Libevent

1. 事件的底层处理框架
    - 一个函数
2. 消息循环
    - 一个函数
3. 创建事件
    - 不带缓冲区 - event
        - 几个函数
    - 带缓冲区 - bufferevent
        - 几个函数
4. 资源释放
    - 几个函数

libevent - 基本概念

1. libevent是干什么的
   - 开源的库，提高开发效率
     - 封装了socket通信
     - 封装了IO多路转接
   - 精简，专注于网络，性能高
   - 事件驱动
2. libevent库的安装
   - 官方网站：http://libevent.org
   - 源码包下载：
     - 1.4.x -- 适合源码学习
     - 2.x
   - 源码包的安装
     - ./configure
       - --prefix == /usr/xxxxx
       - 检测安装环境
       - 生成makefile
     - make
       - 编译源代码
       - 生成一些库
         - 动态，静态
         - 可执行程序
     - sudo make install
       - 将数据拷贝到对应的目录
       - 如果目录不存在，创建该目录
       - 默认目录
         - /usr/local
           - /usr/local/include
           - /usr/local/bin
           - /usr/local/lib
3. libevent库的使用
   - 编译程序的时候指定 -levent 即可
   - 常用头文件：
     - #include <event2/event.h>
     - #include <event2/listener.h>
4. 示例程序演示
   - hello-world.c

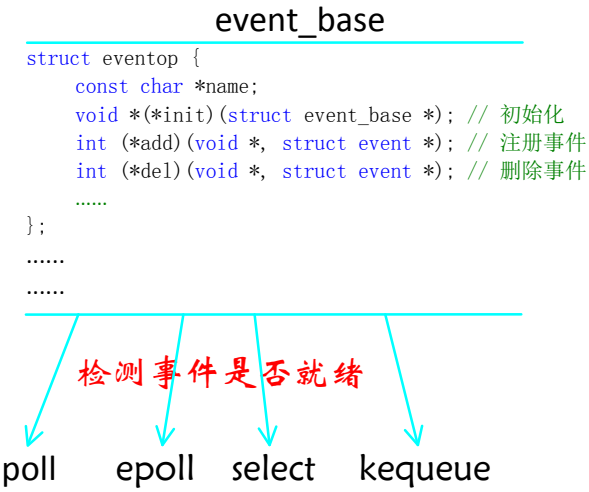
memcached

- 浏览源代码

事件处理框架 - event_base

1. 使用 libevent 函数之前需要分配一个或者多个 event_base 结构体。每个event_base 结构体持有一个事件集合,可以检测以确定哪个事件是激活的。
   - 相当于epoll红黑树的树根
   - 底座
   - 抽象层，完成对event_base的封装
   - 每个 event_base 都有一种用于检测哪种事件已经就绪的 “方法”,或者说后端.

1. 创建根节点
2. 添加监听接节点
3. 检测



2.
- 创建event_base
  - struct event_base* event_base_new(void);
  - 失败返回NULL
- 释放event_base
  - event_base_free(struct event_base* base);
- 循环监听base对应的事件，等待条件满足
  - event_base_dispatch();

3.
- 查看event_base封装的后端
  - const char **event_get_supported_methods(void);
    - char* str[];

    const char * event_base_get_method(
    const struct event_base *base
    );

- event_base和fork
  - 子进程创建成功之后，父进程可以继续使用event_base
  - 子进程中需要继续使用event_base需要重新进程初始化
    - int event_reinit(struct event_base* base);

========= 使用套路

2017年6月7日 10:52

1. 创建一个事件处理框架
2. 创建一事件
3. 事件添加到事件处理框架上
4. 开始事件循环
5. 释放资源
   - 释放event_base
     - event_base_free(struct event_base* base);
   -

流水线

event_base_new(void)
其中有消息循环,需要自己启动

买一头死猪

事件循环 - event_loop

一旦有了一个已经注册了某些事件的 event_base, 就需要让 libevent 等待事件并且通知事件的发生。

```
#define EVLOOP_ONCE                    0x01
```
事件只会被触发一次

事件没有被触发, 阻塞等

```
#define EVLOOP_NONBLOCK                0x02
```
非阻塞 等方式去做事件检测

不关心事件是否被触发了

```
#define EVLOOP_NO_EXIT_ON_EMPTY  0x04
```
没有事件的时候, 也不退出轮询检测

1. int event_base_loop(struct event_base *base, int flags);
   a. 正常退出返回0, 失败返回-1
2. int event_base_dispatch(struct event_base* base);
   - 等同于没有设置标志的 event_base_loop ( )
   - 将一直运行,直到没有已经注册的事件了,或者调用 了 event_base_loopbreak() 或者 event_base_loopexit() 为止。
3. 循环停止

- 返回值: 成功 0, 失败 -1

```
struct timeval {
    long    tv_sec;
    long    tv_usec;
};
```

- 如果 event_base 当前正在执行激活事件的回调 ,它将在执行完当前正在处理的事件后立即退出

```
int event_base_loopexit(
        struct event_base *base,
        const struct timeval *tv
        );
```

- 让event_base 立即退出循环

```
int event_base_loopbreak(struct event_base *base);
```

事件创建 - event

1. 创建新事件

```
#define EV_TIMEOUT      0x01    // 废弃
#define EV_READ         0x02
#define EV_WRITE        0x04
#define EV_SIGNAL       0x08
#define EV_PERSIST      0x10    // 持续触发
#define EV_ET           0x20    // 边沿模式
typedef void (*event_callback_fn)(evutil_socket_t, short, void *);
struct event *event_new(
    struct event_base *base,
    evutil_socket_t fd, // 文件描述符 - int
    short what,
    event_callback_fn cb, // 事件的处理动作
    void *arg
);
```

   - 调用event_new()函数之后，新事件处于已初始化和非未决状态

2. 释放事件

   void event_free(struct event *event);

3. 设置未决事件

   构造事件之后，在将其添加到 event_base 之前实际上是不能对其做任何操作的。使用event_add()将事件添加到event_base，非未决事件 -> 未决事件.

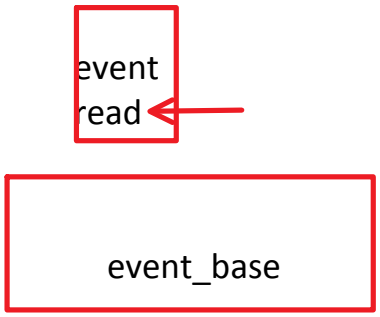
   - int event_add(
         struct event *ev,
         const struct timeval *tv
     );
   - tv:
     - NULL: 事件被触发，对应的回调被调用
     - tv = {0, 100}，如果设置的时间，
       - 在改时间段内检测的事件没被触发，时间到达之后，回调函数还是会被调用
     - 函数调用成功返回0，失败返回-1

4. 设置非未决
   - int event_del(struct event *ev);
     - 对已经初始化的事件调用 event_del()将使其成为非未决和非激活的。如果事件不是未决的或者激活的，调用将没有效果。成功时函数返回 0，失败时返回-1。

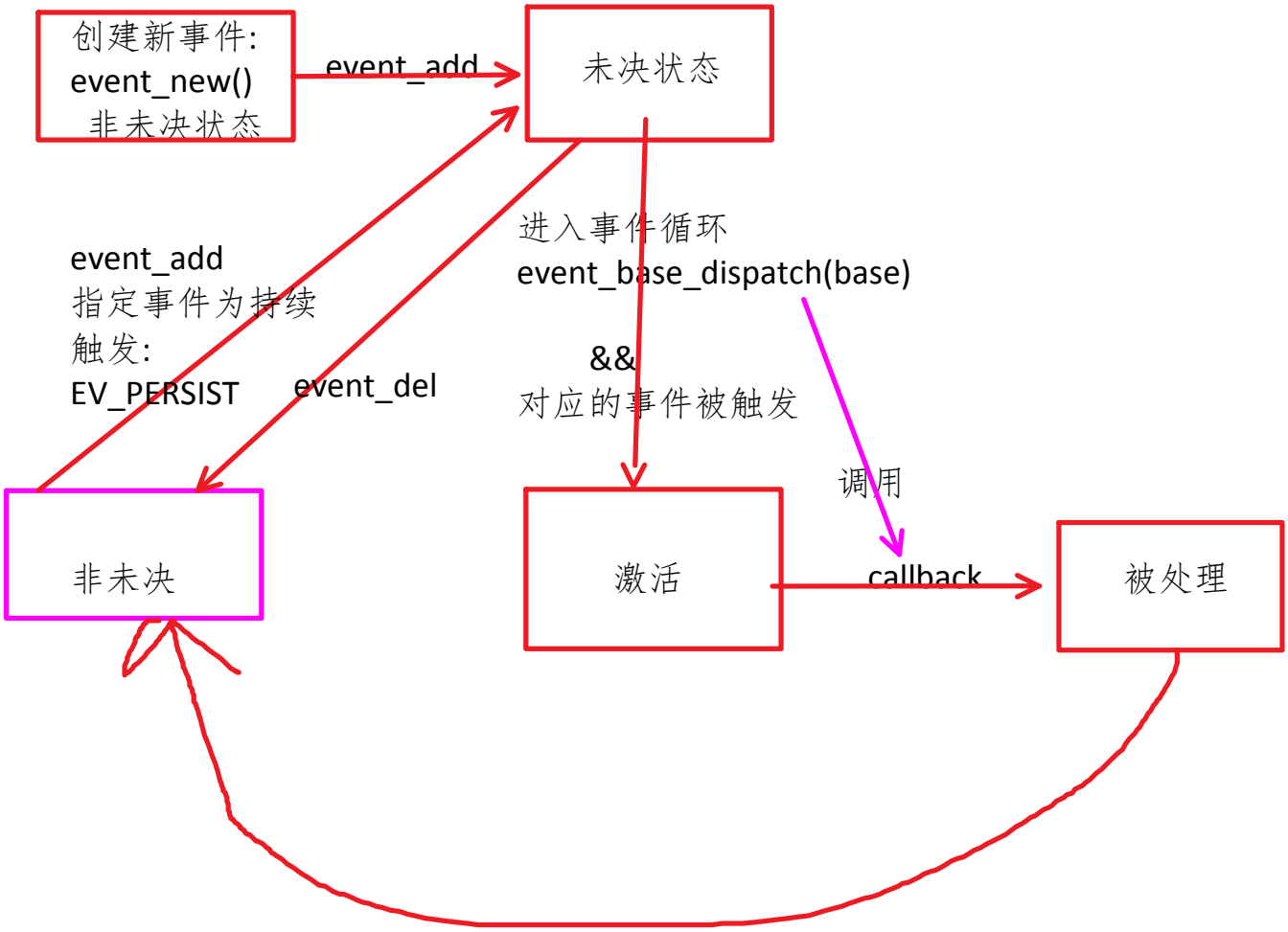
事件的状态转换

fd
- epollin
- epollout

libevent封装了信号相关的操作

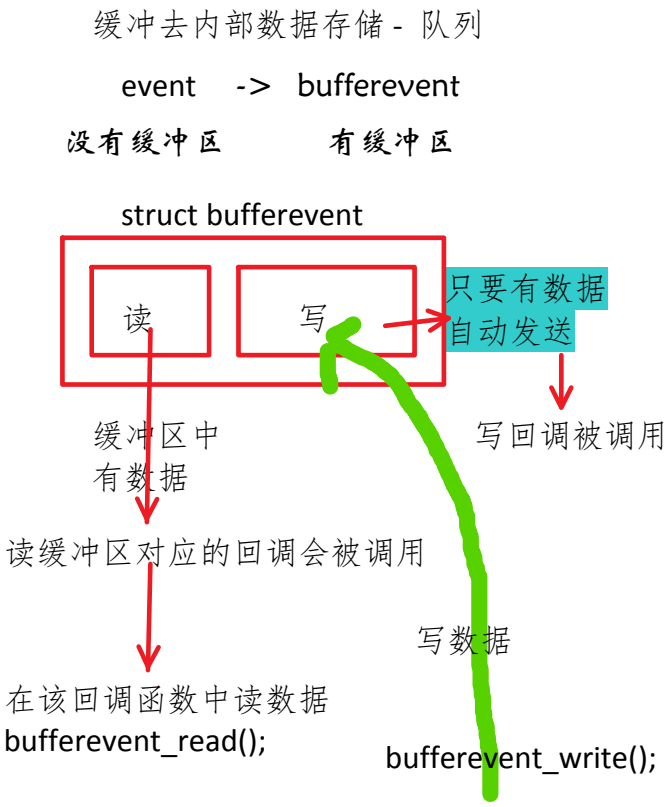


1. 非未决
   - 没有资格被处理
2. 未决:
   - 有资格被处理但是还没有处理


创建新事件:
event_new()
非未决状态
event_add
未决状态
event_add
指定事件为持续
触发:
EV_PERSIST
event_del
进入事件循环
event_base_dispatch(base)
&&
对应的事件被触发
调用
非未决
激活
callback
被处理

数据缓冲区 - Bufferevent

1. event2/bufferevent.h
2. bufferevent 理解:
   - 是libevent为IO缓冲区操作提供的一种通用机制
   - bufferevent 由一个底层的传输端口(如套接字), 一个读取缓冲区和一个写入缓冲区组成。
   - 与通常的事件在底层传输端口已经就绪,可以读取或者写入的时候执行回调不同的是, bufferevent 在读取或者写入了足够量的数据之后调用用户提供的回调。
3. 回调 - 缓冲区对应的操作
   - 每个 bufferevent 有两个数据相关的回调
     - 一个读取回调
       - 从底层传输端口读取了任意量的数据之后会调用读取回调(默认)
     - 一个写入回调
       - 输出缓冲区中足够量的数据被清空到底层传输端口后写入回调会被调用(默认)



使用 bufferevent

1. 创建基于套接字的bufferevent
   - 可以使用 bufferevent_socket_new()创建基于套接字的 bufferevent

```
struct bufferevent *bufferevent_socket_new(
        struct event_base *base,
        evutil_socket_t fd,
        enum bufferevent_options options
);
```

     - options: BEV_OPT_CLOSE_ON_FREE
       - 释放 bufferevent 时关闭底层传输端口。这将关闭底层套接字,释放底层 bufferevent 等
       - 参考手册page53 - bufferevent的选项标志
     - struct bufferevent也是一个 event
     - 成功时函数返回一个 bufferevent,失败则返回 NULL。
2. 在bufferevent上启动链接
   - int bufferevent_socket_connect(

```
        struct bufferevent *bev,
        struct sockaddr *address, - server ip和port
        int addrlen
);
```

     - address 和 addrlen 参数跟标准调用 connect() 的参数相同。如果还没有为bufferevent 设置套接字,调用函数将为其分配一个新的流套接字,并且设置为非阻塞的
     - 如果已经为 bufferevent 设置套接字,调用 bufferevent_socket_connect() 将告知libevent 套接字还未连接,直到连接成功之前不应该对其

进行读取或者写入操作。

- 连接完成之前可以向输出缓冲区添加数据。

3. 释放bufferevent操作

- void bufferevent_free(struct bufferevent *bev);
  - 这个函数释放 bufferevent

4. bufferevent读写缓冲区回调操作

```
typedef void (*bufferevent_data_cb)(
        struct bufferevent *bev,
        void *ctx
);
typedef void (*bufferevent_event_cb)(
        struct bufferevent *bev,
        short events,
        void *ctx
);
```

events参数:

EV_EVENT_READING：读取操作时发生某事件，具体是哪种事件请看其他标志。
BEV_EVENT_WRITING：写入操作时发生某事件，具体是哪种事件请看其他标志。
BEV_EVENT_ERROR：操作时发生错误。关于错误的更多信息，请调 用 EVUTIL_SOCKET_ERROR()。
BEV_EVENT_TIMEOUT：发生超时。
BEV_EVENT_EOF：遇到文件结束指示。
BEV_EVENT_CONNECTED：请求的连接过程已经完成

- 实现客户端的时候可以判断

```
void bufferevent_setcb(
        struct bufferevent *bufev,
        bufferevent_data_cb readcb,
            -- 在读回调中读数据
            -- bufferevent_read()
        bufferevent_data_cb writecb,
            -- NULL
        bufferevent_event_cb eventcb,
          --- NULL
```

```
        void *cbarg
    );
```

5. 禁用、启用缓冲区
    - 禁用之后，对应的回调就不会被调用了

```
void bufferevent_enable(
    struct bufferevent *bufev,
    short events
);
void bufferevent_disable(
    struct bufferevent *bufev,
    short events
);
short bufferevent_get_enabled(
    struct bufferevent *bufev
);
```

可以启用或者禁用 bufferevent 上的 EV_READ、EV_WRITE 或者 EV_READ | EV_WRITE 事件。没有启用读取或者写入事件时，bufferevent 将不会试图进行数据读取或者写入。

6. 操作bufferevent中的数据
    - 向bufferevent的输出缓冲区添加数据
        -
```
int bufferevent_write(
    struct bufferevent *bufev,
    const void *data,
    size_t size
);
```
    - 从bufferevent的输入缓冲区移除数据
        -
```
size_t bufferevent_read(
    struct bufferevent *bufev,
    void *data,
    size_t size
```

```
);
```

链接监听器 - evconnlistener

1. 创建和释放evconnlistener

```
typedef void (*evconnlistener_cb)(
        struct evconnlistener *listener,
        evutil_socket_t sock,
                ◇ 用于通信的文件描述符
        struct sockaddr *addr,
                ◇ 客户端的IP和端口信息
        int len,
        void *ptr
            ◆ 外部传进来的数据
);
```

- struct evconnlistener * evconnlistener_new(
  struct event_base *base,
  evconnlistener_cb cb,
  void *ptr,
  unsigned flags,
  int backlog,
  evutil_socket_t fd
  );
  - 参数flags：
    - 参考手册 - page99-100 [可识别的标志]
      LEV_OPT_CLOSE_ON_FREE
      LEV_OPT_REUSEABLE
- struct evconnlistener *evconnlistener_new_bind(
  struct event_base *base,
  evconnlistener_cb cb, --- 接受连接之后，用户要做的操作
  void *ptr, // 给回调传参
  unsigned flags,
  int backlog,
  --- -1: 使用默认的最大值
  const struct sockaddr *sa,
  ---- 服务器的IP和端口信息
  int socklen);

两个 evconnlistener_new*()函数都分配和返回一个新的连接监听器对象。连接监听器使用 event_base 来得知什么时候在给定的监听套接字上有新的 TCP 连接。新连接到达时，监听器调用你给出的回调函数

- void evconnlistener_free(struct evconnlistener *lev);

socket - server
- 创建监听socket
- 绑定
- 监听
- 等待并接收连接

2. 启用和禁用 evconnlistener
   - int evconnlistener_disable(struct evconnlistener *lev);
   - int evconnlistener_enable(struct evconnlistener *lev);

   这两个函数暂时禁止或者重新允许监听新连接。
3. 调整 evconnlistener 的回调函数

```
void evconnlistener_set_cb(
        struct evconnlistener *lev,
        evconnlistener_cb cb,
        void *arg
);
```

   函数调整 evconnlistener 的回调函数和其参数

*** 重要函数总结 ***

必须要掌握的函数:

- 创建event_base
  - struct event_base* event_base_new(void);
  - 失败返回NULL
- 释放event_base
  - event_base_free(struct event_base* base);
- 事件创建 - 没有缓冲区
  - struct event *event_new()
  - int event_add(
    struct event *ev,
    const struct timeval *tv
    );
  - 释放事件
    void event_free(struct event *event);
- 开始事件循环
  - int event_base_dispatch(struct event_base* base);

================== 套接字通信 ==================

1. 创建带缓冲区的事件

```
struct bufferevent *bufferevent_socket_new(
        struct event_base *base,
        evutil_socket_t fd,
        enum bufferevent_options options
);
```

- 给读写缓冲区设置回调

```
void bufferevent_setcb(
```

```
        struct bufferevent *bufev,
        bufferevent_data_cb readcb,
        bufferevent_data_cb writecb,
        bufferevent_event_cb eventcb,
        void *cbarg
    );
    void bufferevent_enable(
        struct bufferevent *bufev,
        short events
    );
```

- 默认ev_write 是enable
- ev_read是关闭的

2. 套接字通信，客户端连接服务器
    - int bufferevent_socket_connect(
        struct bufferevent *bev,
        struct sockaddr *address, - server ip和port
        int addrlen
    );
3. 服务器端用的函数
    - 创建监听的套接字
    - 绑定
    - 监听
    - 接收连接请求
    - struct evconnlistener *evconnlistener_new_bind(
        struct event_base *base,
        evconnlistener_cb cb,
        void *ptr,
        unsigned flags,
        int backlog,
        const struct sockaddr *sa,
        int socklen);
4. 操作bufferevent中的数据

- 向bufferevent的输出缓冲区添加数据
  - ```
    int bufferevent_write(
        struct bufferevent *bufev,
        const void *data,
        size_t size
    );
    ```
- 从bufferevent的输入缓冲区移除数据
  - ```
    size_t bufferevent_read(
        struct bufferevent *bufev,
        void *data,
        size_t size
    );
    ```

动态库找不到

2017年6月7日 10:43

1. 找到xxx.so放到 /usr/lib /lib -- 不推荐
   - sudo find /usr/local -name "libevent.so"
2. 将xxx.so放到环境变量中
   - LD_LIBRARY_PATH
     - export LD_LIBRARY_PATH=xxxx
       - ~/.bashrc --用户级别
       - /etc/profile - 系统级别
     - 使用命令重新加载
       - . ~/.bashrc
       - . /etc/profile
       - . 等价于 source
3. 修改/etc/ls.so.conf
   a. 动态库路径添加到该文件中 - 绝对路径
   b. sudo ldconfig -v