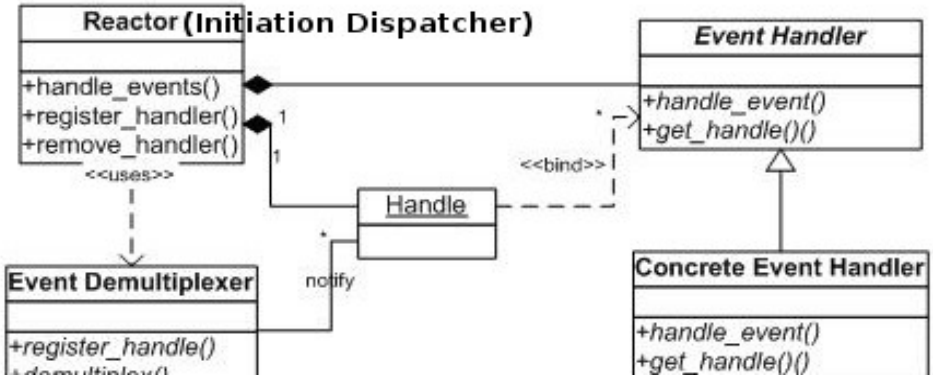
Reactor 模式及在 DSS 中的体现

Reactor 模式是处理并发 I/O 比较常见的一种模式，用于**同步 I/O**，中心思想是将所有要处理的 I/O 事件注册到一个中心 I/O 多路复用器上，同时主线程阻塞在多路复用器上；一旦有 I/O 事件到来或是准备就绪(区别在于多路复用器是边沿触发还是水平触发)，多路复用器返回并将相应 I/O 事件分发到对应的处理器中。

Reactor 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，如果相应的事件发生，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”。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。

Reactor 模式与 Observer 模式在某些方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。

模式框架



1） Handle

Handle 代表操作系统管理的资源，包括：网络链接，打开的文件，计时器，同步对象等等。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

事件分离器，由操作系统提供，在 linux 上一般是 select, poll, epoll 等系统调用，在一个 Handle 集合上等待事件的发生。接受 client 连接，建立对应 client 的事件处理器（Event Handler），并向事件分发器（Reactor）注册此事件处理器（Handler）。

3） Reactor(Initiation Dispatcher)

提供接口：注册，删除和派发 Event Handler。Event Demultiplexer 等待事件的发生，当检测到新的事件，就把事件交给 Initiation Dispatcher，它去回调 Event Handler。

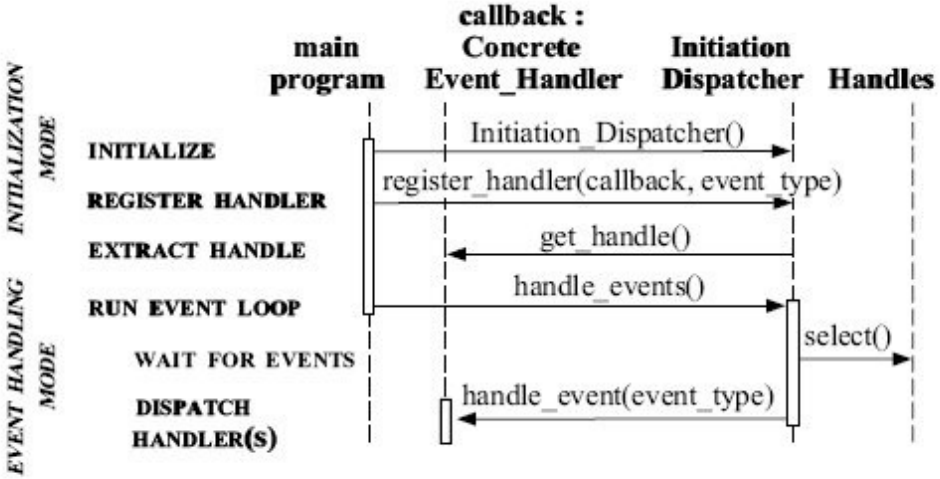
4） Event Handler

事件处理器，负责处理特定事件的处理函数。一般在基本的 Handler 基础上还会有更进一步的层次划分，用来抽象诸如 decode，process 和 encoder 这些过程。比如对 Web Server 而言，decode 通常是 HTTP 请求的解析，process 的过程会进一步涉及到 Listner 和 Servlet 的调用。为了简化设计，**Event Handler 通常被设计成状态机**，按 GoF 的 state pattern 来实现。

5） Concrete Event Handler

继承上面的类，实现钩子方法。应用把 Concrete Event Handler 注册到 Reactor，等待被处理的事件。当事件发生，这些方法被回调。

事件处理流程



模式模型

应用场景举例

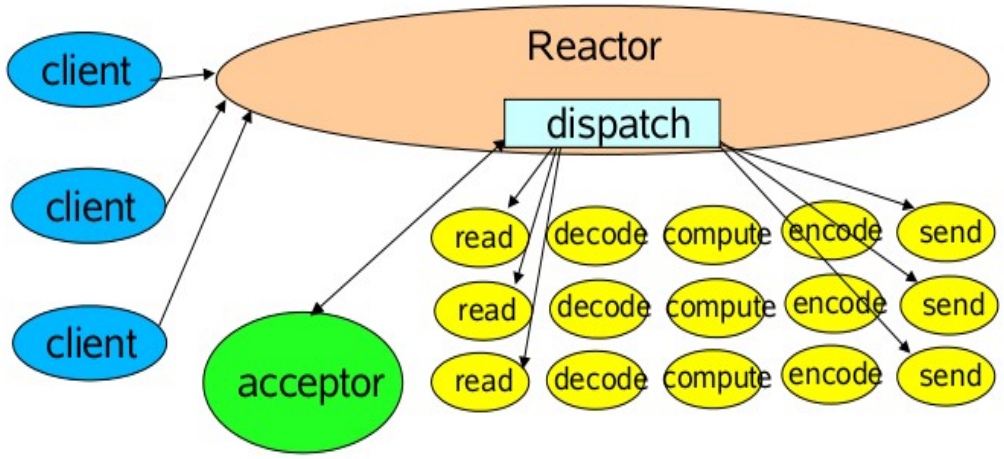
场景：长途客车在路途上，有人上车有人下车，但是乘客总是希望能够在客车上得到休息。

传统做法：每隔一段时间（或每一个站），司机或售票员对每一个乘客询问是否下车。

Reactor 做法：汽车是乘客访问的主体（Reactor），乘客上车后，到售票员（acceptor）处登记，之后乘客便可以休息睡觉去了，当到达乘客所要到达的目的地后，售票员将其唤醒即可。

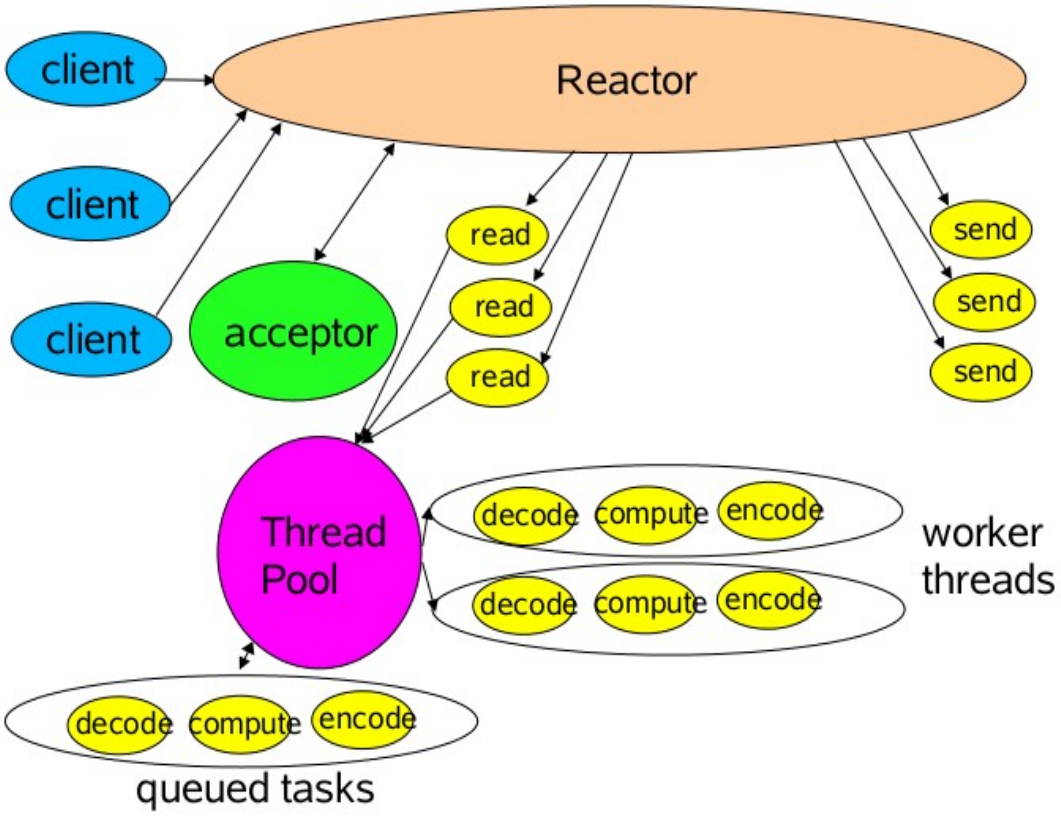
1） 单线程模型

这是最简单的单 Reactor 单线程模型。Reactor 线程是个多面手，负责多路分离套接字，Accept 新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过这种单线程模型不能充分利用多核资源，所以实际使用的不多。



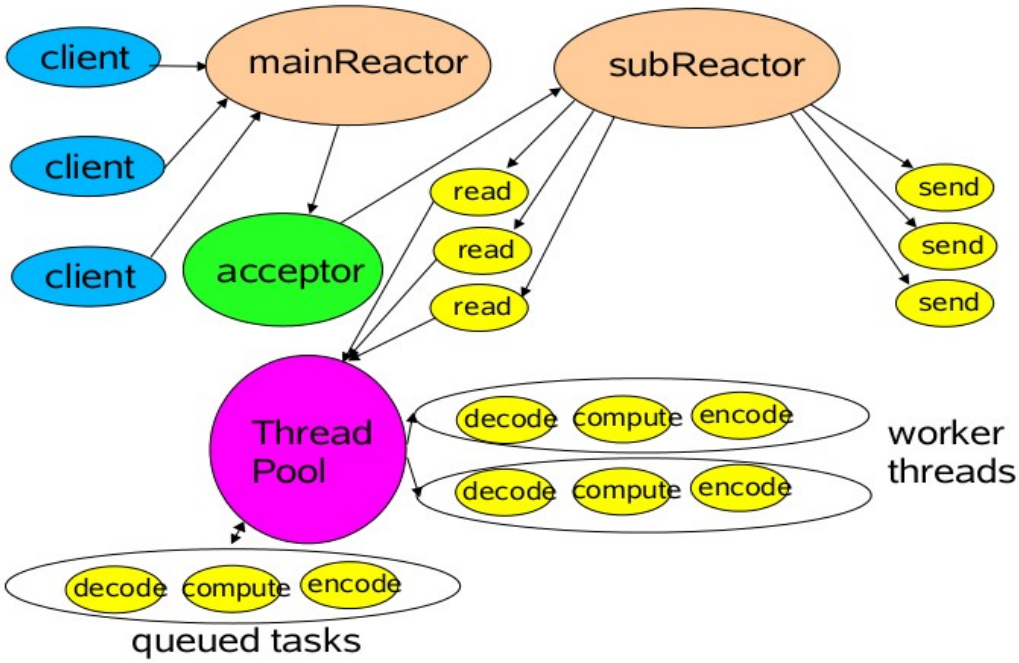
2）多线程模型（单 Reactor）

相比上一种模型，该模型在事件处理器（Handler）链部分采用了多线程（线程池），也是后端程序常用的模型。



3）多线程模型（多 Reactor）

这个模型比起第二种模型，它是将 Reactor 分成两部分，mainReactor 负责监听并 accept 新连接，然后将建立的 socket 通过多路复用器（Acceptor）分派给 subReactor。subReactor 负责多路分离已连接的 socket，读写网络数据；业务处理功能，其交给 worker 线程池完成。通常，subReactor 个数上可与 CPU 个数等同。



优缺点

优点

- 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；
- 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；
- 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；
- 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；

缺点

- 应用受限制：Reactor 模式只能应用在支持 Handle 的操作系统上。虽然可以使用多线程模拟 Reactor，但因为同步控制和上下文切换的要求，这种实现效率低，与 Reactor 模式出发点相违背。
- 非抢占模式：在单线程的实现这种情况下，事件的处理必须不能使用阻塞的 I/O，因此，如果存在长期操作，比如传输大量的数据。使用主动对象，效率可能更好，主动对象可以并发的处理这些任务。
- 难以调试：使用 Reactor 模式的应用程序可能会难以调试，因为程序运行的控制流会在框架和应用相关的处理器之间跳转，不了解框架的应用程序开发人员难一跟着调试。

相关库

ACE

ACE 是一个大型的中间件产品，代码 20 万行左右，过于宏大，一堆的设计模式，架构了一层又一层，使用的时候，要根据情况，看从那一层来进行使用。支持跨平台。

设计模式 ： ACE 主要应用了 Reactor,Proactor 等；

层次架构 ： ACE 底层是 C 风格的 OS 适配层，上一层基于 C++的 wrap 类，再上一层是一些框架 (Accpetor,Connector,Reactor,Proactor 等)，最上一层是框架上服务；

可移植性 ： ACE 支持多种平台，可移植性不存在问题，据说 socket 编程在 linux 下有不少 bugs；

事件分派处理 ： ACE 主要是注册 handler 类，当事件分派时，调用其 handler 的虚挂勾函数。实现 ACE_Handler/ACE_Svc_Handler/ACE_Event_handler 等类的虚函数；

涉及范围 ： ACE 包含了日志，IPC,线程池，共享内存，配置服务，递归锁，定时器等；

线程调度 ： ACE 的 Reactor 是单线程调度，Proactor 支持多线程调度；

发布方式 ： ACE 是开源免费的，不依赖于第三方库，一般应用使用它时，以动态链接的方式发布动态库；

开发难度 ： 基于 ACE 开发应用，对程序员要求比较高，要用好它，必须非常了解其框架。在其框架下开发，往往 new 出一个对象，不知在什么地方释放好。

Libevent

libevent 是一个 C 语言写的网络库，官方主要支持的是类 linux 操作系统，最新的版本添加了对 windows 的 IOCP 的支持。在跨平台方面主要通过 select 模型来进行支持。

设计模式 ： libevent 为 Reactor 模式；

层次架构 ： livevent 在不同的操作系统下，做了多路复用模型的抽象，可以选择使用不同的模型，通过事件函数提供服务；

可移植性 ： libevent 主要支持 linux 平台，freebsd 平台，其他平台下通过 select 模型进行支持，效率不是太高；

事件分派处理 ： libevent 基于注册的事件回调函数来实现事件分发；

涉及范围 ： libevent 只提供了简单的网络 API 的封装，线程池，内存池，递归锁等均需要自己实现；

线程调度 ： libevent 的线程调度需要自己来注册不同的事件句柄；

发布方式 ： libevent 为开源免费的，一般编译为静态库进行使用；

开发难度 ： 基于 libevent 开发应用，相对容易，具体可以参考 memcached 这个开源的应用，里面使用了 libevent 这个库。

Libev

与 libevent 一样，libev 系统也是基于事件循环的系统，它在 poll()、select() 等机制的本机实现的基础上提供基于事件的循环。libev 实现的开销更低，能够实现更好的基准测试结果。

Reactor 举例

基于 libevent 的 http 服务器 demo

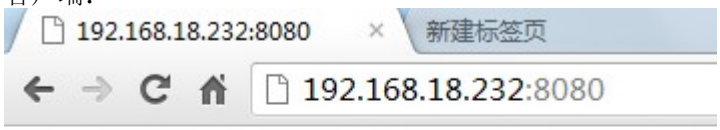
源码：

```
#include <stdio.h>
#include <event2/event.h>
#include <event2/http.h>
#include <event2/buffer.h>

struct _options
{
   int port;
   char *address;
   int verbose;
} options;

void root_handler(struct evhttp_request *req, void *arg)
{
   struct evbuffer *buf = evbuffer_new();
   if(!buf)
   {
      puts("failed to create response buffer");
      return;
   }
   evbuffer_add_printf(buf, "Hello: %s\r\n",
evhttp_request_get_uri(req));
      evhttp_send_reply(req, HTTP_OK, "OK", buf);
}

void generic_handler(struct evhttp_request *req, void
*arg)
{
      struct evbuffer *buf = evbuffer_new();
      if(!buf)
   {
            puts("failed to create response buffer");
            return;
   }
      evbuffer_add_printf(buf, "Requested: %s\n",
evhttp_request_get_uri(req));
      printf("Requested:
%s\n",evhttp_request_get_uri(req));
```

```
    evhttp_send_reply(req, HTTP_OK, "OK", buf);
}

int main(int argc, wchar_t* argv[])
{
   struct evhttp *httpd;
   struct event_base *libbase;

   options.port = 8080;
   options.address = "0.0.0.0";
   options.verbose = 0;

   libbase = event_base_new ();
   httpd = evhttp_new (libbase);
   evhttp_bind_socket (httpd, options.address,
options.port);
   evhttp_set_cb(httpd, "/", root_handler, NULL);
   evhttp_set_gencb (httpd, generic_handler, NULL);
   event_base_dispatch (libbase);

   return 0;
}
```

运行效果

客户端：



服务端：

```
[root@host232 samba]# ./http
Requested: /favicon.ico
```

Reactor 模式在 DSS 中的体现

Darwin 流媒体服务器是由父进程及其 fork 出来的子进程构成的，子进程就是核心服务器。父进程的职责就是等待子进程退出。如果子进程出错退出，则父进程就会 fork 一个新的子进程，从而保证视频服务器继续提供服务。核心服务器的作用是充当 VOD(视频点播)客户端与服务器模块之间的接口，VOD 客户端采用 RTP 和 RTSP 协议向服务器发送请求并接收响应，服务器模块负责处理 VOD 客户端的请求并向 VOD 客户端发送数据包。

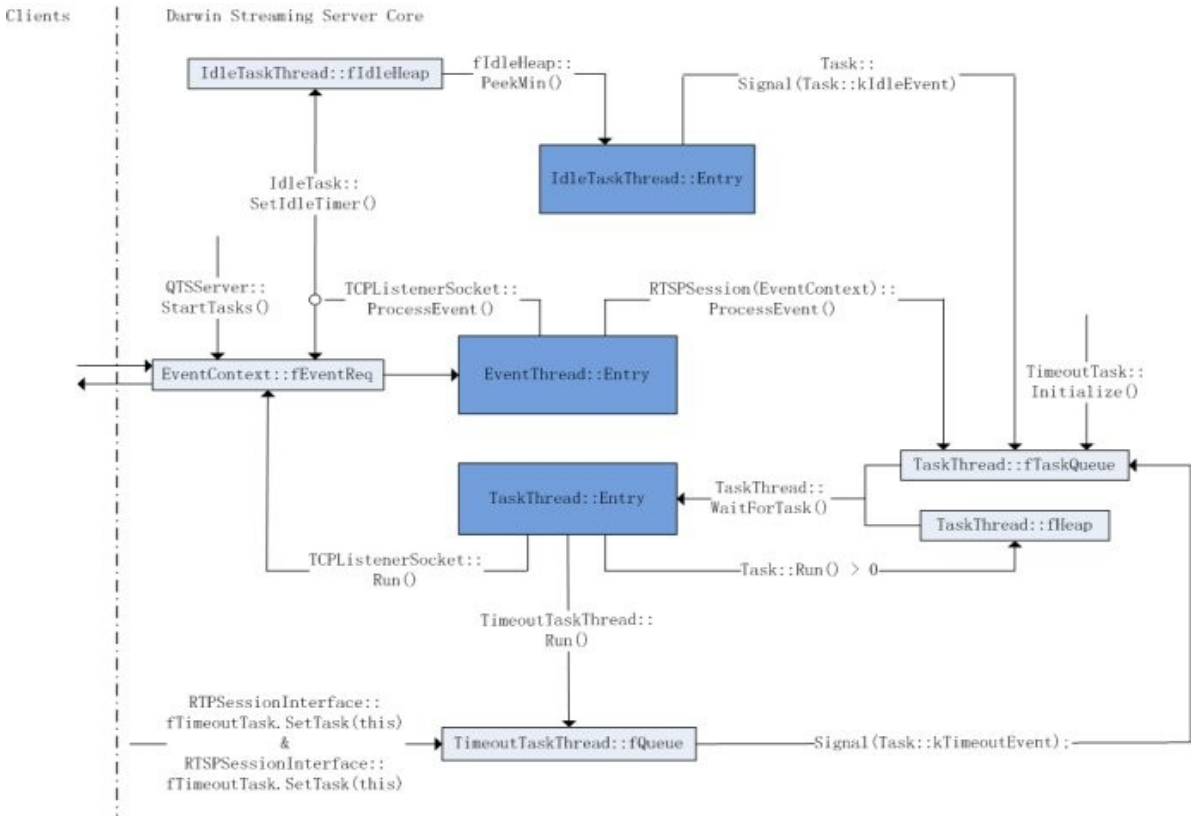
在 DSS 中，除主线程以外，还有有三种类型的线程：

- **TaskThread** ：TaskThread 通过运行 Task 类型对象的 Run 方法来完成相应 Task 的处理。典型的 Task 类型是 RTSPSession 和 RTPSession。TaskThread 的个数是可配置的，***缺省情况下 TaskThread 的个数与处理器的个数一致***。等待被 TaskThread 调用并运行的 Task 放在队列或者堆中。

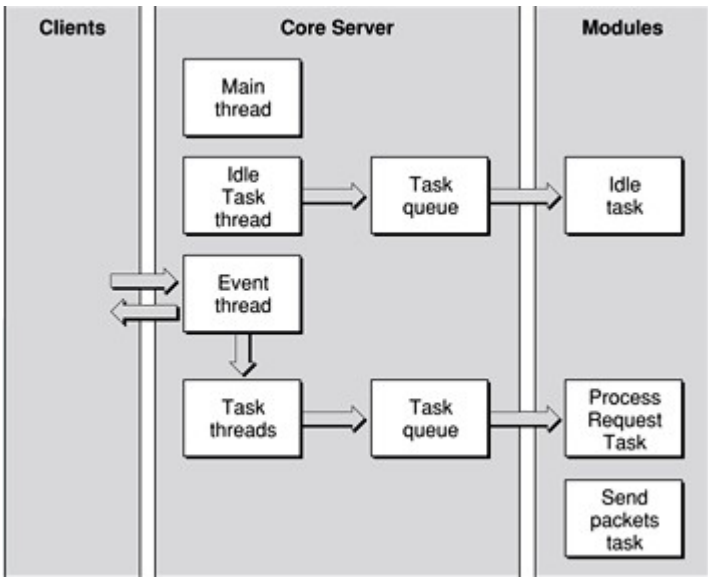
- **EventThread** ：EventThread 负责侦听套接口事件，在 DSS 中，有两种被侦听的事件，分别是建立 RTSP 连接请求的到达和 RTSP 请求的到达。对于 RTSP 连接请求的事件，EventThread 建立一个 RTSP Session，并启动针对相应的 socket 的侦听。对于 RTSP 请求的事件，EventThread 把对应的 RTSPSession 类型的 Task 加入到 TaskThread 的队列中，等待 RTSP 请求被处理。

- **IdleTaskThread** ：IdleTaskThread 管理 IdleTask 类型对象的队列，根据预先设定的定时器触发 IdleTask 的调度。TCPListenerSocket 就是一个 IdleTask 的派生类，当并发连接数达到设定的最大值时，会把派生自 TCPListenerSocket 的 RTSPListenerSocket 加入到 IdleTaskThread 管理的 IdleTask 队列中，暂时停止对 RTSP 端口的侦听，直到被设定好的定时器触发。

下图是 Darwin Streaming Server 核心架构的示意图。在这个示意图中有三种类型的要素，分别是线程，Task 队列或者堆，被侦听的事件。



其中，事件线程（Event thread）是 Event Demultiplexer（事件分离器），任务线程（Task threads）是 Event Handler （事件处理器）。



这里的主线程（Main thread）就是Reactor模式中的Reactor(Initiation Dispatcher)。