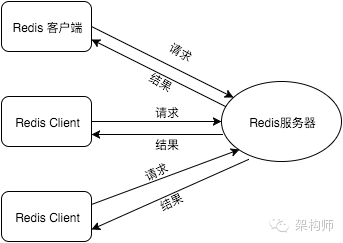
**目录**

* 从Redis的工作模式谈起
* Reactor模式
  + C10K问题
  + I/O多路复用技术
  + Reactor的定义
  + Java中的NIO与Netty
* Redis与Reactor
* 总结
* 参考资料

## ****从Redis的工作模式谈起****

我们在使用Redis的时候，通常是多个客户端连接Redis服务器，然后各自发送命令请求(例如GetSet)到Redis服务器，最后Redis处理这些请求返回结果



那Redis服务端是使用单进程还是多进程，单线程还是多线程来处理客户端请求的呢？

**答案是单进程单线程**

当然，Redis除了处理客户端的命令请求还有诸如RDB持久化AOF重写这样的事情要做，而在做这些事情的时候，Redis会fork**(分叉出)**子进程去完成但对于accept客户端连接处理客户端请求返回命令结果等等这些，Redis是使用主进程及主线程来完成的。我们可能会惊讶Redis在使用单进程及单线程来处理请求为什么会如此高效？在回答这个问题之前，

我们先来讨论一个**I/O多路复用的模式--Reactor**

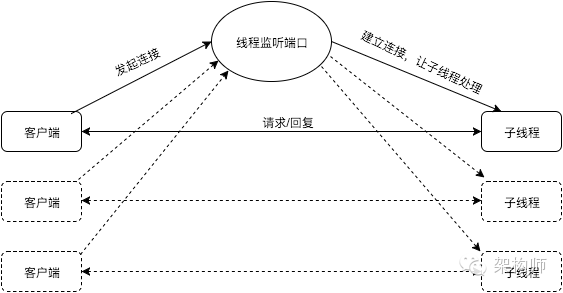
## ****Reactor模式****

### ****C10K问题****

考虑这样一个问题：有10000个客户端需要连上一个服务器并保持TCP连接，客户端会不定时的发送请求给服务器，服务器收到请求后需及时处理并返回结果我们应该怎么解决?

方案一：我们使用一个线程来监听，当一个新的客户端发起连接时，建立连接并new一个线程来处理这个新连接

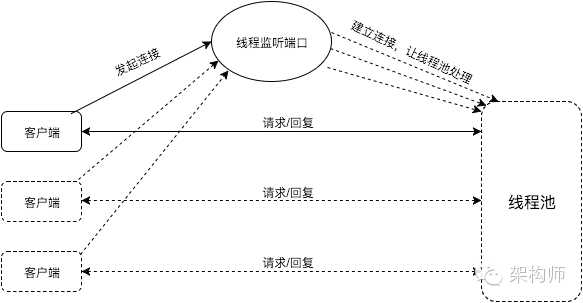
缺点：当客户端数量很多时，服务端线程数过多，即便不压垮服务器，由于CPU有限其性能也极其不理想因此此方案不可用



方案二：**我们使用一个线程监听，当一个新的客户端发起连接时，建立连接并使用线程池处理该连接**

优点：客户端连接数量不会压垮服务端

缺点：服务端处理能力受限于线程池的线程数，而且如果客户端连接中大部分处于空闲状态的话服务端的线程资源被浪费



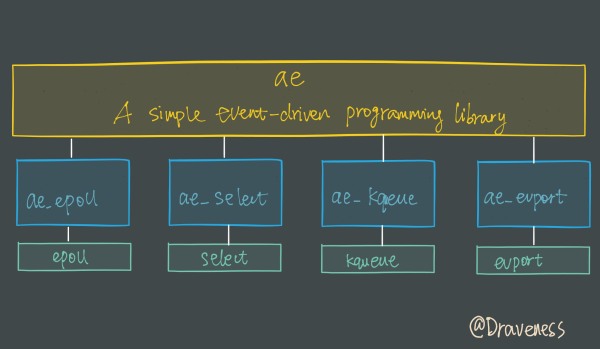
因此，一个线程仅仅处理一个客户端连接无论如何都是不可接受的，那能不能一个线程处理多个连接呢？该线程轮询每个连接，如果某个连接有请求则处理请求，没有请求则处理下一个连接，这样可以实现吗？

答案是肯定的，而且不必轮询,**我们可以通过I/O多路复用技术来解决这个问题**

### ****I/O多路复用技术（三种里最佳）****

现代的UNIX操作系统提供了**select/poll/kqueue/epoll这样的系统调用**，这些系统调用的功能是：你告知我一批套接字（socket），**当这些套接字的可读或可写事件发生时，我通知你这些事件信息。（IO中讲到的，里面的事件分离者。在我的理解有点像中介的味道，在socket和事件处理者中充当传话的角色）、**

**I/O 多路复用模块（整个 I/O 多路复用模块在事件循环看来就是一个输入事件、输出 aeFiredEvent 数组的一个黑箱**）



I/O 多路复用模块封装了底层的 **select、epoll、avport 以及 kqueue** 这些 I/O 多路复用函数，为上层提供了相同的接口。aeCreateFileEvent() 会根据文件描述符的数值大小在事件循环结构体的 I/O 事件表中取一个数据空间，利用系统提供的 I/O 多路复用技术监听感兴趣的 I/O 事件，并设置回调函数。

当如下**任一**情况发生时，会产生套接字的**可读**事件：

* 该套接字的接收缓冲区中的数据字节数大于等于套接字接收缓冲区低水位标记的大小；
* 该套接字的读半部关闭（也就是收到了FIN），对这样的套接字的读操作将返回0（也就是返回EOF）；
* 该套接字是一个监听套接字且已完成的连接数不为0；
* 该套接字有错误待处理，对这样的套接字的读操作将返回-1

当如下**任一**情况发生时，会产生套接字的**可写**事件：

* 该套接字的发送缓冲区中的可用空间字节数大于等于套接字发送缓冲区低水位标记的大小；
* 该套接字的写半部关闭，继续写会产生SIGPIPE信号；
* 非阻塞模式下，connect返回之后，该套接字连接成功或失败；
* 该套接字有错误待处理，对这样的套接字的写操作将返回-1

**事件触发**

这里以 select 版本的 redis api 实现作为讲解，aeApiPoll() 调用了 select() 进入了监听轮询。aeApiPoll() 的 tvp 参数是最小等待时间，它会被预先计算出来，它主要完成：

1. 拷贝读写的 fdset。select() 的调用会破坏传入的 fdset，实际上有两份 fdset，一份作为备份，另一份用作调用。每次调用 select() 之前都从备份中直接拷贝一份；
2. 调用 select()；
3. 被唤醒后，检查 fdset 中的每一个文件描述符，并**将可读或者可写的描述符记录到触发表当中。**

接下来的操作便是**执行相应的回调函数**，代码在上一段中已经贴出：先处理 I/O 事件，再处理定时事件。

**此外，在UNIX系统上，一切皆文件套接字也不例外，每一个套接字都有对应的fd（即文件描述符）**

**我们简单看看这几个系统调用的原型(每一个套接字都有对应的fd（即文件描述符）)**

select(int nfds, fd\_set \*r, fd\_set \*w, fd\_set \*e, struct timeval \*timeout)

对于select()，我们需要传3个集合，r（读），w（写）和e其中，r表示我们对哪些fd的**可读**事件感兴趣，w表示我们对哪些fd的**可写**事件感兴趣每个集合其实是一个bitmap，通过0/1表示我们感兴趣的fd例如，

**如：我们对于fd为6的可读事件感兴趣，那么r集合的第6个bit需要被设置为1这个系统调用会阻塞，直到我们感兴趣的事件（至少一个）发生调用返回时**，内核同样使用这3个集合来存放**fd实际发生的事件信息**也就是说，**调用前这3个集合表示我们感兴趣的事件，调用后这3个集合表示实际发生的事件**

select为最早期的UNIX系统调用，它存在4个问题：

1）**这3个bitmap有大小限制（FD\_SETSIZE，通常为1024）；**

**2）由于这3个集合在返回时会被内核修改，因此我们每次调用时都需要重新设置；**

**3）我们在调用完成后需要扫描这3个集合才能知道哪些fd的读/写事件发生了，一般情况下全量集合比较大而实际发生读/写事件的fd比较少，效率比较低下；4）（将用户传入的数组拷贝到内核空间）内核在每次调用时都需要扫描这3个fd集合(套接字集合)，然后查看哪些fd的事件实际发生，并将它传到设备等待队列。在读/写比较稀疏的情况下同样存在效率问题**

由于存在这些问题，于是人们对select进行了改进，从而有了poll

poll(struct **pollfd** \*fds, int nfds, int timeout)

struct pollfd {

int fd;

short events;

short revents;

}

poll调用需要传递的是一个pollfd结构的数组，**调用返回时结果信息也存放在这个数组里面 pollfd的结构中。**

**poll传递的不是固定大小的bitmap**，因此：

select的问题1解决了；poll将感兴趣事件和实际发生事件分开了，**poll传递的不是固定大小的bitmap。**

select的问题2也解决了：直接传的**pollfd的结构。**

但问题3和问题4仍然没有解决

select问题3比较容易解决，只要系统调用返回的是**实际发生相应事件的fd集合，我们便不需要扫描全量的fd集合**

对于select的问题4，我们为什么需要每次调用都传递全量的fd呢？

内核可不可以在第一次调用的时候记录这些fd，然后我们在以后的调用中不需要再传这些fd呢？

问题的关键在于**无状态对于每一次系统调用，内核不会记录下任何信息，所以每次调用都需要重复传递相同信息**

上帝说要有状态，所以我们有了epoll和kqueue

int epoll\_create(int size);

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

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

**epoll\_create的作用是创建一个context，这个context相当于状态保存者的概念**

**epoll\_ctl的作用是，当你对一个新的fd的读/写事件感兴趣时，通过该调用将fd与相应的感兴趣事件更新到context中**

**epoll\_wait的作用是，等待context中fd的事件发生**

就是这么简单

epoll是Linux中的实现，kqueue则是在FreeBSD的实现

int kqueue(void);

int kevent(int kq, const struct kevent \*changelist, int nchanges, struct kevent \*eventlist, int nevents, const struct timespec \*timeout);

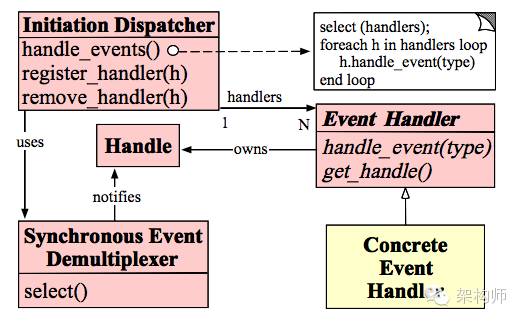
与epoll相同的是，kqueue创建一个context；与epoll不同的是，**kqueue用kevent代替了epoll\_ctl和epoll\_wait**

epoll和kqueue解决了select存在的问题通过它们，

**我们可以高效的通过系统调用来获取多个套接字的读/写事件，从而解决一个线程处理多个连接的问题**

### ****Reactor的定义****

**通过select/poll/epoll/kqueue这些I/O多路复用函数库，我们解决了一个线程处理多个连接的问题**，但整个Reactor模式的完整框架是怎样的呢？参考这篇paper，我们可以对Reactor模式有个完整的描述



**Handles** ：表示**操作系统管理的资源，我们可以理解为fd**

**Synchronous Event Demultiplexer** ：**同步事件分离器，阻塞等待Handles中的事件发生**

**Initiation Dispatcher** ：初始分派器，作用为添加Event handler（事件处理器）删除Event handler以及分派事件给Event handler也就是说，Synchronous Event Demultiplexer负责等待新事件发生，事件发生时通知Initiation Dispatcher，然后Initiation Dispatcher调用event handler处理事件

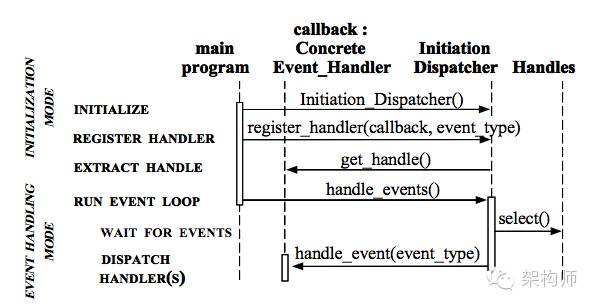
**Event Handler** ：事件处理器的接口

**Concrete Event Handler** ：**事件处理器的实际实现，而且绑定了一个Handle**因为在实际情况中，我们往往不止一种事件处理器，因此这里将事件处理器接口和实现分开，与C++Java这些高级语言中的多态类似

以上各子模块间协作的步骤描述如下：（其实就是项目中所做的基于redis 的异步框架差不多）

1. **我们注册Concrete Event Handler到Initiation Dispatcher中**
2. Initiation Dispatcher**调用每个Event Handler**的get\_handle接口获取其**绑定的Handle**
3. Initiation Dispatcher调用**handle\_events开始事件处理循环在这里**，Initiation Dispatche**r会将步骤2获取的所有Handle都收集起来**，使用Synchronous Event Demultiplexer来**等待这些Handle的事件发生**
4. 当某个（或某几个）Handle的事件发生时，**Synchronous Event Demultiplexer通知Initiation Dispatcher**
5. Initiation Dispatcher根据**发生事件的Handle找出所对应的Handler**
6. Initiation Dispatcher调用**Handler的handle\_event方法处理事件**

时序图如下：



通过以上的叙述，我们清楚了Reactor的大概框架以及涉及到的底层I/O多路复用技术

### ****Java中的NIO与Netty****

谈到Reactor模式，在这里奉上Java大神Doug Lea的Scalable IO in Java，里面提到了Java网络编程中的经典模式NIO**（非堵塞）**以及Reactor，并且有相关代码帮助理解，看完后获益良多

另外，Java的NIO是比较底层的，我们实际在网络编程中还需要自己处理很多问题（譬如socket的读半包），稍不注意就会掉进坑里幸好，我们有了Netty这么一个网络处理框架，免去了很多麻烦

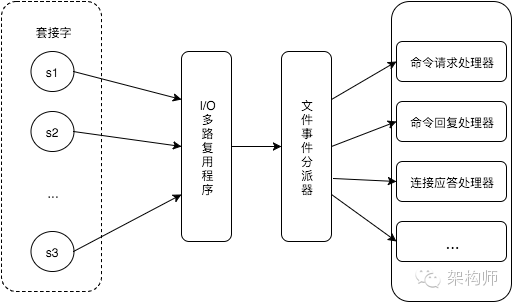
## ****Redis与Reactor****

在上面的讨论中，我们了解了Reactor模式，那么**Redis中又是怎么使用Reactor**模式的呢？

首先，Redis服务器中有两类事件，**文件事件和时间事件**

* 文件事件（file event）：Redis客户端通过socket与Redis服务器连接，而文件事件就是服务器对套接字操作的抽象例如，**客户端发了一个GET命令请求，对于Redis服务器来说就是一个文件事件**
* 时间事件（time event）：**服务器定时或周期性执行的事件例如，定期执行RDB持久化**

在这里我们主要关注Redis处理**文件事件**的模型参考Redis的设计与实现，Redis的文件事件处理模型是这样的：



在这个模型中，Redis服务器用**主线程执行I/O多路复用程序和文件事件分派器以及事件处理器而且，尽管多个文件事件可能会并发出现，Redis服务器是顺序处理各个文件事件的。**

**其实流程如下：**

**1.其中I/O多路复用程序就是多个API在监听轮询，如果没有事件发生就会进入睡眠状态，其实就是 I/O 多路复用 select() epoll() 等的调用，有事件发生会被唤醒，检查 fdset 中的每一个文件描述符，并将可读或者可写的描述符记录到触发表传到分派器。**

**2.处理已触发的 I/O 事件和定时事件。如下：**

1. **我们注册Concrete Event Handler到Initiation Dispatcher中**
2. Initiation Dispatcher**调用每个Event Handler**的get\_handle接口获取其**绑定的Handle**
3. Initiation Dispatcher调用**handle\_events开始事件处理循环在这里**，Initiation Dispatche**r会将步骤2获取的所有Handle都收集起来**，使用Synchronous Event Demultiplexer来**等待这些Handle的事件发生**
4. 当某个（或某几个）Handle的事件发生时，**Synchronous Event Demultiplexer通知Initiation Dispatcher**
5. Initiation Dispatcher根据**发生事件的Handle找出所对应的Handler**
6. Initiation Dispatcher调用**Handler的handle\_event方法处理事件，并返回结果给客户端**

Redis服务器主线程的执行流程在Redis.c的main函数中体现，而关于处理文件事件的主要的有这几行：

int main(int argc, char \*\*argv) {

...

initServer();

...

aeMain();

...

aeDeleteEventLoop(server.el);

return 0;

}

在initServer()中，建立各个事件处理器；在aeMain()中，**执行事件处理循环；**在aeDeleteEventLoop(server.el)中关闭停止事件处理循环；最后退出

## **总结**

多路 I/O 复用模型是利用select、poll、epoll可以同时监察多个流的 I/O 事件的能力，在空闲的时候，会把当前线程阻塞掉，当有一个或多个流有I/O事件时，就从阻塞态中唤醒，于是程序就会轮询一遍所有的流（epoll是只轮询那些真正发出了事件的流），并且只依次顺序的处理就绪的流，这种做法就避免了大量的无用操作。**这里“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗），且Redis在内存中操作数据的速度非常快（内存内的操作不会成为这里的性能瓶颈），主要以上两点造就了Redis具有很高的吞吐量。**

在这篇文章中，我们从Redis的工作模型开始，讨论了C10K问题、I/O多路复用技术、Java的NIO，最后回归到Redis的Reactor模式中。如有纰漏，恳请大家指出，我会一一加以勘正。谢谢！