<https://blog.csdn.net/andyhuabing/article/details/52574411>

首先想一个问题，**为何Redis比Memcached快呢？**  
一般想法：Memcached完全基于内存，而Redis具有持久化保存特性，即使是异步的，Redis也不可能比Memcached快。  
可实际测试情况基本上是：Redis占绝对优势。  
  
  
可能原因有二：  
1、Libevent: Memcached使用、而Redis没有选用。Libevent为了迎合通用性造成代码庞大及牺牲了在特定平台的不少性能。Redis一直坚持设计小巧并去依赖库的思路。  
2、CAS问题：CAS是Memcached中比较方便的一种防止竞争修改资源的方法。  
   CAS实现需要为每个cache key设置一个隐藏的cas token，cas相当value版本号，每次set会将token需要递增，

   因此带来CPU和内存的双重开销、但达到单机10G+ cache以及QPS上万之后这些开销就会给双方相对带来一些

   细微性能差别。

Redis在封装事件的处理采用了Reactor模式，添加了定时事件的处理。  
Redis处理事件是单进程单线程的，而经典Reator模式对事件是串行处理的。  
即如果有一个事件阻塞过久的话会导致整个Redis被阻塞。  
  
  
**下面来简要分析一下Redis AE事件处理模型。**  
  
  
从代码中可以看到它主要支持了epoll、select、kqueue、以及基于Solaris的event ports。  
主要提供了对两种类型的事件驱动：  
1、IO事件（文件事件），包括有IO的读事件和写事件。  
2、定时器事件，包括有一次性定时器和循环定时器。

基本的数据结构：@ae.h

//定义文件事件处理接口（函数指针）

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">**typedef** **void** aeFileProc(**struct** aeEventLoop \*eventLoop, **int** fd, **void** \*clientData, **int** mask);
3. //时间事件处理接口（函数指针），该函数返回定时的时长
4. **typedef** **int** aeTimeProc(**struct** aeEventLoop \*eventLoop, **long** **long** id, **void** \*clientData);
5. **typedef** **void** aeEventFinalizerProc(**struct** aeEventLoop \*eventLoop, **void** \*clientData);
7. //aeMain中使用，在调用处理事件前调用
8. **typedef** **void** aeBeforeSleepProc(**struct** aeEventLoop \*eventLoop);</span>

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">//文件事件结构体
2. **typedef** **struct** aeFileEvent {
3. //读或者写，也用于标识该事件结构体是否正在使用
4. **int** mask; /\* one of AE\_(READABLE|WRITABLE) \*/
5. //读事件的处理函数
6. aeFileProc \*rfileProc;
7. //写事件的处理函数
8. aeFileProc \*wfileProc;
9. //传递给上述两个函数的数据
10. **void** \*clientData;
11. } aeFileEvent;
13. //时间事件
14. **typedef** **struct** aeTimeEvent {
15. //时间事件标识符，用于唯一标识该时间事件，并且用于删除时间事件
16. **long** **long** id; /\* time event identifier. \*/
17. **long** when\_sec; /\* seconds \*/
18. **long** when\_ms; /\* milliseconds \*/
19. //该事件对应的处理程序
20. aeTimeProc \*timeProc;
21. //时间事件的最后一次处理程序，若已设置，则删除时间事件时会被调用
22. aeEventFinalizerProc \*finalizerProc;
23. **void** \*clientData;
24. **struct** aeTimeEvent \*next;
25. } aeTimeEvent;
27. //这里用于保存已触发的事件
28. **typedef** **struct** aeFiredEvent {
29. **int** fd;
30. **int** mask;
31. } aeFiredEvent;</span>

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">/\* State of an event based program \*/
2. **typedef** **struct** aeEventLoop {
3. //最大文件描述符的值
4. **int** maxfd;   /\* highest file descriptor currently registered \*/
5. //文件描述符的最大监听数
6. **int** setsize; /\* max number of file descriptors tracked \*/
7. //用于生成时间事件的唯一标识id
8. **long** **long** timeEventNextId;
9. //用于检测系统时间是否变更（判断标准 now<lastTime）
10. **time\_t** lastTime;     /\* Used to detect system clock skew \*/
11. //注册要使用的文件事件，这里的分离表实现为直接索引，即通过fd来访问，实现事件的分离
12. aeFileEvent \*events; /\* Registered events \*/
13. //已触发的事件
14. aeFiredEvent \*fired; /\* Fired events \*/
15. aeTimeEvent \*timeEventHead;
16. //停止标志，1表示停止
17. **int** stop;
18. //这个是处理底层特定API的数据，对于epoll来说，该结构体包含了epoll fd和epoll\_event
19. **void** \*apidata; /\* This is used for polling API specific data \*/
20. //在调用processEvent前（即如果没有事件则睡眠），调用该处理函数
21. aeBeforeSleepProc \*beforesleep;
22. } aeEventLoop;</span>

1、 aeCreateEventLoop  
  
底层epoll多路复用初始化，然后存放在aeEventLoop中 void \* 类型的apidata，隐藏了底层的实现。

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">**typedef** **struct** aeApiState {
2. **int** epfd;
3. **struct** epoll\_event \*events;
4. } aeApiState;</span>

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">//ae底层的数据创建以及初始化
2. **static** **int** aeApiCreate(aeEventLoop \*eventLoop) {
3. aeApiState \*state = zmalloc(**sizeof**(aeApiState));
5. **if** (!state) **return** -1;
6. //创建setsize个epoll\_event
7. state->events = zmalloc(**sizeof**(**struct** epoll\_event)\*eventLoop->setsize);
8. **if** (!state->events) {
9. zfree(state);
10. **return** -1;
11. }
12. state->epfd = epoll\_create(1024); /\* 1024 is just a hint for the kernel \*/
13. **if** (state->epfd == -1) {
14. zfree(state->events);
15. zfree(state);
16. **return** -1;
17. }
18. eventLoop->apidata = state;
19. **return** 0;
20. }
22. //创建事件循环，setsize为最大事件的的个数，对于epoll来说也是epoll\_event的个数
23. aeEventLoop \*aeCreateEventLoop(**int** setsize) {
24. aeEventLoop \*eventLoop;
25. **int** i;
27. //分配该结构体的内存空间
28. **if** ((eventLoop = zmalloc(**sizeof**(\*eventLoop))) == NULL) **goto** err;
29. eventLoop->events = zmalloc(**sizeof**(aeFileEvent)\*setsize);
30. eventLoop->fired = zmalloc(**sizeof**(aeFiredEvent)\*setsize);
31. **if** (eventLoop->events == NULL || eventLoop->fired == NULL) **goto** err;
33. //初始化最多setsize个事件
34. eventLoop->setsize = setsize;
35. eventLoop->lastTime = time(NULL);
36. eventLoop->timeEventHead = NULL;
37. eventLoop->timeEventNextId = 0;
38. eventLoop->stop = 0;
39. eventLoop->maxfd = -1;
40. eventLoop->beforesleep = NULL;
42. //这一步为创建底层IO处理的数据，如epoll，创建epoll\_event,和epfd
43. **if** (aeApiCreate(eventLoop) == -1) **goto** err;
44. /\* Events with mask == AE\_NONE are not set. So let's initialize the
45. \* vector with it. \*/
46. **for** (i = 0; i < setsize; i++)
47. eventLoop->events[i].mask = AE\_NONE;
48. **return** eventLoop;
50. err:
51. **if** (eventLoop) {
52. zfree(eventLoop->events);
53. zfree(eventLoop->fired);
54. zfree(eventLoop);
55. }
56. **return** NULL;
57. }</span>

这里将最大文件描述符作为参数setSize、后面创建的eventLoop->events、eventLoop->fired都是以此  
来创建、即用文件描述符作为其索引、以最大内存 sizeof(aeFiredEvent) + sizeof(aeFileEvent) 40字节  
乘以总的fd数之内存换取o(1)查找效率是值得的。

2、aeCreateFileEvent  
  
对于创建文件事件，需要传入一个该事件对应的处理程序，当事件发生时，会调用对应的回调函数。  
这里设计的aeFileEvent结构体就是将事件源（FD），事件，事件处理程序关联起来。

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">//添加监听的事件，其中如果该fd对应的事件已经存在，则为修改合并旧的事件
2. **static** **int** aeApiAddEvent(aeEventLoop \*eventLoop, **int** fd, **int** mask) {
3. aeApiState \*state = eventLoop->apidata;
4. **struct** epoll\_event ee = {0}; /\* avoid valgrind warning \*/
5. /\* If the fd was already monitored for some event, we need a MOD
6. \* operation. Otherwise we need an ADD operation. \*/
7. //判断fd是否已经添加了事件的监听
8. **int** op = eventLoop->events[fd].mask == AE\_NONE ?
9. EPOLL\_CTL\_ADD : EPOLL\_CTL\_MOD;
11. ee.events = 0;
12. mask |= eventLoop->events[fd].mask; /\* Merge old events \*/
13. **if** (mask & AE\_READABLE) ee.events |= EPOLLIN;
14. **if** (mask & AE\_WRITABLE) ee.events |= EPOLLOUT;
15. ee.data.fd = fd;
16. **if** (epoll\_ctl(state->epfd,op,fd,&ee) == -1) **return** -1;
17. **return** 0;
18. }
20. //删除指定事件的监听
21. **static** **void** aeApiDelEvent(aeEventLoop \*eventLoop, **int** fd, **int** delmask) {
22. aeApiState \*state = eventLoop->apidata;
23. **struct** epoll\_event ee = {0}; /\* avoid valgrind warning \*/
24. **int** mask = eventLoop->events[fd].mask & (~delmask);
26. ee.events = 0;
27. **if** (mask & AE\_READABLE) ee.events |= EPOLLIN;
28. **if** (mask & AE\_WRITABLE) ee.events |= EPOLLOUT;
29. ee.data.fd = fd;
30. **if** (mask != AE\_NONE) {
31. epoll\_ctl(state->epfd,EPOLL\_CTL\_MOD,fd,&ee);
32. } **else** {
33. /\* Note, Kernel < 2.6.9 requires a non null event pointer even for
34. \* EPOLL\_CTL\_DEL. \*/
35. epoll\_ctl(state->epfd,EPOLL\_CTL\_DEL,fd,&ee);
36. }
37. }
39. //创建文件事件，并将该事件注册到eventLoop中
40. **int** aeCreateFileEvent(aeEventLoop \*eventLoop, **int** fd, **int** mask,
41. aeFileProc \*proc, **void** \*clientData)
42. {
43. **if** (fd >= eventLoop->setsize) {
44. errno = ERANGE;
45. **return** AE\_ERR;
46. }
48. //直接使用fd来获取FileEvent，来后面分离事件时也采用这种方法（直接索引）
49. aeFileEvent \*fe = &eventLoop->events[fd];
51. //该该事件添加eventLoop中或者修改原来的已有的（保留旧的）
52. **if** (aeApiAddEvent(eventLoop, fd, mask) == -1)
53. **return** AE\_ERR;
54. fe->mask |= mask;
56. //将该事件的处理程序放到对应的位置
57. **if** (mask & AE\_READABLE) fe->rfileProc = proc;
58. **if** (mask & AE\_WRITABLE) fe->wfileProc = proc;
60. //设置将要传递给该事件处理程序的数据
61. fe->clientData = clientData;
62. **if** (fd > eventLoop->maxfd)
63. eventLoop->maxfd = fd;
64. **return** AE\_OK;
65. }</span>

3、aeProcessEvents  
  
这个是核心部分，通过epoll\_wait将事件分离出来，从而保存到fired中，对于语句   
aeFileEvent \*fe = &eventLoop->events[eventLoop->fired[j].fd];   
通过触发事件的 fd 在events中直接映射找到与事件关联的结构体，从而实现事件分派。  
Reactor的核心是实现了事件的分离分派。

**[cpp]** [view plain](https://blog.csdn.net/andyhuabing/article/details/52574411) [copy](https://blog.csdn.net/andyhuabing/article/details/52574411)

1. <span style="font-size:18px;">**static** **int** aeApiPoll(aeEventLoop \*eventLoop, **struct** timeval \*tvp) {
2. aeApiState \*state = eventLoop->apidata;
3. **int** retval, numevents = 0;
5. //等待事件产生
6. retval = epoll\_wait(state->epfd,state->events,eventLoop->setsize,
7. tvp ? (tvp->tv\_sec\*1000 + tvp->tv\_usec/1000) : -1);
8. **if** (retval > 0) {
9. **int** j;
11. numevents = retval;
12. **for** (j = 0; j < numevents; j++) {
13. **int** mask = 0;
14. **struct** epoll\_event \*e = state->events+j;
16. **if** (e->events & EPOLLIN) mask |= AE\_READABLE;
17. **if** (e->events & EPOLLOUT) mask |= AE\_WRITABLE;
18. **if** (e->events & EPOLLERR) mask |= AE\_WRITABLE;
19. **if** (e->events & EPOLLHUP) mask |= AE\_WRITABLE;
20. // 利用fired数组记录触发的事件
21. eventLoop->fired[j].fd = e->data.fd;
22. eventLoop->fired[j].mask = mask;
23. }
24. }
25. **return** numevents;
26. }
28. //事件处理程序
29. **int** aeProcessEvents(aeEventLoop \*eventLoop, **int** flags)
30. {
31. **int** processed = 0, numevents;
33. //若什么都没有设置，则直接返回
34. /\* Nothing to do? return ASAP \*/
35. **if** (!(flags & AE\_TIME\_EVENTS) && !(flags & AE\_FILE\_EVENTS)) **return** 0;
37. //如果有文件事件或者设置了时间事件并且没有设置DONT\_WAIT标志
38. /\* Note that we want call select() even if there are no
39. \* file events to process as long as we want to process time
40. \* events, in order to sleep until the next time event is ready
41. \* to fire. \*/
42. **if** (eventLoop->maxfd != -1 ||
43. ((flags & AE\_TIME\_EVENTS) && !(flags & AE\_DONT\_WAIT))) {
44. **int** j;
45. aeTimeEvent \*shortest = NULL;
46. **struct** timeval tv, \*tvp;
48. **if** (flags & AE\_TIME\_EVENTS && !(flags & AE\_DONT\_WAIT))
49. //查找时间最早的时间事件
50. shortest = aeSearchNearestTimer(eventLoop);
51. **if** (shortest) {
52. **long** now\_sec, now\_ms;
54. aeGetTime(&now\_sec, &now\_ms);
55. tvp = &tv;
57. /\* How many milliseconds we need to wait for the next
58. \* time event to fire? \*/
59. **long** **long** ms =
60. (shortest->when\_sec - now\_sec)\*1000 +
61. shortest->when\_ms - now\_ms;
62. // 找到最早的时间事件与当前时间差值就是epoll wait时间
63. **if** (ms > 0) {
64. tvp->tv\_sec = ms/1000;
65. tvp->tv\_usec = (ms % 1000)\*1000;
66. } **else** {
67. tvp->tv\_sec = 0;
68. tvp->tv\_usec = 0;
69. }
70. } **else** {
71. /\* If we have to check for events but need to return
72. \* ASAP because of AE\_DONT\_WAIT we need to set the timeout
73. \* to zero \*/
74. **if** (flags & AE\_DONT\_WAIT) {
75. tv.tv\_sec = tv.tv\_usec = 0;
76. tvp = &tv;
77. } **else** {
78. //如果没有时间事件则可以阻塞、如果此时加入一个Timer event，啥时候唤醒呢？！
79. /\* Otherwise we can block \*/
80. tvp = NULL; /\* wait forever \*/
81. }
82. }
84. numevents = aeApiPoll(eventLoop, tvp);
85. **for** (j = 0; j < numevents; j++) {
86. aeFileEvent \*fe = &eventLoop->events[eventLoop->fired[j].fd];
87. **int** mask = eventLoop->fired[j].mask;
88. **int** fd = eventLoop->fired[j].fd;
89. **int** rfired = 0;
91. /\* note the fe->mask & mask & ... code: maybe an already processed
92. \* event removed an element that fired and we still didn't
93. \* processed, so we check if the event is still valid. \*/
94. **if** (fe->mask & mask & AE\_READABLE) {
95. rfired = 1;
96. fe->rfileProc(eventLoop,fd,fe->clientData,mask);
97. }
98. **if** (fe->mask & mask & AE\_WRITABLE) {
99. //这里的判断是为了防止重复调用
100. **if** (!rfired || fe->wfileProc != fe->rfileProc)
101. fe->wfileProc(eventLoop,fd,fe->clientData,mask);
102. }
103. processed++;
104. }
105. }
106. /\* Check time events \*/
107. **if** (flags & AE\_TIME\_EVENTS)
108. processed += processTimeEvents(eventLoop);
110. **return** processed; /\* return the number of processed file/time events \*/
111. }</span>

**总结如下：**  
1. Reactor模式，串行处理事件  
2. 具有定时事件功能（但是不能过多，因为是使用链表实现的）、O(N)复杂度  
3. 优先处理读事件

版权声明：本文为博主原创文章，未经博主允许不得转载。 https://blog.csdn.net/andyhuabing/article/details/52574411