iOS 微信编译速度优化分享

前言

岁月真是个养猪场，这几年，人胖了，微信代码也翻了。记得 14 年转岗来微信时，用自己笔记本编译微信工程才十来分钟。如今用公司配的 17 年款 27-inch iMac 编译要接近半小时；偶然间更新完代码，又莫名其妙需要全新编译。在这么低的编译效率下，开发心情受到严重影响。于是年初我向上头请示，优化微信编译效率，上头也同意了。

现有方案

在动手之前，先搜索目前已有方案，大概有这几个优化点：

一、优化工程配置

1、将 Debug Information Format 改为 DWARF

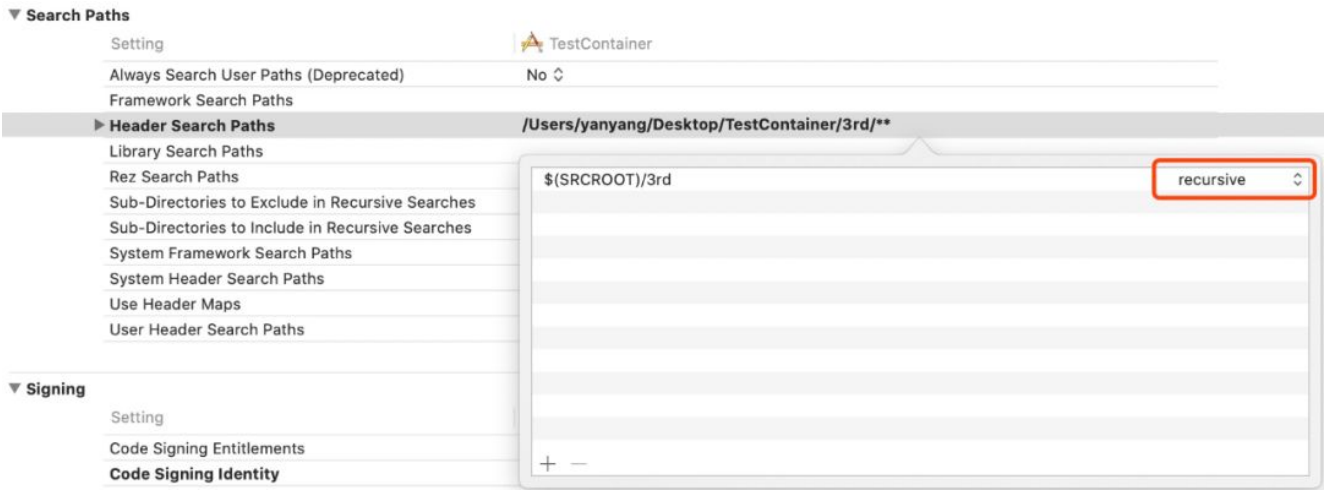
Debug 时是不需要生成符号表，可以检查一下子工程（尤其开源库）有没有设置正确。

2、将 Build Active Architecture Only 改为 Yes

Debug 时是不需要生成全架构，可以检查一下子工程（尤其开源库）有没有设置正确。

3、优化头文件搜索路径

避免工程 `Header Search Paths` 设置了路径递归引用：



Xcode 编译源文件时，会根据 `Header Search Paths` 自动添加 `-I` 参数，如果递归引用的路径下子目录越多，`-I` 参数也越多，编译器预处理头文件效率就越低，所以不能简单的设置路径递归引用。同样 `Framework Search Paths` 也类似处理。

二、使用 CocoaPods 管理第三方库

这是业界常用的做法，利用 cocoapods 插件 `cocoapods-packager` 将任意的 pod 打包成 Static Library，省去重复编译的时间；但缺点是不方便调试源码，如果库代码反复修改，需要重新生成二进制并上传到内部服务器，等等。

三、CCache

CCache 是一个能够把编译的中间产物缓存起来的工具，不需要过多修改项目配置，也不需要修改开发工具链。Xcode 9 有个很偶然的 bug，在源码没有任何修改的情况下经常触发全新编译，用 CCache 很好的解决这一问题。但随着 Xcode 10 修复全量编译问题，这一方案逐步弃用

了。

四、distcc

distcc 是一个分布式编译工具，它原理是把本地多个编译任务分发到网络中多个机器，其他机器编译完成后，再把产物返回给本机上执行链接，最终得到编译结果。

五、硬件解决

如把 `Derived Data` 目录放到由内存创建的虚拟磁盘，或者购买最新款的 iMac Pro...

实践过程

一、优化编译选项

1、优化头文件搜索路径

把一些递归引用路径去了后，整体编译速度快了 20s。

2、关闭 Enable Index-While-Building Functionality

这选项无意中找到的（Xcode 9 的新特性？），默认打开，作用是 Xcode 编译时会顺带建立代码索引，但影响编译速度。关闭后整体编译速度快 80s（Xcode 会换回以前的方式，在空闲时间建立代码索引）。

二、优化 kinda

kinda 是今年引入支付跨平台框架（C++），但编译速度

奇慢，一个源文件编译都要 30s。另外生成的二进制大小在 App 占比较高，感觉有不少冗余代码，理论上减少冗余代码也能加快编译速度。经过分析 LinkMap 文件和使用 Xcode Preprocess 某些源文件，发现有以下问题：

- proto 文件生成的代码较多
- 某个基类/宏使用了大量模版

对于问题一，可以设置 proto 文件选项为 `optimize_for=CODE_SIZE` 来让 protobuf 编译器生成精简版代码。但我是用自己的工具生成（具体原理可看这里），代码更少。

对于问题二，由于模版是编译期间的多态（增加代码膨胀和编译时间），所以可以把模版基类改成虚基类这种运行时的多态；另外推荐使用 `hyper_function` 取代 `std::function`，使得基类用通用函数指针，就能存储任意 lambda 回调函数，从而避免基类模板化。例如：

```
template <typename Request, typename Response>
class BaseCgi {
public:
    BaseCgi(Request request, std::function<void(Response &)
        _request = request;
        _callback = callback;
    }

    void onRequest(std::vector<uint8_t> &outData) {
        _request.toData(outData);
    }
```

```
    void onResponse(std::vector<uint8_t> &inData) {
        Response response;
        response.fromData(inData);
        callback(response);
    }

public:
    Request _request;
    std::function<void(Response &)> _callback;
};

class CgiA : public BaseCgi<RequestA, ResponseA> {
public:
    CgiA(RequestA &request, std::function<void(ResponseA &)
        BaseCgi(request, callback) {}
};
```

可改成:

```
class BaseRequest {
public:
    virtual void toData(std::vector<uint8_t> &outData) = 0;
};

class BaseResponse {
public:
    virtual void fromData(std::vector<uint8_t> &outData) =
};

class BaseCgi {
public:
    template <typename Request, typename Response>
```

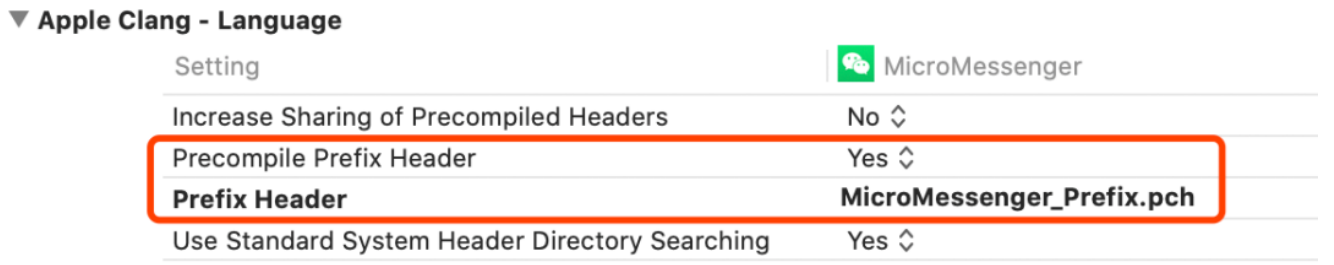
```
    BaseCgi(Request &request, hyper_function<void(Response
        _request = new Request(request);
        _response = new Response;
        _callback = callback;
    }

    void onRequest(std::vector<uint8_t> &outData) {
        _request->toData(outData);
    }

    void onResponse(std::vector<uint8_t> &inData) {
        _response->fromData(inData);
        _callback(*_response);
    }

public:
    BaseRequest *_request;
    BaseResponse *_response;
    hyper_function<void(BaseResponse &)> _callback;
};

class RequestA : public BaseRequest { ... };

class ResponseA : public BaseResponse { ... };

class CgiA : public BaseCgi {
public:
    CgiA(RequestA &request, hyper_function<void(ResponseA &
        BaseCgi(request, callback) {}
};
```

BaseCgi 由模版基类变成只有构造函数是模板的基类，onRequest 和 onResponse 逻辑代码并不因为基类模版

实例化而被“复制黏贴”。经过上述优化，整体编译速度快了 70s，而 kinda 二进制也减少了 60%，效果特别明显。

三、使用 PCH 预编译头文件

PCH（Precompile Prefix Header File）文件，也就是预编译头文件，其文件里的内容能被项目中的其他所有源文件访问。通常放一些通用的宏和头文件，方便编写代码，提高效率。另外 PCH 文件预编译完成后，后面用到 PCH 文件的源文件编译速度也会加快。缺点是 PCH 文件和 PCH 引用到的头文件内容一旦发生变化，引用到 PCH 的所有源文件都要重新编译。所以使用时要谨慎。在 Xcode 里设置 `Prefix Header` 和 `Precompile Prefix Header` 即可使用 PCH 文件并对它进行预编译：



微信使用 PCH 预编译后，编译速度提升非常可观，快了接近 280s。

终极优化

通过上述优化，微信工程的编译时间由原来的 **1,626.4s** 下降到 **1,182.8s**，快了将近 **450s**，但仍然需要 20 分钟，令人不满意。如果继续优化，得从编译器下手。正如我们平常做的客户端性能优化，在优化之前，先分析原理，输出每个地方的耗时，针对耗时做相对应的

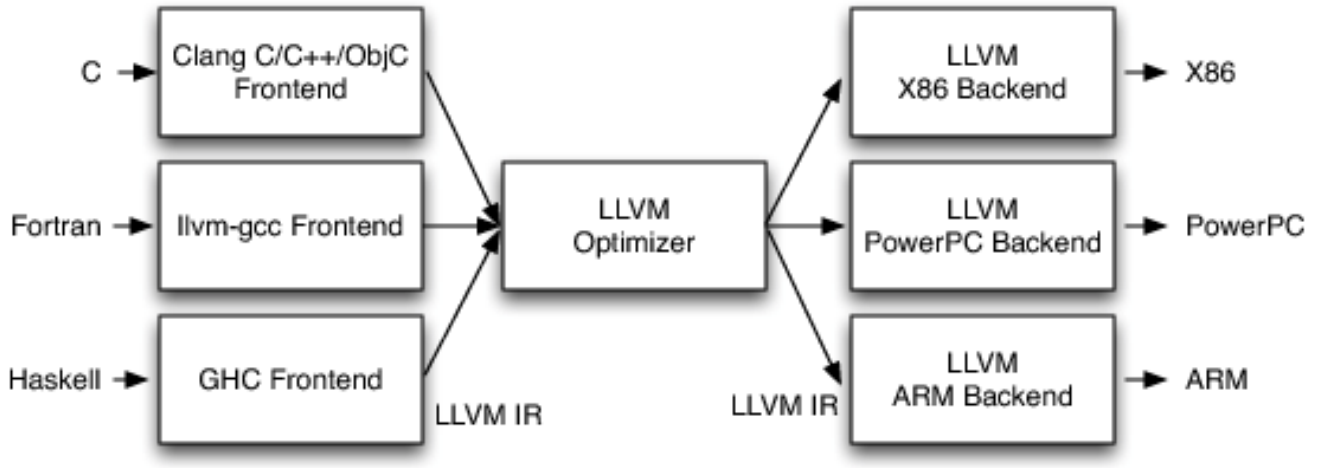
优化。

一、编译原理

编译器，是把一种语言（通常是高级语言）转换为另一种语言（通常是低级语言）的程序。大多数编译器由三部分组成：

- 前端（Frontend）：负责解析源码，检查错误，生成抽象语法树（AST），并把 AST 转化成类汇编中间代码
- 优化器（Optimizer）：对中间代码进行架构无关的优化，提高运行效率，减少代码体积，例如删除 `if (0)` 无效分支
- 后端（Backend）：把中间代码转换成目标平台的机器码

LLVM 实现了更通用的编译框架，它提供了一系列模块化的编译器组件和工具链。首先它定义了一种 LLVM IR（Intermediate Representation，中间表达码）。Frontend 把原始语言转换成 LLVM IR；LLVM Optimizer 优化 LLVM IR；Backend 把 LLVM IR 转换为目标平台的机器语言。这样一来，不管是新的语言，还是新的平台，只要实现对应的 Frontend 和 Backend，新的编译器就出来了。



在 Xcode，C/C++/ObjC 的编译器是 Clang（前端）+LLVM（后端），简称 Clang。Clang 的编译过程有这几个阶段：

```
➜  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
```

1、预处理

这阶段的工作主要是头文件导入，宏展开/替换，预编译指令处理，以及注释的去除。

2、编译

这阶段做的事情比较多，主要有：

- 词法分析（Lexical Analysis）：将代码转换成一系列 token，如大中小括

号 `paren'()' square'[]' brace'{}'`、标识符 `identifier`、字符串 `string_literal`、数字常量 `numeric_constant` 等等

- 语法分析（Semantic Analysis）：将 token 流组成抽象语法树 AST
- 静态分析（Static Analysis）：检查代码错误，例如参数类型是否错误，调用对象方法是否有实现
- 中间代码生成（Code Generation）：将语法树自顶向下遍历逐步翻译成 LLVM IR

3、生成汇编代码

LLVM 将 LLVM IR 生成当前平台的汇编代码，期间 LLVM 根据编译设置的优化级别 `Optimization Level` 做对应的优化（Optimize），例如 Debug 的 `-O0` 不需要优化，而 Release 的 `-Os` 是尽可能优化代码效率并减少体积。

4、生成目标文件

汇编器（Assembler）将汇编代码转换为机器代码，它会创建一个目标对象文件，以 .o 结尾。

5、链接

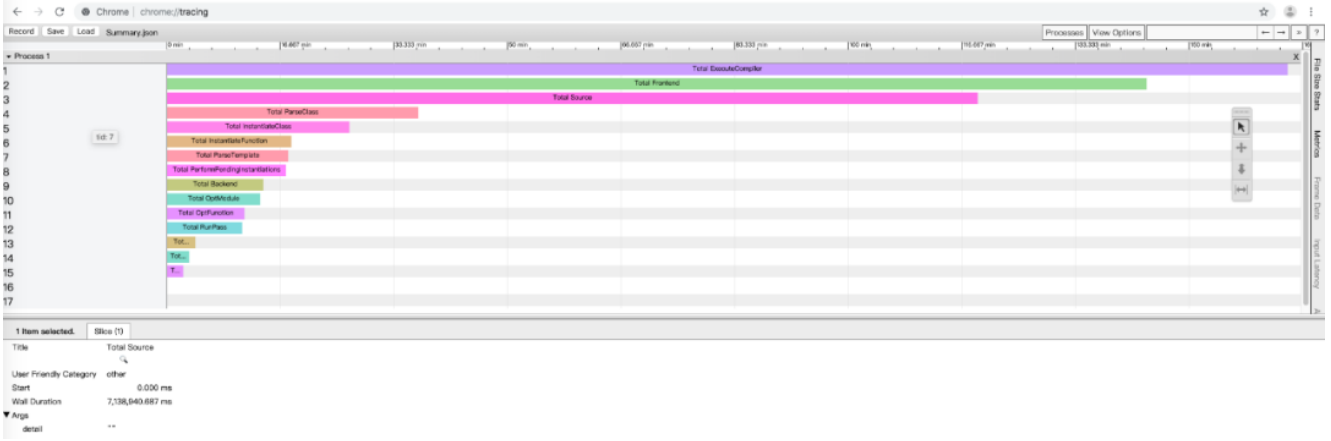
链接器（Linker）把若干个目标文件链接在一起，生成可执行文件。

二、分析耗时

Clang/LLVM 编译器是开源的，我们可以从官网下载其源码，根据上述编译过程，在每个编译阶段埋点输出耗时，生成定制化的编译器。在自己准备动手的前一周，国外大神 **Aras Pranckevičius** 已经在 LLVM 项目提交了 rL357340 修改：clang 增加 `-ftime-trace` 选项，编译时生成 Chrome（chrome://tracing） JSON 格式的耗时报告，列出所有阶段的耗时。效果如下：

- 整体编译（ExecuteCompiler）耗时 8,423.8ms
- 其中前端（Frontend）耗时 5,307.9ms，后端（Backend）耗时 3,009.6ms
- 而前端编译里头文件 SourceA 耗时 xx ms，B 耗时 xx ms，...
- 头文件处理里 Parse ClassA 耗时 xx ms，B 耗时 xx ms，...
- 等等

这就是我想要的耗时报告！接下来修改工程 `CC={YOUR PATH}/clang`，让 Xcode 编译时使用自己的编译器；同时编译选项 `OTHER_CFLAGS` 后面增加 `-ftime-trace`，每个源文件编译后输出耗时报告。最终把所有报告汇聚起来，形成整体的编译耗时：



由整体耗时可以看出，编译器前端处理（Frontend）耗时 7,659.2s，占整体 87%；而前端处理下头文件处理（Source）耗时 7,146.2s，占整体 71.9%！猜测头文件嵌套严重，每个源文件都要引入几十个甚至几百个头文件，每个头文件源码要做预处理、词法分析、语法分析等等。实际上源文件不需要使用某些头文件里的定义（如 class、function），所以编译时间才那么长。

于是又写了个工具，统计所有头文件被引用次数、总处理时间、头文件分组（指一个耗时顶部的头文件所引用到的所有子头文件的集合），列出一份表格（截取 Top10）：

Name	Time(s)	Ref Count	Average(s)	Group
[redacted]/AppUtil+WeChat.h	1187.7	2304	0.5	1
[redacted]/KernelObject+WeChat.h	1124.9	3831	0.3	2
[redacted]/MsgDelegate.h	1053.5	3202	0.3	1
/Users/yanyang/Library/Developer/Xcode/DerivedData/MicroMessenger-dmcukmbfqbvfsxhdjssnbbzmalsq/Build/Product	1051.9	4673	0.2	1
/Applications/Xcode.app/Contents/Developer/Platforms/iPhoneOS.platform/Developer/SDKs/iPhoneOS12.2.sdk/System/	1026.7	3415	0.3	1
[redacted]/MMAsset.h	940.6	3248	0.3	1
[redacted]/EditVideoLogicController.h	937.6	3052	0.3	1
[redacted]/ForwardMessageLogicController.h	903.2	2546	0.4	1
[redacted]/MessageMgr.h	884	2727	0.3	1

Header1 处理时间 1187.7s，被引用 2,304 次；Header2 处理时间 1,124.9s，被引用 3,831 次；后面 Header3～10 都是被 Header1 引用。所以可以尝试优化 TopN 头文件里的头文件引用，尽量不包含其他头文件。

三、解决耗时

通常我们写代码时，如果用到某个类，就直接 include 该类声明所在头文件，但在头文件，我们可以用前置声明解决。因此优化头文件思路很简单，就是能用前置声明，就用前置声明替代 include。实际上改动量非常大，我跟组内另外的同事 vakeee 分工优化 Header1 和 Header2，花了整整 5 个工作日，才改完。效果还是有，整体编译时间减少 80s。

但需要优化的头文件还有几十个，我们不可能继续做这种体力活。因此我们可以做这样的工具，通过 AST 找到代码里出现的标识符（包括类型、函数、宏），以及标识符定义所在文件，然后分析是否需要 include 它定义所在文件。

先看看代码如何转换 AST，如以下代码：

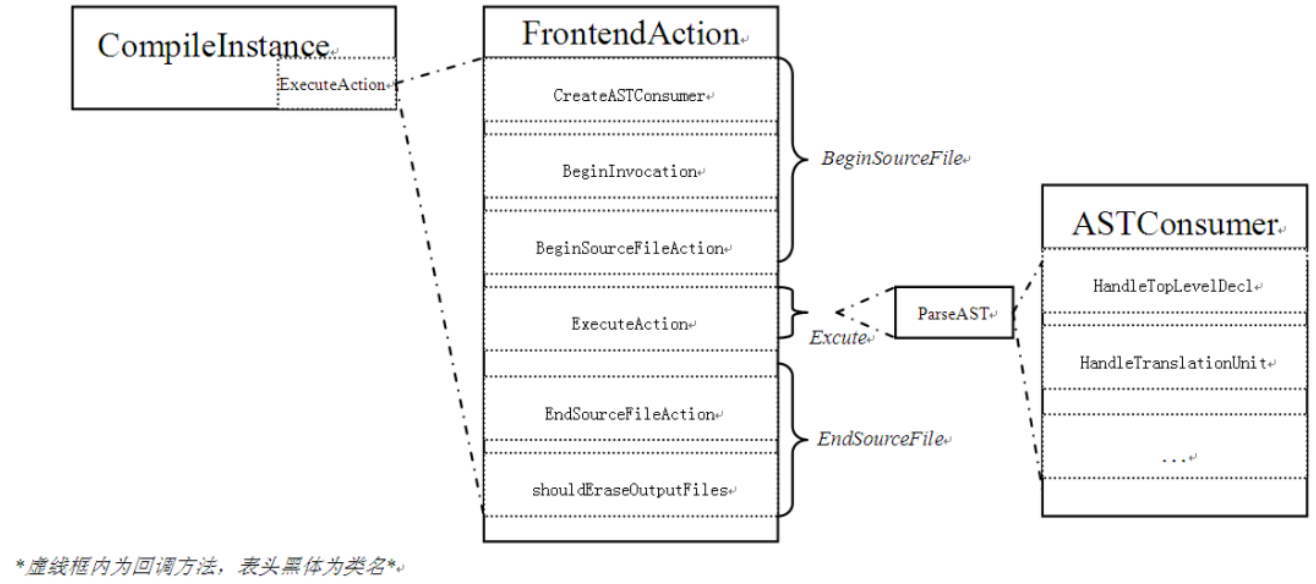
```
// HeaderA.h
struct StructA {
    int val;
};

// HeaderB.h
struct StructB {
    int val;
};

// main.c
#include "HeaderA.h"
#include "HeaderB.h"

int testAndReturn(struct StructA *a, struct StructB *b) {
    return a->val;
```

```
}
```

控制台输入：

```
➜  TestContainer clang -Xclang -ast-dump -fsyntax-only main
TranslationUnitDecl 0x7f8f36834208 <<invalid sloc>> <invali
|-RecordDecl 0x7faa62831d78 <./HeaderA.h:12:1, line:14:1> l
| `-FieldDecl 0x7faa6383da38 <line:13:2, col:6> col:6 refer
|-RecordDecl 0x7faa6383da80 <./HeaderB.h:12:1, line:14:1> l
| `-FieldDecl 0x7faa6383db38 <line:13:2, col:6> col:6 val '
`-FunctionDecl 0x7faa6383de50 <main.c:35:1, line:37:1> line
  |-ParmVarDecl 0x7faa6383dc30 <col:19, col:35> col:35 used
  |-ParmVarDecl 0x7faa6383dd40 <col:38, col:54> col:54 b 's
  `-CompoundStmt 0x7faa6383dfc8 <col:57, line:37:1>
    `-ReturnStmt 0x7faa6383dfb8 <line:36:2, col:12>
      `-ImplicitCastExpr 0x7faa6383dfa0 <col:9, col:12> 'in
        `-MemberExpr 0x7faa6383df70 <col:9, col:12> 'int' l
          `-ImplicitCastExpr 0x7faa6383df58 <col:9> 'struct
            `-DeclRefExpr 0x7faa6383df38 <col:9> 'struct St
```

从上可以看出，每一行包括 AST Node 的类型、所在位置（文件名，行号，列号）和结点描述信息。头文件定义的类也包含进 AST 中。AST Node 常见类型有 Decl（如 RecordDecl 结构体定义，FunctionDecl 函数定义）、Stmt（如 CompoundStmt 函数体括号内实现）。



Clang AST 有三个重要的基类：ASTFrontendAction、ASTConsumer 以及 RecursiveASTVisitor。ClangTool 类读入命令行配置项后初始化 CompilerInstance；CompilerInstance 成员函数 `ExcutionAction` 会调用 ASTFrontendAction 3 个成员函数 `BeginSourceFile`（准备遍历 AST）、`Execute`（解析 AST）、`EndSourceFileAction`（结束遍历）。ASTFrontendAction 有个重要的纯虚函数 `CreateASTConsumer`（会被自己 `BeginSourceFile` 调用），用于返回读取 AST 的 ASTConsumer 对象：

```
class MyFrontendAction : public clang::ASTFrontendAction {
public:
    virtual std::unique_ptr<clang::ASTConsumer> CreateASTCo
        TheRewriter.setSourceMgr(CI.getASTContext().getSour
        return llvm::make_unique<MyASTConsumer>(&CI);
    }
};

int main(int argc, const char **argv) {
    clang::tooling::CommonOptionsParser op(argc, argv, Opts
```

```
    clang::tooling::ClangTool Tool(op.getCompilations(), op
    int result = Tool.run(clang::tooling::newFrontendAction

    return result;
}
```

ASTConsumer 有若干个可以 override 的方法，用来接收 AST 解析过程中的回调，其中之一是工具用到的 `HandleTranslationUnit` 方法。当编译单元 TranslationUnit 的 AST 完整解析后，`HandleTranslationUnit` 会被回调。我们在 `HandleTranslationUnit` 使用 RecursiveASTVisitor 对象以深度优先的方式遍历 AST 所有结点：

```
class MyASTVisitor
: public clang::RecursiveASTVisitor<MyASTVisitor> {
public:
    explicit MyASTVisitor(clang::ASTContext *Ctx) {}

    bool VisitFunctionDecl(clang::FunctionDecl* decl) {
        // FunctionDecl 下的所有参数声明允许前置声明取代 include
        // 如上面 Demo 代码里 StructA、StructB
        return true;
    }

    bool VisitMemberExpr(clang::MemberExpr* expr) {
        // 被引用的成员所在的类，需要 include 它定义所在文件
        // 如 StructA
        return true;
    }

    bool VisitXXX(XXX) {
```

```
        return true;
    }

    // 同一个类型，可能出现若干次判定结果
    // 如果其中一个判断的结果需要 include，则 include
    // 否则使用前置声明代替 include
    // 例如 StructA 只能 include，StructB 可以前置声明
};

class MyASTConsumer : public clang::ASTConsumer {
private:
    MyASTVisitor Visitor;
public:
    explicit MyASTConsumer(clang::CompilerInstance *aCI)
    : Visitor(&(aCI->getASTContext())) {}

    void HandleTranslationUnit(clang::ASTContext &context)
        clang::TranslationUnitDecl *decl = context.getTrans
        Visitor.TraverseTranslationUnitDecl(decl);
    }
};
```

工具框架大致如上所示。不过早在 2011 年 Google 内部做了个基于 Clang libTooling 的工具 include-what-you-use，用来整理 C/C++ 头文件，效果如下：

```
➜  include-what-you-use main.c
HeaderA.h has correct #includes/fwd-decls)

HeaderB.h has correct #includes/fwd-decls)

main.c should add these lines:
```

```
struct StructB;

main.c should remove these lines:
- #include "HeaderB.h"  // lines 2-2

The full include-list for main.c:
#include "HeaderA.h"  // for StructA
struct StructB;
```

我们在 IWYU 基础上，增加了 ObjC 语言的支持，并增强它的逻辑，让结果更好看（通常 IWYU 处理完后，会引入很多头文件和前置声明，我们做剪枝处理，进一步去掉多余的头文件和前置声明，篇幅限制就不多做解释了）。

微信源码通过工具优化头文件引入后，整体编译时间降到了 **710s**。另外头文件依赖的减少，也能降低因修改头文件引起大规模源码重编的可能性。我们再用编译耗时分析工具分析当前瓶颈：

	Name	Time(s)	Ref Count	Average(s)	Group
2	[redacted]./WCDB/WCDB.framework/Headers/WCTConvertible.h	1334.9	1185	1.1	1
3	[redacted]k/WCDB/WCDB.framework/Headers/WINQ.h	1333.3	1185	1.1	2
4	[redacted].c/WCDB/WCDB.framework/Headers/WCTCommon.h	1265.2	1130	1.1	3
5	[redacted]/WCDB/WCDB.framework/Headers/SQL.hpp	1250.8	1185	1.1	4
6	[redacted]./WCDB/WCDB.framework/Headers/ColumnType.hpp	1230	1185	1	5
7	[redacted]./WCDB/WCDB.framework/Headers/WCTCoding.h	1210.1	1137	1.1	6
8	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/string	1059.6	3950	0.3	7
9	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/ostream	1023	2675	0.4	8
10	[redacted]/WCDB/WCDB.framework/Headers/Syntax.h	997.2	1185	0.8	9
11	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/sstream	869.2	1825	0.5	10
12	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/algorithm	850.8	4279	0.2	11
13	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/ios	731.6	2674	0.3	10
14	/Applications/Xcode.app/Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/include/c++/v1/map	712	3429	0.2	12

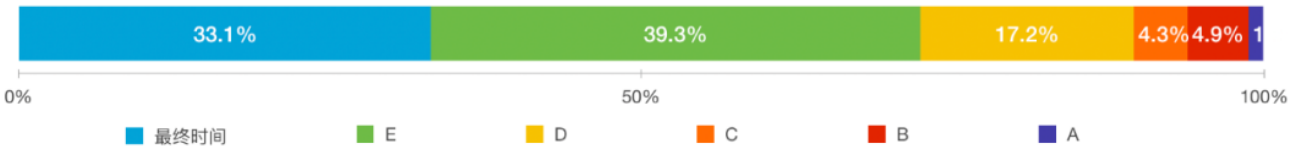
WCDB 头文件处理时间太长了，业务代码（如 Model 类）没有很好的隔离 WCDB 代码，把 WINQ 暴露出去，外面被动 include WCDB 头文件。解决方法有很多，例如 WCDB 相关放 category 头文件（XXModel+WCDB.h）里引入，或者跟其他库一样，把 `<WCDB/WCDB.h>` 放 PCH。

最终编译时间优化到 540s 以下，是原来的三分之一，编

译效率得到巨大的提升。

优化总结

总结微信的编译优化方案：



- A、优化头文件搜索路径
- B、关闭 Enable Index-While-Building Functionality
- C、优化 PB/模版，减少冗余代码
- D、使用 PCH 预编译
- E、使用工具优化头文件引入；尽量避免头文件里包含 C++ 标准库

未来展望

期待公司的蓝盾分布式编译 for ObjC；另外可以把业务代码模块化，项目文件按模块加载，目前 kinda/小程序/mars 在很好的实践中。

参考文献

- 如何将 iOS 项目的编译速度提高5倍
- 深入剖析 iOS 编译 Clang / LLVM

- Clang之语法抽象语法树AST
- time-trace: timeline / flame chart profiler for Clang
- Introduction to the Clang AST