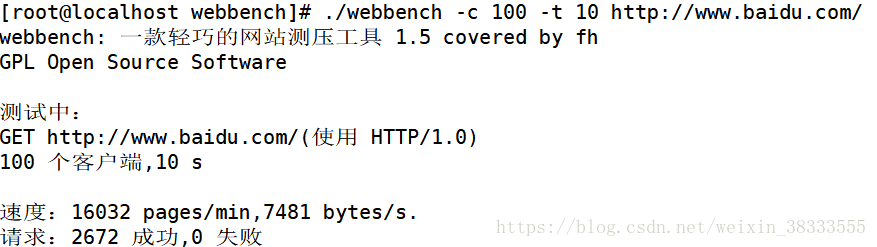
Webench是一款轻量级的网站测压工具，最多可以对网站模拟3w左右的并发请求，可以控制时间、是否使用缓存、是否等待服务器回复等等，且对中小型网站有明显的效果，基本上可以测出中小型网站的承受能力，对于大型的网站，如百度、淘宝这些巨型网站没有意义，因为其承受能力非常大。同时测试结果也受自身网速、以及自身主机的性能与内存的限制，性能好、内存大的主机可以模拟的并发就明显要多。

Webbench用C语言编写，运行于linux平台，下载源码后直接编译即可使用，非常迅速快捷，对于中小型网站的制作者，在上线前用webbench进行系列并发测试不失为一个好的测试方法。

使用方法也是非常的简单，例如对百度进行测试



Webbench实现的核心原理是：父进程fork若干个子进程，每个子进程在用户要求时间或默认的时间内对目标web循环发出实际访问请求，父子进程通过管道进行通信，子进程通过管道写端向父进程传递在若干次请求访问完毕后记录到的总信息，父进程通过管道读端读取子进程发来的相关信息，子进程在时间到后结束，父进程在所有子进程退出后统计并给用户显示最后的测试结果，然后退出。

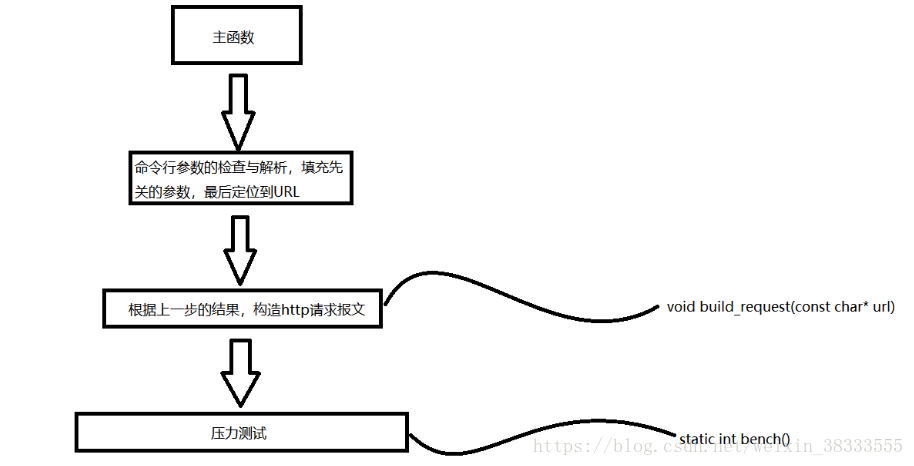
下面我从我实现的思路来详细说一下每个过程，非常显然的地方就略过了，有不明白的地方提问后会尽快回复。

源代码主要有三个源文件：**Socket.c\Socket.h\webbench.c**

其中Sokcet.c与Socket.h封装了对于目标网站的TCP套接字的构造，其中Socket函数用于获取连接目标网站TCP套接字，这一部分后面就不做说明了，重点讲解webbench.c，其中有些对于各种错误的处理就不在此加以说明了，详见末尾的源码。

**详细实现过程**

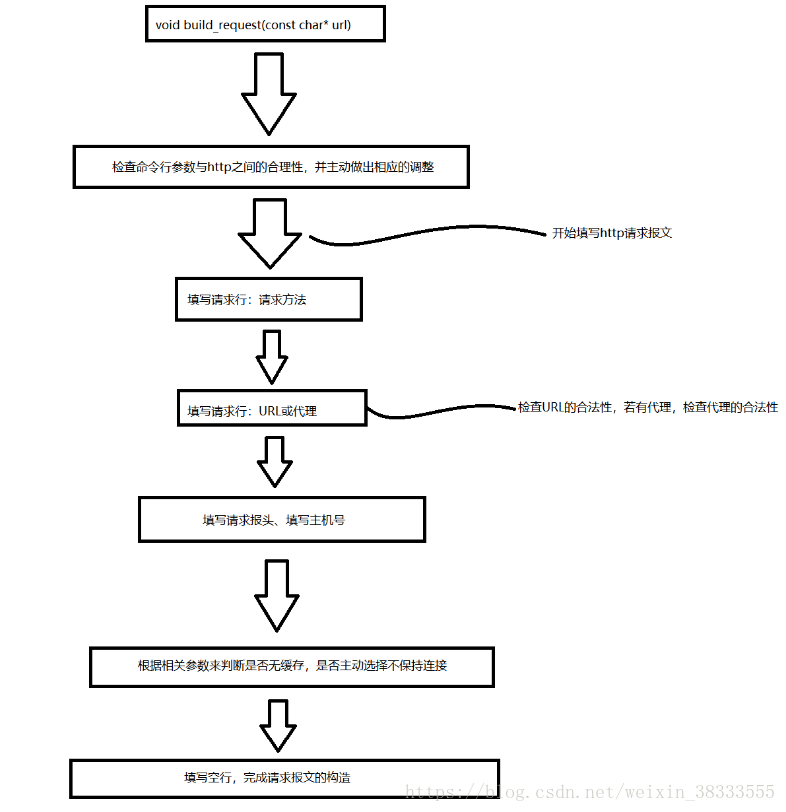
主函数的思路如下

**  
命令行参数解析**

命令行参数解析当然是linux下的经典做法，通过getopt\_long库函数，该函数非常强大，若是没有这个函数，多命令行参数解析会显得很麻烦，很复杂，但有了这个函数就非常简单了，若你不了解这个函数的使用方法，那么点击这篇[博客](https://blog.csdn.net/coding__madman/article/details/51043733)或者自己查阅相关博客进行学习，不然你肯定是看不懂代码的，在此就不做多余的讲解了。在解析命令行参数的过程中，会对每个不同的参数做出一些相应的操作，若解析到有问题的参数会结束并打印帮助手册。命令行参数解析完毕后，就进入请求报文构造函数。

**构造http请求报文**

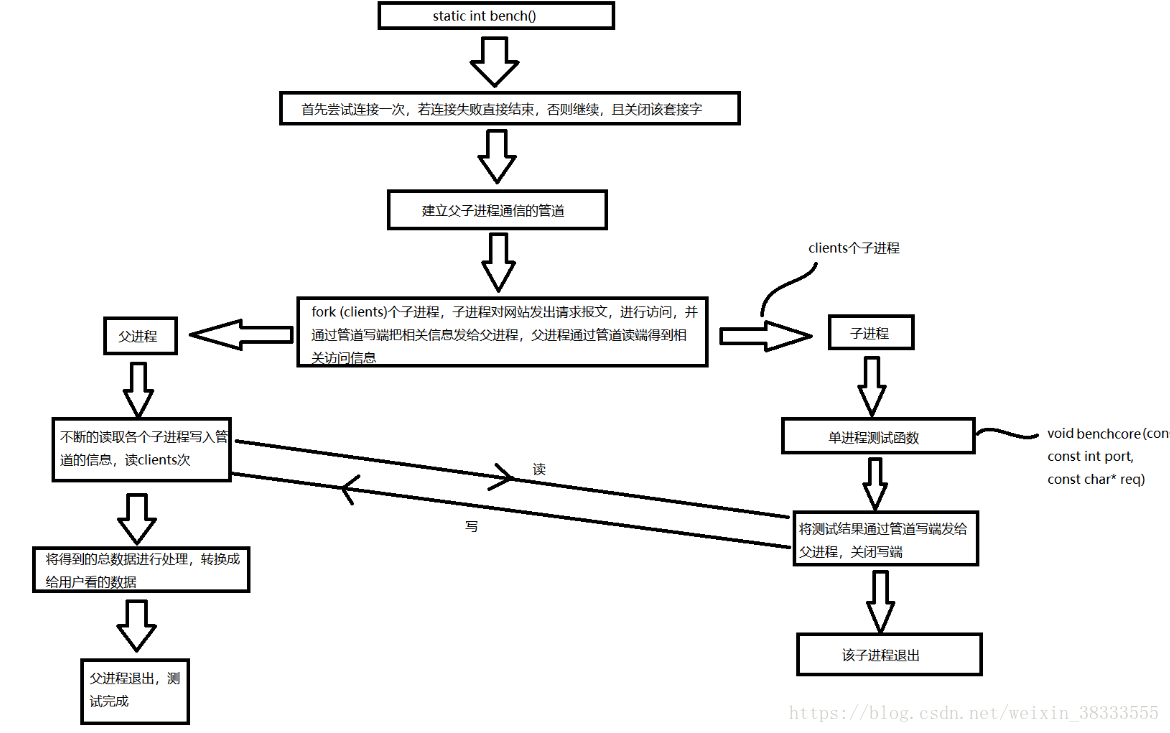
该模块由build\_request函数实现，该函数流程如下



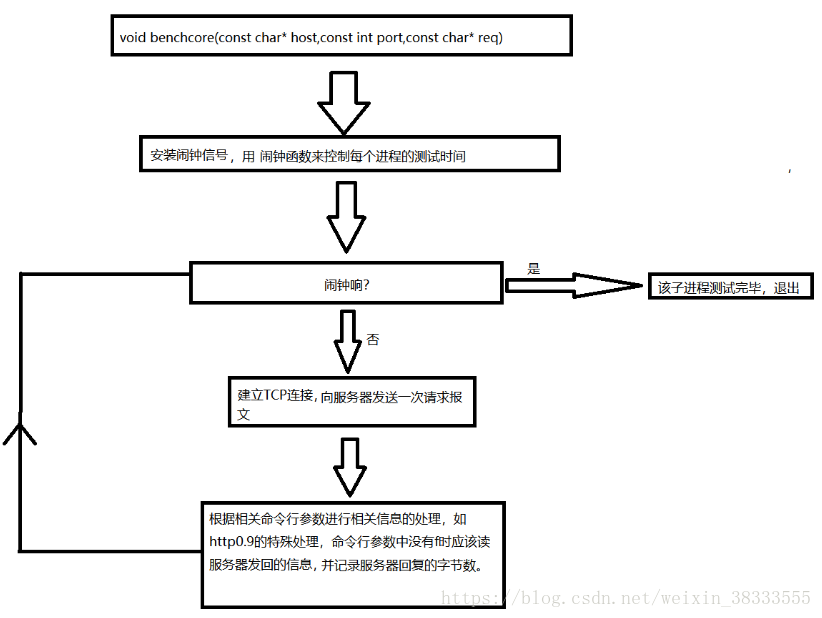
这个模块需要进行大量的错误分析，分析用户有可能出现的错误填写，并及时终止，如果在这一步不做好这件事，后面会出现几乎找不出的错误。然后就是在填写报文时，一定要严格的注意格式，不要随意少空格、回车换行，保证报文填写的正确。

**压力测试**

 这一模块由bench函数来完成



 这一过程中关键是恰当的利用管道在父子进程间通信，其中函数benchcore函数时每个子进程在要求时间内发送请求报文的函数，该函数会记录请求的成功次数、失败次数、以及服务器回复的字节数。



过程分析得再清楚，不如代码清楚来得好（Socket.c与Socket.h不用做说明，就不放了，在末尾的github上）

1. #include"Socket.h"
2. #include<unistd.h>
3. #include<stdio.h>
4. #include<sys/param.h>
5. #include<rpc/types.h>
6. #include<getopt.h>
7. #include<strings.h>
8. #include<time.h>
9. #include<signal.h>
10. #include<string.h>
11. #include<error.h>
13. static void usage(void)
14. {
15. fprintf(stderr,
16. "webbench [选项参数]... URL\n"
17. " -f|--force 不等待服务器响应\n"
18. " -r|--reload 重新请求加载（无缓存）\n"
19. " -t|--time <sec> 运行时间，单位：秒，默认为30秒\n"
20. " -p|--proxy <server:port> 使用代理服务器发送请求\n"
21. " -c|--clients <n> 创建多少个客户端，默认为1个\n"
22. " -9|--http09 使用http0.9协议来构造请求\n"
23. " -1|--http10 使用http1.0协议来构造请求\n"
24. " -2|--http11 使用http1.1协议来构造请求\n"
25. " --get 使用GET请求方法\n"
26. " --head 使用HEAD请求方法\n"
27. " --options 使用OPTIONS请求方法\n"
28. " --trace 使用TRACE请求方法\n"
29. " -?|-h|--help 显示帮助信息\n"
30. " -V|--version 显示版本信息\n" );
31. };
33. *//http请求方法*
34. #define METHOD\_GET 0
35. #define METHOD\_HEAD 1
36. #define METHOD\_OPTIONS 2
37. #define METHOD\_TRACE 3
39. *//相关参数选项的默认值*
40. int method = METHOD\_GET;
41. int clients = 1;
42. int force = 0; *//默认需要等待服务器相应*
43. int force\_reload = 0; *//默认不重新发送请求*
44. int proxyport = 80; *//默认访问80端口，http国际惯例*
45. char\* proxyhost = NULL; *//默认无代理服务器，因此初值为空*
46. int benchtime = 30; *//默认模拟请求时间*
48. *//所用协议版本*
49. int http = 1; *//0:http0.9 1:http1.0 2:http1.1*
51. *//用于父子进程通信的管道*
52. int mypipe[2];
53. *//存放目标服务器的网络地址*
54. char host[MAXHOSTNAMELEN];
56. *//存放请求报文的字节流*
57. #define REQUEST\_SIZE 2048
58. char request[REQUEST\_SIZE];


62. *//构造长选项与短选项的对应*
63. static const struct option long\_options[]=
64. {
65. {"force",no\_argument,&force,1},
66. {"reload",no\_argument,&force\_reload,1},
67. {"time",required\_argument,NULL,'t'},
68. {"help",no\_argument,NULL,'?'},
69. {"http09",no\_argument,NULL,9},
70. {"http10",no\_argument,NULL,1},
71. {"http11",no\_argument,NULL,2},
72. {"get",no\_argument,&method,METHOD\_GET},
73. {"head",no\_argument,&method,METHOD\_HEAD},
74. {"options",no\_argument,&method,METHOD\_OPTIONS},
75. {"trace",no\_argument,&method,METHOD\_TRACE},
76. {"version",no\_argument,NULL,'V'},
77. {"proxy",required\_argument,NULL,'p'},
78. {"clients",required\_argument,NULL,'c'},
79. {NULL,0,NULL,0}
80. };
82. int speed = 0;
83. int failed = 0;
84. long long bytes = 0;
85. int timeout = 0;
86. void build\_request(const char\* url);
87. static int bench();
88. static void alarm\_handler(int signal);
89. void benchcore(const char\* host,const int port,const char\* req);
91. int main(int argc,char\* argv[])
92. {
93. int opt = 0;
94. int options\_index = 0;
95. char\* tmp = NULL;
97. *//首先进行命令行参数的处理*
98. *//1.没有输入选项*
99. if(argc == 1)
100. {
101. usage();
102. return 1;
103. }
105. *//2.有输入选项则一个一个解析*
106. while((opt = getopt\_long(argc,argv,"frt:p:c:?V912",long\_options,&options\_index)) != EOF)
107. {
108. switch(opt)
109. {
110. case 'f':
111. force = 1;
112. break;
113. case 'r':
114. force\_reload = 1;
115. break;
116. case '9':
117. http = 0;
118. break;
119. case '1':
120. http = 1;
121. break;
122. case '2':
123. http = 2;
124. break;
125. case 'V':
126. printf("WebBench 1.5 covered by fh\n");
127. exit(0);
129. case 't':
130. benchtime = atoi(optarg); *//optarg指向选项后的参数*
131. break;
132. case 'c':
133. clients = atoi(optarg); *//与上同*
134. break;
135. case 'p': *//使用代理服务器，则设置其代理网络号和端口号，格式：-p server:port*
136. tmp = strrchr(optarg,':'); *//查找':'在optarg中最后出现的位置*
137. proxyhost = optarg; *//*
139. if(tmp == NULL) *//说明没有端口号*
140. {
141. break;
142. }
143. if(tmp == optarg) *//端口号在optarg最开头,说明缺失主机名*
144. {
145. fprintf(stderr,"选项参数错误，代理服务器 %s:缺失主机名",optarg);
146. return 2;
147. }
148. if(tmp == optarg + strlen(optarg)-1) *//':'在optarg末位，说明缺少端口号*
149. {
150. fprintf(stderr,"选项参数错我，代理服务器 %s 缺少端口号",optarg);
151. return 2;
152. }
154. \*tmp = '\0'; *//将optarg从':'开始截断*
155. proxyport = atoi(tmp+1); *//把代理服务器端口号设置好*
156. break;
158. case '?':
159. usage();
160. exit(0);
161. break;
163. default:
164. usage();
165. return 2;
166. break;
167. }
168. }
170. *//选项参数解析完毕后，刚好是读到URL，此时argv[optind]指向URL*
172. if(optind == argc) *//这样说明没有输入URL，不明白的话自己写一条命令行看看*
173. {
174. fprintf(stderr,"缺少URL参数\n");
175. usage();
176. return 2;
177. }
179. if(benchtime == 0)
180. benchtime = 30;
182. fprintf(stderr,"webbench: 一款轻巧的网站测压工具 1.5 covered by fh\nGPL Open Source Software\n");
184. *//OK,我们解析完命令行后，首先先构造http请求报文*
185. build\_request(argv[optind]); *//参数当然是URL*
187. *//请求报文构造好了*
188. *//开始测压*
189. printf("\n测试中：\n");
191. switch(method)
192. {
193. case METHOD\_OPTIONS:
194. printf("OPTIONS");
195. break;
197. case METHOD\_HEAD:
198. printf("HEAD");
199. break;
201. case METHOD\_TRACE:
202. printf("TRACE");
203. break;
205. case METHOD\_GET:
207. default:
208. printf("GET");
209. break;
210. }
212. printf(" %s",argv[optind]);
214. switch(http)
215. {
216. case 0:
217. printf("(使用 HTTP/0.9)");
218. break;
220. case 1:
221. printf("(使用 HTTP/1.0)");
222. break;
224. case 2:
225. printf("(使用 HTTP/1.1)");
226. break;
227. }
229. printf("\n");
231. printf("%d 个客户端",clients);
233. printf(",%d s",benchtime);
235. if(force)
236. printf(",选择提前关闭连接");
238. if(proxyhost != NULL)
239. printf(",经由代理服务器 %s:%d ",proxyhost,proxyport);
241. if(force\_reload)
242. printf(",选择无缓存");
244. printf("\n"); *//换行不能少！库函数是默认行缓冲，子进程会复制整个缓冲区，*
245. *//若不换行刷新缓冲区，子进程会把缓冲区的的也打出来！*
246. *//而换行后缓冲区就刷新了，子进程的标准库函数的那块缓冲区就不会有前面的这些了*
248. *//真正开始压力测试*
249. return bench();
250. }
252. void build\_request(const char\* url)
253. {
254. char tmp[10];
255. int i = 0;
257. bzero(host,MAXHOSTNAMELEN);
258. bzero(request,REQUEST\_SIZE);
260. *//缓存和代理都是http1.0后才有的*
261. *//无缓存和代理都要在http1.0以上才能使用*
262. *//因此这里要处理一下，不然可能会出问题*
263. if(force\_reload && proxyhost != NULL && http < 1)
264. http = 1;
265. *//HEAD请求是http1.0后才有*
266. if(method == METHOD\_HEAD && http < 1)
267. http = 1;
268. *//OPTIONS和TRACE都是http1.1才有*
269. if(method == METHOD\_OPTIONS && http < 2)
270. http = 2;
271. if(method == METHOD\_TRACE && http < 2)
272. http = 2;
274. *//开始填写http请求*
275. *//请求行*
276. *//填写请求方法*
277. switch(method)
278. {
279. case METHOD\_HEAD:
280. strcpy(request,"HEAD");
281. break;
282. case METHOD\_OPTIONS:
283. strcpy(request,"OPTIONS");
284. break;
285. case METHOD\_TRACE:
286. strcpy(request,"TRACE");
287. break;
288. default:
289. case METHOD\_GET:
290. strcpy(request,"GET");
291. }
293. strcat(request," ");
295. *//判断URL的合法性*
296. *//1.URL中没有"://"*
297. if(strstr(url,"://") == NULL)
298. {
299. fprintf(stderr,"\n%s:是一个不合法的URL\n",url);
300. exit(2);
301. }
303. *//2.URL过长*
304. if(strlen(url) > 1500)
305. {
306. fprintf(stderr,"URL 长度过过长\n");
307. exit(2);
308. }
310. *//3.没有代理服务器却填写错误*
311. if(proxyhost == NULL) *//若无代理*
312. {
313. if(strncasecmp("http://",url,7) != 0) *//忽略大小写比较前7位*
314. {
315. fprintf(stderr,"\nurl无法解析，是否需要但没有选择使用代理服务器的选项？\n");
316. usage();
317. exit(2);
318. }
319. }
321. *//定位url中主机名开始的位置*
322. *//比如 http://www.xxx.com/*
323. i = strstr(url,"://") - url + 3;
325. *//4.在主机名开始的位置找是否有'/'，若没有则非法*
326. if(strchr(url + i,'/') == NULL)
327. {
328. fprintf(stderr,"\nURL非法：主机名没有以'/'结尾\n");
329. exit(2);
330. }
332. *//判断完URL合法性后继续填写URL到请求行*
334. *//无代理时*
335. if(proxyhost == NULL)
336. {
337. *//有端口号时，填写端口号*
338. if(index(url+i,':') != NULL && index(url,':') < index(url,'/'))
339. {
340. *//设置域名或IP*
341. strncpy(host,url+i,strchr(url+i,':') - url - i);
343. bzero(tmp,10);
344. strncpy(tmp,index(url+i,':')+1,strchr(url+i,'/') - index(url+i,':')-1);
346. *//设置端口号*
347. proxyport = atoi(tmp);
348. *//避免写了':'却没写端口号*
349. if(proxyport == 0)
350. proxyport = 80;
351. }
352. else *//无端口号*
353. {
354. strncpy(host,url+i,strcspn(url+i,"/")); *//找到url+i到第一个”/"之间的字符个数*
355. }
356. }
357. else *//有代理服务器就简单了，直接填就行，不用自己处理*
358. {
359. strcat(request,url);
360. }
362. *//填写http协议版本到请求行*
363. if(http == 1)
364. strcat(request," HTTP/1.0");
365. if(http == 2)
366. strcat(request," HTTP/1.1");
368. strcat(request,"\r\n");
370. *//请求报头*
372. if(http > 0)
373. strcat(request,"User-Agent:WebBench 1.5\r\n");
375. *//填写域名或IP*
376. if(proxyhost == NULL && http > 0)
377. {
378. strcat(request,"Host: ");
379. strcat(request,host);
380. strcat(request,"\r\n");
381. }
383. *//若选择强制重新加载，则填写无缓存*
384. if(force\_reload && proxyhost != NULL)
385. {
386. strcat(request,"Pragma: no-cache\r\n");
387. }
389. *//我们目的是构造请求给网站，不需要传输任何内容，当然不必用长连接*
390. *//否则太多的连接维护会造成太大的消耗，大大降低可构造的请求数与客户端数*
391. *//http1.1后是默认keep-alive的*
392. if(http > 1)
393. strcat(request,"Connection: close\r\n");
395. *//填入空行后就构造完成了*
396. if(http > 0)
397. strcat(request,"\r\n");
398. }
400. *//父进程的作用：创建子进程，读子进程测试到的数据，然后处理*
401. static int bench()
402. {
403. int i = 0,j = 0;
404. long long k = 0;
405. pid\_t pid = 0;
406. FILE\* f = NULL;
408. *//尝试建立连接一次*
409. i = Socket(proxyhost == NULL?host:proxyhost,proxyport);
411. if(i < 0)
412. {
413. fprintf(stderr,"\n连接服务器失败，中断测试\n");
414. return 3;
415. }
417. close(i);*//尝试连接成功了，关闭该连接*
419. *//建立父子进程通信的管道*
420. if(pipe(mypipe))
421. {
422. perror("通信管道建立失败");
423. return 3;
424. }
426. *//让子进程去测试，建立多少个子进程进行连接由参数clients决定*
427. for(i = 0;i < clients;i++)
428. {
429. pid = fork();
430. if(pid <= 0)
431. {
432. sleep(1);
433. break; *//失败或者子进程都结束循环，否则该子进程可能继续fork了，显然不可以*
434. }
435. }
437. *//处理fork失败的情况*
438. if(pid < 0)
439. {
440. fprintf(stderr,"第 %d 子进程创建失败",i);
441. perror("创建子进程失败");
442. return 3;
443. }
445. *//子进程执行流*
446. if(pid == 0)
447. {
448. *//由子进程来发出请求报文*
449. benchcore(proxyhost == NULL?host : proxyhost,proxyport,request);
451. *//子进程获得管道写端的文件指针*
452. f = fdopen(mypipe[1],"w");
453. if(f == NULL)
454. {
455. perror("管道写端打开失败");
456. return 3;
457. }
459. *//向管道中写入该子进程在一定时间内请求成功的次数*
460. *//失败的次数*
461. *//读取到的服务器回复的总字节数*
463. fprintf(f,"%d %d %lld\n",speed,failed,bytes);
464. fclose(f); *//关闭写端*
465. return 0;
466. }
467. else
468. {
469. *//子进程获得管道读端的文件指针*
470. f = fdopen(mypipe[0],"r");
472. if(f == NULL)
473. {
474. perror("管道读端打开失败");
475. return 3;
476. }
478. *//fopen标准IO函数是自带缓冲区的，*
479. *//我们的输入数据非常短，并且要求数据要及时，*
480. *//因此没有缓冲是最合适的*
481. *//我们不需要缓冲区*
482. *//因此把缓冲类型设置为\_IONBF*
483. setvbuf(f,NULL,\_IONBF,0);
485. speed = 0; *//连接成功的总次数，后面除以时间可以得到速度*
486. failed = 0; *//失败的请求数*
487. bytes = 0; *//服务器回复的总字节数*
489. *//唯一的父进程不停的读*
490. while(1)
491. {
492. pid = fscanf(f,"%d %d %lld",&i,&j,&k);*//得到成功读入的参数个数*
493. if(pid < 3)
494. {
495. fprintf(stderr,"某个子进程死亡\n");
496. break;
497. }
499. speed += i;
500. failed += j;
501. bytes += k;
503. *//我们创建了clients,正常情况下要读clients次*
504. if(--clients == 0)
505. break;
506. }
508. fclose(f);
510. *//统计处理结果*
511. printf("\n速度：%d pages/min,%d bytes/s.\n请求：%d 成功,%d 失败\n",\
512. (int)((speed+failed)/(benchtime/60.0f)),\
513. (int)(bytes/(float)benchtime),\
514. speed,failed);
515. }
517. return i;
518. }
520. *//闹钟信号处理函数*
521. static void alarm\_handler(int signal)
522. {
523. timeout = 1;
524. }
525. *//子进程真正的向服务器发出请求报文并以其得到此期间的相关数据*
526. void benchcore(const char\* host,const int port,const char\* req)
527. {
528. int rlen;
529. char buf[1500];
530. int s,i;
531. struct sigaction sa;
533. *//安装闹钟信号的处理函数*
534. sa.sa\_handler = alarm\_handler;
535. sa.sa\_flags = 0;
536. if(sigaction(SIGALRM,&sa,NULL))
537. exit(3);
539. *//设置闹钟函数*
540. alarm(benchtime);
542. rlen = strlen(req);
544. nexttry:
545. while(1)
546. {
547. *//只有在收到闹钟信号后会使 time = 1*
548. *//即该子进程的工作结束了*
549. if(timeout)
550. {
551. if(failed > 0)
552. {
553. failed--;
554. }
555. return;
556. }
558. *//建立到目标网站服务器的tcp连接，发送http请求*
559. s = Socket(host,port);
560. if(s < 0)
561. {
562. failed++; *//连接失败*
563. continue;
564. }
566. *//发送请求报文*
567. if(rlen != write(s,req,rlen))
568. {
569. failed++;
570. close(s); *//写失败了也不能忘了关闭套接字*
571. continue;
572. }
574. *//http0.9的特殊处理*
575. *//因为http0.9是在服务器回复后自动断开连接的，不keep-alive*
576. *//在此可以提前先彻底关闭套接字的写的一半，如果失败了那么肯定是个不正常的状态,*
577. *//如果关闭成功则继续往后，因为可能还有需要接收服务器的恢复内容*
578. *//但是写这一半是一定可以关闭了，作为客户端进程上不需要再写了*
579. *//因此我们主动破坏套接字的写端，但是这不是关闭套接字，关闭还是得close*
580. *//事实上，关闭写端后，服务器没写完的数据也不会再写了，这个就不考虑了*
581. if(http == 0)
582. {
583. if(shutdown(s,1))
584. {
585. failed++;
586. close(s);
587. continue;
588. }
589. }
591. *//-f没有设置时默认等待服务器的回复*
592. if(force == 0)
593. {
594. while(1)
595. {
596. if(timeout)
597. break;
598. i = read(s,buf,1500); *//读服务器发回的数据到buf中*
600. if(i < 0)
601. {
602. failed++; *//读失败*
603. close(s); *//失败后一定要关闭套接字，不然失败个数多时会严重浪费资源*
604. goto nexttry; *//这次失败了那么继续下一次连接，与发出请求*
605. }
606. else
607. {
608. if(i == 0) *//读完了*
609. break;
610. else
611. bytes += i; *//统计服务器回复的字节数*
612. }
613. }
614. }
615. if(close(s))
616. {
617. failed++;
618. continue;
619. }
620. speed++;
621. }
622. }

[源码](https://github.com/l-f-h/webbench-1.5)

git clone <https://github.com/l-f-h/webbench-1.5.git>

make&&make clean

 最后想说一说关于webbench，20年前左右一位大佬写了最初的开源版本，后来不断有人去更新，1.5版本是多进程版本，受益于多进程，其优点是比较稳定，各进程毕竟不会互相影响，父子进程间通信时基本不会出Bug，也不需要加任何锁，但是缺点个人感觉也是比较明显的。

首先，从代码的角度来讲，我自己研究了大半天才把整个理清楚，因为最初的版本用了太多的全局变量，实在是对于阅读源码的人不友好，但是在实现的时候就还好了，毕竟需要什么就在全局加什么，反而思路清楚。（似乎这个问题也不好避免）

其次，再者就是多线程来做肯定性能要好得多，毕竟进程的开销实在是不小，开2g内存跑的话差不多3000个进程就不行了，但是现在都动不动就8g起，似乎也不太在意这个问题，但是线程无论如何都会好很多的。

最后就是一个细节问题，在统计服务器回复的字节数时个人认为应该检测其是否回复了正确信息（2xx）

过一段时间搞一个多线程版本，顺便解决一下一些问题。

前段时间测自己的web服务器发现我漏了对于http1.1没有填充好报文，当时自己没有测到，希望各位看到这篇博客的人能够注意下这一点！