## 网络编程简介

**金孟哲**

**jinmengzhe@360.cn**

如何使网络服务器能够处理数以万计的客户端连接，这个问题被称为C10K Problem。在很多系统中，网络框架的性能直接决定了系统的整体性能，因此研究解决高性能网络编程框架问题具有十分重要的意义。

## 一、网络编程模型

在C10K Problem中，给出了一些常见的解决大量并发连接的方案和模型，在此根据自己理解去除了一些不实际的方案，并做了一些整理。

### 1.1、PPC/TPC模型

典型的Apache模型（Process Per Connection，简称PPC），TPC（Thread Per Connection）模型，这两种模型思想类似，就是让每一个到来的连接都一边自己做事直到完成。只是PPC是为每个连接开了一个进程，而TPC开了一个线程。可是当连接多了之后，如此多的进程/线程切换需要大量的开销；这类模型能接受的最大连接数都不会高，一般在几百个左右。

### 1.2、异步网络编程模型

异步网络编程模型都依赖于I/O多路复用模式。一般地,I/O多路复用机制都依赖于一个事件多路分离器(Event Demultiplexer)。**分离器对象可将来自事件源的I/O事件分离出来，并分发到对应的read/write事件处理器(Event Handler)**。开发人员预先注册需要处理的事件及其事件处理器（或回调函数）；事件分离器负责将请求事件传递给事件处理器。两个与事件分离器有关的模式是**Reactor和Proactor**。Reactor模式采用同步IO，而Proactor采用异步IO。

在Reactor中，事件分离器负责等待文件描述符或socket为读写操作准备就绪，然后将就绪事件传递给对应的处理器，最后由处理器负责完成实际的读写工作。

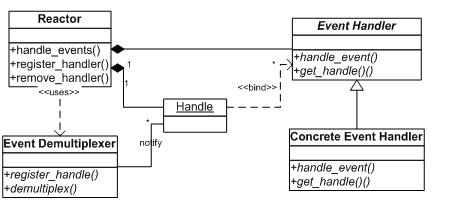
而在Proactor模式中，处理器--或者兼任处理器的事件分离器，只负责发起异步读写操作。**IO操作本身由操作系统来完成**。传递给操作系统的参数需要包括用户定义的数据缓冲区地址和数据大小，操作系统才能从中得到写出操作所需数据，或写入从socket读到的数据。事件分离器捕获IO操作完成事件，然后将事件传递给对应处理器。

* 在Reactor中实现读：
  + 注册读就绪事件和相应的事件处理器
  + 事件分离器等待事件
  + 事件到来，激活分离器，分离器调用事件对应的处理器
  + 事件处理器完成实际的读操作，处理读到的数据，注册新事件，然后返还控制权
* 在Proactor中实现读：
  + 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，它关注的是完成事件。
  + 事件分离器等待操作完成事件
  + 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成。
  + 事件分离器呼唤处理器。
  + 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

可以看出，两个模式的相同点，都是对某个IO事件的事件通知(即告诉某个模块，这个IO操作可以进行或已经完成)。在结构上，两者也有相同点：demultiplexor负责提交IO操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调handler；不同点在于，**异步情况下(Proactor)，当回调handler时，表示IO操作已经完成；同步情况下(Reactor)，回调handler时，表示IO设备可以进行某个操作(can read or can write)。**

#### 1.2.1 Reactor模式框架

使用Proactor模式需要操作系统支持异步接口，因此在日常中比较常见的是Reactor模式的系统调用接口。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。



* 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。

* 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。
* 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。

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);**

**// ...**

**};**

* 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。

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;**

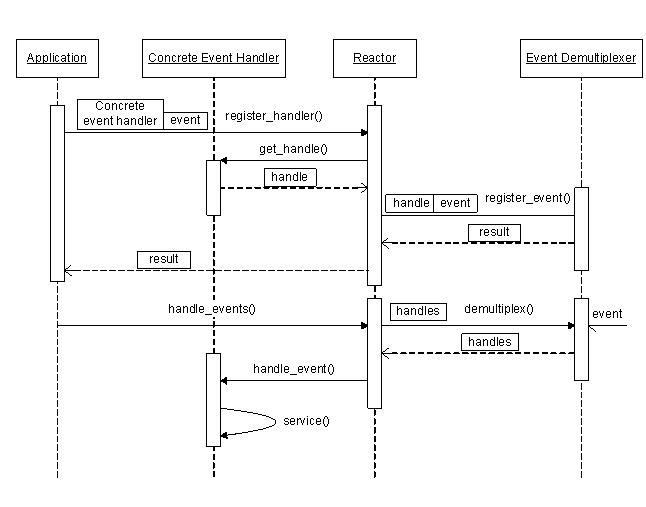
**virtual HANDLE get\_handle() = 0;**

**// ...**

**};**

**1.2.2 Reactor事件处理流程**

前面说过Reactor将事件流“逆置”了，使用Reactor模式后，事件控制流可以参见下面的序列图。



#### 1.3 Select，poll和epoll

在Linux环境中，比较常见的I/O多路复用机制就是Select，poll和epoll，下面对这三种机制进行分析和比较，并对epoll的使用进行介绍。

##### 1.3.1 select模型

1. 最大并发数限制，因为一个进程所打开的FD（文件描述符）是有限制的，由FD\_SETSIZE设置，默认值是1024/2048，因此Select模型的最大并发数就被相应限制了。
2. 效率问题，select每次调用都会线性扫描全部的FD集合，这样效率就会呈现线性下降，把FD\_SETSIZE改大的后果就是所有FD处理都慢慢来
3. 内核/用户空间 内存拷贝问题，如何让内核把FD消息通知给用户空间呢？在这个问题上select采取了内存拷贝方法。

**int res = select(maxfd+1, &readfds, NULL, NULL, 120);**

**if (res > 0) {**

**for (int i = 0; i < MAX\_CONNECTION; i++) {**

**if (FD\_ISSET(allConnection[i],&readfds)) {**

**handleEvent(allConnection[i]);**

**}**

**}**

**}**

##### 1.3.2 poll模型

基本上效率和select是相同的，select缺点的2和3都没有改掉。

##### 1.3.3 epoll模型

1. Epoll没有最大并发连接的限制，上限是最大可以打开文件的数目，这个数字一般远大于2048, 一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。
2. 效率提升，Epoll最大的优点就在于它只管你“活跃”的连接，而跟连接总数无关，应用程序就能直接定位到事件，而不必遍历整个FD集合，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。

**int res = epoll\_wait(epfd, events, 20, 120);**

**for(int i = 0; i < res; i++) {**

**handleEvent(events[n]);**

**}**

1. 内存拷贝，Epoll在这点上使用了“共享内存”，这个内存拷贝也省略了。

###### 1.3.4 使用epoll

Epoll的接口很简单，只有三个函数，十分易用。

**int epoll\_create(int size);**

生成一个epoll专用的文件描述符，其实是申请一个内核空间，用来存放你想关注的socket fd上是否发生以及发生了什么事件。size就是你在这个Epoll fd上能关注的最大socket fd数，大小自定，只要内存足够。

**int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);**

控制某个Epoll文件描述符上的事件：注册、修改、删除。其中参数epfd是epoll\_create()创建Epoll专用的文件描述符。相对于select模型中的FD\_SET和FD\_CLR宏。

**int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);**

等待I/O事件的发生；参数说明：

* epfd:由epoll\_create()生成的Epoll专用的文件描述符；
* epoll\_event:用于回传代处理事件的数组；
* maxevents:每次能处理的事件数；
* timeout:等待I/O事件发生的超时值；
* 返回发生事件数。

上面讲到了Reactor的基本概念、框架和处理流程，并对基于Reactor模型的select，poll和epoll进行了比较分析后，再来对比看网络编程框架就会更容易理解了。

## 二、Libeasy网络编程框架

Libeasy底层使用的是Libev事件库，在分析Libeasy代码前，首先对Libev有相关了解。

#### 3.1 Libev简介

* Libev是什么？

Libev is an event loop: you register interest in certain events (such as a file descriptor being readable or a timeout occurring), and it will manage these event sources and provide your program with events.

Libev是一个event loop：向libev注册感兴趣的events，比如Socket可读事件，libev会对所注册的事件的源进行管理，并在事件发生时触发相应的程序。通过event watcher来注册事件

* libev定义的watcher类型：
* **ev\_io // io 读写类型watcher**
* **ev\_timer // 定时器 类watcher**
* ev\_periodic
* ev\_signal
* ev\_child
* ev\_stat
* ev\_idle
* ev\_prepare
* ev\_check
* ev\_embed
* ev\_fork
* ev\_cleanup
* **ev\_async // 线程同步信号watcher**

在libev中watcher还能支持优先级

###### 3.1.1 libev使用

下面以一个简单例子程序说明libev的使用。这段程序实现从标准输入异步读取数据，5.5秒内没有数据到来则超时的功能。

**#include <ev.h>**

**#include <stdio.h>**

**ev\_io stdin\_watcher;**

**ev\_timer timeout\_watcher;**

**// all watcher callbacks have a similar signature**

**// this callback is called when data is readable on stdin**

**static void stdin\_cb (EV\_P\_ ev\_io \*w, int revents) {**

**puts ("stdin ready");**

**// for one-shot events, one must manually stop the watcher**

**// with its corresponding stop function.**

**ev\_io\_stop (EV\_A\_ w);**

**// this causes all nested ev\_run's to stop iterating**

**ev\_break (EV\_A\_ EVBREAK\_ALL);**

**}**

**// another callback, this time for a time-out**

**static void timeout\_cb (EV\_P\_ ev\_timer \*w, int revents) {**

**puts ("timeout");**

**// this causes the innermost ev\_run to stop iterating**

**ev\_break (EV\_A\_ EVBREAK\_ONE);**

**}**

**int main (void) {**

**// use the default event loop unless you have special needs**

**struct ev\_loop \*loop = EV\_DEFAULT;**

**// initialise an io watcher, then start it**

**// this one will watch for stdin to become readable**

**ev\_io\_init (&stdin\_watcher, stdin\_cb, /\*STDIN\_FILENO\*/ 0, EV\_READ);**

**ev\_io\_start (loop, &stdin\_watcher);**

**// initialise a timer watcher, then start it**

**// simple non-repeating 5.5 second timeout**

**ev\_timer\_init (&timeout\_watcher, timeout\_cb, 5.5, 0.);**

**ev\_timer\_start (loop, &timeout\_watcher);**

**// now wait for events to arrive**

**ev\_run (loop, 0);**

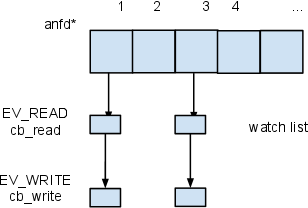
**// break was called, so exit**

**return 0;**

**}**

###### 3.1.2 Libev和Libevent比较

libevent和libev架构近似相同，对于非定时器类型，libevent使用**双向链表**管理，而libev则是使用**数组**来管理。如我们所知，新的fd总是系统可用的最小fd，所以这个长度可以进行大小限制的，我们用一个连续的数组来存储fd/watch 信息。如下图，我们用anfd[fd]就可以找到对应的fd/watcher 信息，当然可能遇到anfd超出我们的buffer长度情形，这是我们用类似relloc 的函数来做数组迁移、扩大容量，但这种概率是很小的，所以其对系统性能的影响可以忽略不计。



我们用anfd[fd]找到的结构体中，有一个指向io\_watch\_list的头指针，以epoll为例，当epoll\_wait返回一个fd\_event时 ，我们就可以直接定位到对应fd的watch\_list，这个watch\_list的长度一般不会超过3 ，fd\_event会有一个导致触发的事件，我们用这个事件依次和各个watch注册的event做 **“&” 操作**, 如果不为0，则把对应的watch加入到待处理队列pendings中(当我们启用watcher优先级模式时，pendings是个2维数组，此时仅考虑普通模式)所以我们可以看到，这个操作是非常非常快。



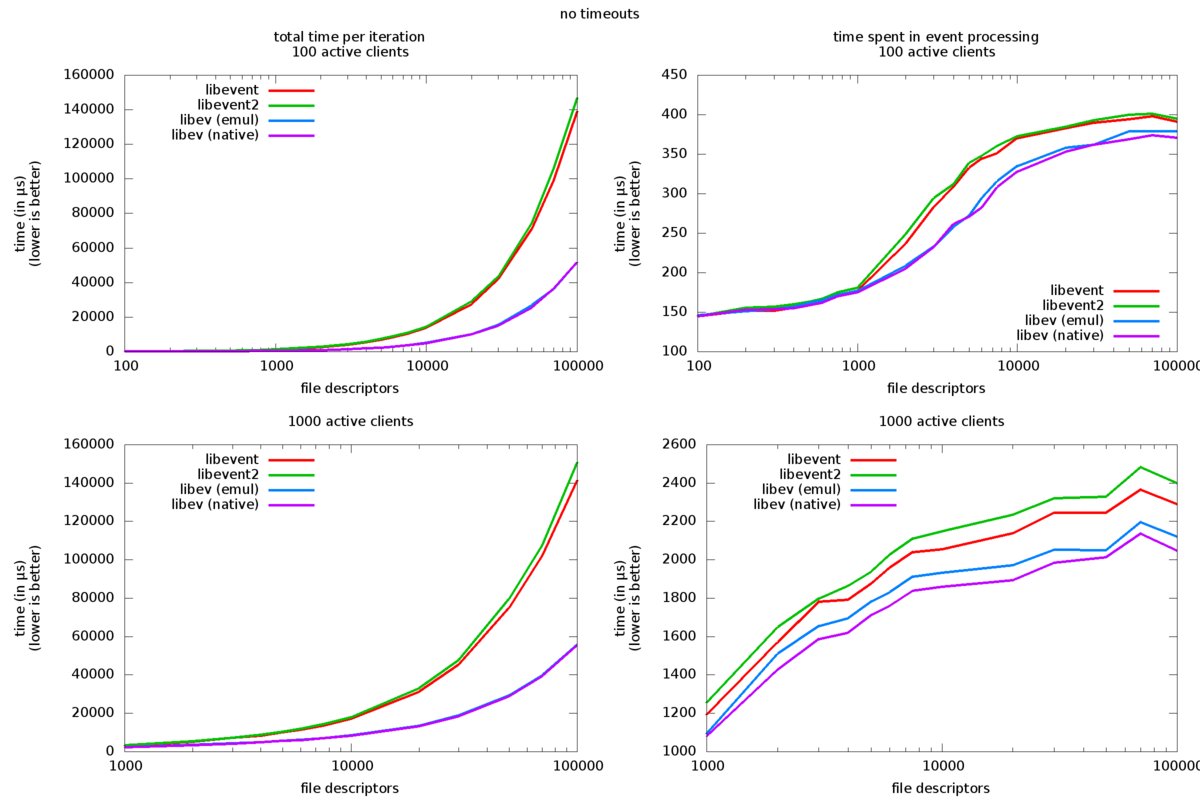
再看添加watch的场景，把watch插入到相应的链表中，这个操作也是直接定位，然后在fdchange队列中，加入对应的fd（如果这个fd已经被添加过，则不会发生这一步，我们通过anfd[fd]中一个bool 值来判断）

注意，假如我们在某个fd上已经有个watch 注册了read事件，这时我们又再添加一个watch，还是read 事件，但是不同的回调函数，在此种情况下，我们不应该调用epoll\_ctrl 之类的系统调用，因为我们的events集合是没有改变的，所以为了达到这个目的，anfd[fd]结构体中，还有一个events事件，它是原先的**所有watcher的事件的“|”操作**，向系统的epoll从新添加描述符的操作是在下次事件迭代开始前进行的，当我们依次扫描fdchangs，找到对应的anfd结构，如果发现先前的events与当前所有的watcher的“|”操作结果不等，则表示我们需要调用epoll\_ctrl之类的函数来进行更改，反之不做操作，作为一条原则，在调用系统调用前，我们已经做了充分的检查，确保不进行多余的系统调用。

再来看删除和更新一个watcher造作，基于以上分析，这个操作也是近乎O(1) 的，当然，如果events事件更改，可能会发生一次系统调用。

所以我们对io watcher的操作，在我们的用户层面上，几乎总是是O(1)的复杂度，当然如果牵涉到epoll 文件结构的更新，我们的系统调用 epoll\_ctrl 在内核中还是 O(lgn)的复杂度，但我们已经在我们所能掌控的范围内做到最好了。

###### 3.1.3 性能测试对比



结论：The cost for setting up or changing event watchers is clearly much higher for libevent than for libev，详细性能对比测试参考这<http://libev.schmorp.de/bench.html>

#### 3.2 libeasy

3.2.2 Server端使用

1、启动流程

**eio\_ = easy\_eio\_create(eio\_, io\_thread\_count\_);**

easy\_eio\_create(eio\_, io\_thread\_count\_)做了如下几件事：

1. 分配一个easy\_pool\_t的内存区，存放easy\_io\_t对象
2. 设置一些tcp参数，比如tcp\_nodelay（tcp\_cork），cpu亲核性等参数
3. 分配线程池的内存区并初始化
4. 对每个线程构建client\_list，client\_array, 初始化双向链表conn\_list session\_list request\_list
5. 设置listen watcher的ev回调函数为easy\_connection\_on\_listen
6. 调用easy\_baseth\_init初始化io线程

**easy\_listen\_t\* listen = easy\_connection\_add\_listen(eio\_, NULL, port\_, &handler\_);**

1. 从eio->pool中为easy\_listen\_t和listen watcher（在这里listen的watcher数默认为2个）分配空间
2. 开始监听某个地址
3. 初始化每个read\_watcher
4. 关注listen fd的读事件，设置其回调函数easy\_connection\_on\_accep（在这里仅仅是初始化read\_watcher, 还没有激活，激活在每个IO线程启动调用easy\_io\_on\_thread\_start的时候做。一旦激活后，当有连接到来的时候，触发easy\_connection\_on\_accept）

**rc = easy\_eio\_start(eio\_);**

1. 调用pthread\_create启动每个io线程，线程执行函数easy\_io\_on\_thread\_start，在easy\_io\_on\_thread\_start中
   1. 设置io线程的cpu亲核性sched\_setaffinity
   2. 如果不是listen\_all或者只有一个线程，则发出ev\_async\_send唤醒下一个线程的listen\_watcher(实现连接请求的负载均衡)
2. 线程执行ev\_run

**easy\_eio\_wait(eio\_);**

调用pthead\_join等待线程结束

2、处理流程

* 当连接到来时触发easy\_connection\_on\_accept

1. 调用accept获得连接fd，构建connection（easy\_connection\_new），设置非阻塞，初始化connection参数和read、write、timeout的watcher
2. 切换listen线程，从自己切换到下一个io线程，调用ev\_async\_send激活下一个io线程的listen\_watcher，实现负载均衡
3. 将connection加入到线程的connected\_list线程列表中，并开启该连接上的read、write、timeout的watcher

* 当数据包到来时触发easy\_connection\_on\_readable回调函数

1. 检查当前IO线程同时正在处理的请求是否超过EASY\_IOTH\_DOING\_REQ\_CNT(8192)，当前连接上的请求数是否超过EASY\_CONN\_DOING\_REQ\_CNT（1024），如果超过，则调用easy\_connection\_destroy(c)将连接销毁掉, 提供了一种负载保护机制
2. 构建message空间
3. 调用read读取socket数据
4. 作为服务端调用easy\_connection\_do\_request
   1. 从message中解包
   2. 调用easy\_connection\_recycle\_message看是否需要释放老的message，构建新的message空间
   3. 调用hanler的process处理数据包，如果返回easy\_ok则调用easy\_connection\_request\_done
   4. 对发送数据进行打包
   5. 对返回码是EASY\_AGAIN的request将其放入session\_list中
   6. 对返回码是EASY\_OK的request将其放入request\_done\_list中，更新统计计数
   7. 统计计数更新
   8. 调用easy\_connection\_write\_socket发送数据包
   9. 调用easy\_connection\_evio\_start中ev\_io\_start(c->loop, &c->read\_watcher);开启该连接的读watcher
   10. 调用easy\_connection\_redispatch\_thread进行负载均衡

如果负载均衡被禁或者该连接的message\_list和output不为空，则直接返回，否则调用easy\_thread\_pool\_rr从线程池中选择一个io线程，将该连接从原来io线程上移除（停止读写timeout 的watcher），将该连接加入到新的io线程中的conn\_list中，调用ev\_async\_send唤醒新的io线程，在easy\_connection\_on\_wakeup中调用easy\_connection\_evio\_start将该连接的read、write、timeou的watcher再打开。

* 当socket可写时触发easy\_connection\_on\_writable回调函数：

1. 调用easy\_connection\_write\_socket写数据
2. 如果没有数据可写，将该连接的write\_watcher停掉

3.2.3 客户端使用

libeasy作为客户端时，将每个发往libeasy服务器端的请求包封装成一个session(easy\_session\_t)，客户端将这个session放入连接的队列中然后返回，随后收到包后，将相应的session从连接的发送队列中删除。详细流程如下：

**easy\_session\_t \*easy\_session\_create(int64\_t asize)**

这个函数主要就做了一件事分配一个内存池easy\_pool\_t，在内存池头部放置一个easy\_session\_t，剩下部分存放实际的数据包Packet，然后将session的type设置为EASY\_TYPE\_SESSION。

###### 异步请求

**int easy\_client\_dispatch(easy\_io\_t \*eio, easy\_addr\_t addr, easy\_session\_t \*s)**

1. 根据socket addr从线程池中选择一个线程,将session加入该线程的session\_list，然后将该线程唤醒
2. 线程唤醒后调用easy\_connection\_send\_session\_list
3. 其中首先调用easy\_connection\_do\_client，这里首先在该线程的client\_list中查找该addr的client，如果没找到，则新建一个client，初始化将其加入client\_list，如果该client的connect未建立，调用easy\_connection\_do\_connect建立该连接，然后返回该连接
4. easy\_connection\_do\_connect中首先创建一个新的connection结构，和一个socket，设置非阻塞，并调用connect进行连接，初始化该连接的read、write、timeout watcher（连接建立前是write，建立后是read）
5. 调用easy\_connection\_session\_build，其中调用encode函数对数据包进行打包，调用easy\_hash\_dlist\_add(c->send\_queue, s->packet\_id, &s->send\_queue\_hash, &s->send\_queue\_list)将这个session添加到连接的发送队列中。这个函数将session添加到发送队列的同时，同时将相应的项添加到hash表的相应的bucket的链表头
6. 开启timeout watcher
7. 调用easy\_connection\_write\_socket发送数据包

* 当回复数据包到达触发easy\_connection\_on\_readable回调函数时

1. 初始化一个easy\_message\_t存放数据包
2. 从内核缓冲区读入数据到应用层输入缓冲区中，然后调用easy\_connection\_do\_response进行处理
3. 先解包，将该packet\_id数据包从发包队列中删除，更新统计信息，停止timeout watcher，
4. 如果是同步请求，则调用session的process函数，从而调用easy\_client\_wait\_process函数，唤醒客户端接收数据包

* 当超时时间到还没有收到回复数据包时触发easy\_connection\_on\_timeout\_mesg回调函数

1. 从发送队列中删除请求数据包
2. 调用session的process函数，从而调用easy\_client\_wait\_process函数，唤醒客户端接
3. 释放此连接

###### 同步请求

**void \*easy\_client\_send(easy\_io\_t \*eio, easy\_addr\_t addr, easy\_session\_t \*s)**

同步请求是通过异步请求实现的，easy\_client\_send方法封装了异步请求接口easy\_client\_dispatch

1. easy\_client\_send将session的process置为easy\_client\_wait\_process方法
2. 初始化一个easy\_client\_wait\_t wobj
3. 调用easy\_client\_dispatch方法发送异步请求
4. 客户端调用wait在wobj包装的信号量上等待
5. 当这个请求收到包的时候触发session的process函数，回调easy\_client\_wait\_process方法，其中会给wobj发送信号唤醒客户端，返回session封装的请求的ipacket

###### 3.2.4 特性总结

1. 多个IO线程/epoll，大大提升了数据包处理性能，特别是处理小数据包的性能

针对多核处理器，libeasy使用多个IO线程来充分发挥处理器性能，提升IO处理能力。特别是针对小数据包IO处理请求数较多的情况下，性能提升十分明显。

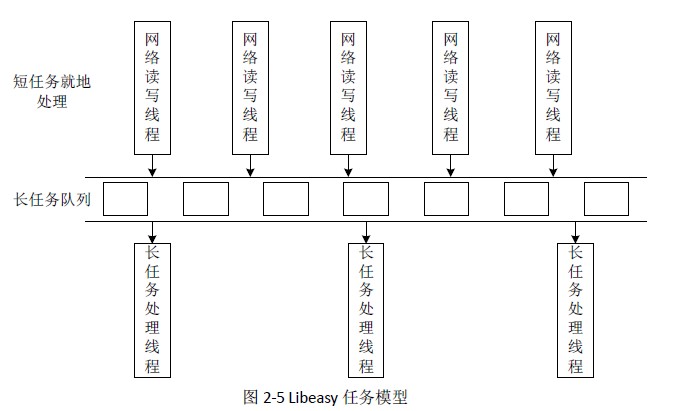
1. 短任务和长任务区分，处理短任务更加高效（编码了内存拷贝，线程切换）

同步处理

对于**短任务**而言，调用用户process回调函数返回EASY\_OK的数据包直接被加入该连接的发送队列，发送给客户端，这样避免了数据包的内存拷贝和线程切换开销。

异步处理

对于耗时较长的**长任务**而言，如果放在网络库的IO线程内执行，可能会阻塞住IO线程，所以需要异步处理。



1. 应用线程CPU亲核性，避免线程调度开销，提升处理性能

开启亲核特性将线程与指定CPU核进行绑定，避免了线程迁移导致的CPU cache失效，同时它允许我们精确控制线程和cpu核的关系，从而根据需要划分CPU核的使用。

**sched\_setaffinity(pid\_t pid, unsigned int cpusetsize, cpu\_set\_t \*mask)**

该函数设置进程为pid的这个进程,让它运行在mask所设定的CPU上.如果pid的值为0,则表示指定的是当前进程,使当前进程运行在mask所设定的那些CPU上.第二个参数cpusetsize是mask所指定的数的长度.通常设定为sizeof(cpu\_set\_t).如果当前pid所指定的进程此时没有运行在mask所指定的任意一个CPU上,则该指定的进程会从其它CPU上迁移到mask的指定的一个CPU上运行.

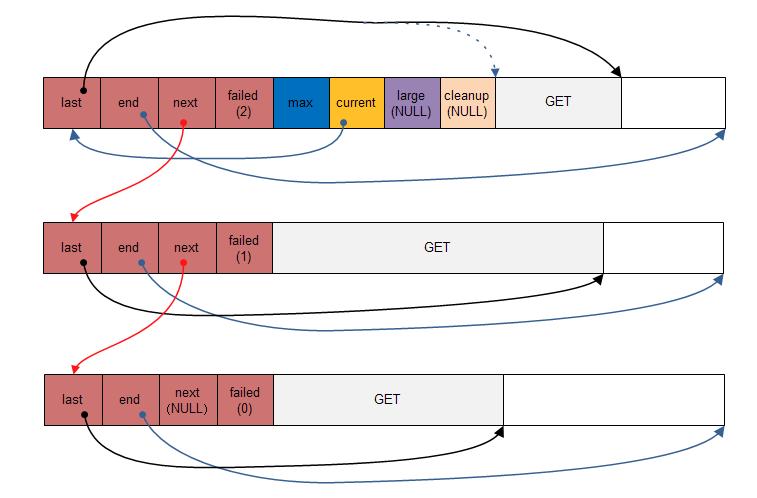
1. 内存管理，减少小内存申请开销，避免内存碎片化

Libeasy的内存管理和nginx一致，有兴趣的可以去学习下，下面大致介绍其思想。

1) 创建一个内存池

2) 分配小块内存(size <= max)

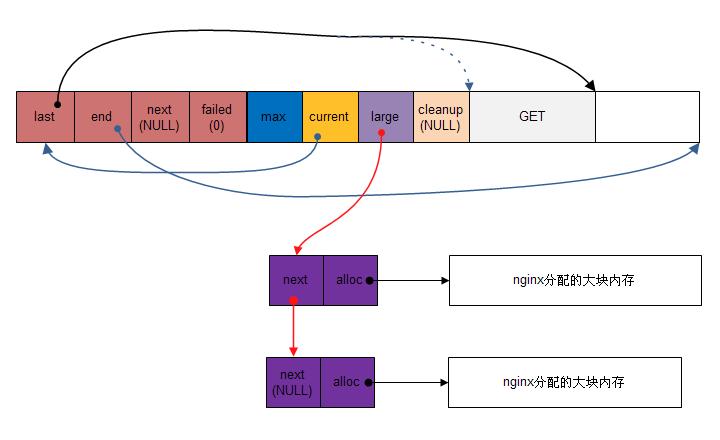
小块内存分配模型：



上图这个内存池模型是由上3个小内存池构成的，由于第一个内存池上剩余的内存不够分配了，于是就创建了第二个新的内存池，第三个内存池是由于前面两个内存池的剩余部分都不够分配，所以创建了第三个内存池来满足用户的需求。由图可见：**所有的小内存池是由一个单向链表维护在一起的**。这里还有两个字段需要关注，failed和current字段。failed表示的是当前这个内存池的剩余可用内存不能满足用户分配请求的次数，如果下一个内存池也不能满足，那么它的failed也会加1，直到满足请求为止（如果没有现成的内存池来满足，会再创建一个新的内存池）。current字段会随着failed的增加而发生改变，如果current指向的内存池的failed达到了一个阈值，current就指向下一个内存池了。

3)、大块内存的分配(size > max)

大块内存的分配请求不会直接在内存池上分配内存来满足，而是直接向操作系统申请这么一块内存（就像直接使用malloc分配内存一样），然后将这块内存挂到内存池头部的large字段下。**内存池的作用在于解决小块内存池的频繁申请问题，对于这种大块内存，是可以忍受直接申请的**。同样，用图形展示大块内存申请模型：



4)、内存释放

nginx利用了web server应用的特殊场景来完成；一个web server总是不停的接受connection和request，所以nginx就将内存池分了不同的等级，有进程级的内存池、connection级的内存池、request级的内存池。也就是说，创建好一个worker进程的时候，同时为这个worker进程创建一个内存池，待有新的连接到来后，就在worker进程的内存池上为该连接创建起一个内存池；连接上到来一个request后，又在连接的内存池上为request创建起一个内存池。这样，在request被处理完后，就会释放request的整个内存池，连接断开后，就会释放连接的内存池。

5)、总结

通过内存的分配和释放可以看出，nginx只是**将小块内存的申请聚集到一起申请(内存池)，然后一起释放，避免了频繁申请小内存，降低内存碎片的产生**等问题。

1. 网络流量自动负载均衡，充分发挥多核性能

1、在连接到来时，正在listen的IO线程接受连接，将其加入本线程的连接队列中，之后主动唤醒下一个线程执行listen。通过**切换listen线程**来使每个线程上处理的连接数大致相同。

2、每一个连接上的流量是不同的，因此在每次有读写请求，**计算该线程上近一段时间内请求速率**，触发负载均衡，将该连接移动到其它线程上，使每个线程处理的IO请求数大致相同。

1. 将encode和decode接口暴露给应用层，实现网络编程框架与协议的分离

Libeasy将网络数据包打包解包接口暴露给应用层，由用户定义数据包内容的格式，实现了网络编程框架与协议的分离，能够支持http等其他协议类型，格式更改更加方便。

1. 底层采用libev，对于事件的注册和更改速度更快

###### 参考资料

* 1. C10K Problem
  2. Unix环境高级编程
  3. Unix网络编程
  4. Nginx、Libevent
  5. Libev [http://pod.tst.eu/http://cvs.schmorp.de/libev/ev.pod#WHAT\_TO\_READ\_WHEN\_IN\_A\_HURRY](http://pod.tst.eu/http:/cvs.schmorp.de/libev/ev.pod#WHAT_TO_READ_WHEN_IN_A_HURRY)
  6. Libeasy源码分析等<http://www.cnblogs.com/foxmailed/archive/2013/02/17/2908180.html>