ProxyLab 实验报告

李梓童 2017202121

➢ 遇到的问题及解决方案

- PART I 主要工作流程如下：服务器端接受请求，解析（举例）GET http://www.ruc.edu /index.html HTTP/1.1，把它转换为 GET /index.html HTTP/1.0, 同时获取 host、port，proxy 自己作为 client 向目标发送 HTTP 1.0 请求。
  header 部分，修改下面 4 个值：Host: www.ruc.edu; User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:10.0.3) Gecko/20120305 Firefox/10.0.3; Connection: close; Proxy-Connection: close
  转发后，把接受到的信息再作为 proxy 的输出，向原客户端转发。
  代码框架上参考了 tiny。如 read_request_header、clienterror 实际都是 tiny 的扩充。
- 对 SIGPIPE 信号的处理：调用 Signal(SIGPIPE, SIG_IGN)，将 SIGPIPE 忽略掉。
  如果尝试两次发送数据到一个已经被关闭的 socket 上，kernel 会发送 SIGPIPE 信号，这在默认情况下会导致当前程序终止，这样不符合 server 持久性要求，所以选择忽略。另外，往已关闭的 pipe 里写会使 errno 等于 EPIPE，而往已关闭的 pipe 里读会使 errno 等于 ECONNRESET，对于这两种情况也不应使程序终止，故在错误处理函数中将 exit 换成了 return。
- 对各种 uri 的处理要考虑多种情况。例如对于前面带“http”前缀的 uri，如果有前缀就要判断这个前缀是不是正确的。根据“:”“/”来判断 port 和 host。代码针对第一个字符是“:”的情况进行了排除，如果第一个字符是“/”则继续进行。这个函数对错误情况的考虑应该可以再细致些。
- 一些容易出 bug 和做了优化的地方：
  is_cache_hit 函数中，在比较 cache_block 信息、确定是否命中的过程中，初始版本是每个 cache_block 的 tag 和目标 command 进行全部比对，后来先比较 tag 和 command 的长度，如果不同直接跳过，提高了比较效率。
  find_replace_block 函数中，需要分两种情况考虑缓存中的 block。一种是还没有使用过，recently_used==0 的情况，可以直接返回替换 block；另一种是当每一个块都使用过了，选择其中 recently_used 最小的值。两种情况不太好合并，最后还是分开来处理。同样，在 rid_block（对块进行驱逐）函数中，也需要考虑到 recently_used!=0，如果这个 block 没有使用过则不需要驱逐，这里也容易忽略。
  Thread 函数里要注意 detach self。
  Doit 函数里，对 cache 大小限制做了优化。例如，只有当 object_size < MAX_OBJECT_SIZE 的时候，才将该 block 放入缓存。用 ptr_in_buf 指针指向真实 cache_block 的 object，读取的时候记录读的 pos，如果超过 MAX_OBJECT_SIZE 则回归初始位置，用这个方式来限制 object 大小。
  做 Cache 的时候用到了很多指针，容易发生引用、解指的错误，需要特别注意。

➢ 使用的多线程解决方案

- 在 main 函数的 while 循环里加入 pthread_create()，制造线程 thread。
  线程 thread 主要执行下面这些功能：detach，否则线程结束后不会释放资源直到有别的线程 join；free 参数指针（这步容易漏）；doit()，真正工作的主体；close socket。

Doit 函数从单线程变为多线程，主要工作的程序没有变，但是后面做到缓存算法、读取写入缓存的时候，要考虑对共享变量 cache 的改动，也就需要用锁，这在后面会作出说明。

➢ 使用的缓存算法

- 不严格的 LRU 算法。
  在寻找 client 需要的内容的时候，如果找到了 cache，则直接从 cache 里读取数据，读取的时候 update 缓存结构里的 recently_used，作为 LRU 策略的依据；如果没有，就另外开一个 cache_block。当 cache 放不下这个 cache_block 的时候，选择 recently_used 数值最小的 block 进行驱逐。更新 recently_used 的时候并没有加锁，读缓存的时候，如果有别的线程也在更新同一个 block，那么就看调度结果了。

➢ 关于缓存的多线程读写问题的解决方案

- 多线程读取写入缓存的时候，选择读者优先的读写者模式。参考课堂 ppt 上的内容，只有当 readcnt==0（当前没有读者了）的时候，我们才开始写。写入的时候，整个写入部分内容加了一把大锁，从查找 least used block 到把新的 block 写入 cache，都属于写的部分。

➢ 反思总结

- Proxylab 的主要内容：单线程代理->实现并发->共享进程之间的数据，缓存；需要用到 web server、并行编程的有关知识。
- 感觉 proxylab 针对不同需求要考虑的情况较之 Fslab 要少，webserver 的工作需求在教材里 tiny 已经给了个 demo，rio 读写包的健壮性也减少了意外发生的情况，因此 proxylab 主要要考虑来自 cache 的意外错误和来自用户不合法输入内容的意外错误。对于来自用户不合法输入的错误，我们可以通过类似于正则表达式解析的思想来判断它是否符合要求，重新包装后再发送给真正的 server。对于 cache 的错误，主要限制 cache 的大小等，要从 cache 条目的构造中入手，另外，如前所述，对于 cache 里的指针要多留意。对于 proxy 持久性的维护，粗略的处理方式是将所有的 error 处理结果变成 return 而不是 exit，这里不能确定除了 EPIPE、ECONNRESET 还会有什么错误信号，所以只将这两个错误的导致结果改成 return。

➢ 参考资料

《socket 通信中 EPIPE 错误》（CDSN）
《CSAPP》chapter 10 - 12