高性能网络编程6--reactor反应堆与定时器管理

2013年12月20日 19:37:23 阅读数：20364

反应堆开发模型被绝大多数高性能服务器所选择，上一篇所介绍的IO多路复用是它的实现基础。定时触发功能通常是服务器必备组件，反应堆模型往往还不得不将定时器的管理囊括在内。本篇将介绍反应堆模型的特点和用法。

首先我们要谈谈，网络编程界为什么需要反应堆？有了IO复用，有了epoll，我们已经可以使服务器并发几十万连接的同时，维持高TPS了，难道这还不够吗？
我的答案是，技术层面足够了，但在软件工程层面却是不够的。
程序使用IO复用的难点在哪里呢？1个请求虽然由多次IO处理完成，但相比传统的单线程完整处理请求生命期的方法，IO复用在人的大脑思维中并不自然，因为，程序员编程中，处理请求A的时候，假定A请求必须经过多个IO操作A1-An（两次IO间可能间隔很长时间），每经过一次IO操作，再调用IO复用时，IO复用的调用返回里，非常可能不再有A，而是返回了请求B。即请求A会经常被请求B打断，处理请求B时，又被C打断。这种思维下，编程容易出错。
形象的说，传统编程方法就好像是到了银行营业厅里，每个窗口前排了长队，业务员们在窗口后一个个的解决客户们的请求。一个业务员可以尽情思考着客户A依次提出的问题，例如：
“我要买2万XX理财产品。 ”
“看清楚了，5万起售。”
“等等，查下我活期余额。”
“余额5万。”
“那就买 5万吧。”
业务员开始录入信息。
” 对了，XX理财产品年利率8%？”
“是预期8%，最低无利息保本。 “
” 早不说，拜拜，我去买余额宝。 “
业务员无表情的删着已经录入的信息进行事务回滚。
” 下一个！ “

用了IO复用则是大师业务员开始挑战极限，在超大营业厅里给客户们人手一个牌子，黑压压的客户们都在大厅中，有问题时举牌申请提问，大师目光敏锐点名指定某人提问，该客户迅速得到大师的答复后，要经过一段时间思考，查查自己的银袋子，咨询下LD，才能再次进行下一个提问，直到得到完整的满意答复退出大厅。例如：大师刚指导A填写转帐单的某一项，B又来申请兑换泰铢，给了B兑换单后，C又来办理定转活，然后D与F在争抢有限的圆珠笔时出现了不和谐现象，被大师叫停业务，暂时等待。
这就是基于事件驱动的IO复用编程比起传统1线程1请求的方式来，有难度的设计点了，客户们都是上帝，既不能出错，还不能厚此薄彼。

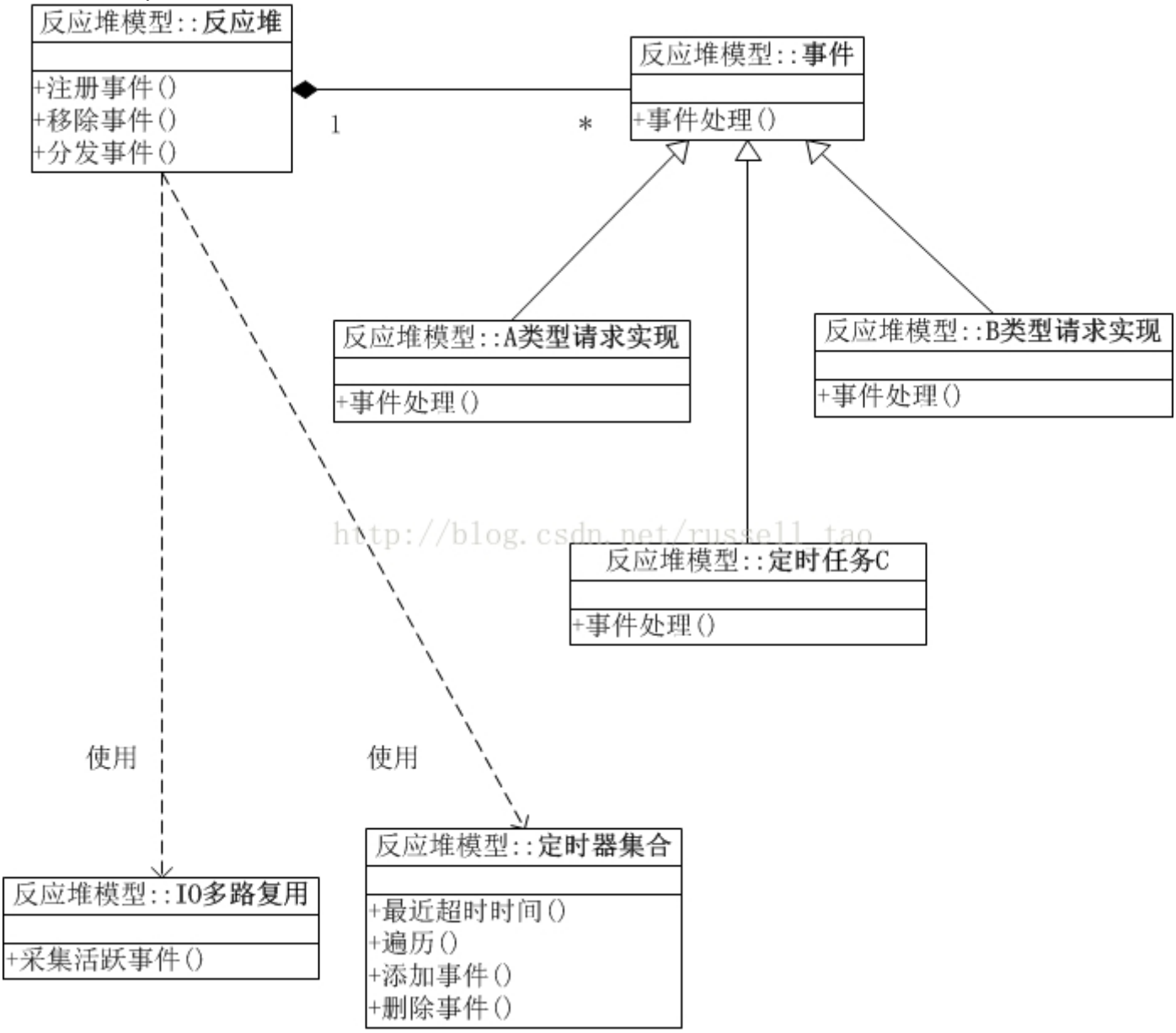
当没有反应堆时，我们可能的设计方法是这样的：大师把每个客户的提问都记录下来，当客户A提问时，首先查阅A之前问过什么做过什么，这叫联系上下文，然后再根据上下文和当前提问查阅有关的银行规章制度，有针对性的回答A，并把回答也记录下来。当圆满回答了A的所有问题后，删除A的所有记录。

回到码农生涯，即，某一瞬间，服务器共有10万个并发连接，此时，一次IO复用接口的调用返回了100个活跃的连接等待处理。先根据这100个连接找出其对应的对象，这并不难，epoll的返回连接数据结构里就有这样的指针可以用。接着，循环的处理每一个连接，找出这个对象此刻的上下文状态，再使用read、write这样的网络IO获取此次的操作内容，结合上下文状态查询此时应当选择哪个业务方法处理，调用相应方法完成操作后，若请求结束，则删除对象及其上下文。

这样，我们就陷入了面向过程编程方法之中了，在面向应用、快速响应为王的移动互联网时代，这样做早晚得把自己玩死。我们的主程序需要关注各种不同类型的请求，在不同状态下，对于不同的请求命令选择不同的业务处理方法。这会导致随着请求类型的增加，请求状态的增加，请求命令的增加，主程序复杂度快速膨胀，导致维护越来越困难，苦逼的程序员再也不敢轻易接新需求、重构。

反应堆是解决上述软件工程问题的一种途径，它也许并不优雅，开发效率上也不是最高的，但其执行效率与面向过程的使用IO复用却几乎是等价的，所以，无论是nginx、memcached、redis等等这些高性能组件的代名词，都义无反顾的一头扎进了反应堆的怀抱中。

反应堆模式可以在软件工程层面，将事件驱动框架分离出具体业务，将不同类型请求之间用OO的思想分离。通常，反应堆不仅使用IO复用处理网络事件驱动，还会实现定时器来处理时间事件的驱动（请求的超时处理或者定时任务的处理），就像下面的示意图：



这幅图有5点意思：

（1）处理应用时基于OO思想，不同的类型的请求处理间是分离的。例如，A类型请求是用户注册请求，B类型请求是查询用户头像，那么当我们把用户头像新增多种分辨率图片时，更改B类型请求的代码处理逻辑时，完全不涉及A类型请求代码的修改。

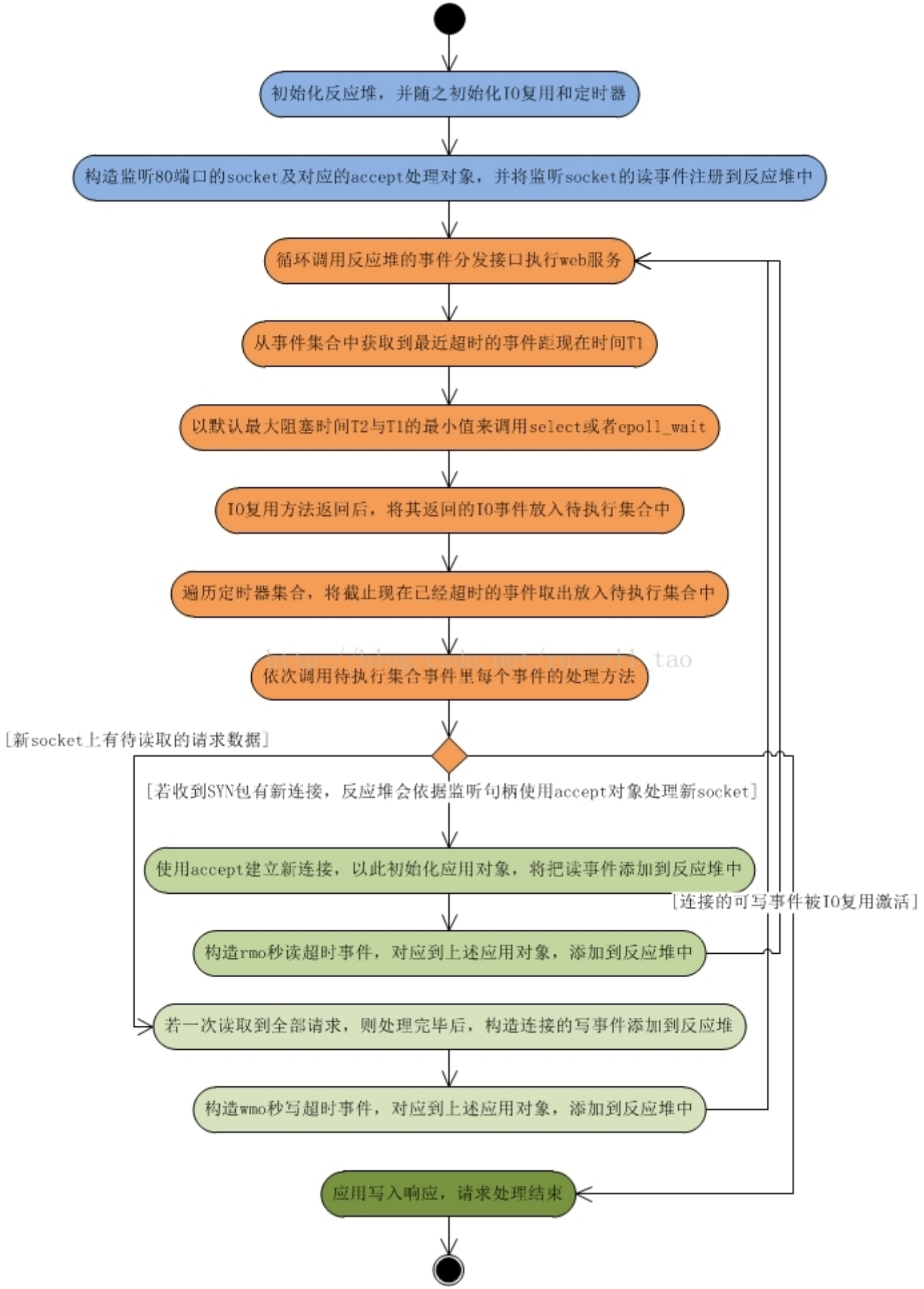
（2）应用处理请求的逻辑，与事件分发框架完全分离。什么意思呢？即写应用处理时，不用去管何时调用IO复用，不用去管什么调用epoll_wait，去处理它返回的多个socket连接。应用代码中，只关心如何读取、发送socket上的数据，如何处理业务逻辑。事件分发框架有一个抽象的事件接口，所有的应用必须实现抽象的事件接口，通过这种抽象才把应用与框架进行分离。

（3）反应堆上提供注册、移除事件方法，供应用代码使用，而分发事件方法，通常是循环的调用而已，是否提供给应用代码调用，还是由框架简单粗暴的直接循环使用，这是框架的自由。

（4）IO多路复用也是一个抽象，它可以是具体的select，也可以是epoll，它们只必须提供采集到某一瞬间所有待监控连接中活跃的连接。

（5）定时器也是由反应堆对象使用，它必须至少提供4个方法，包括添加、删除定时器事件，这该由应用代码调用。最近超时时间是需要的，这会被反应堆对象使用，用于确认select或者epoll_wait执行时的阻塞超时时间，防止IO的等待影响了定时事件的处理。遍历也是由反应堆框架使用，用于处理定时事件。

下面用极简流程来形象说明下反应堆是如何处理一个请求的，下图中桔色部分皆为反应堆的分发事件流程：



可以看到，分发IO、定时器事件都由反应堆框架来完成，应用代码只会关注于如何处理可读、可写事件。
当然，上图是极度简化的流程，实际上要处理的异常情况都没有列入。

这里可以看到，为什么定时器集合需要提供最近超时事件距离现在的时间？因为，调用epoll_wait或者select时，并不能够始终传入-1作为timeout参数。因为，我们的服务器主营业务往往是网络请求处理，如果网络请求很少时，那么CPU的所有时间都会被频繁却又不必要的epoll_wait调用所占用。在服务器闲时使进程的CPU利用率降低是很有意义的，它可以使服务器上其他进程得到更多的执行机会，也可以延长服务器的寿命，还可以省电。这样，就需要传入准确的timeout最大阻塞时间给epoll_wait了。

什么样的timeout时间才是准确的呢？这等价于，我们需要准确的分析，什么样的时段进程可以真正休息，进入sleep状态？
一个没有意义的答案是：不需要进程执行任务的时间段内是可以休息的。
这就要求我们仔细想想，进程做了哪几类任务，例如：
1、所有网络包的处理，例如TCP连接的建立、读写、关闭，基本上所有的正常请求都由网络包来驱动的。对这类任务而言，没有新的网络分组到达本机时，就是可以使进程休息的时段。
2、定时器的管理，它与网络、IO复用无关，虽然它们在业务上可能有相关性。定时器里的事件需要及时的触发执行，不能因为其他原因，例如阻塞在epoll_wait上时耽误了定时事件的处理。当一段时间内，可以预判没有定时事件达到触发条件时（这也是提供接口查询最近一个定时事件距当下的时间的意义所在），对定时任务的管理而言，进程就可以休息了。
3、其他类型的任务，例如磁盘IO执行完成，或者收到其他进程的signal信号，等等，这些任务明显不需要执行的时间段内，进程可以休息。

于是，使用反应堆模型的进程代码中，通常除了epoll_wait这样的IO复用外，其他调用都会基于无阻塞的方式使用。所以，epoll_wait的timeout超时时间，就是除网络外，其他任务所能允许的进程睡眠时间。而只考虑常见的定时器任务时，就像上图中那样，只需要定时器集合能够提供最近超时事件到现在的时间即可。

从这里也可以推导出，定时器集合通常会采用有序容器这样的数据结构，好处是：
1、容易取到最近超时事件的时间。
2、可以从最近超时事件开始，向后依次遍历已经超时的事件，直到第一个没有超时的事件为止即可停止遍历，不用全部遍历到。

因此，粗暴的采用无序的数据结构，例如普通的链表，通常是不足取的。但事无绝对，redis就是用了个毫无顺序的链表，原因何在？因为redis的客户端连接没有超时概念，所以对于并发的成千上万个连上，都不会因为超时被断开。redis的定时器唯一的用途在于定时的将内存数据刷到磁盘上，这样的定时事件通常只有个位数，其性能无关紧要。

如果定时事件非常多，综合插入、遍历、删除的使用频率，使用树的机会最多，例如小根堆（libevent）、二叉平衡树（nginx红黑树）。当然，场景特殊时，尽可以用有序数组、跳跃表等等实现。

综上所述，反应堆模型开发效率上比起直接使用IO复用要高，它通常是单线程的，设计目标是希望单线程使用一颗CPU的全部资源，但也有附带优点，即每个事件处理中很多时候可以不考虑共享资源的互斥访问。可是缺点也是明显的，现在的硬件发展，已经不再遵循摩尔定律，CPU的频率受制于材料的限制不再有大的提升，而改为是从核数的增加上提升能力，当程序需要使用多核资源时，反应堆模型就会悲剧，为何呢？
如果程序业务很简单，例如只是简单的访问一些提供了并发访问的服务，就可以直接开启多个反应堆，每个反应堆对应一颗CPU核心，这些反应堆上跑的请求互不相关，这是完全可以利用多核的。例如Nginx这样的http静态服务器。
如果程序比较复杂，例如一块内存数据的处理希望由多核共同完成，这样反应堆模型就很难做到了，需要昂贵的代价，引入许多复杂的机制。所以，大家就可以理解像redis、nodejs这样的服务，为什么只能是单线程，为什么memcached简单些的服务确可以是多线程