最新版Web服务器项目详解 - 07 定时器处理非活动连接（上）

原创 互联网猿 两猿社 2020-04-22 11:50

基础知识

非活跃，是指客户端（这里是浏览器）与服务器端建立连接后，长时间不交换数据，一直占用服务器端的文件描述符，导致连接资源的浪费。

定时事件，是指固定一段时间之后触发某段代码，由该段代码处理一个事件，如从内核事件表删除事件，并关闭文件描述符，释放连接资源。

定时器，是指利用结构体或其他形式，将多种定时事件进行封装起来。具体的，这里只涉及一种定时事件，即定期检测非活跃连接，这里将该定时事件与连接资源封装为一个结构体定时器。

定时器容器，是指使用某种容器类数据结构，将上述多个定时器组合起来，便于对定时事件统一管理。具体的，项目中使用升序链表将所有定时器串联组织起来。

整体概述

本项目中，服务器主循环为每一个连接创建一个定时器，并对每个连接进行定时。另外，利用升序时间链表容器将所有定时器串联起来，若主循环接收到定时通知，则在链表中依次执行定时任务。

Linux 下提供了三种定时的方法:

- socket选项SO_RECVTIMEO和SO_SNDTIMEO
- SIGALRM信号
- I/O复用系统调用的超时参数

三种方法没有一劳永逸的应用场景，也没有绝对的优劣。由于项目中使用的是 SIGALRM 信号，这里仅对其进行介绍，另外两种方法可以查阅游双的 Linux高性能服务器编程 第11章 定时器 。

具体的，利用 alarm 函数周期性地触发 SIGALRM 信号，信号处理函数利用管道通知主循环，主循环接收到该信号后对升序链表上所有定时器进行处理，若该段时间内没有交换数据，则将该连接关闭，释放所占用的资源。

从上面的简要描述中，可以看出定时器处理非活动连接模块，主要分为两部分，其一为定时方法与信号通知流程，其二为定时器及其容器设计与定时任务的处理。

本文内容

本篇将介绍定时方法与信号通知流程，具体的涉及到基础API、信号通知流程和代码实现。

基础API，描述 sigaction 结构体、 sigaction 函数、 sigfillset 函数、 SIGALRM 信号、 SIGTERM 信号、 alarm 函数、 socketpair 函数、 send 函数。

信号通知流程，介绍统一事件源和信号处理机制。

代码实现，结合代码对信号处理函数的设计与使用进行详解。

基础API

为了更好的源码阅读体验，这里提前对代码中使用的一些API进行简要介绍，更丰富的用法可以自行查阅资料。

sigaction结构体

```
1  struct sigaction {
2      void (*sa_handler)(int);
3      void (*sa_sigaction)(int, siginfo_t *, void *);
4      sigset_t sa_mask;
5      int sa_flags;
6      void (*sa_restorer)(void);
7  }
```

- sa_handler是一个函数指针，指向信号处理函数
- sa_sigaction同样是信号处理函数，有三个参数，可以获得关于信号更详细的信息
- sa_mask用来指定在信号处理函数执行期间需要被屏蔽的信号
- sa_flags用于指定信号处理的行为
    - SA_RESTART，使被信号打断的系统调用自动重新发起
    - SA_NOCLDSTOP，使父进程在它的子进程暂停或继续运行时不会收到 SIGCHLD 信号
    - SA_NOCLDWAIT，使父进程在它的子进程退出时不会收到 SIGCHLD 信号，这时子进程如果退出也不会成为僵尸进程
    - SA_NODEFER，使对信号的屏蔽无效，即在信号处理函数执行期间仍能发出这个信号

    - SA_RESETHAND，信号处理之后重新设置为默认的处理方式
    - SA_SIGINFO，使用 sa_sigaction 成员而不是 sa_handler 作为信号处理函数
- sa_restorer一般不使用

sigaction函数

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

- signum表示操作的信号。
- act表示对信号设置新的处理方式。
- oldact表示信号原来的处理方式。
- 返回值，0 表示成功，-1 表示有错误发生。

sigfillset函数

```
#include <signal.h>

int sigfillset(sigset_t *set);
```

用来将参数set信号集初始化，然后把所有的信号加入到此信号集里。

SIGALRM、SIGTERM信号

```
#define SIGALRM  14      //由alarm系统调用产生timer时钟信号
#define SIGTERM  15      //终端发送的终止信号
```

alarm函数

```
#include <unistd.h>;

unsigned int alarm(unsigned int seconds);
```

设置信号传送闹钟，即用来设置信号SIGALRM在经过参数seconds秒数后发送给目前的进程。如果未设置信号SIGALRM的处理函数，那么alarm()默认处理终止进程.

socketpair函数

在linux下，使用socketpair函数能够创建一对套接字进行通信，项目中使用管道通信。

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv[2]);
```

- domain表示协议族，PF_UNIX或者AF_UNIX

- type表示协议，可以是SOCK_STREAM或者SOCK_DGRAM，SOCK_STREAM基于TCP，SOCK_DGRAM基于UDP
- protocol表示类型，只能为0
- sv[2]表示套节字柄对，该两个句柄作用相同，均能进行读写双向操作
- 返回结果， 0为创建成功，-1为创建失败

send函数

```
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

当套接字发送缓冲区变满时，send通常会阻塞，除非套接字设置为非阻塞模式，当缓冲区变满时，返回EAGAIN或者EWOULDBLOCK错误，此时可以调用select函数来监视何时可以发送数据。

信号通知流程

Linux下的信号采用的异步处理机制，信号处理函数和当前进程是两条不同的执行路线。具体的，当进程收到信号时，操作系统会中断进程当前的正常流程，转而进入信号处理函数执行操作，完成后再返回中断的地方继续执行。

为避免信号竞态现象发生，信号处理期间系统不会再次触发它。所以，为确保该信号不被屏蔽太久，信号处理函数需要尽可能快地执行完毕。

一般的信号处理函数需要处理该信号对应的逻辑，当该逻辑比较复杂时，信号处理函数执行时间过长，会导致信号屏蔽太久。

这里的解决方案是，信号处理函数仅仅发送信号通知程序主循环，将信号对应的处理逻辑放在程序主循环中，由主循环执行信号对应的逻辑代码。

统一事件源

统一事件源，是指将信号事件与其他事件一样被处理。

具体的，信号处理函数使用管道将信号传递给主循环，信号处理函数往管道的写端写入信号值，主循环则从管道的读端读出信号值，使用I/O复用系统调用来监听管道读端的可读事件，这样信号事件与其他文件描述符都可以通过epoll来监测，从而实现统一处理。

信号处理机制

每个进程之中，都有存着一个表，里面存着每种信号所代表的含义，内核通过设置表项中每一个位来标识对应的信号类型。

- 信号的接收
    - 接收信号的任务是由内核代理的，当内核接收到信号后，会将其放到对应进程的信号队列中，同时向进程发送一个中断，使其陷入内核态。注意，此时信号还只是在队列中，对进程来说暂时是不知道有信号到来的。
- 信号的检测
    - 进程从内核态返回到用户态前进行信号检测
    - 进程在内核态中，从睡眠状态被唤醒的时候进行信号检测
    - 进程陷入内核态后，有两种场景会对信号进行检测：
    - 当发现有新信号时，便会进入下一步，信号的处理。
- 信号的处理
    - (**内核**)信号处理函数是运行在用户态的，调用处理函数前，内核会将当前内核栈的内容备份拷贝到用户栈上，并且修改指令寄存器（eip）将其指向信号处理函数。
    - (**用户**)接下来进程返回到用户态中，执行相应的信号处理函数。
    - (**内核**)信号处理函数执行完成后，还需要返回内核态，检查是否还有其它信号未处理。
    - (**用户**)如果所有信号都处理完成，就会将内核栈恢复（从用户栈的备份拷贝回来），同时恢复指令寄存器（eip）将其指向中断前的运行位置，最后回到用户态继续执行进程。

至此，一个完整的信号处理流程便结束了，如果同时有多个信号到达，上面的处理流程会在第2步和第3步骤间重复进行。

代码分析

信号处理函数

自定义信号处理函数，创建sigaction结构体变量，设置信号函数。

```
1  //信号处理函数
2  void sig_handler(int sig)
3  {
```

```
    //为保证函数的可重入性，保留原来的errno
    //可重入性表示中断后再次进入该函数，环境变量与之前相同，不会丢失数据
    int save_errno = errno;
    int msg = sig;

    //将信号值从管道写端写入，传输字符类型，而非整型
    send(pipefd[1], (char *)&msg, 1, 0);

    //将原来的errno赋值为当前的errno
    errno = save_errno;
}
```

信号处理函数中仅仅通过管道发送信号值，不处理信号对应的逻辑，缩短异步执行时间，减少对主程序的影响。

```
//设置信号函数
void addsig(int sig, void(handler)(int), bool restart = true)
{
    //创建sigaction结构体变量
    struct sigaction sa;
    memset(&sa, '\0', sizeof(sa));

    //信号处理函数中仅仅发送信号值，不做对应逻辑处理
    sa.sa_handler = handler;
    if (restart)
        sa.sa_flags |= SA_RESTART;
    //将所有信号添加到信号集中
    sigfillset(&sa.sa_mask);

    //执行sigaction函数
    assert(sigaction(sig, &sa, NULL) != -1);
}
```

项目中设置信号函数，仅关注SIGTERM和SIGALRM两个信号。

信号通知逻辑

- 创建管道，其中管道写端写入信号值，管道读端通过I/O复用系统监测读事件
- 设置信号处理函数SIGALRM（时间到了触发）和SIGTERM（kill会触发，Ctrl+C）
  - 通过struct sigaction结构体和sigaction函数注册信号捕捉函数
  - 在结构体的handler参数设置信号处理函数，具体的，从管道写端写入信号的名字
- 利用I/O复用系统监听管道读端文件描述符的可读事件
- 信息值传递给主循环，主循环再根据接收到的信号值执行目标信号对应的逻辑代码

代码分析

```
//创建管道套接字
ret = socketpair(PF_UNIX, SOCK_STREAM, 0, pipefd);
assert(ret != -1);

//设置管道写端为非阻塞，为什么写端要非阻塞？
setnonblocking(pipefd[1]);

//设置管道读端为ET非阻塞
addfd(epollfd, pipefd[0], false);
```

```
10
11  //传递给主循环的信号值，这里只关注SIGALRM和SIGTERM
12  addsig(SIGALRM, sig_handler, false);
13  addsig(SIGTERM, sig_handler, false);
14
15  //循环条件
16  bool stop_server = false;
17
18  //超时标志
19  bool timeout = false;
20
21  //每隔TIMESLOT时间触发SIGALRM信号
22  alarm(TIMESLOT);
23
24  while (!stop_server)
25  {
26      //监测发生事件的文件描述符
27      int number = epoll_wait(epollfd, events, MAX_EVENT_NUMBER, -1);
28      if (number < 0 && errno != EINTR)
29      {
30          break;
31      }
32
33      //轮询文件描述符
34      for (int i = 0; i < number; i++)
35      {
36          int sockfd = events[i].data.fd;
37
38          //管道读端对应文件描述符发生读事件
39          if ((sockfd == pipefd[0]) && (events[i].events & EPOLLIN))
40          {
41              int sig;
42              char signals[1024];
43
44              //从管道读端读出信号值，成功返回字节数，失败返回-1
45              //正常情况下，这里的ret返回值总是1，只有14和15两个ASCII码对应的字符
46              ret = recv(pipefd[0], signals, sizeof(signals), 0);
47              if (ret == -1)
48              {
49                  // handle the error
50                  continue;
51              }
52              else if (ret == 0)
53              {
54                  continue;
55              }
56              else
57              {
58                  //处理信号值对应的逻辑
59                  for (int i = 0; i < ret; ++i)
60                  {
61                      //这里面明明是字符
62                      switch (signals[i])
63                      {
64                      //这里是整型
65                      case SIGALRM:
66                      {
67                          timeout = true;
68                          break;
69                      }
70                      case SIGTERM:
71                      {
72                          stop_server = true;
73                      }
74                      }
75                  }
76              }
```

```
77              }
78          }
79      }
```

为什么管道写端要非阻塞?

send是将信息发送给套接字缓冲区，如果缓冲区满了，则会阻塞，这时候会进一步增加信号处理函数的执行时间，为此，将其修改为非阻塞。

没有对非阻塞返回值处理，如果阻塞是不是意味着这一次定时事件失效了?

是的，但定时事件是非必须立即处理的事件，可以允许这样的情况发生。

管道传递的是什么类型? switch-case的变量冲突?

信号本身是整型数值，管道中传递的是ASCII码表中整型数值对应的字符。

switch的变量一般为字符或整型，当switch的变量为字符时，case中可以是字符，也可以是字符对应的ASCII码。

如果本文对你有帮助， 阅读原文 star一下服务器项目，我们需要你的星星^_^.

完。