用GNUprofiler提高代码运行速度 下 PDF转换可能丢失图片或格式，建议阅读原文 https://www.100test.com/kao_ti2020/144/2021_2022__E7_94_A8GNUprof_c103_144235.htm 共享库的支持 正如在前面曾经介绍的，对于代码剖析的支持是由编译器增加的，因此如果希望从共享库（包括 C 库 libc.a）中获得剖析信息，就需要使用 -pg 来编译这些库。幸运的是，很多发行版都提供了已经启用代码剖析支持而编译的 C 库版本（libc_p.a）。 在我使用的发行版 gentoo 中，需要将“ profile ”添加到 USE 标志中，并重新执行 emerge glibc.当这个过程完成之后，就会看到 /usr/lib/libc_p.a 文件已经创建好了。对于没有按照标准提供 libc_p 的发行版本来说，需要检查它是否可以单独安装，或者可能需要自己下载 glibc 的源代码并进行编译。 在获得 libc_p.a 文件之后，就可以简单地重新编译前面的例子了，方法如下： gcc example1.c -g -pg -o example1 -O2 -lc_p 然后，可以像以前一样运行这个应用程序，并获得 flat profile 或 call graph，应该会看到很多 C 运行函数，包括 printf（这些函数在我们的测试函数中并不是太重要）。 用户时间与内核时间 现在我们已经知道如何使用 gprof 了，接下来可以简单且有效地对应用程序进行分析了，希望可以消除性能瓶颈。 不过现在您可能已经注意到了 gprof 的最大缺陷：它只能分析应用程序在运行 过程中所消耗掉的用户 时间。通常来说，应用程序在运行时既要花费一些时间来运行用户代码，也要花费一些时间来运行“ 系统代码 ”，例如内核系统调用。 如果对清单 1 稍加修改，就可以清楚地看出这个问题： 清单 5. 为清单 1 添加系统调用分析功能 #include int a(void) { sleep(1). return

0.}int b(void) { sleep(4). return 0.}int main(int argc, char** argv){ int iterations. if(argc != 2) { printf("Usage %s \n", argv[0]). exit(-1). } else iterations = atoi(argv[1]). printf("No of iterations = %d\n", iterations). while(iterations--) { a(). b(). }}