**[【转】Linux程序编译速度提高方法](http://www.cnblogs.com/jacktu/archive/2010/07/15/1777974.html)**

项目越来越大，每次需要重新编译整个项目都是一件很浪费时间的事情。Research了一下，找到以下可以帮助提高速度的方法，总结一下。

tmpfs

有人说在Windows下用了RAMDisk把一个项目编译时间从4.5小时减少到了5分钟，也许这个数字是有点夸张了，不过粗想想，把文件放到内存上做编译应该是比在磁盘上快多了吧，尤其如果编译器需要生成很多临时文件的话。

这个做法的实现成本最低，在Linux中，直接mount一个tmpfs就可以了。而且对所编译的工程没有任何要求，也不用改动编译环境。

mount -t tmpfs tmpfs ~/build -o size=1G

用2.6.32.2的Linux Kernel来测试一下编译速度：

用物理磁盘：40分16秒

用tmpfs：39分56秒

呃……没什么变化。看来编译慢很大程度上瓶颈并不在IO上面。但对于一个实际项目来说，编译过程中可能还会有打包等IO密集的操作，所以只要可能，用tmpfs是有益无害的。当然对于大项目来说，你需要有足够的内存才能负担得起这个tmpfs的开销。

make -j

既然IO不是瓶颈，那CPU就应该是一个影响编译速度的重要因素了。

用make -j带一个参数，可以把项目在进行并行编译，比如在一台双核的机器上，完全可以用make -j4，让make最多允许4个编译命令同时执行，这样可以更有效的利用CPU资源。

还是用Kernel来测试：

用make： 40分16秒

用make -j4：23分16秒

用make -j8：22分59秒

由此看来，在多核CPU上，适当的进行并行编译还是可以明显提高编译速度的。但并行的任务不宜太多，一般是以CPU的核心数目的两倍为宜。

不过这个方案不是完全没有cost的，如果项目的Makefile不规范，没有正确的设置好依赖关系，并行编译的结果就是编译不能正常进行。如果依赖关系设置过于保守，则可能本身编译的可并行度就下降了，也不能取得最佳的效果。

ccache

ccache用于把编译的中间结果进行缓存，以便在再次编译的时候可以节省时间。这对于玩Kernel来说实在是再好不过了，因为经常需要修改一些Kernel的代码，然后再重新编译，而这两次编译大部分东西可能都没有发生变化。对于平时开发项目来说，也是一样。为什么不是直接用make所支持的增量编译呢？还是因为现实中，因为Makefile的不规范，很可能这种“聪明”的方案根本不能正常工作，只有每次make clean再make才行。

安装完ccache后，可以在/usr/local/bin下建立gcc，g++，c++，cc的symbolic link，链到/usr/bin/ccache上。总之确认系统在调用gcc等命令时会调用到ccache就可以了（通常情况下/usr/local /bin会在PATH中排在/usr/bin前面）。

继续测试：

用ccache的第一次编译(make -j4)：23分38秒

用ccache的第二次编译(make -j4)：8分48秒

用ccache的第三次编译(修改若干配置，make -j4)：23分48秒

看来修改配置（我改了CPU类型...）对ccache的影响是很大的，因为基本头文件发生变化后，就导致所有缓存数据都无效了，必须重头来做。但如果只是修改一些.c文件的代码，ccache的效果还是相当明显的。而且使用ccache对项目没有特别的依赖，布署成本很低，这在日常工作中很实用。

可以用ccache -s来查看cache的使用和命中情况：

cache directory                     /home/lifanxi/.ccachecache hit                           7165cache miss                         14283called for link                       71not a C/C++ file                     120no input file                       3045files in cache                     28566cache size                          81.7 Mbytesmax cache size                     976.6 Mbytes

可以看到，显然只有第二编次译时cache命中了，cache miss是第一次和第三次编译带来的。两次cache占用了81.7M的磁盘，还是完全可以接受的。

distcc

一台机器的能力有限，可以联合多台电脑一起来编译。这在公司的日常开发中也是可行的，因为可能每个开发人员都有自己的开发编译环境，它们的编译器版本一般是一致的，公司的网络也通常具有较好的性能。这时就是distcc大显身手的时候了。

使用distcc，并不像想象中那样要求每台电脑都具有完全一致的环境，它只要求源代码可以用make -j并行编译，并且参与分布式编译的电脑系统中具有相同的编译器。因为它的原理只是把预处理好的源文件分发到多台计算机上，预处理、编译后的目标文件的链接和其它除编译以外的工作仍然是在发起编译的主控电脑上完成，所以只要求发起编译的那台机器具备一套完整的编译环境就可以了。

distcc安装后，可以启动一下它的服务：

/usr/bin/distccd  --daemon --allow 10.64.0.0/16

默认的3632端口允许来自同一个网络的distcc连接。

然后设置一下DISTCC\_HOSTS环境变量，设置可以参与编译的机器列表。通常localhost也参与编译，但如果可以参与编译的机器很多，则可以把localhost从这个列表中去掉，这样本机就完全只是进行预处理、分发和链接了，编译都在别的机器上完成。因为机器很多时，localhost的处理负担很重，所以它就不再“兼职”编译了。

export DISTCC\_HOSTS="localhost 10.64.25.1 10.64.25.2 10.64.25.3"

然后与ccache类似把g++，gcc等常用的命令链接到/usr/bin/distcc上就可以了。

在make的时候，也必须用-j参数，一般是参数可以用所有参用编译的计算机CPU内核总数的两倍做为并行的任务数。

同样测试一下：

一台双核计算机，make -j4：23分16秒

两台双核计算机，make -j4：16分40秒

两台双核计算机，make -j8：15分49秒

跟最开始用一台双核时的23分钟相比，还是快了不少的。如果有更多的计算机加入，也可以得到更好的效果。

在编译过程中可以用distccmon-text来查看编译任务的分配情况。distcc也可以与ccache同时使用，通过设置一个环境变量就可以做到，非常方便。

总结一下：

tmpfs： 解决IO瓶颈，充分利用本机内存资源

make -j： 充分利用本机计算资源

distcc： 利用多台计算机资源

ccache： 减少重复编译相同代码的时间

这些工具的好处都在于布署的成本相对较低，综合利用这些工具，就可以轻轻松松的节省相当可观的时间。上面介绍的都是这些工具最基本的用法，更多的用法可以参考它们各自的man page