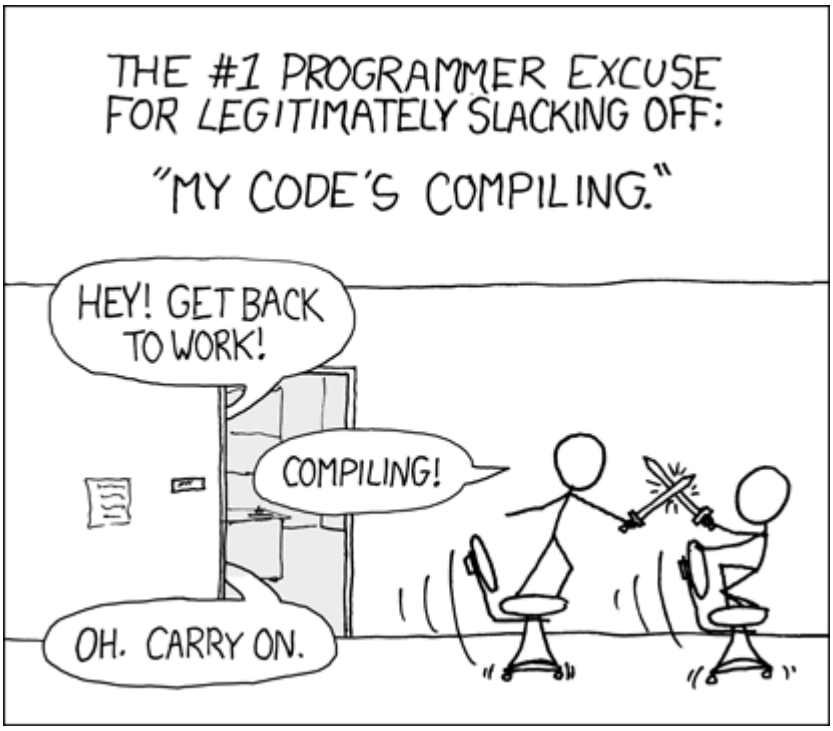
编译缓存工具 CCache 介绍与基本使用

利用编译缓存显著提升项目构建速度



编译时间过长是影响大型项目研发效率的一大难题。就市面上被普遍使用的构建系统，其确实存在一定的编译缓存机制，使得用户在重复进行编译操作时（例如修改了一个 Bug），尽可能复用之前的编译成果以节省编译时间。例如 CMake 使用 `CMakeCache.txt` 来对编译缓存项进行追踪。但类似的机制存在一些限制，如 `CMakeCache.txt` 只能管理它自己所在项目的构建目录，如果用户因为一些原因清空了整个构建目录，构建缓存也将被一同清理；又如果多人协作同一个代码项目，他们各自都有属于自己的用户目录和工作目录（甚至自己的工作电脑），自然不可能指望他们可以共享编译缓存，即使这些缓存实质上确实是完全相同的，毕竟是同一份源码编译出的同样的东西。

为了解决这个问题，我们可以引入专业的编译缓存管理工具：CCache。

CCache 简介

> Ccache is a compiler cache. It speeds up recompilation by caching the result of previous compilations and detecting when the same compilation is being done again.

你可以把 CCache 理解为**编译缓存的共享存储空间**。它管理一个目录 `cache_dir`，这个目录负责存放和记录你编译产生的所有缓存项。当你每次进行编译时，若 CCache 在缓存中发现（缓存命中）了你将要编译的某个项目（比如你没有改某个 `.cpp` 文件从而生成的完全相同的 `.cpp.o` 文件），将会跳过真正的编译流程，而直接从缓存中将命中的项目提取出来“假装”完成了编译。这对于大型项目增量编译而言，意义非凡。这保证了我们每一次对项目代码进行修改并编译的时候，几乎总是能做到严格的增量编译，从而不去浪费时间在一遍又一遍的编译那些重复的东西上。

总体来说，编译缓存是一种“空间换时间”的思路：利用空余的磁盘空间存储尽可能多的编译缓存项目，并用专门的工具和缓存命中算法加以管理的策略。更进一步思考，编译缓存的存储位置不一定在本机，完全可以放在某个网络服务器上（为保证速度，通常用局域网服务器搭建），工作在同一个局域网的机器可以一起利用服务器来获取编译缓存，机器自身在编译产生成果的同时同时还可以反过来向服务器贡献编译缓存。这样的工作流程有时会被成为“编译农场（Compiling Farm）”。

CCache 安装与配置

> 参考：官网 4.8.3 版本完整使用说明书

下载

CCache 的安装非常简单。其最新版 4.8.3 可以在官网上直接下载。推荐直接下载二进制发布版（binary release），当然愿意的话也可以下载源码编译。

> ⚠ 注：不推荐用 `apt` 安装，上面只有较老旧的 3.x 版本。

下载后解压，目录内只需要关注唯一一个可执行文件 `ccache`。为执行方便，可以将其复制到 `PATH` 所包含的目录中。官网推荐复制到 `/usr/local/bin` 目录：

```
# 先解压并进入 CCache 安装包目录
$ sudo cp ccache /usr/local/bin/
```

需要的话重新载入终端，测试“安装”结果：

```
$ ccache -V
ccache version 4.8.3
Features: file-storage http-storage redis+unix-storage redis-storage

Copyright (C) 2002-2007 Andrew Tridgell
Copyright (C) 2009-2023 Joel Rosdahl and other contributors

See <https://ccache.dev/credits.html> for a complete list of contributors.

This program is free software; you can redistribute it and/or modify it under
the terms of the GNU General Public License as published by the Free Software
Foundation; either version 3 of the License, or (at your option) any later
version.
```

配置

通过 `-p` 或 `--show-config` 参数，可以查看 CCache 的默认配置：

```
$ ccache -p
(default) absolute_paths_in_stderr = false
(default) base_dir =
(default) cache_dir = /home/mriiiron/.cache/ccache
(default) compiler =
(default) compiler_check = mtime
(default) compiler_type = auto
(default) compression = true
(default) compression_level = 0
(default) cpp_extension =
(default) debug = false
(default) debug_dir =
(default) depend_mode = false
(default) direct_mode = true
(default) disable = false
(default) extra_files_to_hash =
(default) file_clone = false
(default) hard_link = false
(default) hash_dir = true
(default) ignore_headers_in_manifest =
(default) ignore_options =
(default) inode_cache = true
(default) keep_comments_cpp = false
(default) log_file =
(default) max_files = 0
(default) max_size = 5.0 GiB
(default) msvc_dep_prefix = Note: including file:
(default) namespace =
(default) path =
(default) pch_external_checksum = false
(default) prefix_command =
(default) prefix_command_cpp =
(default) read_only = false
(default) read_only_direct = false
(default) recache = false
(default) remote_only = false
(default) remote_storage =
(default) reshare = false
(default) run_second_cpp = true
(default) sloppiness =
(default) stats = true
(default) stats_log =
(default) temporary_dir = /run/user/1000/ccache-tmp
(default) umask =
```

默认配置一般已经可以满足大部分情况下使用。目前需要关注的可能有：

- `max_size`：描述 CCache 将预留多少磁盘空间用于存放编译缓存。默认 5GB。如果编译较大的项目且磁盘空间富余，可以适当考虑预留多一些空间；
- `cache_dir`：CCache 存放编译缓存的目录位置。可以看到这个目录位于当前的用户目录下，也就是说，如果不同的用户使用同一台机器，他们将使用不同的缓存目录，需注意。

若需修改配置，可以使用 `-M` 或 `--max-size` 参数，例如将 `max_size` 设置为 10GB：

```
$ ccache -M 10
Set cache size limit to 10.0 GiB
```

检查是否设置成功：

```
$ ccache -p | grep max_size
(/home/mriiiron/.config/ccache/ccache.conf) max_size = 10.0 GiB
```

注意这里对 `max_size` 的配置已经被写入了自动建立的配置文件中。

> 关于配置项的完整说明，参见官网手册 。

使用

CCache 通常有两种使用方式：

1. 直接使用：通过 `ccache` 命令直接调用，编译命令本身统统接在 `ccache` 后面作为其参数。例如：`ccache gcc -c example.c`；
2. 通过“劫持” `gcc`、`g++` 等命令使用。这样当每次调用 `gcc` 等命令时，实际已经通过了 `ccache` 的处理。

第一种方式一般仅仅在测试 CCache 功能等场合时使用。实际工作中因为我们通常都是使用 Makefile 等手段完成编译，配置编译工具链来调用 `ccache` 既麻烦又不现实，所以一般使用第二种方式。官网推荐构建软链接的方式完成“劫持”：

```
$ sudo ln -s ccache /usr/local/bin/gcc
$ sudo ln -s ccache /usr/local/bin/g++
$ sudo ln -s ccache /usr/local/bin/cc
$ sudo ln -s ccache /usr/local/bin/c++
```

这样就把 `gcc`、`g++`、`cc`、`c++` 四个命令通过软链接指向了 `ccache`。验证如下：

```
$ ll /usr/local/bin/gcc

```

```
lrwxrwxrwx 1 root root 6 Sep 20 11:42 /usr/local/bin/gcc -> ccache*
```

> 通常安装好 GCC 后， `gcc` 等命令本身位于 `/usr/bin/gcc` ，命令本身是指向实际 GCC 版本的软链接，再通过多重软连接等手段指向实际的 GCC 可执行文件。这个在每台机器上可能都有所不同，例如：
>
> ```
> $ ll /usr/bin/gcc
> lrwxrwxrwx 1 root root 14 Sep 20 11:40 /usr/bin/gcc -> /usr/bin/gcc-
> $ ll /usr/bin/gcc-9
> lrwxrwxrwx 1 root root 22 Oct 24  2022 /usr/bin/gcc-9 -> > x86_64-li
> $ which x86_64-linux-gnu-gcc-9
> /usr/bin/x86_64-linux-gnu-gcc-9
> ```
>
> 因此我们选择不覆盖原本 `/usr/bin` 里面的软链接，而把我们指向 `ccache` 的软链接放在更高执行优先的 `/usr/local/bin` 之中。这样既可以使 `gcc` 等命令完美优先指向 `ccache` ，同时也尽可能不去破坏原有环境。

整合到 CMake

如果现在我们直接使用 `make` 编译，可能会发现编译缓存并没有起到作用。原因在于 CMake 默认告诉 Makefile 的编译命令，可能直接是 `/usr/bin` 下面的软链接，这样便绕开了我们定义在 `/usr/local/bin` 下面的软链接。可以用 `make` 命令的 `VERBOSE` 参数验证如下：

```
# conan install ..
# cmake ..
$ make VERBOSE=1
```

通过输出可以检查 `make` 实际调用的命令：

```
...
...
[ 75%] Building CXX object CMakeFiles/larkgui.dir/src/lark-gui/component/lbasi
/usr/bin/g++ ...
...
...
```

如果证实了上述情况，我们可以借助设置 CMake 的 `CMAKE_<LANG>_COMPILER` 变量来控制 `make` 具体使用的编译器命令。例如我们分别设置用于 C 和 C++ 语言的编译器命令：

```
set (CMAKE_C_COMPILER "gcc")
set (CMAKE_CXX_COMPILER "g++")
```

这样就用我们之前配置的 `gcc` 软链接（也就是指向 `ccache` 的 `/usr/local/bin/gcc`）代替了 `/usr/bin/gcc` 软链接，这样就可以正确的使用 CCache 了。

体验 CCache 编译缓存

CCache 提供了 `-s` 或 `--show-stats` 命令参数对缓存使用情况进行统计，我们可以借助其来监控 CCache 的使用情况。例如：

```
$ ccache -s
Cacheable calls:    1142 / 1206 (94.69%)
  Hits:              951 / 1142 (83.27%)
    Direct:          951 /  951 (100.0%)
    Preprocessed:      0 /  951 ( 0.00%)
  Misses:            191 / 1142 (16.73%)
Uncacheable calls:    64 / 1206 ( 5.31%)
Local storage:
  Cache size (GiB):  0.0 /  5.0 ( 0.08%)
  Hits:              951 / 1142 (83.27%)
  Misses:            191 / 1142 (16.73%)
```

在刚配置好 CCache 的机器上进行初次构建，由于缓存尚未建立，编译速度并未有所提升，但此时编译缓存已经慢慢建立。第一次执行 `ccache -s` 可能看见 `Cacheable calls` 为零，但随着后续的编译操作进行，将产生越来越多的缓存命中，再执行 `ccache -s` 就可看见变化，同时可以明显感知到编译速度的提升。

由于编译缓存归于 CCache 管理，因此即使你完全删除了构建目录（在 `build` 里面 `rm -rf *` 之类），下一次 `make` 的时候同样可以享受编译缓存带来的加速体验。

以上便是 CCache 基本的本地使用方式。事实上 CCache 还支持网络使用、跨用户共享等高级功能，这超出了本文的介绍范围。关于 CCache 的高级使用，还请参阅官方文档。

参考资料

- CCache 官网

© 2023 中科合迅。 保留所有权利。 | Powered by Wiki.js