# 概述

Libevent是一个用C语言编写的、轻量级的开源高性能事件通知库，主要有以下几个亮点：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。

## 特点

Libevent是开源社区一款高性能的I/O框架库，其具有如下特点：

1、跨平台支持。Libevent支持Linux、UNIX和Windows。

2、统一事件源。libevent对i/o事件、信号和定时事件提供统一的处理。

3、线程安全。libevent使用libevent\_pthreads库来提供线程安全支持。

4、基于reactor模式的实现。

## 优点

1、响应快，不必为单个同步事件所阻塞；

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。

# 原理

Libevent确实方便了开发人员，对于定时器、信号处理、关心的文件或者套接字，只需要挂载到event\_base上面，设置好对应的回调函数和参数就可以了，当对应的事件发生时，Libevent会自动调度相应的回调函数进行处理。

参考：<https://blog.csdn.net/century_sunshine/article/details/80213533>

<https://zhuanlan.zhihu.com/p/370666536>

## Reactor模式

### 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；当这些事件发生时，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，还是被据了；“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，还是被据了；你不用自己打电话去问结果，事实上也不能，你没有HR的留联系方式。

### 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；

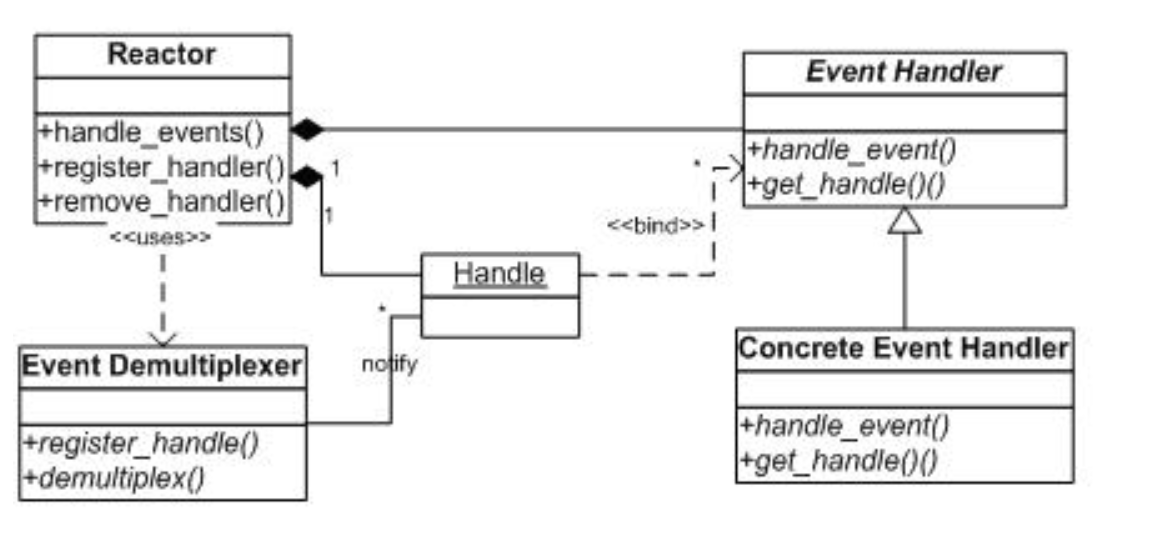
2、编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销；

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。

### 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;

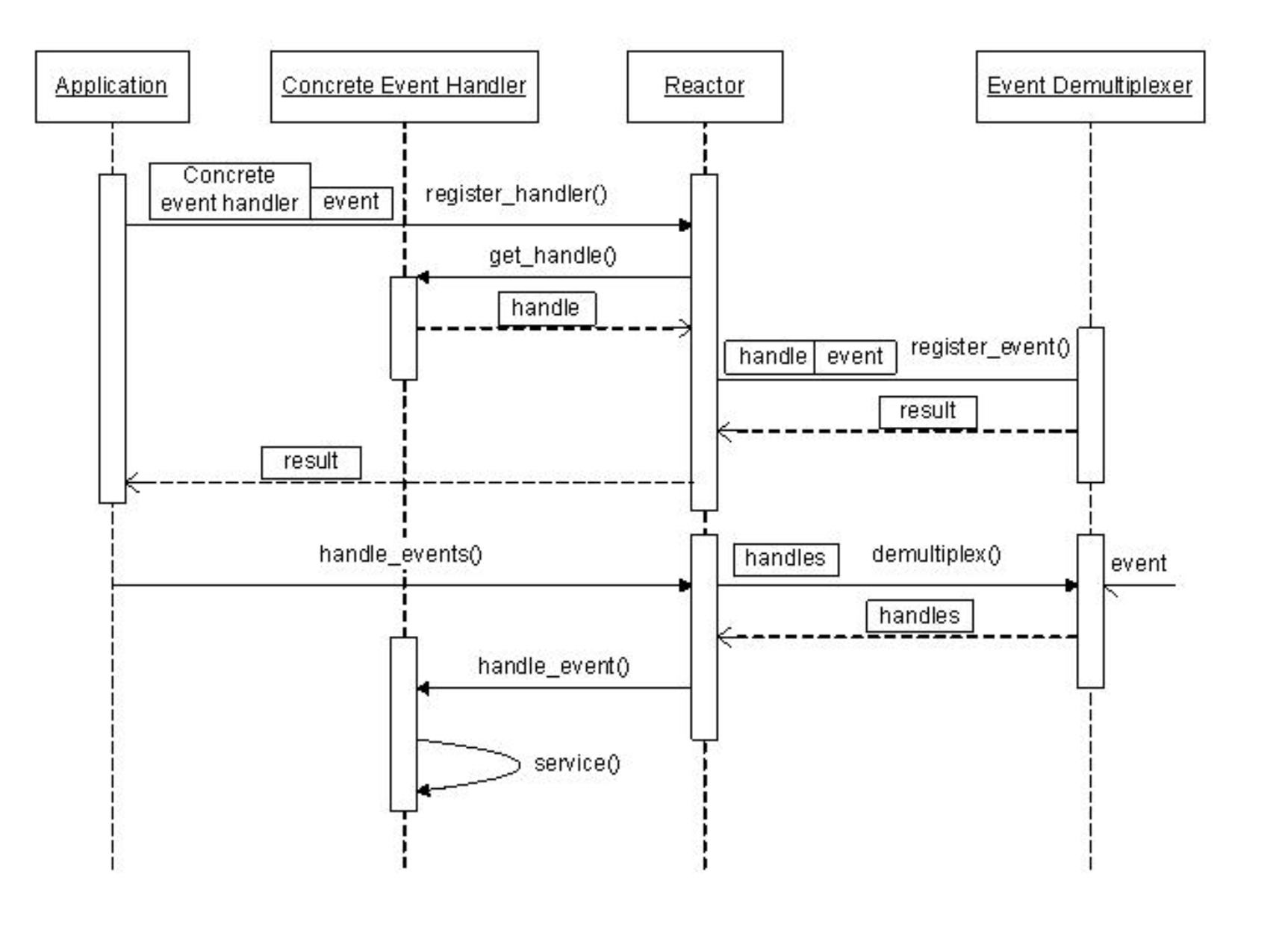
virtual HANDLE get\_handle() = 0;

// ...

};

### 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：



## 基本应用场景

基本应用场景也是使用libevnet的基本流程，下面来考虑一个最简单的场景，使用livevent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

evtimer\_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);

示例代码：

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);

}

## 事件处理流程

当应用程序向libevent注册一个事件后，libevent内部是怎么样进行处理的呢？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；

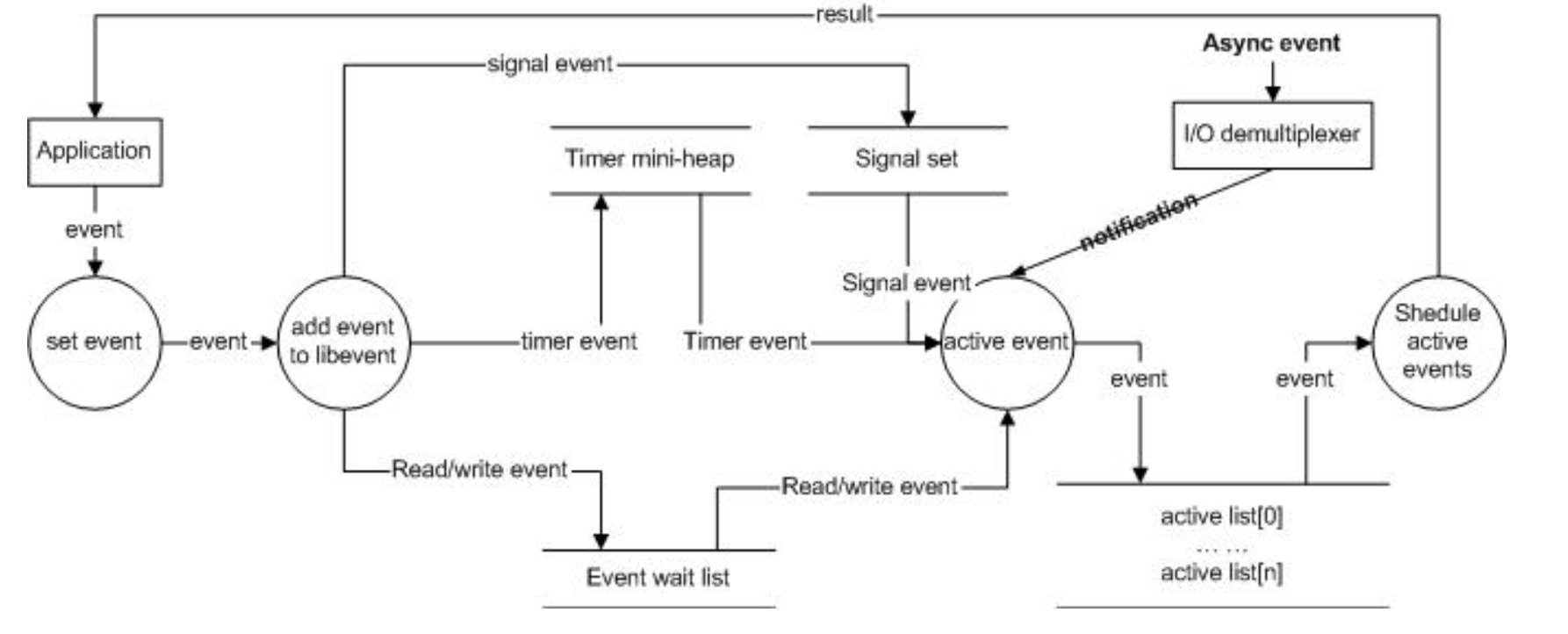
2、向libevent添加该事件event。对于定时事件，libevent使用一个小根堆管理，key为超时时间；对于Signal和I/O事件，libevent将其放入到等待链表（wait list）中，这是一个双向链表结构；

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；



# 源码

## 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。

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；

# 应用

## Memcache

### 原理

1、main [memcached.c]

memcached启动时候执行memcached.c中的main函数，在加载了好长的初始化配置之后，定义并初始化event\_base；

static struct event\_base \*main\_base;

main\_base = event\_init();

然后通过调用memcached\_thread\_init，创建工作者线程：

memcached\_thread\_init(settings.num\_threads, main\_base);

创建定时器clock\_handler(0, 0, 0);，这个基于Libevent创建的定时器每一秒钟执行一次，用以更新current\_time这个表示自从进程启动后的时间长度。

然后针对服务端参数指定的侦听(ip:port/unix socket)类型，分别调用server\_socket\_unix/server\_sockets函数，绑定指定地址，并为创建的socket添加connect事件，核心代码如下

// unix socket

listen\_conn = conn\_new(sfd, conn\_listening, EV\_READ|EV\_PERSIST,1,

local\_transport, main\_base)))

// tcp

listen\_conn\_add=conn\_new(sfd,conn\_listening,EV\_READ|EV\_PERSIST,1,

transport, main\_base))

这个conn\_new不仅仅在这里用以侦听套接字分配资源、创建事件侦听，之后所有客户端连接的套接字也会用这个函数。这个函数最终回调的响应函数是event\_handler，然后最终调用函数drive\_machine，这个函数内部是一个复杂的有限状态机，会处理所有与套接字相关的连接、关闭、读写等操作。

listen套接字当接收到客户请求的时候，如果连接OK，并且没有超过最大连接数目，就调用dispatch\_conn\_new接收请求。这个函数中，会轮询选择要添加的工作线程，然后创建一个等待item，并添加到对应线程的new\_conn\_queue队列上去，然后向这个线程的读取队列里面写入’c’一个字节表明有一个新的请求，然后对应线程管道读事件就会被触发，执行处理回调函数。

主线main\_base进入Libevent事件循环中：

/\* enter the event loop \*/

if (event\_base\_loop(main\_base, 0) != 0) {

retval = EXIT\_FAILURE;

}

2、memcached\_thread\_init [thread.c]

上面我们关注的核心在于调用memcached\_thread\_init这个函数创建nthreads个工作者线程。

typedef struct {

pthread\_t thread\_id; /\* unique ID of this thread \*/

struct event\_base \*base; /\* libevent handle this thread uses \*/

struct event notify\_event; /\* listen event for notify pipe \*/

int notify\_receive\_fd; /\* receiving end of notify pipe \*/

int notify\_send\_fd; /\* sending end of notify pipe \*/

struct thread\_stats stats; /\* Stats generated by this thread \*/

struct conn\_queue \*new\_conn\_queue; /\*queue of new connections to handle\*/

cache\_t \*suffix\_cache; /\* suffix cache \*/

} LIBEVENT\_THREAD;

void memcached\_thread\_init(int nthreads, struct event\_base \*main\_base) {

...

threads = calloc(nthreads, sizeof(LIBEVENT\_THREAD));

dispatcher\_thread.base = main\_base;

dispatcher\_thread.thread\_id = pthread\_self();

for (i = 0; i < nthreads; i++) {

int fds[2];

threads[i].notify\_receive\_fd = fds[0];

threads[i].notify\_send\_fd = fds[1];

setup\_thread(&threads[i]);

/\* Reserve three fds for the libevent base, and two for the pipe \*/

stats.reserved\_fds += 5;

}

/\* Create threads after we've done all the libevent setup. \*/

for (i = 0; i < nthreads; i++) {

create\_worker(worker\_libevent, &threads[i]);

}

...

}

上面把非核心的代码剔除掉，就可以看清memcached\_thread\_init所做的具体工作了。

1）为每个线程创建LIBEVENT\_THREAD结构体，并把自我分发线程的信息记录在dispatcher\_thread中；

2）对每个线程的结构体LIBEVENT\_THREAD初始化，然后通过pipe创建匿名管道，pipefd[0]指向读端，而pipefd[1]指向写端，然后每个线程就通过这个匿名管道同其他线程进行通信；

static void setup\_thread(LIBEVENT\_THREAD \*me) {

...

me->base = event\_init();

/\* Listen for notifications from other threads \*/

event\_set(&me->notify\_event, me->notify\_receive\_fd,

EV\_READ | EV\_PERSIST, thread\_libevent\_process, me);

event\_base\_set(me->base, &me->notify\_event);

event\_add(&me->notify\_event, 0);

me->new\_conn\_queue = malloc(sizeof(struct conn\_queue));

cq\_init(me->new\_conn\_queue);

me->suffix\_cache = cache\_create("suffix", SUFFIX\_SIZE, sizeof(char\*), NULL, NULL);

...

}

其中的setup\_thread函数中，为每一个线程创建一个event\_base，然后添加之前管道的读写时间侦听；同时每个线程还创建了一个等待队列，所有的新请求会添加到这个等待队列上面去。

在匿名管道的读写事件的相应函数thread\_libevent\_process上，会尝试读取一个字节,如果是上面写入的’c’，就表明有待处理的请求，然后就从等待队列new\_conn\_queue中取出一个item，然后处理。处理的方式就是确认这个连接，分配相应的资源，然后再丢到上面的那个event\_handler->drive\_machine的状态机中去!

3）调用create\_worker(worker\_libevent, &threads[i]);实行真正创建线程操作，其内部就是一个pthread\_create；

### 总结

Memcached工作方式可以描述如下：

软件启动的时候，创建event\_base，并且根据设置类型创建侦听的tcp/udp socket或者unix socket，然后为这些套接字创建读侦听事件，加入到event\_base上，等待客户端连接；

创建工作线程池，每个工作线程创建自己的event\_base；创建一个等待队列，新连接的客户请求都会挂在这个队列上；创建一个匿名管道，并为管道创建读写侦听事件；

当新的客户端连接上来有请求时候，主线程的侦听事件回调函数会被激活，条件满足后接受这个连接，然后选取一个工作线程，创建等待item挂到其队列上，然后向其管道写入一个c，对应线程管道读事件被激活，读取一个c，并从队列中取出一个请求处理；

Memcache对所有socket的处理都是event\_handler->drive\_machine中处理的。

可以说，memcached在线程池在等待连接和事件处理中都充分利用了Libevent的异步事件，所以效率是非常之高的。自己的那个线程池，主要是将所有的任务都放到一个链表队列中，当线程发现没有任务的时候，就会用pthread\_cond\_wait阻塞睡眠，当主线程发现等待的任务太多，就会用pthread\_cond\_signal唤醒睡眠线程（不会惊群）。

总体会让人感觉，把任务事先分给各个队列，吞吐量要大一些，让任务阻塞在select、poll、epoll上，会比自己控制睡眠唤醒要高效可靠！

## TDSQL

TDSQL采用基于libevent的事件管理机制进行通信。