目 录

[1. 介绍 1](#_Toc1176845868)

[2. 命令行选项 1](#_Toc601459567)

[2.1. 控制错误和警告信息的选项 1](#_Toc596668886)

[2.1.1. 格式化诊断信息 2](#_Toc103361969)

[2.1.2. 单独警告组 5](#_Toc620925078)

[2.2. 控制Clang崩溃诊断选项 6](#_Toc563970338)

[3. 语言和无关目标的特性 6](#_Toc1303923145)

[3.1. 控制错误和警告 6](#_Toc1087754203)

[3.1.1. 控制Clang如何显示诊断信息 7](#_Toc545269824)

[3.1.2. 诊断映射和诊断分类 7](#_Toc1593250539)

[3.1.3. 通过命令行标志控制诊断和通过Pragmas控制诊断 7](#_Toc332426790)

[3.1.4. 通过系统头文件控制诊断 8](#_Toc906620410)

[3.1.5. 使能所有警告 9](#_Toc1466252806)

[3.1.6. 控制静态分析器诊断 9](#_Toc403327262)

[3.2. 预编译的头文件 9](#_Toc1350903203)

[3.2.1. 生成、使用和重定位PCH文件 9](#_Toc816406897)

[3.3. 控制代码生成 10](#_Toc51368325)

[3.4. 控制调试信息大小 12](#_Toc574162116)

[3.5. 注释解析选项 12](#_Toc97575658)

[4. C语言特性 12](#_Toc601479848)

[4.1. clang支持的扩展 12](#_Toc838900797)

[4.2. 各个标准模式之间的差异 13](#_Toc585602930)

[4.3. 尚未实现的GCC扩展 13](#_Toc967282691)

[4.4. 故意不支持的GCC扩展 14](#_Toc635153400)

[4.5. Microsoft扩展 14](#_Toc1726340217)

[5. C++、Objective-C、Objective-C++语言特性 15](#_Toc1121111875)

[5.1. C++语言 15](#_Toc967360320)

[5.2. Objective-C语言特性 15](#_Toc2034687883)

[5.3. Objective-C++特性 15](#_Toc577808917)

[6. 特定目标特性和限制 15](#_Toc642291985)

[6.1. CPU架构特性和限制 15](#_Toc1126621144)

[6.2. 操作系统特性和限制 16](#_Toc1754654785)

[7. clang-cl 16](#_Toc1243751552)

[7.1. 命令行选项 17](#_Toc1723290030)

[7.1.1. /fallback 选项 18](#_Toc1858016754)

[8. 附录 18](#_Toc1864676630)

# 介绍

Clang编译器是一个面向C编程语言家族的开源编译器，旨在成为这些语言中类实现最好的。Clang建立在LLVM优化器和代码生成器，允许它提供高质量的优化和多目标的代码生成支持。更多信息，请参见Clang网站和LLVM网站。

这份文档描述了有关使用Clang作为编译器的终端用户的重要注记，记录了支持的特性，命令行选项等等。如果你对使用Clang来构建一个处理代码的工具，请看”Clang” CFE Internal Manual。如果你对Clang Static Analyzer感兴趣，请参考它的网站。

Clang被设计来支持C编程语言家族的，其中包含了C，Object-C，C++和Object-C++等。对于特定语言的信息，请参见对应语言的节：

* C 语言: K&R C, ANSI C89, ISO C90, ISO C94 (C89+AMD1),ISO C99(+TC1,TC2,TC3).
* Objective-C 语言: ObjC 1, ObjC 2, ObjC 2.1, 加上取决于基本语言的变体
* C++ 语言
* Objective C++ 语言

对于这些基础语言和他们的方言，Clang支持很广泛的语言扩展，他们记录在对应语言的节中。这些扩展被提供来兼容GCC，Microsoft和或者其他流行的编译器来提高功能通过Clang-specific特性。Clang驱动器和语言特性设计来合理的兼容GNU GCC编译器，从GCC迁移到Clang消除合并。在大多数情况下，代码“just works”。

对于特定语言特性，Clang具有针对CPU架构或者编译的目的操作系统的广泛特性。请参见Target-Specific Features and Limitations 一节获取详细信息。

剩下的介绍引入了在整个手册中使用的基本的编译器名词和一个使用Clang作为命令行编译器的使用介绍。

**名词**

前端、解析器、后端、预处理器、未定义行为、诊断、优化器

**基本使用**

给新手的介绍如何使用一个C编译器

编译+连接然后连接调试信息使能优化选择一个语言来使用，默认是C99。基于扩展的Autosenses。使用makefile。

# 命令行选项

这一节是一个到其他各节的索引。没有深入讨论其他各节覆盖的内容。然而，第一部分介绍了语言选择和其他高级选项例如-c、-g等等。

## 控制错误和警告信息的选项

* -Werror

将警告转换成错误。

* -Wno-error=foo

保持警告“foo”不被转换成错误，即使-Werror被指定。

* -Wfoo

使能警告“foo”。

* -w

禁用所有警告。

* -Weverything

使能所有警告。

* -pedantic

警告语言扩展。

* -pedantic-errors

把语言扩展视作错误。

* -Wsystem-headers

使能来自系统头文件的警告。

* -ferror-limit=123

在诊断出123个错误之后停止诊断。默认是20，错误限制可以通过-ferror-limit=0来禁用。

* -ftemplate-backtrace-limit=123

最多实例化123个模板在模板实例化回溯对于单个警告或错误。限制的默认是10，也可以通过-ftemplate-backtrace-limit=0来禁用。

### 格式化诊断信息

Clang默认旨在生成漂亮的诊断信息，特被对于clang的新用户。然而，不同的人具有不同的喜好，并且有时候Clang被另一个程序调用想要解析简单和一致的输出，而不是一个人。对于这些情形，Clang提供了一个广泛的范围的选项来控制它生成的诊断信息的输出格式。

* -f[no-]show-column

在诊断信息中打印列数。

这个选项，默认是开启的，控制是否Clang打印一个诊断信息的列数。举个例子，当使能之后，Clang将会打印如下:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

当被禁用之后，Clang将会打印”test.c:28:warning…”而没有列号。

打印出的列号从一行开始计数；小心你的源代码中包含多字节字符。

* -f[no-]show-source-location

在诊断信息中打印源 文件/行/列 信息。

这个选项，默认是开启的，控制Clang是否打印一个诊断的文件名、行号和列号。举个例子，当被使能之后，Clang将会有如下输出:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

当被禁用之后，Clang将不会打印”test.c:28:8”部分。

* -f[no-]caret-diagnostics

在诊断信息中打印源代码文件行和范围。这个选项，默认是开启的，控制Clang在遇到一个诊断时候是否打印源行、代码范围和插入记号。举个例子，当被使能之后，Clang将会有如下输出:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

* -f[no-]color-diagnostics

这个选项，在一个检测到兼容彩色的中断终端上默认是开启的，控制Clang是否带颜色输出。

当被使能之后，Clang将会使用高亮指定诊断中特殊部分，例如:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

当被禁用时候，Clang将只输出:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

* -fdiagnostics-format=clang/msvc/vi

改变诊断输出使得更高的匹配IDE和命令行工具。

这个选项控制诊断信息中文件名、行号和列的输出格式。这个选项和它的效果格式化一个简单变换诊断，如下:

clang (默认)

|  |
| --- |
| t.c:3:11: warning: conversion specifies type 'char \*' but the argument has type 'int' |

msvc

|  |
| --- |
| t.c(3,11) : warning: conversion specifies type 'char \*' but the argument has type 'int' |

vi

|  |
| --- |
| t.c +3:11: warning: conversion specifies type 'char \*' but the argument has type 'int' |

* -f[no-]diagnostics-show-name

使能显示诊断名称。这个选项，默认是关闭的，控制Clang是否打印相关名称。

* -f[no-]diagnostics-show-option

在诊断行中使能[-Woption]信息。

这个选项，默认是开启的，控制输出一个警告诊断时候Clang是否打印相关的警告组选项名称。举个例子，在这个输出中:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

传递-fno-diagnostics-show-option将会阻止Clang在诊断中打印[-Wextra-tokens]信息。这个信息告诉你需要使能或者禁止诊断的标志，不论是从命令行还是#pragma GCC diagnostic。

* -fdiagnostics-show-category=none/id/name

使能在诊断行打印分类信息。

这个选项，默认是”none”,控制Clang在生成诊断的时候是否打印关联的分类。每个诊断信息可能关联或者无关联到一个类，如果有一个，它被列在诊断行中的类域中(在[]中)。

举个例子，一个格式化字符串警告将会产生如下三行基于这个选项的设置:

|  |
| --- |
| t.c:3:11: warning: conversion specifies type 'char \*' but the argument has type 'int' [-Wformat]  t.c:3:11: warning: conversion specifies type 'char \*' but the argument has type 'int' [-Wformat,1]  t.c:3:11: warning: conversion specifies type 'char \*' but the argument has type 'int' |

* [-Wformat,Format String]

分类可以被客户端使用需要按照类把诊断信息分组的话，所以它应当是一个高的级别。我们只需要几十个，而不是成百上千。

* -f[no-]diagnostics-fixit-info

在诊断输出中使能”FixIt”信息。

这个选项，默认是开启的，控制Clang当它知道时是否打印如何修复特定诊断信息。举个例子，在这个输出中:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^  // |

传递-fno-diagnostics-fixit-info将会阻止Clang打印末尾的”//“行。这个信息对不了解是什么错误的用户是非常有用的，但是可能会迷惑机器解析。

* -fdiagnostics-print-source-range-info

打印机器可以解析的有关源范围的信息。这个选项使得Clang以机器可解析格式在 文件/行/列 后打印有关源范围信息。这个信息是一些花括号中的简单序列，每个范围列出了开始和结束 行/列 位置。举个例子，在这个输出中:

|  |
| --- |
| exprs.c:47:15:{47:8-47:14}{47:17-47:24}: error: invalid operands to binary expression ('int \*' and '\_Complex float')  P = (P-42) + Gamma\*4;  ~~~~~~ ^ ~~~~~~~ |

{}是由 -fdiagnostics-print-source-range-info 产生的。

打印的列号从行开始计数；小心你的文件中多字节字符。

* -fdiagnostics-parseable-fixits

以机器可解析格式打印Fix-It。

这个选项使得Clang以一种机器可解析的格式在诊断末尾打印可用的Fix-It信息。下边的例子展示了格式:

|  |
| --- |
| fix-it:"t.cpp":{7:25-7:29}:"Gamma" |

输出中的范围是一个半开范围，所以在这个例子中列t.cpp中25起的字符到但是不包含行7列29的字符串应当被”Gamma”替换。这个范围或者取代字符串都可能为空(分别代表严格的插入和严格擦除)。文件名和插入字符串逃逸反斜杠(“\“),tab(“\t”),新行(“\n”)，双引号(“\””)和不可打印字符(八进制”\xxx”)。

打印的列号从行首开始计数；小心文件中的多字节字符。

* -fno-elide-type

在模板类型打印中关闭省略。

默认的模板类型打印省略尽可能多的模板参数，移除在模板类型中相同的，只留下不同的。添加这个标志将会打印所有模板参数。如果终端支持，高亮将出现在不同的参数。

默认:

|  |
| --- |
| t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<double, [...]>>>' for 1st argument; |

-fno-elide-type:

|  |
| --- |
| t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<int, map<float, int>>>' to 'vector<map<int, map<double, int>>>' for 1st argument; |

* -fdiagnostics-show-template-tree

模板类型区分打印文本的树。

对于大型的模板类型，这个选项将会导致一个故意的文本树，一个参数一行，具有不同的行内标记。这与 -fno-elide-type 兼容。

默认:

|  |
| --- |
| t.cc:4:5: note: candidate function not viable: no known conversion from 'vector<map<[...], map<float, [...]>>>' to 'vector<map<[...], map<double, [...]>>>' for 1st argument; |

使用 -fdiagnostics-show-template-tree

|  |
| --- |
| t.cc:4:5: note: candidate function not viable: no known conversion for 1st argument;  vector<  map<  [...],  map<  [float != float],  [...]>>> |

### 单独警告组

TODO: 从tblgen生成这个。为每个警告组定义一个锚。

* -Wextra-tokens

警告一个预处理指令尾部的过度的标识符。

这个选项，默认是开启的，使能警告一个预处理指令尾部的过度的标识符。举例如下:

|  |
| --- |
| test.c:28:8: warning: extra tokens at end of #endif directive [-Wextra-tokens]  #endif bad  ^ |

这些额外的标识符不严格符合，通常最好注释掉他们。

* -Wambiguous-member-template

|  |
| --- |
| 警告有关不合格的成员模板的使用，在使用点名字解析到另一个模板。 |

这个选项，默认是开启的，使能警告如下代码中:

|  |
| --- |
| template<typename T> struct set{};  template<typename T> struct trait { typedef const T& type; };  struct Value {  template<typename T> void set(typename trait<T>::type value) {}  };  void foo() {  Value v;  v.set<double>(3.2);  } |

C++[basic.lookup.classref] 需要这个成为一个错误，但是，因为工作比较苦难，Clang把它降级为一个警告作为一个扩展。

* -Wbind-to-temporary-copy

警告关于一个不可用的复制构造器当绑定一个引用到一个暂时的。

这个选项，默认是开启的，使能警告有关绑定一个引用到一个临时的，当临时的没有一个可用的复制构造子。举个例子:

|  |
| --- |
| struct NonCopyable {  NonCopyable();  private:  NonCopyable(const NonCopyable&);  };  void foo(const NonCopyable&);  void bar() {  foo(NonCopyable()); // Disallowed in C++98; allowed in C++11.  }  struct NonCopyable2 {  NonCopyable2();  NonCopyable2(NonCopyable2&);  };  void foo(const NonCopyable2&);  void bar() {  foo(NonCopyable2()); // Disallowed in C++98; allowed in C++11.  } |

注意如果 NonCopyable2::NonCopyable2() 具有一个默认参数，它的实例化产生一个编译错误，这个错误在C++98模式中仍然是一个硬错误，即使这个警告被关闭。

## 控制Clang崩溃诊断选项

听起来不可信，Clang不时的崩溃。通常，这只发生在those living on the bleeding edge。Clang可以帮助你填一个bug报告。特别的，Clang生成预编译的源文件额相关的运行脚本在一个崩溃之上。这些文件应当被附件到bug报告。下边是命令行选项来控制崩溃诊断。

* -fno-crash-diagnostics

禁止在一个clang崩溃期间自动生成预编译的源文件。

-fno-crash-diagnostics标志可以对于加速生成处理有帮助。

<https://llvm.org/docs/tutorial/index.html>

# 语言和无关目标的特性

## 控制错误和警告

Clang提供了许多方式控制代码构建导致它引起错误和警告信息，并且和它们如何显示到终端。

### 控制Clang如何显示诊断信息

当Clang生成一个诊断，在它的输出中包含了丰富的信息，并且给你精细的控制什么信息要输出。Clang具有打印这个信息的能力，以下是控制选项:

1. 一个 文件/行/列 预示器显示代码中生成诊断的地方 [-fshow-column, -fshow-source-location]。
2. 一个诊断信息作为注记、警告、错误和致命错误的分类。
3. 一个文本字符串描述问题是什么。
4. 一个选项预示如何控制这个诊断(对于支持的诊断) [-fdiagnostics-show-option]。
5. 一个对于诊断的的高级别分类给想要更具诊断类别分类的客户(对于支持的诊断) [-fdiagnostics-show-category]。
6. 问题产生的源代码行，并且具有一个花括号和范围预示重要位置[-fcaret-diagnostics]。
7. “FixIt”信息，简洁的解释如何修复问题 (当 Clang 确定它知道的时候) [-fdiagnostics-fixit-info]。
8. 一个机器可以解析的范围表示(默认关闭的) [-fdiagnostics-print-source-range-info]。

有关更多信息，请参见格式化诊断信息。

### 诊断映射和诊断分类

诊断映射

所有诊断被隐射到五类中的一类:

1. Ignored 忽略
2. Note 注记
3. Warning 警告
4. Error 错误
5. Fatal 致命错误

诊断分类

虽然默认没有显示，诊断可以每个都关联到高级别的类别。 This category is intended to make it possible to triage builds that produce a large number of errors or warnings in a grouped way.

类别默认没有被显示，但是它们可以使用-fdiagnostics-show-category 选项开启。当设置到 “name” 时候，类别被以文本打印在诊断中。当被设置为“id”,将会打印类别号。 类别名称和ID的隐射可以通过执行 ‘clang —print-diagnostic-categories‘ 来获得。

### 通过命令行标志控制诊断和通过Pragmas控制诊断

TODO: -W flags, -pedantic, etc

通过Pragmas控制诊断

Clang也可以通过源代码中的pragmas控制什么诊断是要使能的。这在关闭代码中特定区间的诊断时候是非常有用的。Clang支持GCC的pragma来兼容已经存在代码，和几个扩展。

pragma 可以控制任何命令行可以使用的警告。警告可以被设置为忽略、警告、错误或者致命错误。下面的例子代码将会告诉Clang或者GCC去忽略-Wall警告:

|  |
| --- |
| #pragma GCC diagnostic ignored "-Wall" |

作为对GCC的pragma提供的所有功能的附加，Clang还允许你push和pop当前的警告信息。当写一个头文件要被其他人编译的时候是非常有用的，因为你不知道它们提供的警告标志。

下面的例子中，-Wmultichar只被一行代码忽略，在这之后诊断返回之前存在的状态。

|  |
| --- |
| #pragma clang diagnostic push  #pragma clang diagnostic ignored "-Wmultichar"  char b = 'df'; // no warning.  #pragma clang diagnostic pop |

push和pop pragmas将会保存和恢复编译器的全部诊断状态，不论它如何被设置。这意味着有可能push和pop围绕GCC兼容诊断和Clang将会push和pop适当的，而GCC将会忽略push和pop作为未知pragmas。注意虽然Clang支持GCC pragma，Clang和GCC不支持完全相同的警告集合，所以即使使用GCC兼容#pragmas，仍然没有保证它们在两个编译器中行为相同。

作为控制编译器生成的警告和错误的附加，有可能通过如下pragmas生成定制的警告和错误信息:

|  |
| --- |
| // The following will produce warning messages  #pragma message "some diagnostic message"  #pragma GCC warning "TODO: replace deprecated feature"  // The following will produce an error message  #pragma GCC error "Not supported" |

这些pragmas操作与#warning和#error预处理指令相似，除了它们可能通过C99 \_Pragma操作被潜入预处理器宏:

|  |
| --- |
| #define STR(X) #X  #define DEFER(M,...) M(\_\_VA\_ARGS\_\_)  #define CUSTOM\_ERROR(X) \_Pragma(STR(GCC error(X " at line " DEFER(STR,\_\_LINE\_\_))))  CUSTOM\_ERROR("Feature not available"); |

### 通过系统头文件控制诊断

当警告发生在系统头文件的时候会被压制。默认的， 一个被包含的文件被当作是系统头文件，如果它是在-isystem指定的include 路径中找到的话，但是这也可以被通过好几种方式重载。

system\_header pragma可以被用于标记当前文件是系统头文件。从pragma位置开始在这个文件中将不会有警告产生。

|  |
| --- |
| char a = 'xy'; // warning  #pragma clang system\_header  char b = 'ab'; // no warning |

-isystem-prefix 和 -ino-system-prefix 命令行参数可以被用来重载一个include路径的子集作为系统头文件。当名称在一个#include指令被发现在一个头文件搜索路径并且具有system前缀，这个头文件就被当作一个系统头文件。最后一个后缀在命令行上匹配指定的头文件名。如下:

|  |
| --- |
| $ clang -Ifoo -isystem bar -isystem-prefix x/ -ino-system-prefix x/y/ |

这里，#include “x/a.h” 被当作包含一个系统头文件，即使这个文件在foo中被找到， #include “x/y/b.h” 不被当作系统头文件，即使这个文件在bar中。

一个 #include 指令找到一个文件相对于当前目录被当作系统头文件，如果包含文件被当作系统头文件。

### 使能所有警告

作为对传统-W标志的附加，可以通过传递-Weverything使能所有警告。这个工作和使用-Werror预期相同，并且也包含从-pedantic而来的警告。

注意：当组合-w(禁用所有警告)时，起作用的是这个标志。

### 控制静态分析器诊断

虽然不是编译器的严格部分，Clang的静态分析器产生的诊断也可以被用户通过改变源代码影响。更多信息见于可用的注释和分析器的FAQ页面。

## 预编译的头文件

预编译的头文件是一种很多编译器都使用来减少编译时间的通用方式。该方法的基本动机是相同的头文件(并且通常很大)被包含进不同的源文件，是很常见的。结果，编译时间通常可以被很大程度的提高，通过抓住一些编译器处理头文件的(多余的)工作。预编译的头文件, 代表了很多种方式中的一种来实现这种优化， 是逐字文件代表一个磁盘上的包含大量减少处理对应的头文件的相同工作的信息缓存。每个编译器的预编译的头文件细节都不相同，预编译的头文件在具有大量头文件的系统上加速程序的编译是非常高效的(例如 Mac OS/X)。

### 生成、使用和重定位PCH文件

生成PCH文件

使用Clang生成一个PCH文件，可以使用-x -header 选项调用Clang。这映射到GCC中生成PCH文件的接口:

|  |
| --- |
| $ gcc -x c-header test.h -o test.h.gch  $ clang -x c-header test.h -o test.h.pch |

使用PCH文件

一个PCH文件可以被用作一个前缀头文件，当一个-include选项被传递到clang时:

|  |
| --- |
| $ clang -include test.h test.c -o test |

clang驱动将会首先检查是否一个PCH文件test.h可用；如果是，test.h 的内容 (和它包含的文件) 将被从这个PCH文件处理。否则, Clang 回溯到直接处理 test.h 的内容。这映射到GCC的行为。

注意：

Clang不会自动使用源文件中直接包含的PCH文件。举例如下:

|  |
| --- |
| $ clang -x c-header test.h -o test.h.pch  $ cat test.c |

代码：

|  |
| --- |
| #include "test.h" |

编译

|  |
| --- |
| $ clang test.c -o test |

在这个例子中，clang将不会自动使用PCH文件test.h，由于test.h被直接包含进源文件并没有使用-include指令在命令行上指定。

重定位PCH文件

有时候需要从还不是最终头文件位置的头文件创建一个预编译的头文件。举个例子，有人可能像创建一个预编译的头文件在编译树，然后想要安装头文件。Clang允许创建“relocatable” 预编译的头文件，使用一个给定的位置(在编译目录中)，后边可以从一个安装的位置使用。

创建一个可重定位的预编译的头文件，把你的文件放进一个模拟安装位置的子目录。举个例子，你想要创建一个可重定位的预编译的头文件mylib.h，它将会被安装到/usr/include，那么创建一个子目录build/usr/include 并放 mylib.h 进这个子目录。如果 mylib.h 取决于其他头文件，它们可以存储在 build/usr/include 中模拟安装位置。

创建一个可重定位的预编译的头文件需要两个附加参数，首先，传递 --relocatable-pch 标志来预示结果PCH文件应当可重定位。第二，传递 -isysroot /path/to/build，它将使得所有你的库依赖到编译目录。举个例子:

|  |
| --- |
| # clang -x c-header --relocatable-pch -isysroot /path/to/build /path/to/build/mylib.h mylib.h.pch |

在加载可重定位PCH文件的时候，在PCH文件中使用的各种头文件将在系统头文件根目录找到。举个例子，mylib.h可以在/usr/include/mylib.h找到。如果头文件安装在其他系统根目录，-isysroot选项可以用来提供一个不同的系统根目录来找头文件。举个例子，-isysroot /Developer/SDKs/MacOSX10.4u.sdk 将会在/Developer/SDKs/MacOSX10.4u.sdk/usr/include/mylib.h 找mylib.h。

可重定位预编译的头文件被用于有限的情形，编译环境被高度控制并且预编译的头文件不能在头文件被安装之后生成。

## 控制代码生成

Clang提供了很多种方式来控制代码生成。选项列表如下:

|  |
| --- |
| -fsanitize=check1,check2,… |

开启运行时对于多种未定义的或者可疑的行为进行检查。

这个选项控制Clang是否添加运行时对于多种未定义的或者可疑的行为进行检查，默认是关闭的。如果一个检查失败，将会有一个运行时的一个诊断信息生成来解释这个问题。主要检查有:

* -fsanitize=address: AddressSanitizer, 一个内存错误检查器。
* -fsanitize=init-order: 使得 AddressSanitizer 检查动态初始化顺序问题。Implied by -fsanitize=address.
* -fsanitize=address-full: AddressSanitizer 具有所有下面所列的实验性质的特性。
* -fsanitize=integer: 使能检查未定义的或者可疑的整数行为。
* -fsanitize=thread: ThreadSanitizer, 一个数据竞争检测器。
* -fsanitize=memory: MemorySanitizer, 一个实验性质的未初始化读检查器。还不适合广泛使用。
* -fsanitize=undefined: 快速的，兼容的未定义行为监测器。使能未定义行为检测，具有很小的运行环境开销并对地址空间布局或ABI没有影响。这包含了下面所列的所有的检测而不单单是 unsigned-integer-overflow 。
* -fsanitize=undefined-trap: 这包含所有被 -fsantiize=undefined 包含的 sanitizers, 除了那些需要运行环境支持的。这一组的 sanitizers 通常与 -fsanitize-undefined-trap-on-error 标志结合使用, 将会导致陷阱被触发, 而不是到运行库的调用。这包含了下面所列的所有检查而不单单是 unsigned-integer-overflow 和 vptr 。

下面是更多的更细的可用检查:

* -fsanitize=alignment: 使用一个未对齐的指针或者引用。
* -fsanitize=bool: 加载一个既不是真也不是假的bool值。
* -fsanitize=bounds: 数组索引越界, 以防数组边界可以静态检测。
* -fsanitize=enum: 加载一个枚举类型的值，但是值不在那个枚举类型范围内。
* -fsanitize=float-cast-overflow: 转换到, 从, 或者浮点类型之间，其目标可能会溢出。
* -fsanitize=float-divide-by-zero: 浮点除零。
* -fsanitize=integer-divide-by-zero: 整数除零。
* -fsanitize=null: 使用一个空指针或者创建一个空引用。
* -fsanitize=object-size: 尝试使用优化器可以探测到不属于访问对象的字节。 对象的大小使用 \_\_builtin\_object\_size 检测, 并且结果可能会探测到多个问题在高层次的优化。
* -fsanitize=return: 在 C++ 中, 到达一个具有返回值类型函数的末尾而没有返回值。
* -fsanitize=shift: 移位操作符的移位大小超过了位宽或者小于零，或者左边是负值。 对于有符号数移位, 检查C中的有符号溢出，在C++中检查无符号溢出。
* -fsanitize=signed-integer-overflow: 有符号整数溢出, 包含所有通过 -ftrapv 添加的检查, 并且检查有符号除法溢出 (INT\_MIN / -1)。
* -fsanitize=unreachable: 如果控制流到达 \_\_builtin\_unreachable.
* -fsanitize=unsigned-integer-overflow: 无符号整数溢出。
* -fsanitize=vla-bound: 可变长数组边界值非正。
* -fsanitize=vptr: 使用一个vptr预示着具有错误动态类型的对象，或者它的生命长度还未开始或者已经结束。与 -fno-rtti 兼容。

AddressSanitizer 的实验性质的特性(还未准备好被广泛使用, 需要明确指定 -fsanitize=address):

* -fsanitize=use-after-return: 检查 use-after-return 错误 (在函数退出之后访问局部变量)。
* -fsanitize=use-after-scope: 检查 use-after-scope 错误 (在有效域范围外访问局部变量)。

MemorySanitizer 的额外特性(需要明确指定 -fsanitize=memory):

* -fsanitize-memory-track-origins: 使能MemorySanitizer中的原始跟踪。 加上一个额外的区域到 MemorySanitizer 报告指向堆或者栈分配未初始化位的来源。执行速度减慢 1.5x-2x.

为了链接到合适的运行库，连接时必须提供 -fsanitize= 参数。不可能在相同的程序中结合 -fsanitize=address 与 -fsanitize=thread 检查。

* -f[no-]address-sanitizer

过时的 -f[no-]sanitize=address 的同义词。

* -f[no-]thread-sanitizer

过时的 -f[no-]sanitize=thread 的同义词。

* -fcatch-undefined-behavior

过时的 -fsanitize=undefined 的同义词。

* -fno-assume-sane-operator-new

不要假定C++的新操作是健壮的。

这个选项告诉编译器不要假定C++全局新操作符将会一直返回一个指针而不是其他指针的别名，在函数返回的时候。

* -ftrap-function=[name]

指令代码生成器发出一个函数调用到指定的函数名称 \_\_builtin\_trap()。

LLVM代码生成器翻译 \_\_builtin\_trap() 到一个陷阱指令，如果被目标ISA支持的话。否则，内建被翻译为到abort的调用。如果这个选项被设置，代码生成器将会一直使用内建到一个指定函数而不论是否目标ISA具有陷阱指令。这个选项对于那些不能正确处理陷阱，或者需要定制行为的环境(例如深层嵌入)。

* -ftls-model=[model]

选择要使用的TLS模式。

有效值为: global-dynamic, local-dynamic, initial-exec 和 local-exec. 默认值为 global-dynamic。 编译器可能会使用一个不同的模式，如果选择的模式不被目标支持或者具有更具效率的模式可用。TLS模式可以通过每个变量使用tls\_model属性来重载。

## 控制调试信息大小

clang的调试信息类型生成可以设置为下表中的一种。如果有多个标志，使用最后一个。

* -g0

不生成任何调试信息(默认)。

* -gline-tables-only

只生成行号表。

这种调试信息允许使用函数名，文件名和行号进行观察栈跟踪(通过这类工具例如 gdb 或者 addr2line)。它不包含任何其他数据 (举个例子: 局部变量的描述或者函数参数)。

* -g

生成完整的调试信息。

## 注释解析选项

Clang解析Doxygen和non-Doxygen风格的文档注释并且把它们附加到合适的声明注记。默认的，只解析Doxygen风格注释并且忽略以 // and /\* 开头的普通注释。

* -fparse-all-comments

解析所有注释作为文档注释(包括以// and /\* 开头的普通注释)。

# C语言特性

clang中对于标准C的支持是全部特性的，除了C99浮点附注。

## clang支持的扩展

见于 Clang 语言扩展

## 各个标准模式之间的差异

clang支持-std选项，它改变clang使用的语言模式。支持的C语言模式有 c89, gnu89, c94, c99, gnu99 和 其他这些模式的别名。如果没有指定-std选项，默认的是gnu99模式。

所有c\*和gnu\*模式之间的区别:

* c\*模式定义了“STRICT\_ANSI”。
* 指定目标定义定义没有前置下划线，像”linux”,被定义在gnu\* 模式中。
* gnu\* 模式中，Trigaphs模式是关闭的；它们可以被-trigraphs选项使能。
* 在gnu\* 模式中，解析器识别 “asm” 和 “typeof” 作为关键字；变体 “asm” and “typeof” 在所有模式中均被识别。
* 在一些平台上的 gnu\* 模式中，Apple 的 “blocks” 扩展被识别；也可以通过 “-fblocks” 选项在任何模式中使能。
* 按照标准，数组是VLA，但是可以通过前端被常量折叠作为固定大小数组。这发生在例如 “int X[(1, 2)];”，技术上是VLA。c\*模式严格顺从并把它们作为VLA。

\*89 and \*99 模式之间的区别:

* \*99模式默认实现了C99中指定的“inline”，而 \*89 模式实现GNU 版本。这对于单个函数可以使用 gnu\_inline 属性重载。
* Digraphs在c89模式中不被识别。
* “for”, “if”, “switch”, “while”, 或者 “do” 语句中定义的名称范围不同。(例如:“if ((struct x {int x;}\*)0) {}”.)
* \*89中 STDC\_VERSION 未被定义。
* c89中“inline”不被识别为关键字。
* \*89 模式中不识别“restrict”作为关键字。
* \*99 模式中可以使用逗号在数字常量表达式中。
* \*89 模式中，不是左值的数组未被不明显的引进
* 一些警告不同。

c94 模式和 c89 模式是相同的，除了digraphs在c94模式中被使能 (FIXME: 并且 STDC\_VERSION 应当被定义!).

## 尚未实现的GCC扩展

clang尝试与gcc尽可能的兼容，但是一些gcc扩展还未实现:

* clang不支持#pragma weak(bug 3679)。由于在bug中描述的使用，这很可能在某刻被实现，至少是部分上。
* clang不支持十进制浮点类型 (\_Decimal32 and 同类)或者定点类型(\_Fract and 同类);还没有人表现出对这些特性的兴趣，所以还很难说它们什么时候会被实现。
* clang不支持函数嵌套；这是一个不经常使用的复杂特性，所以它可能不会在近期实现。在C++11中，可以通过赋值lambda函数给局部变量来仿真，例如:

|  |
| --- |
| auto const local\_function = [&]($int parameter) {  // Do something  };  ...  local\_function(1); |

* clang不支持全局寄存器变量；这不太可能很快实现，因为它需要附加的LLVM后端支持。
* clang不支持可变数组成员静态初始化。这是一个不常用的扩展，但是可以根据用户需求实现。
* clang不支持builtin\_va\_arg\_pack/builtin\_va\_arg\_pack\_len。这个很少使用，但是在一些有意思的地方，例如glibc头文件，所以可能会根据用户需求实现。注意因为clang假装像GCC 4.2，并且这个扩展是4.3引进的，glibc头文件将不会在此刻尝试使用clang的这个扩展。
* clang不支持gcc的函数参数向前声明扩展；这个还没有在实际代码中使用，所以它可能永远也不会被实现。

这个不是完全列表；如果你在这个表中找到丢失了一个不支持的扩展；请发邮件到cfe-dev。这张表暂时不包含C++；见C++ 语言特性。同样的，这张表也不包含大部分实现的扩展的bug；请参见bug tracker找已知bug(FIXME:有一节是关于bug报告的吗？)

## 故意不支持的GCC扩展

* clang不支持结构体具有变长数组的gcc扩展。这个有好几个理由:一，它的实现比较微妙，二，这个扩展是完全没有文档的，三，这个扩展很少被使用。注意clang不支持可变数组成员(出于一个结构体末尾的具有0长或者未指定长度的数组)。
* clang没有一个等价于gcc的”fold”;这意味着clang不接受一些gcc可能在上下文中接受的概念，那里就需要一个常量表达式。像”x-x”,x是变量。
* clang不支持\_\_builtin\_apply和其同类；这个扩展着实难以可信的实现。

## Microsoft扩展

clang对于Microsoft Visual C++具有一些实验性的支持；使用命令行选项 -fms-extensions 来开启。这对于 Windows 目标是默认的。注意支持是不完全的；使能 Microsoft 扩展将会默默的丢掉特定构建(包含\_\_declspec和Microsoft-type 汇编语句)。

clang具有一个 -fms-compatibility 标志使得clang接受足够的非有效C++来解析大多数Microsoft头文件。这个标志对于 Windows 默认是开启的。

-fdelayed-template-parsing 让clang延迟所有模板实例化直到一个翻译单元结尾。这个对于 Windows 目标默认是开启的。

* clang允许使用-fmsc-version=设置 \_MSC\_VER。默认为1300，和 Visual C/C++ 2003相同。可以使用任何数字并且可以很大的影响clang可以编译的Windows SDK和c++stdlib 头文件。这个选项将会被移除，当clang支持MS扩展的头文件全集。
* clang不支持Microsoft 的匿名记录成员可以使用用户定义类型被声明的扩展。
* clang支持 Microsoft “#pragma pack” 特性来控制记录布局。GCC 也包含对此项特性的支持，然而 MSVC 和 GCC 是不兼容的， clang 跟随 the MSVC 定义.
* clang对于 Windows 目标默认到C++11。

# C++、Objective-C、Objective-C++语言特性

C++ 语言特性

clang完全实现了所有C++98标准，除了导出模板(在C++11中被移除)，并且很多C++11特性也被实现。

## C++语言

控制实现限制

-fbracket-depth=N

设置圆括号、方括号和大括号的嵌套限制为N。默认是256。

-fconstexpr-depth=N

设置常量表达式函数调用的递归层数为N。默认是512。

-ftemplate-depth=N

设置模板实例化递归嵌套层数为N。默认是1024。

## Objective-C语言特性

略

## Objective-C++特性

略

# 特定目标特性和限制

## CPU架构特性和限制

X86

对于 X86 的支持(包含32位和64位)在Darwin上被认为是稳定的，Linux，FreeBSD额Dragonfly BSD: 经过测试可以正确编译很多大的C，C++，Objective-C和Objective-C++基本代码。

ARM

对于ARM的支持(特别对于ARMv6 和 ARMv7) 在Darwin上被认为是稳定的(iOS): 经过测试可以正确编译许多大的C，C++，Objective-C和Objective-C++基本代码。Clang只支持有限数目的ARM架构。例如它还不完全支持ARMv5。

其他平台

clang目前包含一些对PPC和Sparc的支持；然而，仍然没有代码生成的重要片段，并且它们并没有经过显著性检验。

clang包含了对MSP430嵌入式处理器的有限支持，但是clang支持和LLVM后端支持还都是出于重度实验阶段的。

此刻其他平台是完全不支持的。在一个新平台上添加解析和语义分析需要的最小支持是非常容易的；见于 clang 源代码 lib/Basic/Targets.cpp 中。这一层次的支持对于把简单程序转换到 LLVM IR 是足够的。此刻转换到 LLVM IR 的适当支持需要添加代码到lib/CodeGen/CGCall.cpp; 然而这可能很快就会变化。生成汇编需要一个合适的LLVM后端。

## 操作系统特性和限制

Darwin(Mac OS/X)

没有

Windows

Cygming的支持尚出于实验阶段。

参见Microsoft Extensions。

Cygwin

在Cygwin-1.7下工作。

MinGW32

Clang在一些mingw32发布上工作。Clang假定的目录如下：

c:/mingw/include

c:/mingw/lib

c:/mingw/lib/gcc/mingw32/4.[3-5].0/include/c++

在MSYS上，一些测试会失败。

MinGW-w64

对于32位(i686-w64-mingw32)和64位(x86\_64-w64-mingw32)，Clang作如下假定:

GCC versions 4.5.0 to 4.5.3, 4.6.0 to 4.6.2, or 4.7.0 (对于 C++ 头文件搜索路径)

some\_directory/bin/gcc.exe

some\_directory/bin/clang.exe

some\_directory/bin/clang++.exe

some\_directory/bin/../include/c++/GCC\_version

some\_directory/bin/../include/c++/GCC\_version/x86\_64-w64-mingw32

some\_directory/bin/../include/c++/GCC\_version/i686-w64-mingw32

some\_directory/bin/../include/c++/GCC\_version/backward

some\_directory/bin/../x86\_64-w64-mingw32/include

some\_directory/bin/../i686-w64-mingw32/include

some\_directory/bin/../include

对于你可以在MinGW-w64官方网站上可以找到的任何工具链，此目录结构是标准的。

Clang预期PATH中为i686-w64-mingw32(或者x86\_64-w64-mingw32)编译的GCC可执行文件”gcc.exe”

在x86\_64-w64-mingw32上，一些测试可能会失败。

# clang-cl

clang-cl 是一个Clang驱动器的另一个命令行接口的选择，被设计用来兼容Visual C++ 的编译器，cl.exe。

为了使clang-cl从命令行运行的时候能够找到系统头文件，库和连接器，它应当在一个Visual Studio 本地工具命令提示符或者设置了环境变量常规的命令提示符，例如vcvars32.bat

clang-cl 也可以在Visual Studio中使用，通过使用LLVM平台工具集。

## 命令行选项

为了兼容cl.exe，clang-cl 支持大多数相同的命令行选项。这些选项可以使用/或者-开始。它也支持一些Clang的核心选项，例如｀-w｀选项。

clang-cl识别的选项，但是目前还没有被支持的，会被忽略并显示一个警告。例如：

clang-cl.exe: warning: argument unused during compilation: '/Zi'

可以使用-Qunused-arguments选项来关闭为使用的参数警告。

clang-cl不认识的选项将会引发错误。如果它们被冠以/，它们将会被误认为是一个文件名：

clang-cl.exe: error: no such file or directory: '/foobar'

如果有发现cl.exe有的，clang-cl却不理解的命令，请提交一个bug。

执行clang-cl /?来查看支持的选项列表：

/? | Display available options

/c | Compile only

/D <macro[=value]> Define macro

/fallback Fall back to cl.exe if clang-cl fails to compile

/Fe<file or directory> Set output executable file or directory (ends in / or \)

/FI<value> Include file before parsing

/Fo<file or directory> Set output object file, or directory (ends in / or \)

/GF- Disable string pooling

/GR- Disable RTTI

/GR Enable RTTI

/help Display available options

/I <dir> Add directory to include search path

/J Make char type unsigned

/LDd Create debug DLL

/LD Create DLL

/link <options> Forward options to the linker

/MDd Use DLL debug run-time

/MD Use DLL run-time

/MTd Use static debug run-time

/MT Use static run-time

/Ob0 Disable inlining

/Od Disable optimization

/Oi- Disable use of builtin functions

/Oi Enable use of builtin functions

/Os Optimize for size

/Ot Optimize for speed

/Ox Maximum optimization

/Oy- Disable frame pointer omission

/Oy Enable frame pointer omission

/O<n> Optimization level

/P Only run the preprocessor

/showIncludes Print info about included files to stderr

/TC Treat all source files as C

/Tc <filename> Specify a C source file

/TP Treat all source files as C++

/Tp <filename> Specify a C++ source file

/U <macro> Undefine macro

/W0 Disable all warnings

/W1 Enable -Wall

/W2 Enable -Wall

/W3 Enable -Wall

/W4 Enable -Wall

/Wall Enable -Wall

/WX- Do not treat warnings as errors

/WX Treat warnings as errors

/w Disable all warnings

/Zs Syntax-check only

### /fallback 选项

当clang-cl带有/fallback选项运行的时候，它首先会尝试编译文件自身。对于它编译失败的文件，它将会回退并调用cl.exe来编译。

这个选项意在被临时用作一种方式来编译项目，当clang-cl不能成功编译所有文件的时候。clang-cl编译一个文件失败或者可能是引文它无法为一些C++特性产生代码，或者因为它无法解析一些Microsoft语言扩展。

# 附录