Nuitka 全命令

Option：

--help

显示此帮助信息并退出

--version

显示版本信息和提交 `bug` 报告所需的重要细节，然后退出。默认关闭。

--module

创建一个扩展模块可执行文件，而不是一个程序。默认关闭。

--standalone

启用独立模式输出。这允许您将创建的二进制文件转移到其他机器上，而不需要现有的 `Python` 安装。这也意味着它会变大。它隐含了这些选项： `--follow-imports` 和 `--python-flag=no_site` 。默认关闭。

--onefile

在独立模式的基础上，启用 `onefile` 模式。这意味着创建并使用的不是一个文件夹，而是一个压缩的可执行文件。默认关闭。

--python-debug

是否使用 `debug` 版本。默认使用您用来运行 `Nuitka` 的版本，很可能是非 `debug` 版本。

--python-flag=FLAG

要使用的 Python 标志。默认是您用来运行 Nuitka 的标志，这强制执行一个特定的模式。这些选项也存在于标准 Python 可执行文件中。当前支持的有："-S"（alias "no_site"），"static_hashes"（不使用散列随机化），"no_warnings"（不给出 Python 运行时警告），"-O"（alias "no_asserts"），"no_docstrings"（不使用文档字符串），"-u"（alias "unbuffered"）和"-m"。默认为空。

--python-for-scons=PATH

如果使用 Python3.3 或 Python3.4，提供一个 Python 二进制文件的路径供 Scons 使用。否则 Nuitka 可以使用您用来运行 Nuitka 的二进制文件或从 Windows 注册表中获取一个 Python 安装。在 Windows 上需要 Python 3.5 或更高版本。在非 Windows 上，Python 2.6 或 2.7 也可以。

控制结果中包含的模块和包：

--include-package=PACKAGE

包含一个完整的包。提供为一个 Python 命名空间，例如 "`some_package.sub_package`"，然后 `Nuitka` 会找到它，并将其及其下面找到的所有模块包含在它创建的二进制文件或扩展模块中，并使代码可以导入它。为了避免不需要的子包，例如测试，您可以这样做"`--nofollow-import-to=*.tests`"。默认为空。

使用这个选项后，所有在 `PACKAGE` 中被列出来的包，例如 `--include-package=av,PySide6,faster_whisper,transformers,ctranslate2` ，所有这些包都 **将会被完整编译**，这将 **增加编译时间**，同时可能解决一部分编译后出现模块或者子模块无法找到的错误，例如： `ModelFoundError` 、 `importError` 、 `No model Named XXXX` 。

--include-module=MODULE

包含一个单一的模块。提供为一个 `Python` 命名空间，例如 "`some_package.some_module`"，然后 Nuitka 会找到它，并将其包含在它创建的二进制文件或扩展模块中，并使代码可以导入它。默认为空。

--include-plugin-directory=MODULE/PACKAGE

在该目录中找到的代码也将被包含，就像它们每个都作为主文件给出一样。覆盖所有其他包含选项。您应该更喜欢按名称而不是文件名的其他包含选项，因为它们通过在“sys.path”中来查找东西。这个选项仅用于非常特殊的使用案例。可以多次给定。默认为空。

--include-plugin-files=PATTERN

包含匹配 PATTERN 的文件。覆盖所有其他 follow 选项。可以多次给出。默认为空。

--prefer-source-code

对于已经编译的扩展模块，如果同时存在源文件和扩展模块，通常使用扩展模块，但从可用的源代码编译模块性能最佳。如果不需要，使用 `--no-prefer-source-code` 来禁用关于此的警告。默认关闭。

对导入模块的跟踪控制：

--follow-imports

递归到所有导入的模块。在 `--standalone` 模式下默认开启，否则关闭。

--follow-import-to=MODULE/PACKAGE

如果使用了这个模块，或者如果是一个包，跟踪到整个包。可以多次给出。默认为空。

--nofollow-import-to=MODULE/PACKAGE

即使使用了也不跟踪这个模块，或者如果是一个包，在任何情况下也不跟踪整个包，覆盖所有其他选项。可以多次给出。默认为空。

--nofollow-imports

完全不递归到任何导入的模块，覆盖所有其他包含选项，不能用于独立模式。默认关闭。

--follow-stdlib

递归到标准库中导入的模块。这将极大地增加编译时间，目前也没有很好地测试过，有时也不起作用。默认关闭。

Onefile 选项：

--onefile-tempdir-spec=ONEFILE_TEMPDIR_SPEC

在 `onefile` 模式下解压缩到此文件夹。默认 `'%TEMP%/onefile_%PID%_%TIME%`，即用户临时目录，由于不是静态的，退出时会被删除。例如，使用类似' `CACHE_DIR%/%COMPANY%/%PRODUCT%/%VERSION%` 的字符串，这是一个很好的静态缓存路径，因此不会被删除。

--onefile-child-grace-time=GRACE_TIME_MS

当停止子进程时，例如由于 `CTRL-C` 或关闭等， `Python` 代码会收到一个 `KeyboardInterrupt`，它可以处理，例如刷新数据。这是在硬关闭子进程前的宽限时间（单位毫秒）。默认值为 5000。

数据文件：

--include-package-data=PACKAGE

包含给定包名的数据文件。 `DLL` 和扩展模块不是数据文件，不会像这样包含。可以像下面指示的那样使用通配符。默认情况下不包含包的数据文件，但包配置可以做到这一点。这只会包含非 `DLL`、非扩展模块的实际数据文件。冒号后可选地可以给出一个文件名模式，仅选择匹配的文件。例子：

- `--include-package-data=package_name` （所有文件）
- `--include-package-data=package_name=*.txt` （仅某种类型）

- `--include-package-data=package_name=some_filename.dat`（具体文件）默认为空。

--include-data-files=DESC

通过发行版中的文件名包含数据文件。有许多允许的形式。

- 使用 `--include-data-files=/path/to/file/*.txt=folder_name/some.txt` 它会复制一个单一文件，如果是多个会抱怨。
- 使用 `--include-data-files=/path/to/files/*.txt=folder_name/` 它会把所有匹配的文件放入那个文件夹。
- 要进行递归复制，有一个带 3 个值的表单，即 `--include-data-files=/path/to/scan=folder_name=**/*.txt` 这会保留目录结构。默认为空。

--include-data-dir=DIRECTORY

从发行版中包含完整的目录中的数据文件。这是递归的。如果您想要非递归包含，请使用带通配符的 `--include-data-files` 。一个例子是 `--include-data-dir=/path/some_dir=data/some_dir` ，用于简单复制整个目录。所有文件都将被复制，如果您想排除文件，您需要事先将其删除，或使用 `--noinclude-data-files` 选项将其删除。默认为空。

--noinclude-data-files=PATTERN

不要包含匹配给定的文件名模式的数据文件。这是针对目标文件名，而不是源路径。因此，要忽略来自 `package_name` 的数据文件的文件模式，应匹配为 `package_name/*.txt` 。或者对于整个目录，简单地使用 `package_name` 。默认为空。

--list-package-data=LIST_PACKAGE_DATA

输出为给定包名找到的数据文件。默认不执行。

DLL 文件：

--noinclude-dlls=PATTERN

不要包含匹配给定的文件名模式的 `DLL` 文件。这是针对目标文件名，而不是源路径。因此，要忽略包含在包 `package_name` 中的某个 `DLL` **"someDLL"**，应匹配为 `package_name/someDLL.*` 。默认为空。

--list-package-dlls=LIST_PACKAGE_DLLS

输出为给定包名找到的 DLL。默认不执行。

控制 Nuitka 给出的警告：

--warn-implicit-exceptions

启用对编译时检测到的隐式异常的警告。

--warn-unusual-code

启用对编译时检测到的不寻常代码的警告。

--assume-yes-for-downloads

允许Nuitka在必要时下载外部代码，例如dependency walker， ccache，甚至在Windows上是gcc。要禁用，从空设备重定向输入，例如"</dev/null" 或 "<NUL： "。默认是提示。

--nowarn-mnemonic=MNEMONIC

禁用给定助记符的警告。这些是为了确保您知道某些主题，通常指向 Nuitka 网站。助记符是 URL 末尾的部分，没有 HTML 后缀。可以多次给出，并接受 shell 模式。默认为空。

编译后立即执行：

--run

立即执行创建的二进制文件（或导入编译后的模块）。默认关闭。

--debugger

在调试器中执行，例如"`gdb`"或"`lldb`"以自动获取堆栈跟踪。默认关闭。

--execute-with-pythonpath

当使用`--run`立即执行创建的二进制文件或模块时，不重置`PYTHONPATH`环境变量。当所有模块都成功包含时，您不应该需要`PYTHONPATH`了，对于独立模式尤其如此。

编译选择：

--user-package-configuration-file=YAML_FILENAME

用户提供的带有包配置的 Yaml 文件。您可以包含 DLL、删除臃肿、添加隐藏依赖项。有关要使用的格式的完整描述，请查看用户手册。可以多次给出。默认为空。

--full-compat

强制与 CPython 绝对兼容。甚至不允许与 CPython 行为的轻微偏差，例如没有更好的回溯或异常消息，这些并不是真的不兼容，只是不同或更差。这仅用于测试，不应使用。

--file-reference-choice=MODE

选择"**file**"将采用的值。对于"运行时"（独立二进制模式和模块模式的默认值），创建的二进制文件和模块使用它们自己的位置来推断"**file**"的值。包含的包假装在该位置下面的目录中。这允许您在部署中包含数据文件。如果您仅仅是寻求加速，那么使用"原始"值会对您更好，其中将使用源文件的位置。使用"冻结"时，使用类似"<frozen module_name>"的标记。为了兼容性，"**file**"的值将始终具有".py"后缀，无论它实际上是什么。

--module-name-choice=MODE

选择"**name**"和"**package**"将采用的值。对于"运行时"（模块模式的默认值），创建的模块使用父包来推断"**package**"的值，以保持完全兼容。"原始"的值（其他模式的默认值）允许进行更静态的优化，但对于可以加载到任何包中的模块不兼容。

输出选择：

--output-filename=FILENAME

指定如何命名可执行文件。对于扩展模块没有选择，也不用于独立模式，使用它将是一个错误。这可能包括需要存在的路径信息。默认为此平台上的可执行文件格式，如：`“<program_name>”.exe`

--output-dir=DIRECTORY

指定应将中间和最终输出文件放入的位置。该目录将填充构建文件夹、`dist`文件夹、二进制文件等。默认为当前目录。

--remove-output

生成模块或可执行文件后删除构建目录。默认关闭。

--no-pyi-file

为`Nuitka`创建的扩展模块不要创建`.pyi`文件。这用于检测隐式导入。默认关闭。

调试功能：

--debug

执行所有可能的自检以查找 Nuitka 中的错误，不要用于生产。默认关闭。

--unstripped

在生成的对象文件中保留调试信息，以获得更好的调试器交互。默认关闭。

--profile

启用基于 vmprof 的编译时间概要分析。当前不工作。默认关闭。

--internal-graph

创建优化过程内部的图。不要用于整个程序，仅用于小的测试用例。默认关闭。

--trace-execution

跟踪执行输出，在执行每行代码之前输出该行。默认关闭。

--recompile-c-only

这不是增量编译，而仅用于 Nuitka 开发。获取现有文件并简单地再次将它们编译为 C。允许编译编辑后的 C 文件以快速调试生成的源代码中的更改，例如看代码是否通过，输出的值等，默认关闭。取决于编译 Python 源代码以确定应查看的文件。

--xml=XML_FILENAME

将内部程序结构，优化结果以 XML 形式写入给定的文件名。

--generate-c-only

仅生成 C 源代码，不编译为二进制文件或模块。这是为了调试和代码覆盖分析，不会浪费 CPU。默认关闭。不要认为您可以直接使用它。

--experimental=FLAG

使用标记为“实验性”的功能。如果没有实验性功能存在于代码中，可能没有效果。每个实验功能使用秘密标签（检查源代码）。

--low-memory

尝试使用更少的内存，通过派生较少的 C 编译作业和使用较少内存的选项。用于嵌入式机器。如果出现内存不足的问题，请使用此选项。默认关闭。

--create-environment-from-report=CREATE_ENVIRONMENT_FROM_REPORT

从给定报告文件（例如“--report=compilation-report.xml”）创建给定不存在路径上的新 virtualenv。默认不执行。

后端 C 编译器选择：

--clang

强制使用 clang。在 Windows上，这需要一个有效的 Visual Studio 版本进行支持。默认关闭。

--mingw64

在 Windows 上强制使用 MinGW64。默认关闭，除非使用带 MinGW Python 的 MSYS2。与 `--clang` 命令一起使用就可以使用 `MSYS2` 的 clang 编译器，否则即使指定了 `--clang` 参数也将使用 `MSVC` 版本的 clang 编译器。

--msvc=MSVC_VERSION

在 Windows上强制使用特定的 MSVC 版本。允许的值例如“14.3”（MSVC 2022）和其他 MSVC 版本号，指定“list”获取已安装编译器的列表，或使用“latest”。如果已安装，默认为最新的 MSVC，否则使用 MinGW64。

--jobs=N

指定允许的并行 C 编译器作业数。默认为系统 CPU 数。

--lto=choice

使用链接时间优化（MSVC，gcc，clang）。允许的值为“yes”、“no”和“auto”（已知工作时）。默认为“auto”。

--static-libpython=choice

使用 Python 的静态链接库。允许的值为“yes”、“no”和“auto”（已知工作时）。默认为“auto”。

缓存控制：

--disable-cache=DISABLED_CACHES

禁用所选缓存，指定 `all` 表示全部缓存。当前允许的值有： `all` ， `ccache` ， `bytecode` ， `dll-dependencies` 。可以多次给出或用逗号分隔的值。

默认不开启。

--clean-cache=CLEAN_CACHES

在执行之前清除给定的缓存，指定 `all` 表示全部缓存。当前允许的值有： `all` ， `ccache` ， `bytecode` ， `dll-dependencies` 。可以多次给出或用逗号分隔的值。
默认不开启。

--disable-bytecode-cache

不要重用标准库中包含的字节码的模块的依赖关系分析结果。与 `--disable-cache=bytecode` 相同。

--disable-ccache

不要试图使用 `ccache` （ `gcc` 、 `clang` 等）或 `clcache` （ `MSVC` 、 `clangcl` ）。与 `--disable-cache=ccache` 相同。

--disable-dll-dependency-cache

禁用依赖项行走器缓存。这将导致创建发行版文件夹时间大大增加，但可能在缓存可能导致错误时使用。与 `--disable-cache=dll-dependencies` 相同。

--force-dll-dependency-cache-update

强制更新依赖项行走器缓存。这将导致创建发行版文件夹时间大大增加，但在怀疑缓存导致错误或已知需要更新时可能会使用。

PGO 编译选择：

--pgo

启用C级别的基于Profile的优化（PGO），通过首先执行一个专门的构建进行分析运行，然后使用结果反馈到C编译中。注意：这是实验性的，在 Nuitka 的独立模式下还不工作。默认关闭。

--pgo-args=PGO_ARGS

在进行基于Profile的优化时要传递的参数。这些在 PGO 分析运行期间传给特殊构建的可执行文件。默认为空。

--pgo-executable=PGO_EXECUTABLE

收集配置文件信息时要执行的命令。仅在需要通过脚本启动它时使用。默认使用创建的程序。

跟踪功能：

--report=REPORT_FILENAME

在 XML 输出文件中报告模块、数据文件、编译、插件等详细信息。这在问题报告中也非常有用。例如，这些报告可以与“--create-environment-from-report”一起使用，轻松重新创建环境，但包含大量信息。默认关闭。

--report-template=REPORT_DESC

通过模板报告。提供模板和输出文件名"template.rst.j2：output.rst"。对于内置模板，请查看用户手册以了解这些模板。可以多次给定。默认为空。

--quiet

禁用所有信息输出，但显示警告。默认关闭。

--show-scons

使用详细信息运行 C 构建后端 Scons，显示执行的命令、检测到的编译器。默认关闭。

--no-progressbar

禁用进度条。默认关闭。

--show-progress

已废弃：提供编译进度信息和统计信息。禁用正常的进度条。默认关闭。

--show-memory

提供内存信息和统计信息。默认关闭。

--show-modules

提供包含的模块和 DLL 的信息。已废弃：您应该改用 '--report' 文件。默认关闭。

--show-modules-output=PATH

`--show-modules` 的输出位置，应该是一个文件名。默认是标准输出。

--verbose

输出所采取操作的详细信息，尤其是优化方面。可以变得很多。默认关闭。

--verbose-output=PATH

`--verbose` 的输出位置，应该是一个文件名。默认是标准输出。

一般的 OS 控制：

--disable-console

当为 Windows 或 macOS 编译时，禁用控制台窗口并创建 GUI 应用程序。默认关闭。

--enable-console

当为 Windows 或 macOS 编译时，启用控制台窗口并创建控制台应用程序。这会禁用某些模块的提示，例如“PySide”，它建议禁用它。默认为 true。

--force-stdout-spec=FORCE_STDOUT_SPEC

强制程序的标准输出到此位置。对于禁用了控制台的程序以及使用 Nuitka 商业版的 Windows 服务插件的程序很有用。默认不活动，例如使用'%PROGRAM%.out.txt'，即靠近程序的文件。

--force-stderr-spec=FORCE_STDERR_SPEC

强制程序的标准错误输出到此位置。对于禁用了控制台的程序以及使用 Nuitka商业版的 Windows 服务插件的程序很有用。默认不活动，例如使用'%PROGRAM%.err.txt'，即靠近程序的文件。

Windows 特有的控制：

--windows-icon-from-ico=ICON_PATH

添加可执行文件图标。可以多次给定不同分辨率或包含多个图标的文件。在后一种情况下，您还可以用 # 作为后缀，其中 n 是从 1 开始的整数索引，指定要包含的特定图标，忽略所有其他图标。

--windows-icon-from-exe=ICON_EXE_PATH

从这个已存在的可执行文件（仅Windows）复制可执行文件图标。

--onefile-windows-splash-screen-image=SPLASH_SCREEN_IMAGE

当为 Windows 和 onefile 编译时，加载应用程序时显示此内容。默认关闭。

--windows-uac-admin

请求 Windows 用户控制，在执行时授予管理权限。（仅Windows）。默认关闭。

--windows-uac-uiaccess

请求 Windows 用户控制，强制仅从少数文件夹运行，远程桌面访问。（仅Windows）。默认关闭。

macOS 特定的控制：

--macos-target-arch=MACOS_TARGET_ARCH

这应该在哪些体系结构上运行。默认和限制是运行 Python 允许的。默认是“native”，即用来运行 Python 的体系结构。

--macos-create-app-bundle

当为 macOS 编译时，创建一个包而不是一个普通的二进制应用程序。目前还在实验和不完整。目前这是解锁禁用控制台的唯一方法。默认关闭。

--macos-app-icon=ICON_PATH

为应用程序包添加图标。只能给一次。默认为可用的 Python 图标。

--macos-signed-app-name=MACOS_SIGNED_APP_NAME

用于 macOS 签名的应用程序名称。遵循 “com.YourCompany.AppName” 命名约定可以获得最佳效果，因为这些必须是全局唯一的，并有可能授予受保护的 API 访问权限。

--macos-app-name=MACOS_APP_NAME

要在 macOS 包信息中使用的产品名称。默认为二进制文件的基本文件名。

--macos-app-mode=MODE

应用程序包的应用模式。当启动窗口并希望出现在 Docker 中时，默认值“gui”很合适。如果从不启动窗口，应用程序是一个“后台”应用程序。对于稍后显示的 UI 元素，“ui-element” 介于两者之间。该应用程序不会出现在 dock 中，但在打开窗口时可以完全访问桌面。

--macos-sign-identity=MACOS_APP_VERSION

在 macOS 上签名时，默认情况下将使用临时标识，但使用此选项可以指定要使用的其他标识。代码签名在 macOS 上现在是强制性的，不能禁用。如果未给出，默认为“临时”。

--macos-sign-notarization

在为公证签名时，使用来自 Apple 的适当 TeamID 标识，使用所需的运行时签名选项，以便可以被接受。

--macos-app-version=MACOS_APP_VERSION

要在 macOS 包信息中使用的产品版本。如果未给出，默认为“1.0”。

--macos-app-protected-resource=RESOURCE_DESC

请求访问 macOS 保护资源的权限，例如
"NSMicrophoneUsageDescription：Microphone access for recording audio." 请求麦克风访问权限并为用户提供一个信息性文本，并解释为什么需要它。冒号前是一个操作系统标识符，用于访问权限，然后是信息性文本。法律值可以在 Apple Developer 上找到，该选项可以指定多次。默认为空。

Linux 特定的控制：

--linux-icon=ICON_PATH

为 onefile 二进制文件添加可执行文件图标使用。只能给一次。默认为可用的 Python 图标。

二进制版本信息：

--company-name=COMPANY_NAME

要在版本信息中使用的公司名称。默认未使用。

--product-name=PRODUCT_NAME

要在版本信息中使用的产品名称。默认为二进制文件的基本文件名。

--file-version=FILE_VERSION

要在版本信息中使用的文件版本。必须是最多4个数字的序列，例如 1.0 或 1.0.0.0，不允许更多数字，不允许字符串。默认未使用。

--product-version=PRODUCT_VERSION

要在版本信息中使用的产品版本。与文件版本相同的规则。默认未使用。

--file-description=FILE_DESCRIPTION

版本信息中使用的文件说明。目前仅Windows。默认为二进制文件名。

--copyright=COPYRIGHT_TEXT

版本信息中使用的版权。目前仅Windows。默认不显示。

--trademarks=TRADEMARK_TEXT

版本信息中使用的商标。目前仅Windows。默认不显示。

插件控制：

--enable-plugin=PLUGIN_NAME

启用的插件。必须是插件名称。使用'--plugin-list'查询完整列表并退出。默认为空。

--disable-plugin=PLUGIN_NAME

禁用的插件。必须是插件名称。使用'--plugin-list'查询完整列表并退出。禁用大多数标准插件通常不是一个好主意。默认为空。

--plugin-no-detection

插件可以检测它们是否可能被使用，然后您可以通过"--disable-plugin=plugin-that-warned"禁用警告，或者您可以使用此选项完全禁用该机制，这当然也略微加快了编译速度。默认关闭。

--plugin-list

显示所有可用插件的列表并退出。默认关闭。

--user-plugin=PATH

用户插件的文件名。可以多次给出。默认为空。

--show-source-changes

显示编译前对原始 Python 文件内容的更改。主要用于开发插件。默认为 False。

'anti-bloat' 插件选项：

--show-anti-bloat-changes

注释插件所做的更改。

--noinclude-setuptools-mode=NOINCLUDE_SETUPTOOLS_MODE

如果遇到 'setuptools' 或导入，该怎么办。这个包可以带有依赖项变大，绝对应该避免。还处理 'setuptools_scm'。

--noinclude-pytest-mode=NOINCLUDE_PYTEST_MODE

如果遇到 'pytest' 导入，该怎么办。这个包可以带有依赖项变大，绝对应该避免。还处理 'nose' 导入。

--noinclude-unittest-mode=NOINCLUDE_UNITTEST_MODE

如果遇到 unittest 导入，该怎么办。这个包可以带有依赖项变大，绝对应该避免。

--noinclude-IPython-mode=NOINCLUDE_IPYTHON_MODE

如果遇到 IPython 导入，该怎么办。这个包可以带有依赖项变大，绝对应该避免。

--noinclude-dask-mode=NOINCLUDE_DASK_MODE

如果遇到 'dask' 导入，该怎么办。这个包可以带有依赖项变大，绝对应该避免。

--noinclude-numba-mode=NOINCLUDE_NUMBA_MODE

如果遇到 'numba' 导入，该怎么办。这个包可以带有依赖项变大，目前对独立模式不起作用。这个包可以带有依赖项变大，绝对应该避免。

--noinclude-default-mode=NOINCLUDE_DEFAULT_MODE

这实际上为上述选项提供了默认的“警告”值，并可用于打开所有这些。

--noinclude-custom-mode=CUSTOM_CHOICES

如果遇到特定导入，该怎么办。格式为模块名称，可以并且应该是一个顶层包，然后是一个选择，“error”、“warning”、“nofollow”，例如 PyQt5：error。