

# libevent 源码深度剖析

张亮

Email: [sparling.liang@hotmail.com](mailto:sparling.liang@hotmail.com)

回想刚开始写时，就冠以“深度剖析”的名称，也是为了给自己一些压力，以期能写好这一系列文章，对 libevent 源代码的各方面作详细的分析；现在看来也算是达到了最初的目的。希望能给学习和使用 libevent 的朋友们有所帮助。

Email: [sparkling.liang@hotmail.com](mailto:sparkling.liang@hotmail.com)

张亮

# 目录

libevent源码深度剖析.....	1
目录 .....	3
一 序幕.....	5
1 前言.....	5
2 Libevent简介 .....	5
3 学习的好处.....	5
二Reactor模式 .....	6
1 Reactor的事件处理机制 .....	6
2 Reactor模式的优点 .....	6
3 Reactor模式框架 .....	6
4 Reactor事件处理流程 .....	8
5 小结.....	9
三 基本使用场景和事件流程.....	10
1 前言.....	10
2 基本应用场景.....	10
3 实例代码.....	11
4 事件处理流程.....	11
5 小结.....	12
四 libevent源代码文件组织.....	13
1 前言.....	13
2 源代码组织结构.....	13
3 小结.....	14
五 libevent的核心：事件event .....	15
1 libevent的核心-event.....	15
2 libevent对event的管理 .....	16
3 事件设置的接口函数.....	17
4 小结.....	18
六 初见事件处理框架.....	19
1 事件处理框架-event_base.....	19
2 创建和初始化event_base .....	20
3 接口函数.....	20
4 小节.....	23
七 事件主循环.....	24
1 阶段性的胜利.....	24
2 事件处理主循环.....	24
3 I/O和Timer事件的统一 .....	27
4 I/O和Signal事件的统一 .....	27
5 小节.....	27
八 集成信号处理.....	28
1 集成策略——使用socket pair .....	28

2 集成到事件主循环——通知event_base .....	29
4 evsignal_info结构体 .....	30
5 注册、注销signal事件 .....	30
5 小节 .....	31
九 集成定时器事件 .....	32
1 集成到事件主循环 .....	32
2 Timer小根堆 .....	33
3 小节 .....	34
十 支持I/O多路复用技术 .....	35
1 统一的关键 .....	35
2 设置I/O demultiplex机制 .....	35
3 小节 .....	37
十一 时间管理 .....	38
1 初始化检测 .....	38
2 时间缓存 .....	38
3 时间校正 .....	40
4 小节 .....	41
十二 让libevent支持多线程 .....	42
1 错误使用示例 .....	42
2 支持多线程的几种模式 .....	42
3 例子——memcached .....	43
4 小节 .....	44

# 一 序幕

## 1 前言

Libevent 是一个轻量级的开源高性能网络库，使用者众多，研究者更甚，相关文章也不少。写这一系列文章的用意在于，一则分享心得；二则对 libevent 代码和设计思想做系统的、更深层次的分析，写出来，也可供后来者参考。

附带一句：Libevent 是用 c 语言编写的（MS 大牛们都偏爱 c 语言哪），而且几乎是无处不函数指针，学习其源代码也需要相当的 c 语言基础。

## 2 Libevent 简介

上来当然要先夸奖啦，Libevent 有几个显著的亮点：

事件驱动（event-driven），高性能；

轻量级，专注于网络，不如 ACE 那么臃肿庞大；

源代码相当精炼、易读；

跨平台，支持 Windows、Linux、\*BSD 和 Mac Os；

支持多种 I/O 多路复用技术，epoll、poll、dev/poll、select 和 kqueue 等；

支持 I/O，定时器和信号等事件；

注册事件优先级；

Libevent 已经被广泛的应用，作为底层的网络库；比如 memcached、Vomit、Nylon、Netchat 等等。

Libevent 当前的最新稳定版是 1.4.13；这也是本文参照的版本。

## 3 学习的好处

学习 libevent 有助于提升程序设计功力，除了网络程序设计方面外，Libevent 的代码里有很多有用的设计技巧和基础数据结构，比如信息隐藏、函数指针、c 语言的多态支持、链表和堆等等，都有助于提升自身的程序功力。

程序设计不止要了解框架，很多细节之处恰恰也是事关整个系统成败的关键。只对 libevent 本身的框架大概了解，那或许仅仅是一知半解，不深入代码分析，就难以了解其设计的精巧之处，也就难以为自己所用。

事实上 Libevent 本身就是一个典型的 Reactor 模型，理解 Reactor 模式是理解 libevent 的基石；因此下一节将介绍典型的事件驱动设计模式——Reactor 模式。

参考资料：

Libevent: <http://monkey.org/~provos/libevent/>

## 二 Reactor 模式

前面讲到，整个 libevent 本身就是一个 Reactor，因此本节将专门对 Reactor 模式进行必要的介绍，并列出 libevent 中的几个重要组件和 Reactor 的对应关系，在后面的章节中可能还会提到本节介绍的基本概念。

### 1 Reactor 的事件处理机制

首先来回想一下普通函数调用的机制：程序调用某函数→函数执行，程序等待→函数将结果和控制权返回给程序→程序继续处理。

Reactor 释义“反应堆”，是一种事件驱动机制。和普通函数调用的不同之处在于：应用程序不是主动的调用某个 API 完成处理，而是恰恰相反，Reactor 逆置了事件处理流程，应用程序需要提供相应的接口并注册到 Reactor 上，如果相应的时间发生，Reactor 将主动调用应用程序注册的接口，这些接口又称为“回调函数”。使用 Libevent 也是想 Libevent 框架注册相应的事件和回调函数；当这些时间发声时，Libevent 会调用这些回调函数处理相应的事件（I/O 读写、定时和信号）。

用“好莱坞原则”来形容 Reactor 再合适不过了：不要打电话给我们，我们会打电话通知你。

举个例子：你去应聘某 xx 公司，面试结束后。

“普通函数调用机制”公司 HR 比较懒，不会记你的联系方式，那怎么办呢，你只能面试完后自己打电话去问结果；有没有被录取啊，还是被拒了；

“Reactor”公司 HR 就记下了你的联系方式，结果出来后会主动打电话通知你：有没有被录取啊，还是被拒了；你不用自己打电话去问结果，事实上也不能，你没有 HR 的联系方式。

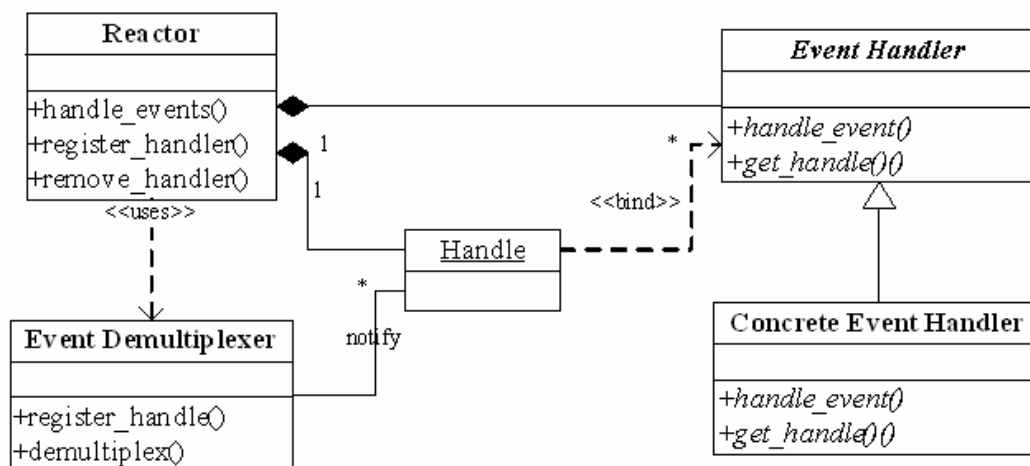
### 2 Reactor 模式的优点

Reactor 模式是编写高性能网络服务器的必备技术之一，它具有如下的优点：

- 1) 响应快，不必为单个同步时间所阻塞，虽然 Reactor 本身依然是同步的；
- 2) 编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销；
- 3) 可扩展性，可以方便的通过增加 Reactor 实例个数来充分利用 CPU 资源；
- 4) 可复用性，reactor 框架本身与具体事件处理逻辑无关，具有很高的复用性；

### 3 Reactor 模式框架

使用 Reactor 模型，必备的几个组件：事件源、Reactor 框架、多路复用机制和事件处理程序，先来看看 Reactor 模型的整体框架，接下来再对每个组件做逐一说明。



### 1) 事件源

Linux 上是文件描述符，Windows 上就是 Socket 或者 Handle 了，这里统一称为“**句柄集**”；程序在指定的句柄上注册关心的事件，比如 I/O 事件。

### 2) event demultiplexer——事件多路分发机制

由操作系统提供的 I/O 多路复用机制，比如 select 和 epoll。

程序首先将其关心的句柄（事件源）及其事件注册到 event demultiplexer 上；

当有事件到达时，event demultiplexer 会发出通知“在已经注册的句柄集中，一个或多个句柄的事件已经就绪”；

程序收到通知后，就可以在非阻塞的情况下对事件进行处理了。

对应到 libevent 中，依然是 select、poll、epoll 等，但是 libevent 使用结构体 eventop 进行了封装，以统一的接口来支持这些 I/O 多路复用机制，达到了对外隐藏底层系统机制的目的。

### 3) Reactor——反应器

**Reactor**，是事件管理的接口，内部使用 event demultiplexer 注册、注销事件；并运行事件循环，当有事件进入“就绪”状态时，调用注册事件的回调函数处理事件。

对应到 libevent 中，就是 event\_base 结构体。

一个典型的Reactor声明方式

```

class Reactor
{
public:
    int register_handler(Event_Handler *pHandler, int event);
    int remove_handler(Event_Handler *pHandler, int event);
    void handle_events(timeval *ptv);
    // ...
};
  
```

### 4) Event Handler——事件处理程序

事件处理程序提供了一组接口，每个接口对应了一种类型的事件，供 Reactor 在相应的事件发生时调用，执行相应的事件处理。通常它会绑定一个有效的句柄。

对应到 libevent 中，就是 event 结构体。

下面是两种典型的 Event Handler 类声明方式，二者互有优缺点。

```

class Event_Handler
{
public:
    virtual void handle_read() = 0;
    virtual void handle_write() = 0;
    virtual void handle_timeout() = 0;
    virtual void handle_close() = 0;
    virtual HANDLE get_handle() = 0;
    // ...
};

```

```

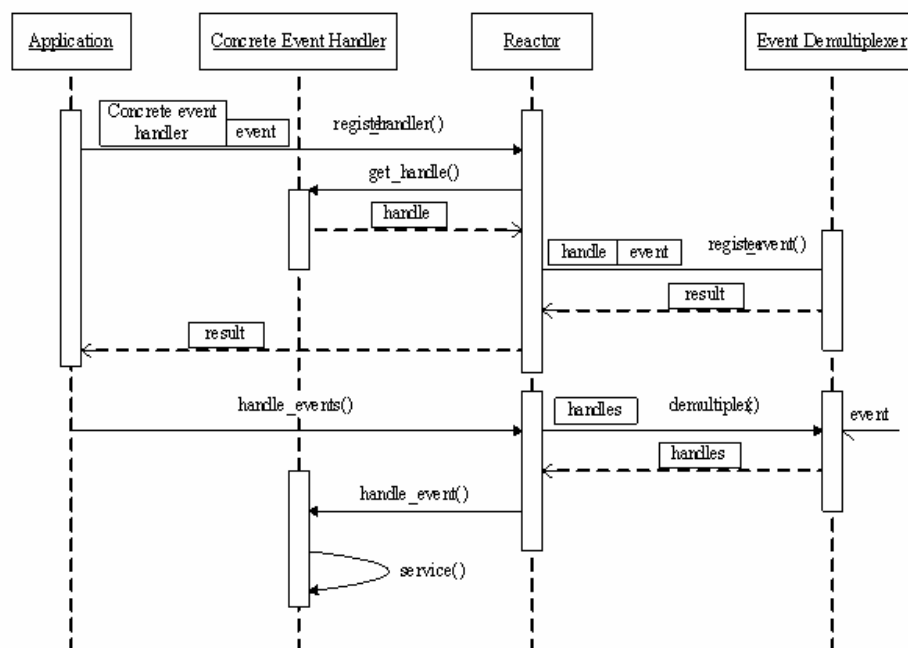
class Event_Handler
{
public:
    // events maybe read/write/timeout/close .etc
    virtual void handle_events(int events) = 0;
    virtual HANDLE get_handle() = 0;
    // ...
};

```

## 4 Reactor 事件处理流程

前面说过 Reactor 将事件流“逆置”了，那么使用 Reactor 模式后，事件控制流是什么样子呢？

可以参见下面的序列图。





## 5 小结

上面讲到了 Reactor 的基本概念、框架和处理流程，对 Reactor 有个基本清晰的了解后，再来对比看 libevent 就会更容易理解了，接下来就正式进入到 libevent 的代码世界了，加油！

参考资料：

Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2

## 三 基本使用场景和事件流程

### 1 前言

学习源代码该从哪里入手？我觉得从程序的**基本使用场景和代码的整体处理流程**入手是个不错的方法，至少从个人的经验上讲，用此方法分析 libevent 是比较有效的。

### 2 基本应用场景

基本应用场景也是使用 libevent 的基本流程，下面来考虑一个最简单的场景，使用 libevent 设置定时器，应用程序只需要执行下面几个简单的步骤即可。

#### 1) 首先初始化 libevent 库，并保存返回的指针

```
struct event_base * base = event_init();
```

实际上这一步相当于初始化一个 Reactor 实例；在初始化 libevent 后，就可以注册事件了。

#### 2) 初始化事件 event，设置回调函数和关注的事件

```
event_set(&ev, timer_cb, NULL);
```

事实上这等价于调用 event\_set(&ev, -1, 0, timer\_cb, NULL);

event\_set 的函数原型是：

```
void event_set(struct event *ev, int fd, short event, void (*cb)(int, short, void *), void *arg)
```

ev: 执行要初始化的 event 对象；

fd: 该 event 绑定的“句柄”，对于信号事件，它就是关注的信号；

event: 在该 fd 上关注的事件类型，它可以是 EV\_READ, EV\_WRITE, EV\_SIGNAL；

cb: 这是一个函数指针，当 fd 上的事件 event 发生时，调用该函数执行处理，它有三个参数，调用时由 event\_base 负责传入，按顺序，实际上就是 event\_set 时的 fd, event 和 arg；

arg: 传递给 cb 函数指针的参数；

由于定时事件不需要 fd，并且定时事件是根据添加时（event\_add）的超时值设定的，因此这里 event 也不需要设置。

这一步相当于初始化一个 event handler，在 libevent 中事件类型保存在 event 结构体中。

**注意：libevent 并不会管理 event 事件集合，这需要应用程序自行管理；**

#### 3) 设置 event 从属的 event\_base

```
event_base_set(base, &ev);
```

这一步相当于指明 event 要注册到哪个 event\_base 实例上；

#### 4) 是正式的添加事件的时候了

```
event_add(&ev, timeout);
```

基本信息都已设置完成，只要简单的调用 event\_add() 函数即可完成，其中 timeout 是定时值；

这一步相当于调用 `Reactor::register_handler()` 函数注册事件。

5) 程序进入无限循环，等待就绪事件并执行事件处理

```
event_base_dispatch(base);
```

### 3 实例代码

上面例子的程序代码如下所示

```
struct event ev;
struct timeval tv;

void time_cb(int fd, short event, void *argc)
{
    printf("timer wakeup\n");
    event_add(&ev, &tv); // reschedule timer
}

int main()
{
    struct event_base *base = event_init();

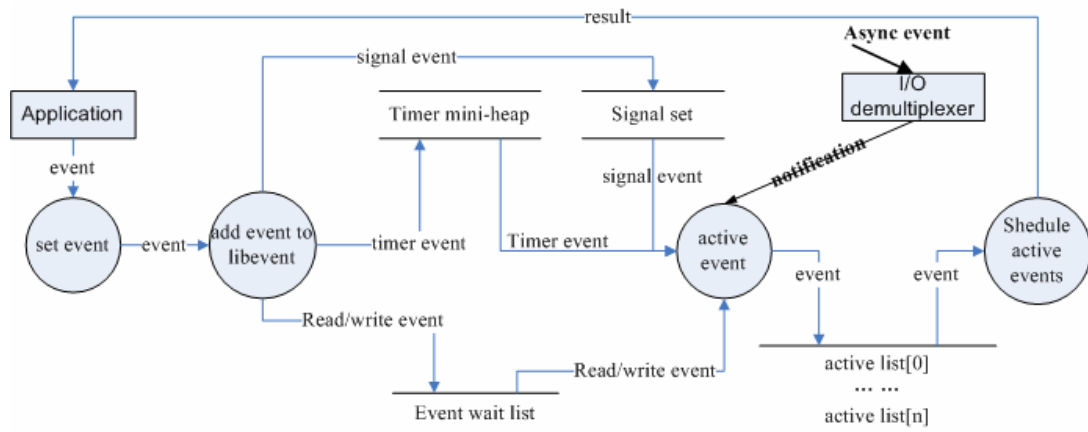
    tv.tv_sec = 10; // 10s period
    tv.tv_usec = 0;
    evtimer_set(&ev, time_cb, NULL);
    event_add(&ev, &tv);
    event_base_dispatch(base);
}
```

### 4 事件处理流程

当应用程序向 `libevent` 注册一个事件后，`libevent` 内部是怎样进行处理的呢？下面的图就给出了这一基本流程。

- 1) 首先应用程序准备并初始化 event，设置好事件类型和回调函数；这对应于前面第步骤 2 和 3；
- 2) 向 `libevent` 添加该事件 event。对于定时事件，`libevent` 使用一个小根堆管理，key 为超时时间；对于 Signal 和 I/O 事件，`libevent` 将其放入到等待链表（wait list）中，这是一个双向链表结构；
- 3) 程序调用 `event_base_dispatch()` 系列函数进入无限循环，等待事件，以 `select()` 函数为例；每次循环前 `libevent` 会检查定时事件的最小超时时间 tv，根据 tv 设置 `select()` 的最大等待时间，以便于后面及时处理超时事件；当 `select()` 返回后，首先检查超时事件，然后检查 I/O 事件；

Libevent 将所有的就绪事件，放入到激活链表中；  
然后对激活链表中的事件，调用事件的回调函数执行事件处理；



## 5 小结

本节介绍了 libevent 的简单实用场景，并旋风般的介绍了 libevent 的事件处理流程，读者应该对 libevent 有了基本的印象，下面将会详细介绍 libevent 的事件管理框架（Reactor 模式中的 Reactor 框架）做详细的介绍，在此之前会对源代码文件做简单的分类。

# 四 libevent 源代码文件组织

## 1 前言

详细分析源代码之前，如果能对其代码文件的基本结构有个大概的认识和分类，对于代码的分析将是大有裨益的。

## 2 源代码组织结构

Libevent 的源代码虽然都在一层文件夹下面，但是其代码分类还是相当清晰的，主要可分为头文件、内部使用的头文件、辅助功能函数、日志、libevent 框架、对系统 I/O 多路复用机制的封装、信号管理、定时事件管理、缓冲区管理、基本数据结构和基于 libevent 的两个实用库等几个部分，有些部分可能就是一个源文件。

源代码中的 test 部分就不在我们关注的范畴了。

### 1) 头文件

主要就是 event.h: 事件宏定义、接口函数声明，主要结构体 event 的声明；

### 2) 内部头文件

xxx-internal.h: 内部数据结构和函数，对外不可见，以达到信息隐藏的目的；

### 3) libevent 框架

event.c: event 整体框架的代码实现；

### 4) 对系统 I/O 多路复用机制的封装

epoll.c: 对 epoll 的封装；

select.c: 对 select 的封装；

devpoll.c: 对 dev/poll 的封装；

kqueue.c: 对 kqueue 的封装；

### 5) 定时事件管理

min-heap.h: 其实就是一个以时间作为 key 的小根堆结构；

### 6) 信号管理

signal.c: 对信号事件的处理；

### 7) 辅助功能函数

evutil.h 和 evutil.c: 一些辅助功能函数，包括创建 socket pair 和一些时间操作函数：加、减和比较等。

### 8) 日志

log.h 和 log.c: log 日志函数

### 9) 缓冲区管理

evbuffer.c 和 buffer.c: libevent 对缓冲区的封装；

### 10) 基本数据结构

compat/sys 下的两个源文件: queue.h 是 libevent 基本数据结构的实现，包括链表，双向链表，队列等；\_libevent\_time.h: 一些用于时间操作的结构体定义、函数和宏定义；

### 11) 实用网络库

http 和 evdns: 是基于 libevent 实现的 http 服务器和异步 dns 查询库;

### 3 小结

本节介绍了 libevent 的组织和分类, 下面将会详细介绍 libevent 的核心部分 event 结构。

## 五 libevent 的核心：事件 event

对事件处理流程有了高层的认识后，本节将详细介绍 libevent 的核心结构 event，以及 libevent 对 event 的管理。

### 1 libevent 的核心-event

Libevent 是基于事件驱动（event-driven）的，从名字也可以看到 event 是整个库的核心。event 就是 Reactor 框架中的事件处理程序组件；它提供了函数接口，供 Reactor 在事件发生时调用，以执行相应的事件处理，通常它会绑定一个有效的句柄。

首先给出 event 结构体的声明，它位于 event.h 文件中：

```
struct event {  
    TAILQ_ENTRY (event) ev_next;  
    TAILQ_ENTRY (event) ev_active_next;  
    TAILQ_ENTRY (event) ev_signal_next;  
    unsigned int min_heap_idx; /* for managing timeouts */  
    struct event_base *ev_base;  
    int ev_fd;  
    short ev_events;  
    short ev_ncalls;  
    short *ev_pncalls; /* Allows deletes in callback */  
    struct timeval ev_timeout;  
    int ev_pri; /* smaller numbers are higher priority */  
    void (*ev_callback)(int, short, void *arg);  
    void *ev_arg;  
    int ev_res; /* result passed to event callback */  
    int ev_flags;  
};
```

下面详细解释一下结构体中各字段的含义。

1) ev\_events: event关注的事件类型，它可以是以下3种类型：

I/O事件： EV\_WRITE和EV\_READ

定时事件： EV\_TIMEOUT

信号： EV\_SIGNAL

辅助选项： EV\_PERSIST，表明是一个永久事件

Libevent中的定义为：

```
#define EV_TIMEOUT 0x01  
#define EV_READ 0x02  
#define EV_WRITE 0x04  
#define EV_SIGNAL 0x08  
#define EV_PERSIST 0x10 /* Persistent event */
```

可以看出事件类型可以使用“|”运算符进行组合，需要说明的是，信号和I/O事件不能同时

设置;

还可以看出libevent使用event结构体将这3种事件的处理统一起来;

2) ev\_next, ev\_active\_next 和 ev\_signal\_next 都是双向链表节点指针; 它们是 libevent 对不同事件类型和在不同的时期, 对事件的管理时使用到的字段。

libevent 使用双向链表保存所有注册的 I/O 和 Signal 事件, ev\_next 就是该 I/O 事件在链表中的位置; 称此链表为“已注册事件链表”;

同样 ev\_signal\_next 就是 signal 事件在 signal 事件链表中的位置;

ev\_active\_next: libevent 将所有的激活事件放入到链表 active list 中, 然后遍历 active list 执行调度, ev\_active\_next 就指明了 event 在 active list 中的位置;

2) min\_heap\_idx 和 ev\_timeout, 如果是 timeout 事件, 它们是 event 在小根堆中的索引和超时值, libevent 使用小根堆来管理定时事件, 这将在后面定时事件处理时专门讲解

3) ev\_base 该事件所属的反应堆实例, 这是一个 event\_base 结构体, 下一节将会详细讲解;

4) ev\_fd, 对于 I/O 事件, 是绑定的文件描述符; 对于 signal 事件, 是绑定的信号;

5) ev\_callback, event 的回调函数, 被 ev\_base 调用, 执行事件处理程序, 这是一个函数指针, 原型为:

```
void (*ev_callback)(int fd, short events, void *arg)
```

其中参数 fd 对应于 ev\_fd; events 对应于 ev\_events; arg 对应于 ev\_arg;

6) ev\_arg: void\*, 表明可以是任意类型的数据, 在设置 event 时指定;

7) eb\_flags: libevent 用于标记 event 信息的字段, 表明其当前的状态, 可能的值有:

```
#define EVLIST_TIMEOUT    0x01 // event在time堆中
#define EVLIST_INSERTED  0x02 // event在已注册事件链表中
#define EVLIST_SIGNAL     0x04 // 未见使用
#define EVLIST_ACTIVE     0x08 // event在激活链表中
#define EVLIST_INTERNAL   0x10 // 内部使用标记
#define EVLIST_INIT       0x80 // event 已被初始化
```

8) ev\_ncalls: 事件就绪执行时, 调用 ev\_callback 的次数, 通常为 1;

9) ev\_pncalls: 指针, 通常指向 ev\_ncalls 或者为 NULL;

10) ev\_res: 记录了当前激活事件的类型;

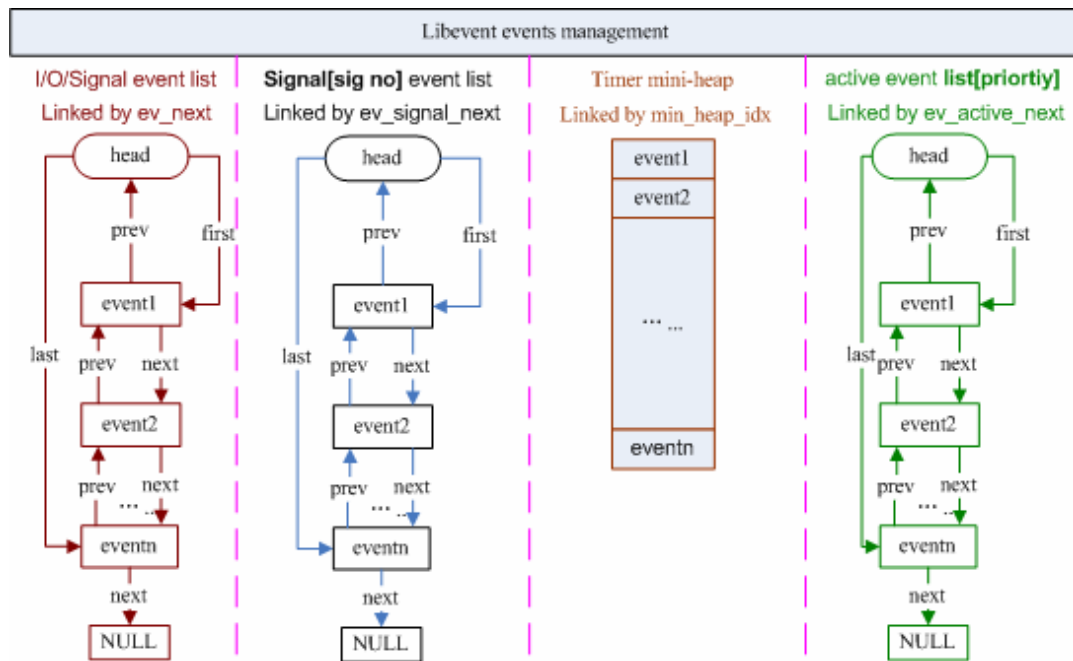
## 2 libevent 对 event 的管理

从 event 结构体中的 3 个链表节点指针和一个堆索引出发, 大体上也能窥出 libevent 对 event 的管理方法了, 可以参见下面的示意图。

每当有事件 event 转变为就绪状态时, libevent 就会把它移入到 active event list[priority] 中, 其中 priority 是 event 的优先级;

接着 libevent 会根据自己的调度策略选择就绪事件, 调用其 cb\_callback()函数执行事件处理; 并根据就绪的句柄和事件类型填充 cb\_callback 函数的参数。





### 3 事件设置的接口函数

要向 libevent 添加一个事件，需要首先设置 event 对象，这通过调用 libevent 提供的函数有：`event_set()`, `event_base_set()`, `event_priority_set()`来完成；下面分别进行讲解。

```
void event_set(struct event *ev, int fd, short events,
               void (*callback)(int, short, void *), void *arg)
```

1. 设置事件 `ev` 绑定的文件描述符或者信号，对于定时事件，设为-1 即可；
2. 设置事件类型，比如 `EV_READ|EV_PERSIST`, `EV_WRITE`, `EV_SIGNAL` 等；
3. 设置事件的回调函数以及参数 `arg`；
4. 初始化其它字段，比如缺省的 `event_base` 和优先级；

```
int event_base_set(struct event_base *base, struct event *ev)
```

设置 event `ev` 将要注册到的 `event_base`；

libevent 有一个全局 `event_base` 指针 `current_base`，默认情况下事件 `ev` 将被注册到 `current_base` 上，使用该函数可以指定不同的 `event_base`；

如果一个进程中存在多个 libevent 实例，则必须要调用该函数为 event 设置不同的 `event_base`；

```
int event_priority_set(struct event *ev, int pri)
```

设置event `ev` 的优先级，没什么可说的，注意的一点就是：当`ev`正处于就绪状态时，不能设置，返回-1。

## 4 小结

本节讲述了libevent的核心event结构,以及libevent支持的事件类型和libevent对event的管理模型;接下来将会描述libevent的事件处理框架,以及其中使用的重要的结构体event\_base。

## 六 初见事件处理框架

前面已经对 libevent 的事件处理框架和 event 结构体做了描述，现在是时候剖析 libevent 对事件的详细处理流程了，本节将分析 libevent 的事件处理框架 event\_base 和 libevent 注册、删除事件的具体流程，可结合前一节 libevent 对 event 的管理。

### 1 事件处理框架-event\_base

回想 Reactor 模式的几个基本组件，本节讲解的部分对应于 Reactor 框架组件。在 libevent 中，这就表现为 event\_base 结构体，结构体声明如下，它位于 event-internal.h 文件中：

```
struct event_base {
    const struct eventop *evsel;
    void *evbase;
    int event_count; /* counts number of total events */
    int event_count_active; /* counts number of active events */
    int event_gotterm; /* Set to terminate loop */
    int event_break; /* Set to terminate loop immediately */
    /* active event management */
    struct event_list **activequeues;
    int nactivequeues;
    /* signal handling info */
    struct evsignal_info sig;
    struct event_list eventqueue;
    struct timeval event_tv;
    struct min_heap timeheap;
    struct timeval tv_cache;
};
```

下面详细解释一下结构体中各字段的含义。

1) evsel 和 evbase 这两个字段的设置可能会让人有些迷惑，这里你可以把 evsel 和 evbase 看作是类和静态函数的关系，比如添加事件时的调用行为：evsel->add(evbase, ev)，实际执行操作的是 evbase；这相当于 class::add(instance, ev)，instance 就是 class 的一个对象实例。evsel 指向了全局变量 static const struct eventop \*eventops[] 中的一个；

前面也说过，libevent 将系统提供的 I/O demultiplex 机制统一封装成了 eventop 结构；因此 eventops[] 包含了 select、poll、kequeue 和 epoll 等等其中的若干个全局实例对象。

evbase 实际上是一个 eventop 实例对象；

先来看看 eventop 结构体，它的成员是一系列的函数指针，在 event-internal.h 文件中：

```
struct eventop {
    const char *name;
    void *(*init)(struct event_base *); // 初始化
    int (*add)(void *, struct event *); // 注册事件
    int (*del)(void *, struct event *); // 删除事件
};
```

```

    int (*dispatch)(struct event_base *, void *, struct timeval *); //
事件分发
    void (*dealloc)(struct event_base *, void *); // 注销, 释放资源
    /* set if we need to reinitialize the event base */
    int need_reinit;
};

```

也就是说, 在 libevent 中, 每种 I/O demultiplex 机制的实现都必须提供这五个函数接口, 来完成自身的初始化、销毁释放; 对事件的注册、注销和分发。

比如对于 epoll, libevent 实现了 5 个对应的接口函数, 并在初始化时并将 eventop 的 5 个函数指针指向这 5 个函数, 那么程序就可以使用 epoll 作为 I/O demultiplex 机制了, 这个在后面会再次提到。

2) activequeues 是一个二级指针, 前面讲过 libevent 支持事件优先级, 因此你可以把它看作是数组, 其中的元素 activequeues[priority]是一个链表, 链表的每个节点指向一个优先级为 priority 的就绪事件 event。

3) eventqueue, 链表, 保存了所有的注册事件 event 的指针。

4) sig 是由来管理信号的结构体, 将在后面信号处理时专门讲解;

5) timeheap 是管理定时事件的小根堆, 将在后面定时事件处理时专门讲解;

6) event\_tv 和 tv\_cache 是 libevent 用于时间管理的变量, 将在后面讲到;

其它各个变量都能因名知意, 就不再啰嗦了。

## 2 创建和初始化 event\_base

创建一个 event\_base 对象也既是创建了一个新的 libevent 实例, 程序需要通过调用 event\_init() (内部调用 event\_base\_new 函数执行具体操作) 函数来创建, 该函数同时还对新生成的 libevent 实例进行了初始化。

该函数首先为 event\_base 实例申请空间, 然后初始化 timer mini-heap, 选择并初始化合适的系统 I/O 的 demultiplexer 机制, 初始化各事件链表;

函数还检测了系统的时间设置, 为后面的时间管理打下基础。

## 3 接口函数

前面提到 Reactor 框架的作用就是提供事件的注册、注销接口; 根据系统提供的事件多路分发机制执行事件循环, 当有事件进入“就绪”状态时, 调用注册事件的回调函数来处理事件。

Libevent 中对应的接口函数主要就是:

```

int event_add(struct event *ev, const struct timeval *timeout);
int event_del(struct event *ev);
int event_base_loop(struct event_base *base, int loops);
void event_active(struct event *event, int res, short events);
void event_process_active(struct event_base *base);

```

本节将按介绍事件注册和删除的代码流程, libevent 的事件循环框架将在下一节再具体描述。

对于定时事件, 这些函数将调用 timer heap 管理接口执行插入和删除操作; 对于 I/O 和

Signal 事件将调用 `eventopadd` 和 `delete` 接口函数执行插入和删除操作（`eventop` 会对 Signal 事件调用 Signal 处理接口执行操作）；这些组件将在后面的内容描述。

### 1) 注册事件

函数原型：

```
int event_add(struct event *ev, const struct timeval *tv)
```

参数：ev：指向要注册的事件；

tv：超时时间；

函数将 ev 注册到 `ev->ev_base` 上，事件类型由 `ev->ev_events` 指明，如果注册成功，ev 将被插入到已注册链表中；如果 tv 不是 NULL，则会同时注册定时事件，将 ev 添加到 timer 堆上；

如果其中有一步操作失败，那么函数保证没有事件会被注册，可以讲这相当于一个原子操作。这个函数也体现了 libevent 细节之处的巧妙设计，且仔细看程序代码，部分有省略，注释直接附在代码中。

```
int event_add(struct event *ev, const struct timeval *tv)
{
    struct event_base *base = ev->ev_base; // 要注册到的event_base
    const struct eventop *evsel = base->evsel;
    void *evbase = base->evbase; // base使用的系统I/O策略
```

```
    // 新的timer事件，调用timer heap接口在堆上预留一个位置
    // 注：这样能保证该操作的原子性：
    // 向系统I/O机制注册可能会失败，而当在堆上预留成功后，
    // 定时事件的添加将肯定不会失败；
    // 而预留位置的可能结果是堆扩充，但是内部元素并不会改变
    if (tv != NULL && !(ev->ev_flags & EVLIST_TIMEOUT)) {
        if (min_heap_reserve(&base->timeheap,
            1 + min_heap_size(&base->timeheap)) == -1)
            return (-1); /* ENOMEM == errno */
    }
```

```
    // 如果事件ev不在已注册或者激活链表中，则调用evbase注册事件
    if ((ev->ev_events & (EV_READ|EV_WRITE|EV_SIGNAL)) &&
        !(ev->ev_flags & (EVLIST_INSERTED|EVLIST_ACTIVE))) {
        res = evsel->add(evbase, ev);
        if (res != -1) // 注册成功，插入event到已注册链表中
            event_queue_insert(base, ev, EVLIST_INSERTED);
    }
```

```
    // 准备添加定时事件
    if (res != -1 && tv != NULL) {
        struct timeval now;
```

```
        // EVLIST_TIMEOUT表明event已经在定时器堆中了，删除旧的
        if (ev->ev_flags & EVLIST_TIMEOUT)
            event_queue_remove(base, ev, EVLIST_TIMEOUT);
```

```

        // 如果事件已经是就绪状态则从激活链表中删除
        if ((ev->ev_flags & EVLIST_ACTIVE) &&
            (ev->ev_res & EV_TIMEOUT)) {
            // 将ev_callback调用次数设置为0
            if (ev->ev_ncalls && ev->ev_pncalls) {
                *ev->ev_pncalls = 0;
            }
            event_queue_remove(base, ev, EVLIST_ACTIVE);
        }
        // 计算时间，并插入到timer小根堆中
        gettimeofday(&now, NULL);
        evutil_timeradd(&now, tv, &ev->ev_timeout);
        event_queue_insert(base, ev, EVLIST_TIMEOUT);
    }
    return (res);
}

```

event\_queue\_insert()负责将事件插入到对应的链表中，下面是程序代码；

event\_queue\_remove()负责将事件从对应的链表中删除，这里就不再重复贴代码了；

```

void event_queue_insert(struct event_base *base, struct event *ev,
int queue)
{
    // ev可能已经在激活列表中了，避免重复插入
    if (ev->ev_flags & queue) {
        if (queue & EVLIST_ACTIVE)
            return;
    }
    // ...
    ev->ev_flags |= queue; // 记录queue标记
    switch (queue) {
        case EVLIST_INSERTED: // I/O或Signal事件，加入已注册事件链表
            TAILQ_INSERT_TAIL(&base->eventqueue, ev, ev_next);
            break;
        case EVLIST_ACTIVE: // 就绪事件，加入激活链表
            base->event_count_active++;
            TAILQ_INSERT_TAIL(base->activequeues[ev->ev_pri], ev,
ev_active_next);
            break;
        case EVLIST_TIMEOUT: // 定时事件，加入堆
            min_heap_push(&base->timeheap, ev);
            break;
    }
}

```

## 2) 删除事件:

函数原型为: `int event_del(struct event *ev);`

该函数将删除事件 `ev`, 对于 I/O 事件, 从 I/O 的 `demultiplexer` 上将事件注销; 对于 `Signal` 事件, 将从 `Signal` 事件链表中删除; 对于定时事件, 将从堆上删除;

同样删除事件的操作则不一定是原子的, 比如删除时间事件之后, 有可能从系统 I/O 机制中注销会失败。

```
int event_del(struct event *ev)
{
    struct event_base *base;
    const struct eventop *evsel;
    void *evbase;

    // ev_base为NULL, 表明ev没有被注册
    if (ev->ev_base == NULL)
        return (-1);
    // 取得ev注册的event_base和eventop指针
    base = ev->ev_base;
    evsel = base->evsel;
    evbase = base->evbase;

    // 将ev_callback调用次数设置为
    if (ev->ev_ncalls && ev->ev_pncalls) {
        *ev->ev_pncalls = 0;
    }

    // 从对应的链表中删除
    if (ev->ev_flags & EVLIST_TIMEOUT)
        event_queue_remove(base, ev, EVLIST_TIMEOUT);
    if (ev->ev_flags & EVLIST_ACTIVE)
        event_queue_remove(base, ev, EVLIST_ACTIVE);
    if (ev->ev_flags & EVLIST_INSERTED) {
        event_queue_remove(base, ev, EVLIST_INSERTED);
        // EVLIST_INSERTED表明是I/O或者Signal事件,
        // 需要调用I/O demultiplexer注销事件
        return (evsel->del(evbase, ev));
    }
    return (0);
}
```

## 4 小节

分析了 `event_base` 这一重要结构体, 初步看到了 `libevent` 对系统的 I/O `demultiplex` 机制的封装 `event_op` 结构, 并结合源代码分析了事件的注册和删除处理, 下面将会接着分析事件管理框架中的主事件循环部分。

# 七 事件主循环

现在我们已经初步了解了 libevent 的 Reactor 组件——event\_base 和事件管理框架，接下来就是 libevent 事件处理的中心部分——事件主循环，根据系统提供的事件多路分发机制执行事件循环，对已注册的就绪事件，调用注册事件的回调函数来处理事件。

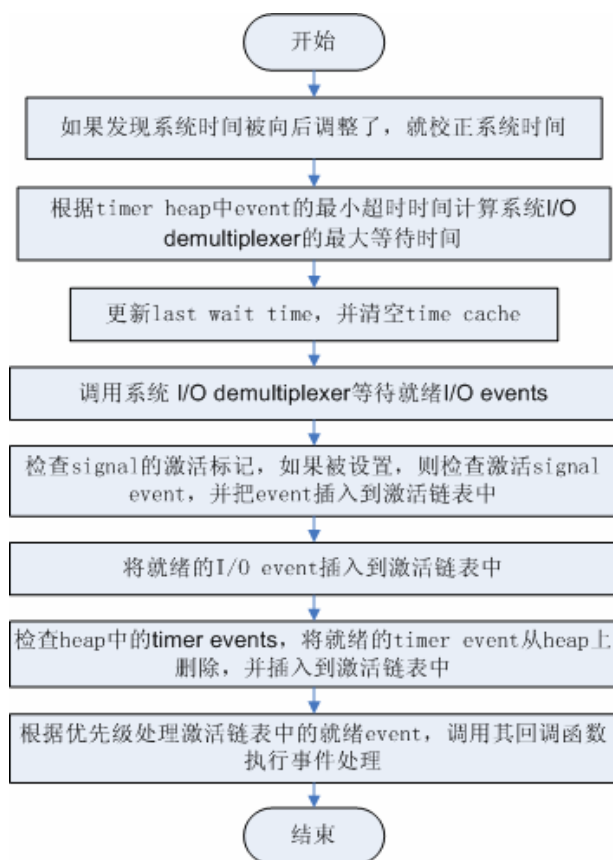
## 1 阶段性的胜利

Libevent 将 I/O 事件、定时器和信号事件处理很好的结合到了一起，本节也会介绍 libevent 是如何做到这一点的。

在看完本节的内容后，读者应该会对 Libevent 的基本框架：事件管理和主循环有比较清晰的认识了，并能够把 libevent 的事件控制流程清晰的串通起来，剩下的就是一些细节的内容了。

## 2 事件处理主循环

Libevent 的事件主循环主要是通过 event\_base\_loop ()函数完成的，其主要操作如下面的流程图所示，event\_base\_loop 所作的就是持续执行下面的循环。





清楚了 `event_base_loop` 所作的主要操作，就可以对比源代码看个究竟了，代码结构还是相当清晰的。

```
int event_base_loop(struct event_base *base, int flags)
{
    const struct eventop *evsel = base->evsel;
    void *evbase = base->evbase;
    struct timeval tv;
    struct timeval *tv_p;
    int res, done;

    // 清空时间缓存
    base->tv_cache.tv_sec = 0;
    // evsignal_base是全局变量，在处理signal时，用于指名signal所属的
    event_base实例
    if (base->sig.ev_signal_added)
        evsignal_base = base;
    done = 0;
    while (!done) { // 事件主循环
        // 查看是否需要跳出循环，程序可以调用event_loopexit_cb() 设置
        event_gotterm标记
        // 调用event_base_loopbreak() 设置event_break标记
        if (base->event_gotterm) {
            base->event_gotterm = 0;
            break;
        }
        if (base->event_break) {
            base->event_break = 0;
            break;
        }
        // 校正系统时间，如果系统使用的是非MONOTONIC时间，用户可能会向
        后调整了系统时间
        // 在timeout_correct函数里，比较last wait time和当前时间，如果
        当前时间 < last wait time
        // 表明时间有问题，这是需要更新timer_heap中所有定时事件的超时时
        间。
        timeout_correct(base, &tv);

        // 根据timer heap中事件的最小超时时间，计算系统I/O demultiplexer
        的最大等待时间
        tv_p = &tv;
        if (!base->event_count_active && !(flags & EVLOOP_NONBLOCK)) {
            timeout_next(base, &tv_p);
        } else {
            // 依然有未处理的就绪时间，就让I/O demultiplexer立即返回，
```

不必等待

// 下面会提到, 在libevent中, 低优先级的就绪事件可能不能立即被处理

```
    evutil_timerclear(&tv);
}
// 如果当前没有注册事件, 就退出
if (!event_haveevents(base)) {
    event_debug(("s: no events registered.", __func__));
    return (1);
}
// 更新last wait time, 并清空time cache
gettime(base, &base->event_tv);
base->tv_cache.tv_sec = 0;
// 调用系统I/O demultiplexer等待就绪I/O events, 可能是
epoll_wait, 或者select等;
// 在evsel->dispatch()中, 会把就绪signal event、I/O event插入到
激活链表中
res = evsel->dispatch(base, evbase, tv_p);
```

```
if (res == -1)
    return (-1);
// 将time cache赋值为当前系统时间
gettime(base, &base->tv_cache);
// 检查heap中的timer events, 将就绪的timer event从heap上删除,
并插入到激活链表中
timeout_process(base);
```

// 调用event\_process\_active()处理激活链表中的就绪event, 调用其回调函数执行事件处理

// 该函数会寻找最高优先级(priority值越小优先级越高)的激活事件链表,

```
// 然后处理链表中的所有就绪事件;
// 因此低优先级的就绪事件可能得不到及时处理;
if (base->event_count_active) {
    event_process_active(base);
    if (!base->event_count_active && (flags & EVLOOP_ONCE))
        done = 1;
} else if (flags & EVLOOP_NONBLOCK)
    done = 1;
}
```

```
// 循环结束, 清空时间缓存
base->tv_cache.tv_sec = 0;
```

```
event_debug(("s: asked to terminate loop.", __func__));  
return (0);  
}
```

### 3 I/O 和 Timer 事件的统一

Libevent 将 Timer 和 Signal 事件都统一到系统的 I/O 的 demultiplex 机制中了，相信读者从上面的流程和代码中也能窥出一斑了，下面就再啰嗦一次了。

首先将 Timer 事件融合到系统 I/O 多路复用机制中，还是相当清晰的，因为系统的 I/O 机制像 `select()` 和 `epoll_wait()` 都允许程序制定一个最大等待时间（也称为最大超时时间）`timeout`，即使没有 I/O 事件发生，它们也保证能在 `timeout` 时间内返回。

那么根据所有 Timer 事件的最小超时时间来设置系统 I/O 的 `timeout` 时间；当系统 I/O 返回时，再激活所有就绪的 Timer 事件就可以了，这样就能将 Timer 事件完美的融合到系统的 I/O 机制中了。

这是在 Reactor 和 Proactor 模式（主动器模式，比如 Windows 上的 IOCP）中处理 Timer 事件的经典方法了，ACE 采用的也是这种方法，大家可以参考 POSA vol2 书中的 Reactor 模式一节。

堆是一种经典的数据结构，向堆中插入、删除元素时间复杂度都是  $O(\lg N)$ ， $N$  为堆中元素的个数，而获取最小 key 值（小根堆）的复杂度为  $O(1)$ ；因此变成了管理 Timer 事件的绝佳人选（当然是非唯一的），libevent 就是采用的堆结构。

### 4 I/O 和 Signal 事件的统一

Signal 是异步事件的经典事例，将 Signal 事件统一到系统的 I/O 多路复用中就不像 Timer 事件那么自然了，Signal 事件的出现对于进程来讲是完全随机的，进程不能只是测试一个变量来判别是否发生了一个信号，而是必须告诉内核“在此信号发生时，请执行如下的操作”。

如果当 Signal 发生时，并不立即调用 event 的 callback 函数处理信号，而是设法通知系统的 I/O 机制，让其返回，然后再统一和 I/O 事件以及 Timer 一起处理，不就可以了嘛。是的，这也是 libevent 中使用的方法。

问题的核心在于，当 Signal 发生时，如何通知系统的 I/O 多路复用机制，这里先买个小关子，放到信号处理一节再详细说明，我想读者肯定也能想出通知的方法，比如使用 pipe。

### 5 小节

介绍了 libevent 的事件主循环，描述了 libevent 是如何处理就绪的 I/O 事件、定时器和信号事件，以及如何将它们无缝的融合到一起。

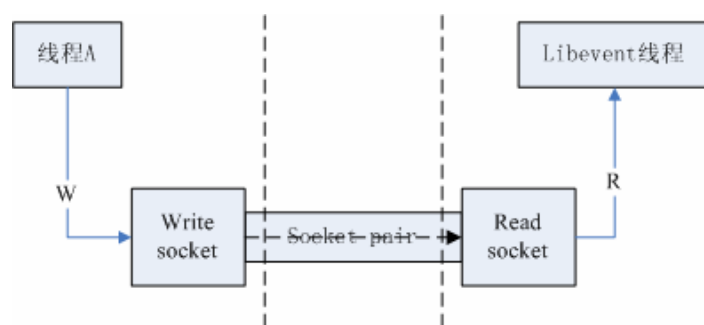
# 八 集成信号处理

现在我们已经了解了 libevent 的基本框架：事件管理框架和事件主循环。上节提到了 libevent 中 I/O 事件和 Signal 以及 Timer 事件的集成，这一节将分析如何将 Signal 集成到事件主循环的框架中。

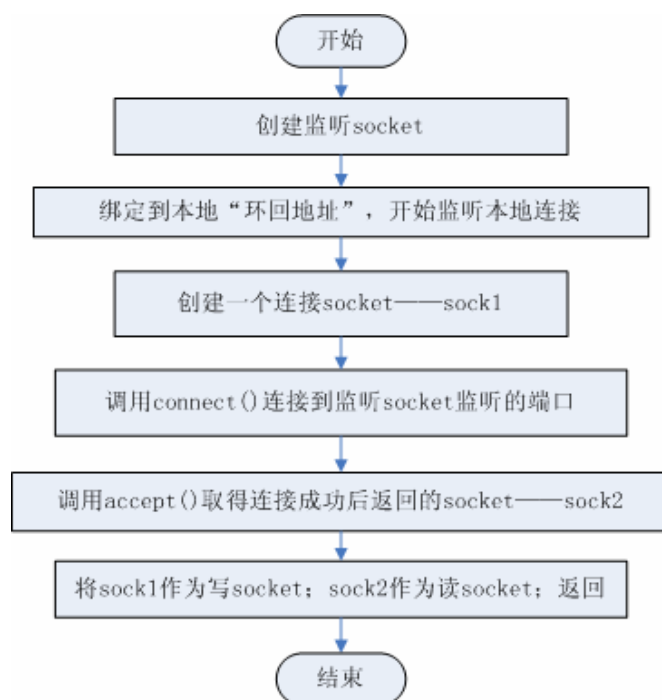
## 1 集成策略——使用 socket pair

前一节已经做了足够多的介绍了，基本方法就是采用“消息机制”。在 libevent 中这是通过 socket pair 完成的，下面就来详细分析一下。

Socket pair 就是一个 socket 对，包含两个 socket，一个读 socket，一个写 socket。工作方式如下图所示：



创建一个 socket pair 并不是复杂的操作，可以参见下面的流程图，清晰起见，其中忽略了一些错误处理和检查。



Libevent 提供了辅助函数 `evutil_socketpair()` 来创建一个 socket pair，可以结合上面的创

建流程来分析该函数。

## 2 集成到事件主循环——通知 event\_base

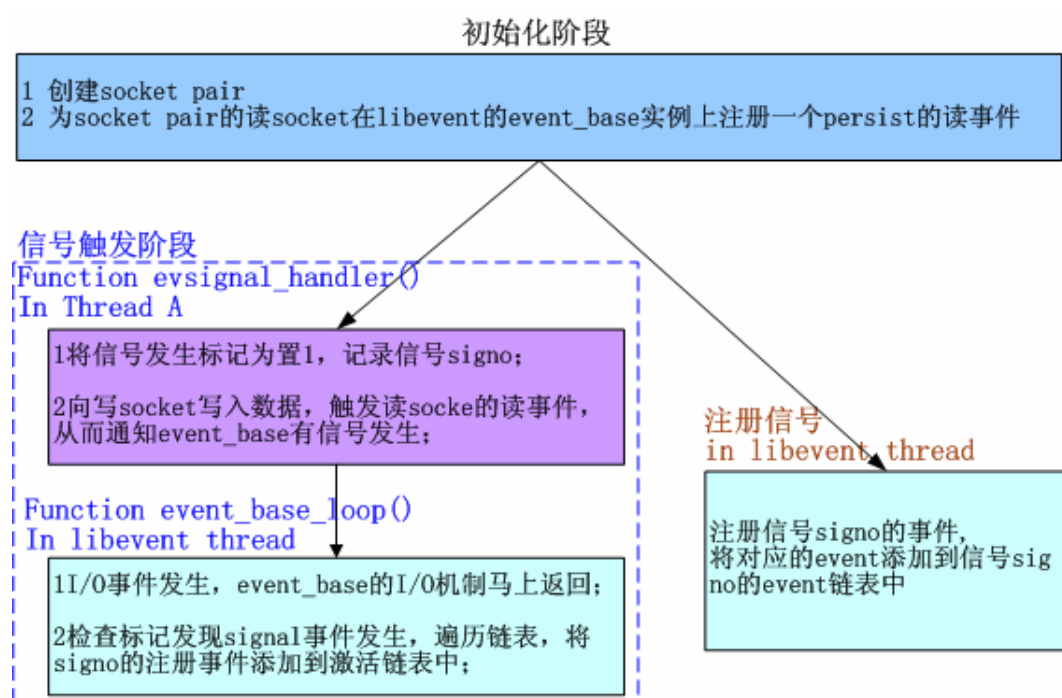
Socket pair 创建好了，可是 libevent 的事件主循环还是不知道 Signal 是否发生了啊，看来我们还差了最后一步，那就是：为 socket pair 的读 socket 在 libevent 的 event\_base 实例上注册一个 persist 的读事件。

这样当向写 socket 写入数据时，读 socket 就会得到通知，触发读事件，从而 event\_base 就能相应的得到通知了。

前面提到过，Libevent 会在事件主循环中检查标记，来确定是否有触发的 signal，如果标记被设置就处理这些 signal，这段代码在各个具体的 I/O 机制中，以 Epoll 为例，在 epoll\_dispatch()函数中，代码片段如下：

```
res = epoll_wait(epollop->epfd, events, epollop->nevents, timeout);
if (res == -1) {
    if (errno != EINTR) {
        event_warn("epoll_wait");
        return (-1);
    }
    evsignal_process(base); // 处理signal事件
    return (0);
} else if (base->sig.evsignal_caught) {
    evsignal_process(base); // 处理signal事件
}
```

完整的处理框架如下所示：



注 1: libevent 中, 初始化阶段并不注册读 socket 的读事件, 而是在注册信号阶段才会测试并注册;

注 2: libevent 中, 检查 I/O 事件是在各系统 I/O 机制的 dispatch()函数中完成的, 该 dispatch()函数在 event\_base\_loop()函数中被调用;

## 4 evsignal\_info 结构体

Libevent 中 Signal 事件的管理是通过结构体 evsignal\_info 完成的, 结构体位于 evsignal.h 文件中, 定义如下:

```
struct evsignal_info {
    struct event ev_signal;
    int ev_signal_pair[2];
    int ev_signal_added;
    volatile sig_atomic_t evsignal_caught;
    struct event_list evsigevents[NSIG];
    sig_atomic_t evsigcaught[NSIG];
#ifdef HAVE_SIGACTION
    struct sigaction **sh_old;
#else
    ev_sighandler_t **sh_old;
#endif
    int sh_old_max;
};
```

下面详细介绍一下个字段的含义和作用:

- 1) ev\_signal, 为 socket pair 的读 socket 向 event\_base 注册读事件时使用的 event 结构体;
- 2) ev\_signal\_pair, socket pair 对, 作用见第一节介绍;
- 3) ev\_signal\_added, 记录 ev\_signal 事件是否已经注册了;
- 4) evsignal\_caught, 是否有信号发生的标记; 是 volatile 类型, 因为它会在另外的线程中被修改;
- 5) evsigevents[NSIG], 数组, evsigevents[signo]表示注册到信号 signo 的事件链表;
- 6) evsigcaught[NSIG], 具体记录每个信号触发的次数, evsigcaught[signo]是记录信号 signo 被触发的次数;
- 7) sh\_old 记录了原来的 signal 处理函数指针, 当信号 signo 注册的 event 被清空时, 需要重新设置其处理函数;

evsignal\_info 的初始化包括, 创建 socket pair, 设置 ev\_signal 事件 (但并没有注册, 而是等到有信号注册时才检查并注册), 并将所有标记置零, 初始化信号的注册事件链表指针等。

## 5 注册、注销 signal 事件

注册 signal 事件是通过 evsignal\_add(struct event \*ev)函数完成的, libevent 对所有的信号注册同一个处理函数 evsignal\_handler(), 该函数将在下一段介绍, 注册过程如下:

- 1 取得 ev 要注册到的信号 signo;

- 2 如果信号 `signo` 未被注册，那么就为 `signo` 注册信号处理函数 `evsignal_handler()`;
- 3 如果事件 `ev_signal` 还没哟注册，就注册 `ev_signal` 事件;
- 4 将事件 `ev` 添加到 `signo` 的 `event` 链表中;

从 `signo` 上注销一个已注册的 `signal` 事件就更简单了，直接从其已注册事件的链表中移除即可。如果事件链表已空，那么就恢复旧的处理函数;

下面的讲解都以 `signal()` 函数为例，`sigaction()` 函数的处理和 `signal()` 相似。

处理函数 `evsignal_handler()` 函数做的事情很简单，就是记录信号的发生次数，并通知 `event_base` 有信号触发，需要处理:

```
static void evsignal_handler(int sig)
{
    int save_errno = errno; // 不覆盖原来的错误代码
    if (evsignal_base == NULL) {
        event_warn("%s: received signal %d, but have no base configured",
            func__, sig);
        return;
    }
    // 记录信号sig的触发次数，并设置event触发标记
    evsignal_base->sig.evsigcaught[sig]++;
    evsignal_base->sig.evsignal_caught = 1;
#ifdef HAVE_SIGACTION
    signal(sig, evsignal_handler); // 重新注册信号
#endif

    // 向写socket写一个字节数据，触发event_base的I/O事件，从而通知其有
    信号触发，需要处理
    send(evsignal_base->sig.ev_signal_pair[0], "a", 1, 0);
    errno = save_errno; // 错误代码
}
```

## 5 小节

本节介绍了 `libevent` 对 `signal` 事件的具体处理框架，包括事件注册、删除和 `socket pair` 通知机制，以及是如何将 `Signal` 事件集成到事件主循环之中的。

## 九 集成定时器事件

现在再来详细分析 libevent 中 I/O 事件和 Timer 事件的集成，与 Signal 相比，Timer 事件的集成会直观和简单很多。Libevent 对堆的调整操作做了一些优化，本节还会描述这些优化方法。

### 1 集成到事件主循环

因为系统的 I/O 机制像 `select()` 和 `epoll_wait()` 都允许程序制定一个最大等待时间（也称为最大超时时间）`timeout`，即使没有 I/O 事件发生，它们也保证能在 `timeout` 时间内返回。

那么根据所有 Timer 事件的最小超时时间来设置系统 I/O 的 `timeout` 时间；当系统 I/O 返回时，再激活所有就绪的 Timer 事件就可以了，这样就能将 Timer 事件完美的融合到系统的 I/O 机制中了。

具体的代码在源文件 `event.c` 的 `event_base_loop()` 中，现在就对比代码来看看这一处理方法：

```
if (!base->event_count_active && !(flags & EVLOOP_NONBLOCK)) {
    // 根据Timer事件计算evsel->dispatch的最大等待时间
    timeout_next(base, &tv_p);
} else {
    // 如果还有活动事件，就不要等待，让evsel->dispatch立即返回
    evutil_timerclear(&tv);
}
// ...
// 调用select() or epoll_wait() 等待就绪I/O事件
res = evsel->dispatch(base, evbase, tv_p);
// ...
// 处理超时事件，将超时事件插入到激活链表中
timeout_process(base);
```

`timeout_next()` 函数根据堆中具有最小超时值的事件和当前时间来计算等待时间，下面看看代码：

```
static int timeout_next(struct event_base *base, struct timeval **tv_p)
{
    struct timeval now;
    struct event *ev;
    struct timeval *tv = *tv_p;

    // 堆的首元素具有最小的超时值
    if ((ev = min_heap_top(&base->timeheap)) == NULL) {
        // 如果没有定时事件，将等待时间设置为NULL，表示一直阻塞直到有I/O事件发生
        *tv_p = NULL;
    }
```



```

        return (0);
    }
    // 取得当前时间
    gettimeofday(&base, &now);
    // 如果超时时间<=当前值，不能等待，需要立即返回
    if (evutil_timercmp(&ev->ev_timeout, &now, <=)) {
        evutil_timerclear(tv);
        return (0);
    }
    // 计算等待的时间=当前时间-最小的超时时间
    evutil_timersub(&ev->ev_timeout, &now, tv);

    return (0);
}

```

## 2 Timer 小根堆

Libevent 使用堆来管理 Timer 事件，其 key 值就是事件的超时时间，源代码位于文件 min\_heap.h 中。

所有的数据结构书中都有关于堆的详细介绍，向堆中插入、删除元素时间复杂度都是  $O(\lg N)$ ，N 为堆中元素的个数，而获取最小 key 值（小根堆）的复杂度为  $O(1)$ 。堆是一个完全二叉树，基本存储方式是一个数组。

Libevent 实现的堆还是比较轻巧的，虽然我不喜欢这种编码方式（搞一些复杂的表达式）。轻巧到什么地方呢，就以插入元素为例，来对比说明，下面伪代码中的 size 表示当前堆的元素个数：

典型的代码逻辑如下：

```

Heap[size++] ← new; // 先放到数组末尾，元素个数+1
// 下面就是 shift_up() 的代码逻辑，不断的将 new 向上调整
_child = size;
while(_child > 0) // 循环
{
    _parent ← (_child-1)/2; // 计算 parent
    if(Heap[_parent].key < Heap[_child].key)
        break; // 调整结束，跳出循环
    swap(_parent, _child); // 交换 parent 和 child
}

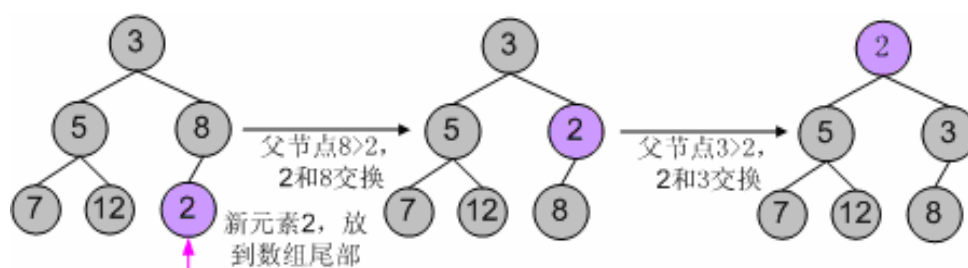
```

而 libevent 的 heap 代码对这一过程做了优化，在插入新元素时，只是为新元素预留了一个位置 hole（初始时 hole 位于数组尾部），但并不立刻将新元素插入到 hole 上，而是不断向上调整 hole 的值，将父节点向下调整，最后确认 hole 就是新元素的所在位置时，才会真正的将新元素插入到 hole 上，因此在调整过程中就比上面的代码少了一次赋值的操作，代码逻辑是：

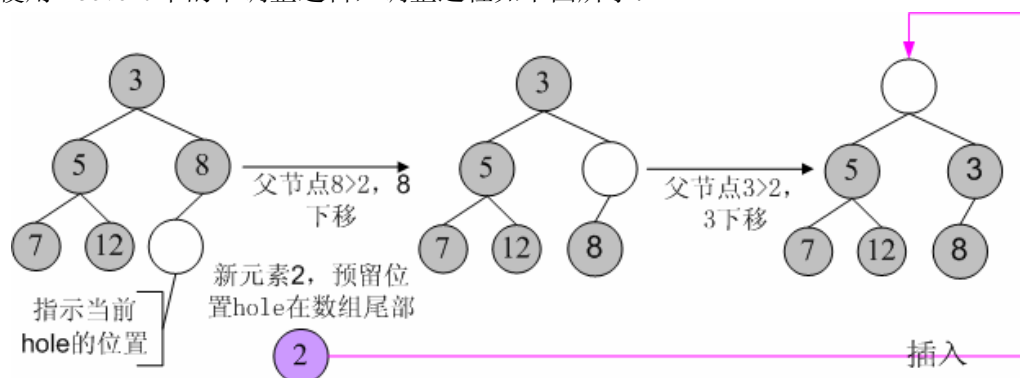
```
// 下面就是 shift_up()的代码逻辑，不断的将 new 的“预留位置” 向上调整
_hole = size; // _hole 就是为 new 预留的位置，但并不立刻将 new 放上
while(_hole>0) // 循环
{
    _parent  $\leftarrow$  (_hole-1)/2; // 计算 parent
    if(Heap[_parent].key < new.key)
        break; // 调整结束，跳出循环
    Heap[_hole] = Heap[_parent]; // 将 parent 向下调整
    _hole = _parent; // 将_hole 调整到_parent
}
Heap[_hole] = new; // 调整结束，将 new 插入到_hole 指示的位置
size++; // 元素个数+1
```

由于每次调整都少做一次赋值操作，在调整路径比较长时，调整效率会比第一种有所提高。libevent 中的 min\_heap\_shift\_up\_()函数就是上面逻辑的具体实现，对应的向下调整函数是 min\_heap\_shift\_down\_()。

举个例子，向一个小根堆 3, 5, 8, 7, 12 中插入新元素 2，使用第一中典型的代码逻辑，其调整过程如下图所示：



使用 libevent 中的堆调整逻辑，调整过程如下图所示：



对于删除和元素修改操作，也遵从相同的逻辑，就不再罗嗦了。

### 3 小节

通过设置系统 I/O 机制的 wait 时间，从而简捷的集成 Timer 事件；主要分析了 libevent 对堆调整操作的优化。

# 十 支持 I/O 多路复用技术

Libevent 的核心是事件驱动、同步非阻塞，为了达到这一目标，必须采用系统提供的 I/O 多路复用技术，而这些在 Windows、Linux、Unix 等不同平台上却各有不同，如何能提供优雅而统一的支持方式，是首要关键的问题，这其实不难，本节就来分析一下。

## 1 统一的关键

Libevent 支持多种 I/O 多路复用技术的关键就在于结构体 `eventop`，这个结构体前面也曾提到过，它的成员是一系列的函数指针，定义在 `event-internal.h` 文件中：

```
struct eventop {  
    const char *name;  
    void *(*init)(struct event_base *); // 初始化  
    int (*add)(void *, struct event *); // 注册事件  
    int (*del)(void *, struct event *); // 删除事件  
    int (*dispatch)(struct event_base *, void *, struct timeval *); // 事件分发  
    void (*dealloc)(struct event_base *, void *); // 注销，释放资源  
    /* set if we need to reinitialize the event base */  
    int need_reinit;  
};
```

在 libevent 中，每种 I/O demultiplex 机制的实现都必须提供这五个函数接口，来完成自身的初始化、销毁释放；对事件的注册、注销和分发。

比如对于 `epoll`，libevent 实现了 5 个对应的接口函数，并在初始化时将 `eventop` 的 5 个函数指针指向这 5 个函数，那么程序就可以使用 `epoll` 作为 I/O demultiplex 机制了。

## 2 设置 I/O demultiplex 机制

Libevent 把所有支持的 I/O demultiplex 机制存储在一个全局静态数组 `eventops` 中，并在初始化时选择使用何种机制，数组内容根据优先级顺序声明如下：

```
/* In order of preference */  
static const struct eventop *eventops[] = {  
#ifdef HAVE_EVENT_PORTS  
    &evportops,  
#endif  
#ifdef HAVE_WORKING_KQUEUE  
    &kqops,  
#endif  
#ifdef HAVE_EPOLL  
    &epollops,  
};
```

```

#endif
#ifdef HAVE_DEVPOLL
    &devpollops,
#endif
#ifdef HAVE_POLL
    &pollops,
#endif
#ifdef HAVE_SELECT
    &selectops,
#endif
#ifdef WIN32
    &win32ops,
#endif
    NULL
};

```

然后 libevent 根据系统配置和编译选项决定使用哪一种 I/O demultiplex 机制，这段代码在函数 `event_base_new()` 中：

```

base->evbase = NULL;
for (i = 0; eventops[i] && !base->evbase; i++) {
    base->evsel = eventops[i];

    base->evbase = base->evsel->init(base);
}

```

可以看出，libevent 在编译阶段选择系统的 I/O demultiplex 机制，而不支持在运行阶段根据配置再次选择。

以 Linux 下面的 `epoll` 为例，实现在源文件 `epoll.c` 中，`eventops` 对象 `epollops` 定义如下：

```

const struct eventop epollops = {
    "epoll",
    epoll_init,
    epoll_add,
    epoll_del,
    epoll_dispatch,
    epoll_dealloc,
    1 /* need reinit */
};

```

变量 `epollops` 中的函数指针具体声明如下，注意到其返回值和参数都和 `eventop` 中的定义严格一致，这是函数指针的语法限制。

```

static void *epoll_init (struct event_base *);
static int epoll_add (void *, struct event *);
static int epoll_del (void *, struct event *);
static int epoll_dispatch(struct event_base *, void *, struct timeval *);
static void epoll_dealloc (struct event_base *, void *);

```

那么如果选择的是 `epoll`，那么调用结构体 `eventop` 的 `init` 和 `dispatch` 函数指针时，实际调用的函数就是 `epoll` 的初始化函数 `epoll_init()` 和事件分发函数 `epoll_dispatch()` 了；

关于 `epoll` 的具体用法这里就不多说了，可以参见介绍 `epoll` 的文章（本人的哈哈）：

<http://blog.csdn.net/sparkliang/archive/2009/11/05/4770655.aspx>

C++ 语言提供了虚函数来实现多态，在 C 语言中，这是通过函数指针实现的。对于各类函数指针的详细说明可以参见文章：

<http://blog.csdn.net/sparkliang/archive/2009/06/09/4254115.aspx>

同样的，上面 `epollops` 以及 `epoll` 的各种函数都直接定义在了 `epoll.c` 源文件中，对外都是不可见的。对于 `libevent` 的使用者而言，完全不会知道它们的存在，对 `epoll` 的使用也是通过 `eventop` 来完成的，达到了信息隐藏的目的。

### 3 小节

支持多种 I/O demultiplex 机制的方法其实挺简单的，借助于函数指针就 OK 了。通过对源代码的分析也可以看出，`Libevent` 是在编译阶段选择系统的 I/O demultiplex 机制的，而不支持在运行阶段根据配置再次选择。

# 十一 时间管理

为了支持定时器，Libevent 必须和系统时间打交道，这一部分的内容也比较简单，主要涉及到时间的加减辅助函数、时间缓存、时间校正和定时器堆的时间值调整等。下面就结合源代码来分析一下。

## 1 初始化检测

Libevent 在初始化时会检测系统时间的类型，通过调用函数 `detect_monotonic()` 完成，它通过调用 `clock_gettime()` 来检测系统是否支持 `monotonic` 时钟类型：

```
static void detect_monotonic(void)
{
    #if defined(HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
        struct timespec  ts;

        if (clock_gettime(CLOCK_MONOTONIC, &ts) == 0)
            use_monotonic = 1; // 系统支持monotonic时间
    #endif
}
```

Monotonic 时间指示的是系统从 boot 后到现在所经过的时间，如果系统支持 Monotonic 时间就将全局变量 `use_monotonic` 设置为 1，设置 `use_monotonic` 到底有什么用，这个在后面说到时间校正时就能看出来了。

## 2 时间缓存

结构体 `event_base` 中的 `tv_cache`，用来记录时间缓存。这个还要从函数 `gettime()` 说起，先来看看该函数的代码：

```
static int gettime(struct event_base *base, struct timeval *tp)
{
    // 如果tv_cache时间缓存已设置，就直接使用
    if (base->tv_cache.tv_sec) {
        *tp = base->tv_cache;
        return (0);
    }

    // 如果支持monotonic，就用clock_gettime获取monotonic时间
    #if defined(HAVE_CLOCK_GETTIME) && defined(CLOCK_MONOTONIC)
        if (use_monotonic) {
            struct timespec  ts;
            if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1)
```

```

        return (-1);
    tp->tv_sec = ts.tv_sec;
    tp->tv_usec = ts.tv_nsec / 1000;
    return (0);
}
#endif
// 否则只能取得系统当前时间
return (evutil_gettimeofday(tp, NULL));
}

```

如果 `tv_cache` 已经设置，那么就直接使用缓存的时间；否则需要再次执行系统调用获取系统时间。

函数 `evutil_gettimeofday()` 用来获取当前系统时间，在 Linux 下其实就是系统调用 `gettimeofday()`；Windows 没有提供函数 `gettimeofday`，而是通过调用 `_ftime()` 来完成的。

在每次系统事件循环中，时间缓存 `tv_cache` 将会被相应的清空和设置，再次来看看下面 `event_base_loop` 的主要代码逻辑：

```

int event_base_loop(struct event_base *base, int flags)
{
    // 清空时间缓存
    base->tv_cache.tv_sec = 0;
    while(!done) {
        timeout_correct(base, &tv); // 时间校正
        // 更新event_tv到tv_cache指示的时间或者当前时间（第一次）
        // event_tv <--- tv_cache
        gettimeofday(base, &base->event_tv);
        // 清空时间缓存-- 时间点1
        base->tv_cache.tv_sec = 0;
        // 等待I/O事件就绪
        res = evsel->dispatch(base, evbase, tv_p);
        // 缓存tv_cache存储了当前时间的值-- 时间点2
        // tv_cache <--- now
        gettimeofday(base, &base->tv_cache);
        // .. 处理就绪事件
    }
    // 退出时也要清空时间缓存
    base->tv_cache.tv_sec = 0;
    return (0);
}

```

时间 `event_tv` 指示了 `dispatch()` 上次返回，也就是 I/O 事件就绪时的时间，第一次进入循环时，由于 `tv_cache` 被清空，因此 `gettimeofday()` 执行系统调用获取当前系统时间；而后将会更新为 `tv_cache` 指示的时间。

时间 `tv_cache` 在 `dispatch()` 返回后被设置为当前系统时间，因此它缓存了本次 I/O 事件就绪时的时间（`event_tv`）。

从代码逻辑里可以看出 `event_tv` 取得的是 `tv_cache` 上一次的值，因此 `event_tv` 应该小于

tv\_cache 的值。

设置时间缓存的优点是不必每次获取时间都执行系统调用，这是个相对费时的操作；在上面标注的时间点 2 到时间点 1 的这段时间（处理就绪事件时），调用 `gettime()` 取得的都是 tv\_cache 缓存的时间。

### 3 时间校正

如果系统支持 `monotonic` 时间，该时间是系统从 `boot` 后到现在所经过的时间，因此不需要执行校正。

根据前面的代码逻辑，如果系统不支持 `monotonic` 时间，用户可能会手动调整时间，如果时间被向前调整了（MS 前面第 7 部分讲成了向后调整，要改正），比如从 5 点调整到了 3 点，那么在时间点 2 取得的值可能会小于上次的时间，这就需要调整了，下面来看看校正的具体代码，由函数 `timeout_correct()` 完成：

```
static void timeout_correct(struct event_base *base, struct timeval *tv)
{
    struct event **pev;
    unsigned int size;
    struct timeval off;
    if (use_monotonic) // monotonic时间就直接返回，无需调整
        return;
    gettime(base, tv); // tv <-- tv_cache
    // 根据前面的分析可以知道event_tv应该小于tv_cache
    // 如果tv < event_tv表明用户向前调整时间了，需要校正时间
    if (evutil_timercmp(tv, &base->event_tv, >=)) {
        base->event_tv = *tv;
        return;
    }
    // 计算时间差值
    evutil_timersub(&base->event_tv, tv, &off);

    // 调整定时事件小根堆
    pev = base->timeheap.p;
    size = base->timeheap.n;
    for (; size-- > 0; ++pev) {
        struct timeval *ev_tv = &(*pev).ev_timeout;
        evutil_timersub(ev_tv, &off, ev_tv);
    }
    base->event_tv = *tv; // 更新event_tv为tv_cache
}
```

在调整小根堆时，因为所有定时事件的时间值都会被减去相同的值，因此虽然堆中元素的时间键值改变了，但是相对关系并没有改变，不会改变堆的整体结构。因此只需要遍历堆中的所有元素，将每个元素的时间键值减去相同的值即可完成调整，不需要重新调整堆的结构。

当然调整完后，要将 `event_tv` 值重新设置为 `tv_cache` 值了。



## 4 小节

主要分析了一下 libevent 对系统时间的处理，时间缓存、时间校正和定时堆的时间值调整等，逻辑还是很简单的，时间的加减、设置等辅助函数则非常简单，主要在头文件 evutil.h 中，就不再多说了。

# 十二 让 libevent 支持多线程

Libevent 本身不是多线程安全的，在多核的时代，如何能充分利用 CPU 的能力呢，这一节来说说如何在多线程环境中使用 libevent，跟源代码并没有太大的关系，纯粹是使用上的技巧。

## 1 错误使用示例

在多核的 CPU 上只使用一个线程始终是对不起 CPU 的处理能力啊，那好吧，那就多创建几个线程，比如下面的简单服务器场景。

- 1 主线程创建工作线程 1；
- 2 接着主线程监听在端口上，等待新的连接；
- 3 在线程 1 中执行 event 事件循环，等待事件到来；
- 4 新连接到来，主线程调用 libevent 接口 event\_add 将新连接注册到 libevent 上；
- ... ..

上面的逻辑看起来没什么错误，在很多服务器设计中都可能用到主线程和工作线程的模式....

可是就在线程 1 注册事件时，主线程很可能也在操作事件，比如删除，修改，通过 libevent 的源代码也能看到，没有同步保护机制，问题麻烦了，看起来不能这样做啊，难道只能使用单线程不成！？

## 2 支持多线程的几种模式

Libevent 并不是线程安全的，但这不代表 libevent 不支持多线程模式，其实方法在前面已经将 signal 事件处理时就接触到了，那就是消息通知机制。

一句话，“你发消息通知我，然后再由我在合适的时间来处理”；

说到这就再多说几句，再打个比方，把你自己比作一个工作线程，而你的头是主线程，你有一个消息信箱来接收别人发给你的消息，当时头有个新任务要指派给你。

### 2.1 暴力抢占

那么第一节中使用的多线程方法相当下面的流程：

- 1 当时你正在做事，比如在写文档；
- 2 你的头找到了一个任务，要指派给你，比如帮他搞个 PPT，哈；
- 3 头命令你马上搞 PPT，你这是不得不停止手头的工作，把 PPT 搞定了再接着写文档；
- ...

### 2.2 纯粹的消息通知机制

那么基于纯粹的消息通知机制的多线程方式就像下面这样：

- 1 当时你正在写文档；
- 2 你的头找到了一个任务，要指派给你，帮他搞个 PPT；
- 3 头发个消息到你信箱，有个 PPT 要帮他搞定，这时你并不鸟他；
- 4 你写好文档，接着检查消息发现头有个 PPT 要你搞定，你开始搞 PPT；

...

第一种的好处是消息可以立即得到处理,但是很方法很粗暴,你必须立即处理这个消息,所以你必须处理好切换问题,省得把文档上的内容不小心写到 PPT 里。在操作系统的进程通信中,消息队列(消息信箱)都是操作系统维护的,你不必关心。

第二种的优点是消息通过消息通知,切换问题省心了,不过消息是不能立即处理的(基于消息通知机制,这个总是难免的),而且所有的内容都通过消息发送,比如 PPT 的格式、内容等等信息,这无疑增加了通信开销。

### 2.3 消息通知+同步层

有个折中机制可以减少消息通信的开销,就是提取一个同步层,还拿上面的例子来说,你把工作安排都存放在一个工作队列中,而且你能够保证“任何人把新任务扔到这个队列”,“自己取出当前第一个任务”等这些操作都能够保证不会把队列搞乱(其实就是个加锁的队列容器)。

再来看看处理过程和上面有什么不同:

- 1 当时你正在写文档;
- 2 你的头找到了一个任务,要指派给你,帮他搞个 PPT;
- 2 头有个 PPT 要你搞定,他把任务 push 到你的工作队列中,包括了 PPT 的格式、内容等信息;
- 3 头发个消息(一个字节)到你信箱,有个 PPT 要帮他搞定,这时你并不鸟他;
- 4 你写好文档,发现有新消息(这预示着有新任务来了),检查工作队列知道头有个 PPT 要你搞定,你开始搞 PPT;

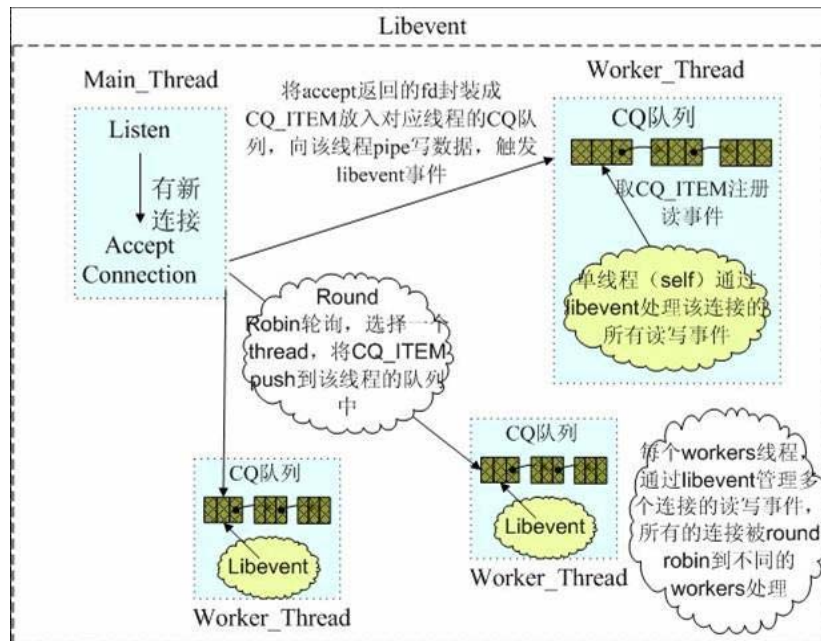
...

工作队列其实就是一个加锁的容器(队列、链表等等),这个很容易实现实现;而消息通知仅需要一个字节,具体的任务都 push 到了在工作队列中,因此想比 2.2 减少了不少通信开销。

多线程编程有很多陷阱,线程间资源的同步互斥不是一两句能说得清的,而且出现 bug 很难跟踪调试;这也有很多的经验和教训,因此如果让我选择,在绝大多数情况下都会选择机制 3 作为实现多线程的方法。

## 3 例子——memcached

Memcached 中的网络部分就是基于 libevent 完成的,其中的多线程模型就是典型的消息通知+同步层机制。下面的图足够说明其多线程模型了,其中有详细的文字说明。



注：该图的具体出处忘记了，感谢原作者。

## 4 小节

本节更是 libevent 的使用方面的技巧，讨论了一下如何让 libevent 支持多线程，以及几种支持多线程的机制，memcached 使用 libevent 的多线程模型。

到此为止，对 libevent 源码的分析也就结束了，其中的基本数据结构比如链表、红黑树等就不再写了，都是经典的数据结构，好参考资料多多。