**如何使用gflags**

**介绍以及与其他命令行标志库的比较**

**命令行标志**是用户在运行可执行文件时在命令行上指定的标志。在命令中

|  |
| --- |
| C++  fgrep -l -f /var/tmp/foo johannes brahms |

-l和-f /var/tmp/foo是两个命令行标志。（johannes和brahms不以破折号开头，是**命令行参数**。）

通常，应用程序会列出允许用户传入的标志以及它们所采用的参数——在此示例中， -l不采用任何参数，而是-f采用字符串（特别是文件名）作为参数。用户可以使用库来帮助解析命令行并将标志存储在某个数据结构中。

Gflags 是 Google 内部使用的命令行标志库，它与其他库（例如getopt()）的不同之处在于，标志定义可以分散在源代码中，而不只是列在一个地方（例如main()）。实际上，这意味着单个源代码文件将定义和使用对该文件有意义的标志。任何链接到该文件的应用程序都将获得这些标志，并且 gflags 库将自动适当地处理该标志。

这种技术大大提高了灵活性，并且易于代码重用。但是，存在两个文件定义相同标志的风险，然后在链接在一起时会出错。

本文档的其余部分介绍了如何使用 commandlineflag 库。它是一个 C++ 库，因此示例也使用 C++ 编写。不过，有一个具有相同功能的 Python 端口，因此本讨论将直接转换为 Python。

**下载和安装**

gflags 库可以从[GitHub](https://github.com/gflags/gflags)下载。你可以使用以下命令克隆该项目：

|  |
| --- |
| C++ git clone https://github.com/gflags/gflags.git |

[INSTALL](https://github.com/gflags/gflags/blob/master/INSTALL.md)文件中提供了构建和安装说明 。 gflags 包的安装包括流行构建系统（如[pkg-config](https://www.freedesktop.org/wiki/Software/pkg-config/)、 [CMake](https://gflags.github.io/gflags/#cmake)和[Bazel ）](https://gflags.github.io/gflags/#bazel)的配置文件。

**使用 CMake 声明对 gflags 的依赖**

[在使用CMake](http://www.cmake.org/)作为其构建系统的项目中使用 gflags非常简单。您可以要求外部安装 gflags 包并使用 CMake 的 find\_package 命令找到它，或者将 gflags 项目作为子树或子模块包含在项目的源代码树中，并使用 CMake 的 add\_subdirectory 命令添加目录。

要使用外部 gflags 安装，请将以下 CMake 代码添加到您的CMakeLists.txt文件中。

查找 gflags 安装。如果 <prefix> 是非标准位置，则必须将gflags\_DIR变量设置为包含 gflags-config.cmake 文件的 <prefix>/lib/cmake/gflags 目录。否则，CMake 应该会自动找到 gflags 安装。

|  |
| --- |
| C++ find\_package(gflags REQUIRED) |

要请求链接特定的导入 gflags 库目标，请使用find\_package 命令的COMPONENTS选项。例如，要强制使用单线程静态库，请使用命令

|  |
| --- |
| C++ find\_package(gflags COMPONENTS nothreads\_static) |

请注意，当安装的 gflags 包不包含请求的库时，这将引发致命错误。因此，如果必须使用特定库，建议仅指定要查找的特定组件。否则，gflags-config.cmake 模块将为您选择合适且可用的库。默认情况下，如果可用，则选择具有共享链接的多线程 gflags 库。

当 gflags 项目的源代码树作为子树或子模块包含在项目的“gflags”目录中时，请将上面的 find\_package 命令替换为add\_subdirectory(gflags)。请参阅文件顶部gflags/CMakeLists.txt 以获取可用的 CMake 变量列表，这些变量可以在此命令之前设置以配置 gflags 库的构建。默认构建设置是单线程静态库的构建，它不需要安装任何 gflags 子项目产品。

最后，添加可执行构建目标，该目标使用 gflags 来解析命令参数，并依赖于导入的 gflags 库目标：

|  |
| --- |
| C++ add\_executable(foo main.cc) target\_link\_libraries(foo gflags::gflags) |

**DEFINE：在程序中定义标志**

定义标志很容易：只需使用您想要的标志类型的适当宏，如在gflags/gflags.h底部所定义 。这是一个示例文件， foo.cc：

|  |
| --- |
| C++  #include <gflags/gflags.h>   DEFINE\_bool(big\_menu, true, "Include 'advanced' options in the menu listing");  DEFINE\_string(languages, "english,french,german",  "comma-separated list of languages to offer in the 'lang' menu"); |

DEFINE\_bool定义一个布尔标志。支持的类型如下：

* DEFINE\_bool：布尔值
* DEFINE\_int32：32位整数
* DEFINE\_int64：64位整数
* DEFINE\_uint64：无符号 64 位整数
* DEFINE\_double： 双精度浮点数
* DEFINE\_string：C++ 字符串

请注意，没有像列表这样的“复杂”类型：我们示例中的“languages”标志是字符串列表，但定义为“string”类型，而不是“list\_of\_string”或类似类型。这是设计使然。我们宁愿只使用简单类型的标志，并允许使用复杂、任意的解析例程来解析它们，而不是尝试将逻辑放入标志库中。

所有 DEFINE 宏都采用相同的三个参数：标志的名称、其默认值以及描述其用途的“帮助”字符串。当用户运行带有[--help](https://gflags.github.io/gflags/#special)标志的应用程序时，会显示“帮助”[字符串](https://gflags.github.io/gflags/#special)。

您可以在可执行文件中的任何源代码文件中定义标志。每个标志只能定义一次！如果您想在多个源文件中访问标志，请在一个文件中定义它，并在其他文件中[声明](https://gflags.github.io/gflags/#declare)它。甚至更好，在foo.cc中定义它并在foo.h中声明它；然后每个人都#includes foo.h可以使用该标志。

在库中定义标志（而不是在 main() 中）非常有效，但也有一些代价。其中一个代价是，库的标志可能没有好的默认值，例如，如果标志包含的文件名在某些环境中可能不存在。为了缓解此类问题，您可以使用[标志验证器](https://gflags.github.io/gflags/#validate)来确保及时通知（以崩溃的形式）无效的标志值。

请注意，虽然此库中的大多数函数都是在 google命名空间中定义的，DEFINE\_foo但（以及 DECLARE\_foo下面[的](https://gflags.github.io/gflags/#declare)）应始终位于全局命名空间中。

**访问 Flag**

所有定义的标志都可作为普通变量供程序使用，只需FLAGS\_添加前缀即可。在上面的示例中，宏定义了两个变量，FLAGS\_big\_menu （一个布尔值）和FLAGS\_languages（一个 C++ 字符串）。

您可以像读取和写入任何其他变量一样读取和写入该标志：

|  |
| --- |
| C++ if (FLAGS\_consider\_made\_up\_languages)  FLAGS\_languages += ",klingon"; // implied by --consider\_made\_up\_languages if (FLAGS\_languages.find("finnish") != string::npos)  HandleFinnish(); |

您还可以通过gflags.h中的特殊函数获取和设置标志值 。不过，这是一种比较罕见的用例。

**DECLARE：在不同的文件中使用标志**

仅当文件顶部带有 DEFINE-ed 标志时，才能按照上一节的方式访问标志。如果不是，您将收到“未知变量”错误。

当您想要使用另一个文件中定义的标志时，可以使用该DECLARE\_type宏。例如，如果我正在编写bar.cc但想要访问 big\_menu 标志，我会将其放在顶部附近bar.cc：

|  |
| --- |
| C++ DECLARE\_bool(big\_menu); |

从功能上讲，这相当于说extern FLAGS\_big\_menu。

请注意，在这种情况下，这样的 extern 声明会在您的文件和定义big\_menu 标志的文件之间引入依赖关系：foo.cc。这种隐式依赖关系在大型项目中可能难以管理。因此，我们建议遵循以下准则：

|  |
| --- |
| 如果您在foo.cc中 DEFINE 一个标志，则根本不要 DECLARE 它，只在紧密相关的测试中 DECLARE 它，或者只在foo.h中 DECLARE 它。 |

如果该标志仅在foo.cc中需要，而其他任何文件中都不需要，则应选择不 DECLARE 路径。如果您想在相关测试文件中修改标志的值以查看它是否按预期运行，请在foo\_test.cc文件中 DECLARE 它 。

如果该标志确实跨越多个文件，则在关联 .h文件中声明它，并让其他文件访问该标志。这 #include将 明确两个文件之间的依赖关系。这会导致该标志成为全局变量。.h#include

**RegisterFlagValidator：对标志值进行健全性检查**

在 DEFINE 标志之后，您可以选择为该标志注册一个验证器函数。如果这样做，在从命令行解析标志之后，每当通过调用SetCommandLineOption()更改其值时 ，都会使用新值作为参数调用验证器函数。如果标志值有效，则验证器函数应返回“true”，否则返回 false。如果函数为标志的新设置返回 false，则标志将保留其当前值。如果它为默认值返回 false，则 ParseCommandLineFlags 将失效。

以下是此功能的使用示例：

|  |
| --- |
| C++ static bool ValidatePort(const char\* flagname, int32 value) {  if (value > 0 && value < 32768) // value is ok  return true;  printf("Invalid value for --%s: %d\n", flagname, (int)value);  return false; } DEFINE\_int32(port, 0, "What port to listen on"); DEFINE\_validator(port, &ValidatePort); |

通过在全局初始化时（DEFINE\_int32 之后）进行注册，我们确保注册发生在开始时解析命令行之前main()。

上面使用的DEFINE\_validator宏调用 RegisterFlagValidator()函数，如果注册成功则返回 true。如果注册失败，则返回 false，因为 a. 第一个参数不引用命令行标志，或者 b. 已经为该标志注册了不同的验证器。返回值可用作名为 的全局静态布尔变量 <flag>\_validator\_registered。

**总结：如何设置标志**

最后一部分是告诉可执行文件处理命令行标志，并FLAGS\_\*根据命令行上看到的内容将变量设置为适当的非默认值。这相当于getopt 库中的getopt()调用，但使用开销要小得多。事实上，它只是一个函数调用：

gflags::ParseCommandLineFlags（＆argc，＆argv，true）；

通常，此代码位于 的开头main()。 argc和argv传入与main()的完全相同 。此例程可能会修改它们，这就是为什么传入指向它们的指针的原因。

最后一个参数称为“remove\_flags”。如果为真，则 ParseCommandLineFlags从中删除标志及其参数argv，并进行argc 适当修改。在这种情况下，在函数调用之后， argv将仅保留命令行参数，而不保留命令行标志。

另一方面，如果remove\_flags为假，则将 ParseCommandLineFlags保持 argc 不变，但将重新排列 argv 中的参数，以便所有标志都位于开头。例如，如果输入是"/bin/foo" "arg1" "-q" "arg2"（合法但很奇怪），函数将重新排列 argv以便读取"/bin/foo", "-q", "arg1", "arg2"。在这种情况下，ParseCommandLineFlags 返回保存第一个命令行参数的 argv 中的索引：即最后一个标志之后的索引。（在此示例中，它将返回 2，因为argv[2]指向arg1。）

无论哪种情况，FLAGS\_\*变量都会根据[命令行传递的](https://gflags.github.io/gflags/#commandline)内容进行修改。

**在命令行上设置标志**

将某个东西设为标志而不是通过编译时常量的原因在于，这样用户就可以在命令行上指定非默认值。以下是他们可能为foo.cc链接的应用程序执行的操作：

|  |
| --- |
| C++ app\_containing\_foo --nobig\_menu -languages="chinese,japanese,korean" ... |

当ParseCommandLineFlags运行时， 这将设置FLAGS\_big\_menu = false;和 。FLAGS\_languages = "chinese,japanese,korean"。

请注意将布尔标志设置为 false 的非典型语法：在其名称前面加上“no”。指定标志的方式有相当大的灵活性。以下是指定“languages”标志的所有方法的示例：

* app\_containing\_foo --languages="chinese,japanese,korean"
* app\_containing\_foo -languages="chinese,japanese,korean"
* app\_containing\_foo --languages "chinese,japanese,korean"
* app\_containing\_foo -languages "chinese,japanese,korean"

对于布尔标志，可能性略有不同：

* app\_containing\_foo --big\_menu
* app\_containing\_foo --nobig\_menu
* app\_containing\_foo --big\_menu=true
* app\_containing\_foo --big\_menu=false

（以及所有这些的单破折号变体）。

尽管具有这种灵活性，但我们建议仅使用一种形式： --variable=value用于非布尔标志和 --variable/--novariable用于布尔标志。这种一致性将使您的代码更具可读性，也是某些特殊用例（如[flagfiles ）](https://gflags.github.io/gflags/#flagfiles)所需的格式。

在命令行上指定未在可执行文件中的某个位置定义的标志是致命错误。如果您出于某种原因需要该功能 - 比如您想对多个可执行文件使用同一组标志，但并非所有可执行文件都定义列表中的每个标志 - 您可以指定[--undefok](https://gflags.github.io/gflags/#special)以抑制错误。

与 getopt() 一样，它--本身将终止标志处理。因此foo -f1 1 -- -f2 2，在 中，f1被视为标志，但实际上-f2不是。

如果多次指定一个标志，则只使用最后一个指定的标志；其他的将被忽略。

请注意，标志没有单字母同义词，就像在 getopt 库中一样，我们也不允许在单个破折号后面“组合”标志，如ls -la。

**更改默认标志值**

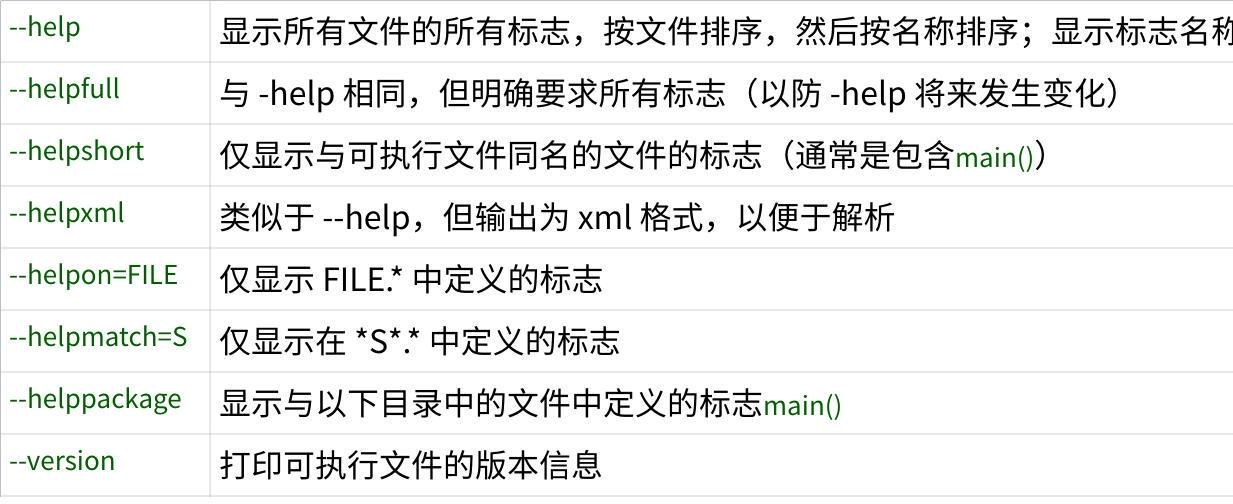
有时，库中定义了一个标志，而您想在一个应用程序中更改其默认值，而不想在其他应用程序中更改。这样做很简单：只需main()在调用之前在 中为标志分配一个新值ParseCommandLineFlags()：

|  |
| --- |
| C++ DECLARE\_bool(lib\_verbose); // mylib has a lib\_verbose flag, default is false int main(int argc, char\*\* argv) {  FLAGS\_lib\_verbose = true; // in my app, I want a verbose lib by default  ParseCommandLineFlags(...); } |

对于此应用程序，用户仍然可以在命令行上设置标志值，但如果不这样做，标志的值将默认为 true。

**特殊标志**

commandlineflags 模块本身定义了一些标志，并且可供所有使用 commandlineflags 的应用程序使用。这些标志分为三类。第一类是“报告”标志，当发现这些标志时，应用程序会打印一些有关自身的信息并退出。



**点击图片可查看完整电子表格**

第二个是影响其他标志如何解析的标志。



**点击图片可查看完整电子表格**

第三是“递归”标志，它导致设置其他标志值：--fromenv，，--tryfromenv。 --flagfile下面将更详细地描述这些。

--fromenv

--fromenv=foo,barfoo表示从环境中读取和标志的值 bar。与此标志配合使用，您必须通过类似以下两行之一的行实际设置环境中的值：

|  |
| --- |
| C++ export FLAGS\_foo=xxx; export FLAGS\_bar=yyy # sh setenv FLAGS\_foo xxx; setenv FLAGS\_bar yyy # tcsh |

这相当于在命令行上--foo=xxx指定 。--bar=yyy

--fromenv=foo请注意，如果 foo在应用程序中某处未定义，则会出现致命错误。（尽管您可以通过 抑制此错误--undefok=foo，就像任何其他标志一样。）

--fromenv=foo如果 FLAGS\_foo在环境中实际上没有定义，那么这也是一个致命错误。

--tryfromenv

--tryfromenv与--fromenv完全相同，只不过如果在环境中未实际定义， 则**不会**出现致命错误 。相反，在这种情况下，只需保留其在应用程序中指定的默认值即可。--tryfromenv=fooFLAGS\_fooFLAGS\_foo

--tryfromenv=foo请注意，如果 foo在应用程序中的某处没有定义，那么仍然是一个错误。

--flagfile

--flagfile=f告诉命令行标志模块读取文件f，并运行该文件中找到的所有标志分配，就好像这些标志是在命令行上指定的一样。

在其最简单的形式中，f应该只是标志分配列表，每行一个。与命令行不同，标志文件需要*用*等号将标志名称与其参数分开。以下是标志文件的示例/tmp/myflags：

--nobig\_菜单  
--languages=英语,法语

有了这个标志文件，以下两行是等效的：

./myapp --foo --nobig\_menus --languages=english,french --bar  
 ./myapp --foo --flagfile=/tmp/myflags --bar

请注意，许多错误在标志文件中被默默抑制。特别是，无法识别的标志名称会被默默忽略，缺少必需值的标志也是如此（例如，标志文件中只显示 --languages）。

标志文件的一般格式比上面简单、常见的情况要复杂一些。它是：文件名序列，每行一个，后面跟着标志序列，每行一个，重复任意多次。标志文件中的文件名可以使用通配符（\*和?），并且只有当前可执行文件的名称与其中一个文件名匹配时，才会处理位于文件名序列之后的标志序列。可以使用标志序列而不是文件名序列来启动标志文件；如果存在这样的标志序列，则这些标志将应用于当前可执行文件，无论它是什么。

以 开头的行将#被视为注释而被忽略。标志文件中的前导空格和空行也会被忽略。

标志文件可以使用--flagfile 标志来包含另一个标志文件。

标志始终按预期顺序处理。也就是说，处理从检查命令行上直接指定的标志开始。如果指定了标志文件，则处理其内容，然后继续处理命令行中的其余标志。

**API**

除了FLAGS\_foo直接访问之外，还可以通过 API 以编程方式访问标志。还可以访问有关标志的信息，例如其默认值和帮助字符串。AFlagSaver可以轻松修改标志，然后稍后自动撤消修改。最后，还有一些不相关但有用的例程可以轻松访问argv主程序之外的部分，包括程序名称（argv[0]）。

有关这些例程以及其他有用的辅助方法（如gflags::SetUsageMessage()和 gflags::SetVersionString）的更多信息，请参阅gflags.h。

**其他注意事项**

如果你的应用程序有这样的代码：

|  |
| --- |
| C++ #define STRIP\_FLAG\_HELP 1 // this must go before the #include! #include <gflags/gflags.h> |

我们将从编译的源代码中删除帮助消息。这可以在一定程度上减小生成的二进制文件的大小，并且出于安全原因也可能有用。