmt 0x7f957018f000 <line:15:5, line:20:5>
|   | `-CompoundStmt 0x7f957018efd8 <line:15:22, line:20:5>
|   |   |-DeclStmt 0x7f957018e8d0 <line:16:9, col:25>
|   |   | `-VarDecl 0x7f957018e850 <col:9, col:23> col:19 used a 'HK_INT_64':'int' cinit
|   |   |   `-IntegerLiteral 0x7f957018e8b0 <col:23> 'int' 10
|   |   |-DeclStmt 0x7f957018ed58 <line:17:9, col:25>
|   |   | `-VarDecl 0x7f957018e8f8 <col:9, col:23> col:19 used b 'HK_INT_64':'int' cinit
|   |   |   `-IntegerLiteral 0x7f957018e958 <col:23> 'int' 20
|   |   `-CallExpr 0x7f957018ef70 <line:18:9, col:26> 'int'
|   |     |-ImplicitCastExpr 0x7f957018ef58 <col:9> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
```

如果导入头文件找不到，那么可以指定SDK

```
clang -isysroot /Applications/Xcode.app/Contents/Developer/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulator12.2.sdk（自己SDK路径） -fmodules -fsyntax-only -Xclang -ast-dump main.m
```

生成中间代码IR(intermediate representation)

完成以上步骤后就开始生成中间代码IR了，代码生成器（Code Generation）会将语法树自顶向下遍历逐步翻译成 LLVM IR。通过下面命令可以生成.ll的文本文件，查看IR代码。

```
clang -S -fobjc-arc -emit-llvm main.m
```

Objective C代码在这一步会进行runtime的桥接：property合成，ARC处理等

IR的基本语法

> @ 全局标识
> % 局部标识
> alloca 开辟空间
> align 内存对齐
> i32 32个bit，4个字节
> store 写入内存
> load 读取数据
> call 调用函数
> ret 返回

IR的优化

LLVM 的优化级别分别是 -O0 -O1 -O2 -O3 -Os(第一个是大写英文字母O)

```
clang -Os -S -fobjc-arc -emit-llvm main.m -o main.ll
```

bitCode

xcode7 以后开启bitcode苹果会做进一步的优化。生成.bc的中间代码。
我们通过优化后的IR代码生成.bc代码

```
clang -emit-llvm -c main.ll -o main.bc
```

生成汇编代码

我们通过最终的.bc或者.ll 代码生成汇编代码

```
clang -S -fobjc-arc main.bc -o main.s
clang -S -fobjc-arc main.ll -o main.s
```

生成汇编代码也可以进行优化

```
clang -Os -S -fobjc-arc main.m -o main.s
```

生成目标文件(汇编器)

目标文件的生成，是汇编器以汇编代码作为输入，将汇编代码转换为机器代码，最后输出目标文件(object file)。

```
clang -fmodules -c main.s -o main.o
```

通过nm命令，查看下main.o中的符号

```
$xcrun nm -nm main.o
                 (undefined) external _printf
0000000000000000 (__TEXT,__text) external _test
000000000000000a (__TEXT,__text) external _main
```

_printf 是一个是**undefined external**的。
undefined表示在当前文件暂时找不到符号_printf
external表示这个符号是外部可以访问的。

生成可执行文件（链接）

连接器把编译产生的.o文件和（.dylib .a）文件，生成一个mach-o文件。

```
clang main.o -o main
```

查看链接之后的符号

```
$xcrun nm -nm main
                 (undefined) external _printf (from libSystem)
                 (undefined) external dyld_stub_binder (from libSyste
m)
0000000100000000 (__TEXT,__text) [referenced dynamically] external __
```

```
mh_execute_header
0000000100000f6d (__TEXT,__text) external _test
0000000100000f77 (__TEXT,__text) external _main
```

Clang插件

LLVM下载

> 由于国内的网络限制，我们需要借助镜像下载LLVM的源码
> https://mirror.tuna.tsinghua.edu.cn/help/llvm/

- 下载llvm项目

```
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/llvm.git
```

- 在LLVM 的 tools 目录下下载 Clang

```
cd llvm/tools
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/clang.git
```

- 在 LLVM 的 projects 目录下下载 compiler-rt，libcxx，libcxxabi

```
cd ../projects
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/compiler-rt.g
it
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/libcxx.git
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/libcxxabi.git
```

- 在 Clang 的 tools 下安装 extra 工具

```
cd ../tools/clang/tools
git clone https://mirrors.tuna.tsinghua.edu.cn/git/llvm/clang-tools-e
xtra.git
```

LLVM 编译

> 由于最新的LLVM只支持cmake来编译了，我们还需要安装cmake。

安装 cmake

- 查看brew 是否安装cmake如果有就跳过下面步骤

```
brew list
```

- 通过brew 安装cmake

```
brew install cmake
```

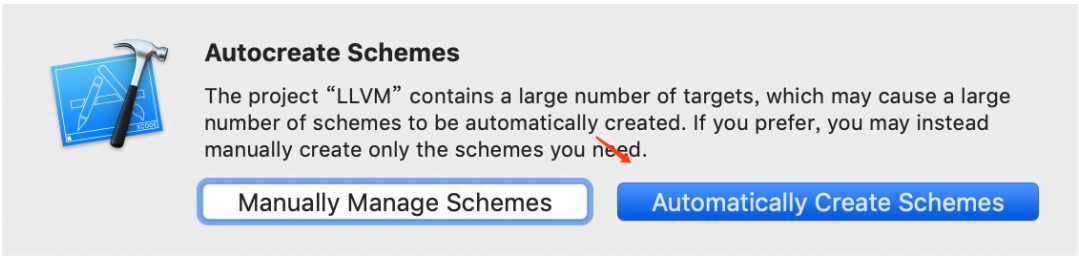
编译 LLVM

通过 xcode 编译 LLVM

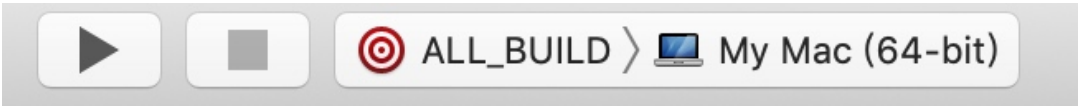
- cmake编译成Xcode项目

```
mkdir build_xcode
cd build_xcode
cmake -G Xcode ../llvm
```

- 使用Xcode编译Clang
  - 选择自动创建Schemes



  - 编译，选择ALL_BUILD Secheme 进行编译，预计1+小时。



    编译的时间会比较长。

通过 ninja 编译 LLVM

- 使用ninja进行编译则还需要安装ninja。 使用`$ brew install ninja`命令即可安装ninja。
- 在llvm源码根目录下新建一个build_ninja目录，最终会在build_ninja目录下生

成build.ninja。

- 在llvm源码根目录下新建一个llvm_release目录，最终编译文件会在llvm_release文件夹路径下。

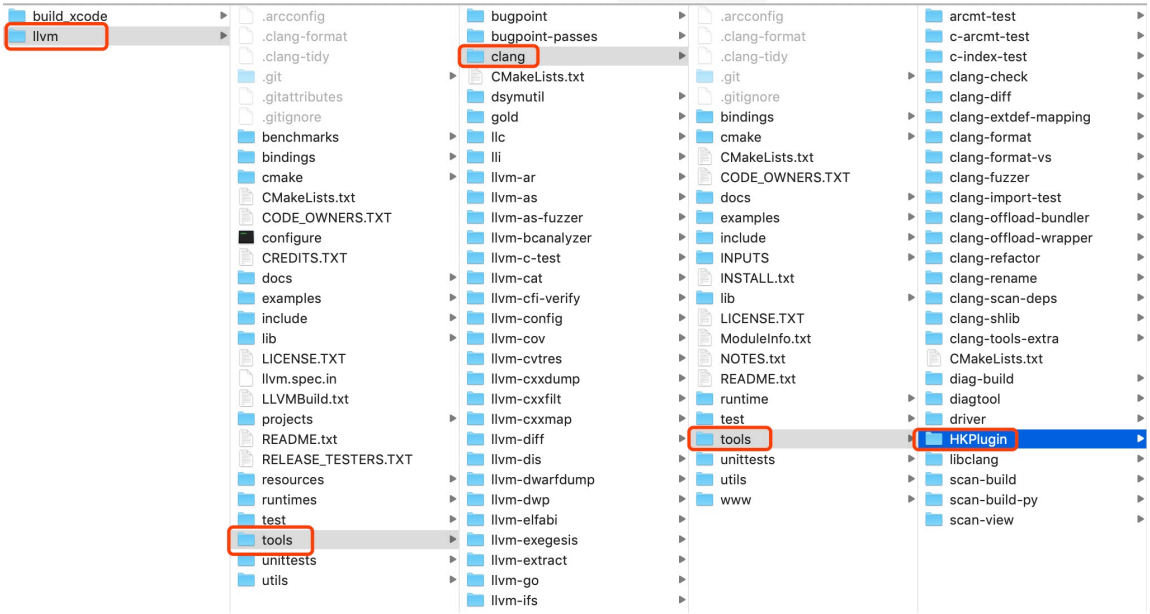
```
$ cd llvm_build
$ cmake -G Ninja ../llvm -DCMAKE_INSTALL_PREFIX= 安装路径（本机为/Users/xxx/xxx/LLVM/llvm_release，注意DCMAKE_INSTALL_PREFIX后面不能有空格。
```

- 依次执行编译、安装指令。

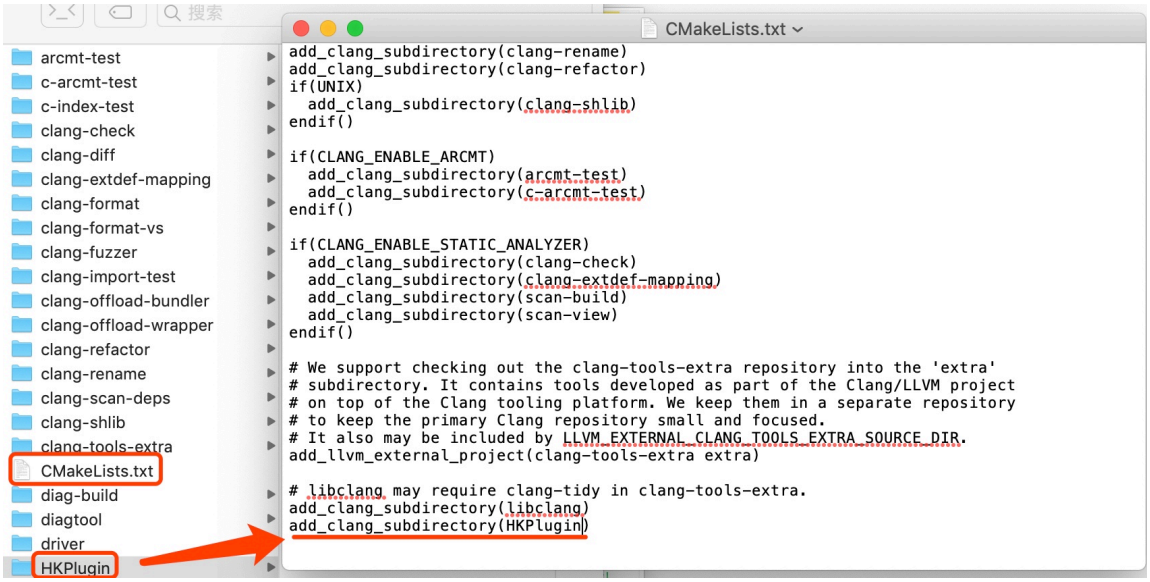
```
$ ninja
$ ninja install
```

创建插件

- 在/llvm/tools/clang/tools目录下新建插件`HKPlugin`

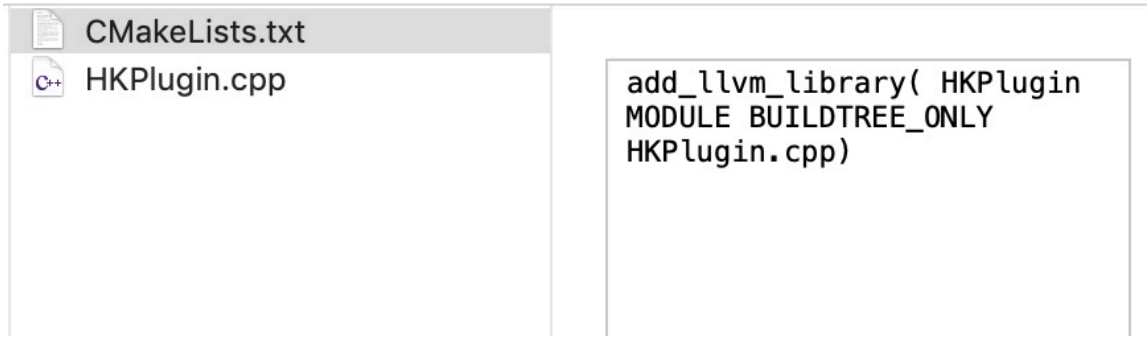


- 修改/llvm/tools/clang/tools目录下的CMakeLists.txt文件，新增add_clang_subdirectory(HKPlugin)。

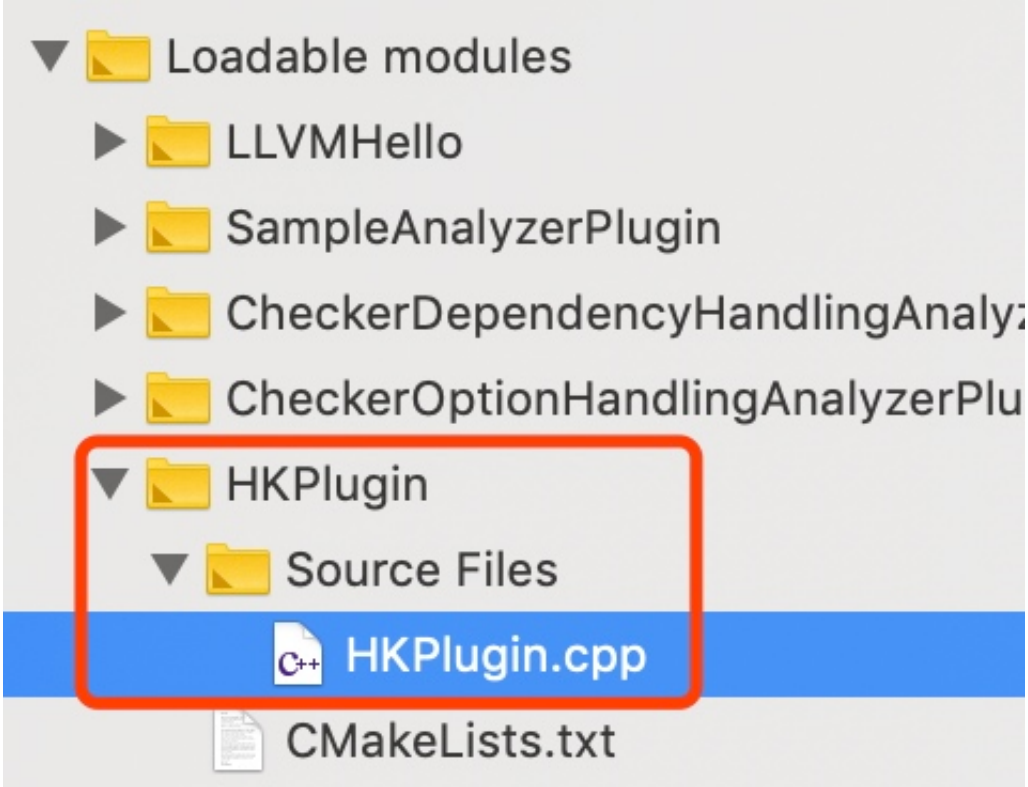


- 在HKPlugin目录下新建一个名为HKPlugi.cpp的文件和CMakeLists.txt的文件。在CMakeLists.txt中写上

```
add_llvm_library( HKPlugin MODULE BUILDTREE_ONLY
  HKPlugin.cpp
)
```



- 接下来利用cmake重新生成一下Xcode项目，在build_xcode中`cmake -G Xcode ../llvm`
- 最后可以在LLVM的Xcode项目中可以看到 Loadable modules 目录下有自己的Plugin目录了。我们可以在里面编写插件代码。



编写插件代码

```
#include <iostream>
#include "clang/AST/AST.h"
#include "clang/AST/DeclObjC.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/ASTMatchers/ASTMatchers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/ASTMatchers/ASTMatchFinder.h"
#include "clang/Frontend/FrontendPluginRegistry.h"

using namespace clang;
using namespace std;
using namespace llvm;
using namespace clang::ast_matchers;

namespace HKPlugin {

    class HKMatchHandler: public MatchFinder::MatchCallback {
    private:
        CompilerInstance &CI;

        bool isUserSourceCode(const string filename) {
            if (filename.empty()) return false;
            if (filename.find("/Applications/Xcode.app/") == 0) retur
n false;
```

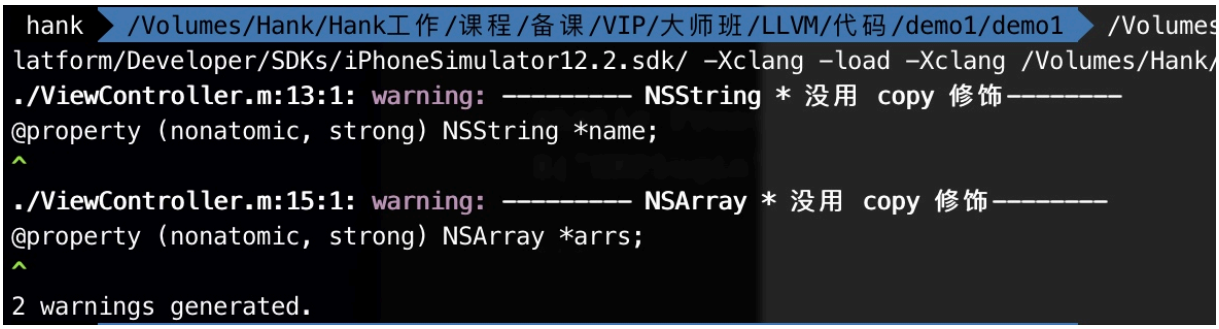
```
            return true;
        }

        bool isShouldUseCopy(const string typeStr) {
            if (typeStr.find("NSString") != string::npos ||
                typeStr.find("NSArray") != string::npos ||
                typeStr.find("NSDictionary") != string::npos/*...*/)
{
                return true;
            }
            return false;
        }
    public:
        HKMatchHandler(CompilerInstance &CI) :CI(CI) {}

        void run(const MatchFinder::MatchResult &Result) {
            const ObjCPropertyDecl *propertyDecl = Result.Nodes.getNo
deAs<ObjCPropertyDecl>("objcPropertyDecl");
            if (propertyDecl && isUserSourceCode(CI.getSourceManager(
).getFilename(propertyDecl->getSourceRange().getBegin()).str()) ) {
                ObjCPropertyDecl::PropertyAttributeKind attrKind = pr
opertyDecl->getPropertyAttributes();
                string typeStr = propertyDecl->getType().getAsString(
);

                if (propertyDecl->getTypeSourceInfo() && isShouldUseC
opy(typeStr) && !(attrKind & ObjCPropertyDecl::OBJC_PR_copy)) {
                    DiagnosticsEngine &diag = CI.getDiagnostics();
                    diag.Report(propertyDecl->getBeginLoc(), diag.get
CustomDiagID(DiagnosticsEngine::Warning, "--------- %0 没用 copy 修饰-
-------")) << typeStr;
                }
            }
        }
    };

    class HKASTConsumer: public ASTConsumer {
    private:
        MatchFinder matcher;
        HKMatchHandler handler;
    public:
        HKASTConsumer(CompilerInstance &CI) :handler(CI) {
            matcher.addMatcher(objcPropertyDecl().bind("objcPropertyD
ecl"), &handler);
        }

        void HandleTranslationUnit(ASTContext &context) {
```

```
            matcher.matchAST(context);
        }
    };

    class HKASTAction: public PluginASTAction {
    public:
        unique_ptr<ASTConsumer> CreateASTConsumer(CompilerInstance &C
I, StringRef iFile) {
            return unique_ptr<HKASTConsumer> (new HKASTConsumer(CI));
        }

        bool ParseArgs(const CompilerInstance &ci, const std::vector<
std::string> &args) {
            return true;
        }
    };
}

static FrontendPluginRegistry::Add<HKPlugin::HKASTAction> X("HKPlugin
", "This is the description of the plugin");
```

测试插件

```
自己编译的clang文件路径 -isysroot /Applications/Xcode.app/Contents/Deve
loper/Platforms/iPhoneSimulator.platform/Developer/SDKs/iPhoneSimulat
or12.2.sdk/ -Xclang -load -Xclang 插件(.dylib)路径 -Xclang -add-plugin
 -Xclang 插件名 -c 源码路径
```

测试结果：

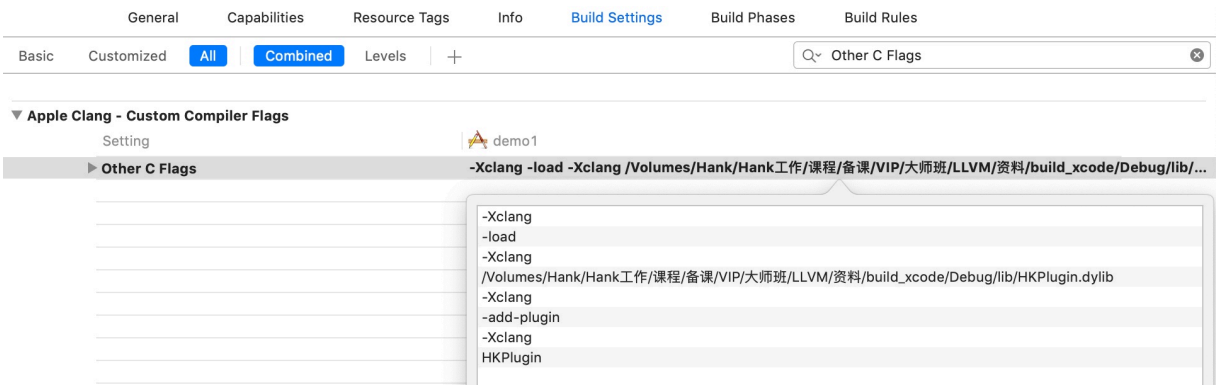


Xcode 集成插件

加载插件

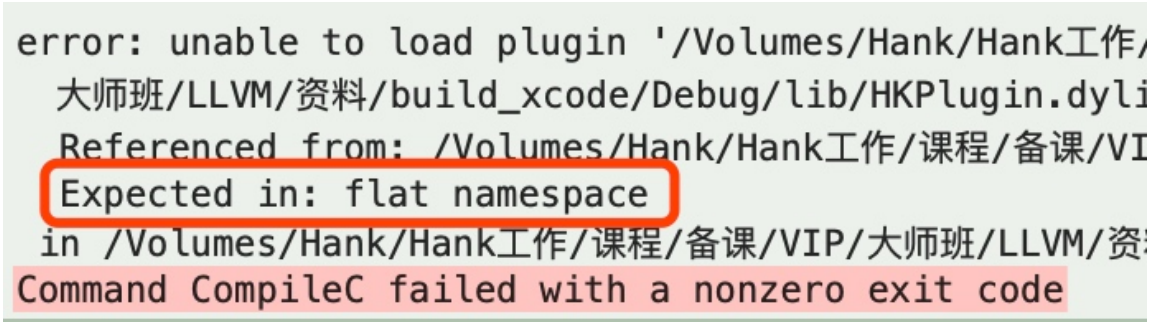
打开测试项目，在Build Settings -> Other C Flags 添加上如下内容：

```
-Xclang -load -Xclang (.dylib)动态库路径 -Xclang -add-plugin -Xclang HKPlugin
```

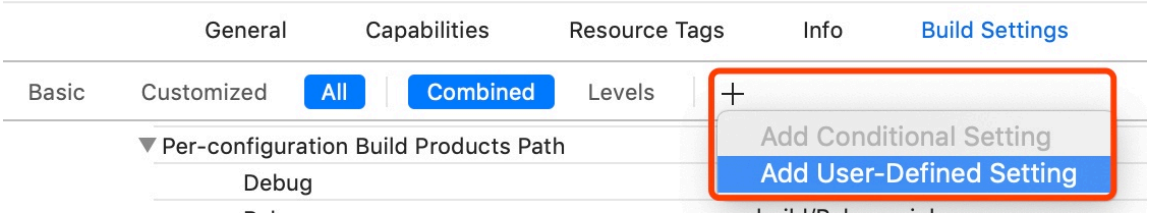


设置编译器

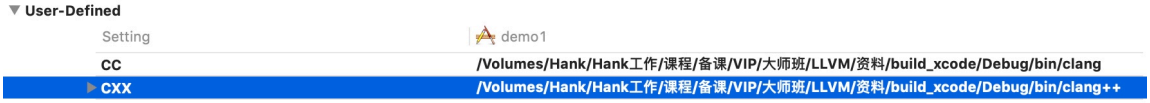
- 由于Clang插件需要使用对应的版本去加载，如果版本不一致则会导致**编译错误**，会出现如下图所示



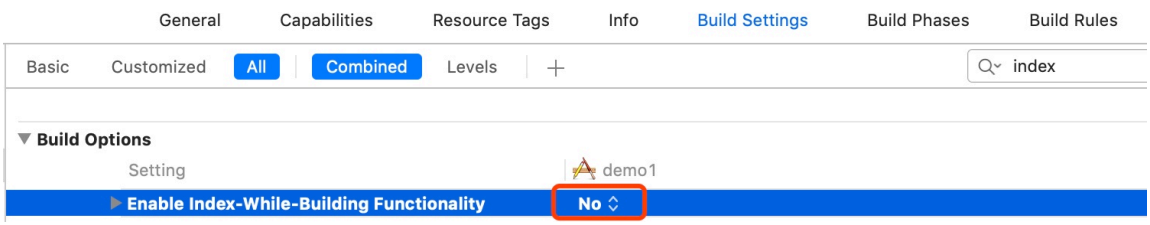
- 在Build Settings栏目中新增两项用户定义的设置



- 分别是CC 和 CXX
  CC对应的是自己编译的clang的绝对路径
  CXX对应的是自己编译的clang++的绝对路径。



- 接下来在Build Settings栏目中搜索index，将Enable Index-Wihle-Building Functionality的Default改为NO。


General
Capabilities
Resource Tags
Info
Build Settings
Build Phases
Build Rules
Basic
Customized
All
Combined
Levels
index
Build Options
Setting
demo1
Enable Index-While-Building Functionality
No