分布式编译

通过本文，您将了解能够通过将编译过程分布到本地网络中的多台机器上，从而加快速度的开源工具选项。

减少基于 C/C++ 的系统的编译时间是所有发布和编译工程师所面对的主要挑战之一。本文研究一些可通过并行活动来加快编译过程的开源工具选项：将编译过程分布到本地网络中的多台机器上。本文中的讨论主要集中于 GNU make，因为它使用比较广泛。

GNU make 中的 -j 选项

默认情况下，make 是一个顺序工作的工具。它按次序调用底层编译器来编译 C/C++ 源。通常，C/C++ 源文件（通常带有 .cpp/.cxx 扩展名）不需以对方为基础即可编译。使用 -j 选项调用 make 来完成该操作。清单 1 显示的是一种典型的用法。

清单 1. 典型的 GNU make 调用

```
make -j10 -f makefile.x86_linux
```

-j -- 10 的参数是编译过程开始后能同时进行的最大编译数。如果没有给 -j 提供任何参数，则所有源文件都会在系统中排队，等待同时编译。在运行多核系统上的编译时，使用 -j 选项特别有用。要使用 -j 选项，必须先解决几个关键问题；这些问题将在下面部分讨论。

使用 -j 选项时的问题和可能的解决方案

首先要检查系统配置。在低内存（<512MB RAM）系统上，同时编译的数量太多会因为分页而使系统变慢。在这种情况下会增加编译时间。您需要进行试验以得出系统的最佳 -j 值。另一种选择是使用 GNU make 工具的 -l 或 -load-average 选项，同时也使用 -j（它只在系统负载小于一定水平时才触发作业）。

还可以使用同一个临时文件进行独立编译。请考虑清单 2 中所示的 make 代码片段。

清单 2. 使用同一个临时文件 y.tab.c 的 makefile。

```
my_parser : main.o parser1.o parser2.o
	g++ -o $* $>


parser1.o : parser1.y
	yacc parser1.y
	g++ -o $* -c y.tab.c

parser2.o : parser2.y
	yacc parser2.y
	g++ -o $* -c y.tab.c
```

假设语法文件 parser1.y 和 parser2.y 位于同一目录中。在有序编译期间，yacc（其中 y.tab.c 是默认文件名）为 parser1 生成文件 y.tab.c，然后为 parser2 生成文件 y.tab.c；但在并行模式下，这会导致冲突。有几种方法可以解决这个问题：将两个 yacc 文件放在单独的文件夹中；或者使用 -b 选项生成两个不同的 C 输出，如清单 3 所示。

清单 3. 使用 yacc 的 -b 选项生成唯一的文件名

```
parser1.o : parser1.y
	yacc parser1.y -b parser1
	g++ -o $* -c parser1.tab.c
```

您必须严密监视 makefile 是否发生这种情况，在这种情况下，在顺序模式下良好编译的脚本会在并行模式下出现混乱。

一些 makefile 规则具有隐式依赖项。请考虑清单 4 中的情况，其中一个 Perl 脚本生成一个被其他源包含的头。

清单 4. 具有隐式依赖项 makefile

```
my_exe: info.h test1.o test2.o
	g++ -o $@ $^

test1.o: test1.cxx
	g++ -c $<

test2.o: test2.cxx
	g++ -c $<

info.h:
	make_header #shell script that generates the header file
```

info.h 头被 test1.cxx 和 test2.cxx 包含。在次序编译模式下，make 从左到右工作，首先生成文件 info.h。但是，在并行编译模式下，make 可以自由并行处理所有依赖项 ——如果 info.h 没有在 test1.cxx 和/或 test2.cxx 编译开始之前生成的话，这可能导致间歇性编译失败。要修复此问题，需要将 info.h 从依赖项列表中删除，并将它放在 test1.o 和 test2.o 的依赖项列表中。另外，最好使用另一个包装器来确保 info.h 只生成一次。清单 5 显示了修改后的 make_header 脚本，而清单 6 显示了 makefile。

清单 5. 修改 make_header 脚本防止多次编写

```
#!/usr/bin/bash

if [ -f info.h ]
```

```
then
  exit
fi

echo "#ifndef __INFO_H" > info.h
echo "#define __INFO_H" > > info.h

echo "#include <iostream>>" > > info.h
echo "using namespace std;" > > info.h
echo "int f1(int);" > > info.h
echo "int f2(int);" > > info.h

echo "#endif" > > info.h
```

清单 6. 修改后的清单 4 中的 makefile

```
my_exe: info.h test1.o test2.o
    g++ -o $@ $^

test1.o: test1.cxx info.h
    g++ -c $<

test2.o: test2.cxx info.h
    g++ -c $<

info.h:
    make_header #shell script that generates the header file
```

一般而言，如果正确创建 makefile，make -j 就能够提取充足的并行项。尽量避免在 makefile 中引入不必要的依赖项。

注意，GNU make 只能提取单台机器的并行项。下一部分将介绍 distcc，这是一个用于在多台机器上共享编译过程的工具。

distcc 简介

distcc 工具可以将 C/C++ 代码的编译分布到多台机器。但这些机器都必须安装 distcc。下面是关于快速安装和配置的说明：

1. 下载 distcc（请参阅 参考资料 部分）。
2. 通过执行 ./configure; make && make install 在所有机器上编译 distcc 源。
3. 编译过程先在某台机器上开始，然后分布所有其他机器（服务器）。在所有服务器中，启动 distccd 守护程序（您必须具有执行操作的根特权）。distccd 位于 /etc/init.d 文件夹。在根模式下启动 distccd 的语法是

   ```
   tcsh-arpan# /etc/init.d/distccd start
   ```

   在用户模式下启动它的语法是

   ```
   tcsh-arpan$ sudo /etc/init.d/distccd
   ```

还可以通过运行 distccd -daemon -j N 在用户模式下运行 distcc 守护进程，其中 N 是您要在给定机器上运行的作业数。

4. 本地机器需要知道应该将编译过程分布到哪些服务器。根据您的 shell，发出与下面命令相似的命令：

```
export DISTCC_HOSTS='localhost tintin asterix pogo'
```

tintin、asterix 和 pogo 是网络中可以驻留编译过程的其他主机；localhost 是本地计算机。

5. 也可以不使用导出指令。您可以创建一个名为 hosts 的文件，将服务器的名称放在该文件中，各个名称使用空格分隔。将该文件复制到 $HOME/.distcc 文件夹。

安装 distcc 之后，惟一需要做的就是触发编译。下面是调用方法：

```
make -j4 CC=distcc -f makefile.x86_linux
```

> distcc 的工作原理
> distcc 将预处理代码发送给网络中的其他指定机器。distccd 守护进程确保编译在远程机器上发生。distcc 的设计目的是与 GNU make 的并行编译（-j）选项一起使用。distcc 本身不是一个编译器；它只是用作 g++ 的一个前端。几乎 g++ 的所有选项都可以按原样传递给 distcc。

使用 distcc 需要记住的几个关键点

要使 distcc 为您工作，必须记住以下几件事情：

- 几台机器必须具有一致的配置。这意味着所有机器上必须安装相同版本的 g++ 编译器，以及相关的编译工具，如 ar、ranlib、libtool 等。操作系统的类型和版本也应该相同。
- 在客户端机器上，distcc 将预处理代码发送给服务器机器。您需要验证 distccd 守护进程正在服务器机器上运行。
- 默认情况下，distcc 在单台机器上调度的作业数是 CPU 的个数 + 2。对于单核机器，这个数是 3。在触发进程时请记住这一点：像 make -j10 CC=distcc 这样的命令行（其中只有三个主机）意味着最初触发 9 个编译作业。
- 保证底层机器可以访问存储源文件的必备文件系统。在基于网络文件系统（Network File System，NFS）的系统中，一些源区域不能被挂载，这将导致编译失败。同时还要仔细监视网络堵塞。
- distcc 用于通过网络编译源代码。链接步骤可能不是并行的。

监视 distcc 编译过程

distcc 安装有一个称为 distccmon-text 的基于控制台的监视工具。在启动编译过程之前，有必要打开一个单独的终端窗口并发出 distccmon-text 5。然后，这个终端每隔 5 秒钟就显示网络中多个节点的编译状态。清单 7 显示了一个监视窗口示例。

> 必须按次序运行的编译过程
> 编译中的有些步骤可能不是并行的 —— 必须在单台机器上才能使用脚本生成某些头、链接等。要更好地处理这种情况，最好将原始 makefile 拆分为多个 makefile，明确划分哪些可以并行运行，哪些不可以，然后按以下方式运行它们：
>
> ```
> tcsh-arpan$ make -f make.init;
> make CC=distcc -j4 -f make.compile_x86;
> make -f make.link
> ```

清单7：distccmon-text 的输出

```
2167  Compile     memory.c                  tintin[0]
2164  Compile     main.cxx                   tintin[1]
2192  Compile     ui_tcl.cxx                asterix[0]
2187  Compile     traverse.c                asterix[1]
```

```
2177  Compile     reports.cxx                pogo[0]
2184  Compile     messghandler.c           pogo[1]
2181  Compile     trace.cpp                  localhost[0]
2189  Compile     remote.c                 localhost[1]
```

使用 ccache 进一步提高编译速度

通常，当在 C/C++ 开发框架中修改头文件时，一般基于 make 的系统最终会重新编译所有源文件。通常，头文件更改只会影响源文件的子集，因此不需要进行耗时的编译清理。还可以使用 ccache，这个工具能大大减少编译清理时间（减少到原来的五分之一至十分之一）。

ccache 用作编译器的缓存。它的工作方式是：从预处理源代码和用于编译源代码的编译器选项创建一个哈希表。在重新编译时，如果 ccache 未在预处理源代码和编译器选项中检测到任何更改，它就检索以前编译输出的缓存副本。这有助于加快编译过程。

安装 ccache

要下载 ccache 的最新版本（2.4），请参考 参考资料 小节。转到 ccache 目录后，发出命令 ./configure -prefix=/usr/bin，接着发出命令 make && make install。如果 ccache 没有安装在 /usr/bin，则检查 ccache 的位置是否定义为 PATH 环境变量的一部分。

Ccache 环境变量

下面是一些可用于自定义 ccache 安装的环境变量：

- CCACHE_DIR —— 指定 ccache 存储预编译输出的文件夹。如果没有定义这个变量，那么缓存输出会默认存储在 $HOME/.ccache 中。
- CCACHE_TEMPDIR —— 指定放置 ccache 生成的临时文件的文件夹。如果没有定义这个变量，那么默认使用 $HOME/.ccache。最好定义这个变量和 CCACHE_DIR —— 大多数组织有一个针对特定文件系统区域的用户配额，如果 $HOME 属于这个区域，那么配额很快就会用完。显式地设置这个缓存区域以避免此类问题。
- CCACHE_DISABLE —— 设置这个选项告诉 ccache 完全调用编译器，从而绕过缓存。这在诊断时使用。
- CCACHE_RECACHE —— 设置这个选项告诉 ccache 忽略缓存中现有的条目并调用编译器；但对于新的条目，则缓存结果。这在诊断时使用。
- CCACHE_LOGFILE —— 设置这个选项告诉 ccache 随机记录该文件在缓存中的统计信息。这对诊断特别有用。
- CCACHE_PREFIX —— 向 ccache 用于完全调用编译器的命令行添加一个前缀。这专门用于将 distcc 和 ccache 连接起来。下一部分将会对此进行详细讨论。

使用 ccache

使用 ccache 时，可以带有 distcc，也可以不带。这不依赖于 -j makefile 选项。ccache 最简单的用法如下：ccache g++ -o <executable name> <source file(s)>。当它与 makefile 一起使用时，就会覆盖 CC 变量；如清单 8 所示。

清单 8. 使用 CC 变量的示例 makefile

```
CC := g++
app1: placer1.o route1.o floorplan1.o
    $(CC) -o $* $^
```

```
placer1.o: placer1.cxx
    $(CC) -o $* -c $<
...
```

使用清单 8 中的 makefile，发出 make 的语法是 make "CC=ccache g++"。

为了同时使用 ccache 和 distcc，需要将 CCACHE_PREFIX 环境变量设置为 distcc，如下所示：export CCACHE_PREFIX=distcc（这个语法适用于 bash shell。如果使用另一种 shell，请相应地修改语法）。

下面是一个使用 ccache 和 distcc 的 make 调用示例：

```
export CCACHE_PREFIX=distcc; make "CC=ccache g++" -j4 -f makefile.x86
```

在编译过程中，shell 提示符下的实际调用类似于：ccache distcc -o placer1.o -c placer1.cxx。注意，只需在本地机器上安装 ccache。ccache 进行第一次检查，确定副本是否存在本地缓存中；如果不存在，就由 distcc 进行分布式编译。

结束语

本文探讨了 GNU make、distcc 和 ccache，这些工具能够并行分布编译过程。它们还有几个可以进一步自定义的其他特性 —— 例如，ccache 有一个限制缓存大小的 -M 选项；distcc 有一个基于 GUI 的监视器 distcc-gnome，它会跟踪网络编译活动（如果使用 -use-gtk 选项编译 distcc，就会创建该监视器）。参考资料 部分中的链接提供更加详细的信息。

标签： distcc ccache
补充话题说明»