Jun,2011

Founder.Com

李浩

[ASIO ANATOMY]

[This document is a draft description of boost::asio. In this paper, firstly, we examine two patterns: Reactor and Proactor and some basic concepts of IOCP (input/output completion port) and etc.]

ASIO 之 剖 析

---Boost ASIO 剖析之初稿

-1: 我预先需要知道哪些知识？

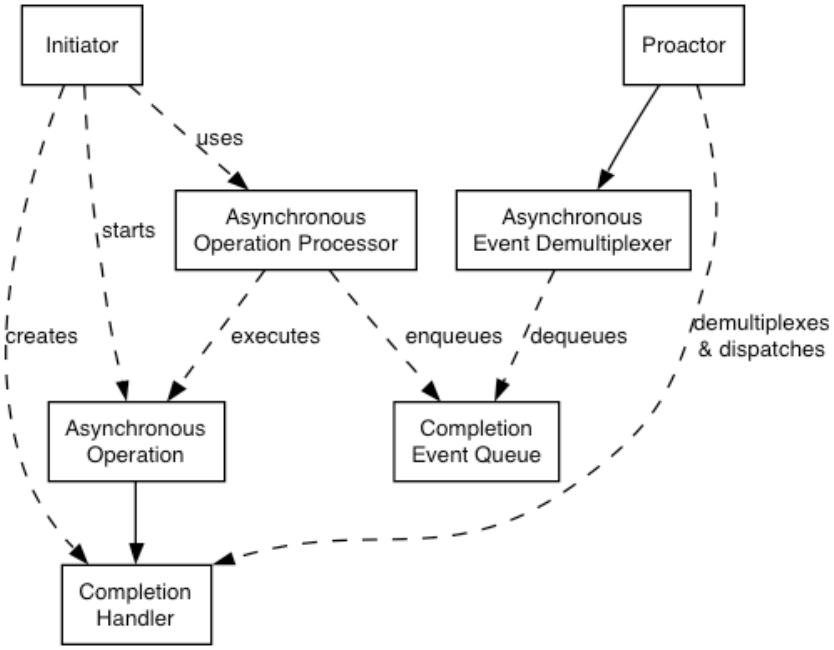
1) socket 通讯的基本知识：如 tcp/ip 基本原理，socket 编程的基本知识，blocking/noblocking。

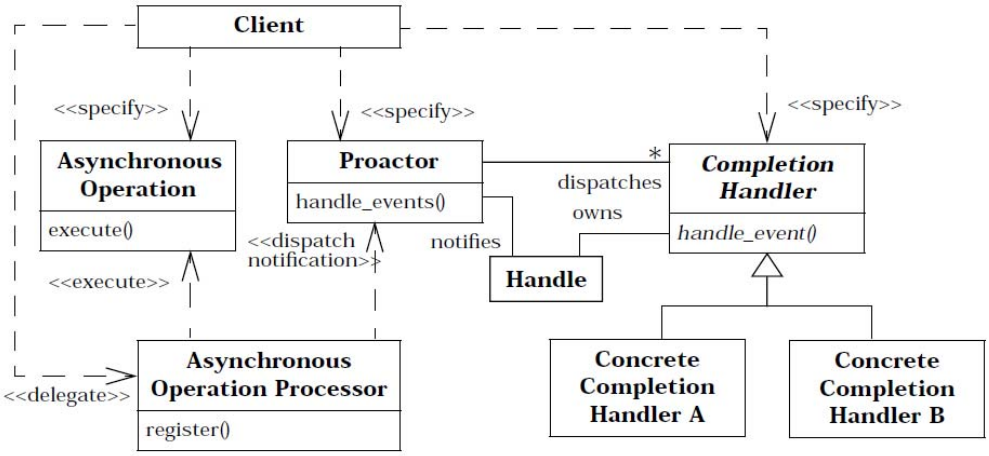
2) 关于 select/poll/epoll 模型的相关基础知识，知道其基本工作原理。

3) IOCP(IO Completion Port)完成端口的基本知识，见附录 2。

4) Reactor 模式。见附录 1。

0：Proactor 模式，描述如下：





其中各个角色的职责及功能如下：

1：Asynchronous Operation

其主要执行相关的异步操作，例如：通过 socket 进行异步读写。

2：Asynchronous Operation Processor

当操作完成后，执行异步操作以及相关完成队列上完成事件的相关操作。从高层用户的角度来看，stream_socket_service 就是一个 Asynchronous Operation Processor。更加具体的就是使用 win_iocp_io_service 或 task_io_service 来完成相应操作完成事件的入队列。在后面的文章中我们将会讨论这两个相关的类。

3：Completion Event Queue

完成事件队列，保存完成事件，直到由异步事件多路复用器从事件队列中取出并将其派发。

4：Completion Handler

由函数对象来处理异步操作的结果。在 boost 中由 boost::bind 来完成相关事件与处理函

数句柄的绑定。在指定的事件发生后，系统将调用其所注册的句柄来对其进行服务。

5：Asynchronous Event Demultiplexer

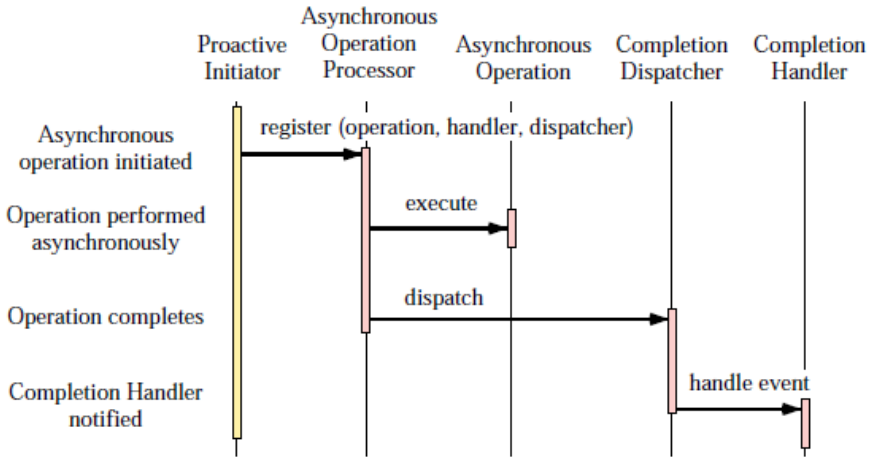
异步事件的多路复用器，阻塞并等待事件发生，并将该完成事件返回其调用者。

6：Proactor

调用异步事件多路复用器，从事件队列中取出事件,并将带有完成事件的完成句柄派发出。在 BOOST 中由 io_service 来描述该概念。

7：Initiator

位于应用层，其发起一个异步操作。Initiator 通过高层接口，例如:basic_stream_socket，与 Asynchronous Operation Processor 交互,其完成类似 stream_socket_service 之类的用户行为服务。一个 Proactor 模式的相关 sample code 如附录 3 所示。而模式中个角色的相互操作的时序图如下所示：



BOOST::ASIO 在具体的实现中对于异步 IO 在不同的平台下其具体实现方式有所不同：在非 Windows 平台下，其采用 Reactor 模式来实现。例如：在 Linux/FreeBSD Unix/Mac OS 平台下所采用的 select/epoll/kqueue；而在 Windows 平台下，则采用 Overlapped IO 来实现。

两种模式(Reactor 与 Proactor)之间对比关系如下：

- Reactor 模式
  - 某个事件处理者宣称它对某个 socket 上的读/写事件很感兴趣；
  - 事件分离者等着这个事件的发生；
  - 当事件发生了，事件分离器被唤醒，这负责通知先前那个事件处理者；
  - 事件处理者收到消息，于是去那个 socket 上读/写数据。 如果需要，它再次宣称对这个 socket 上的读/写事件感兴趣，一直重复上面的步骤；
- Proactor 模式
  - 事件处理者直接投递发一个读/写操作(需 OS 支持这个异步 IO 操作)．这个时候，事件处理者不在关心读/写事件，它只管发这么个请求，它所关注的是读/写操作事件的完成。其只需发出请求命令后，其它事件交由系统，其只需等待操作的完成消息；
  - 事件分离者等着这个读/写事件的完成(与 Reactor 不同)；
  - 事件分离者等待完成事件，操作系统进行具体的 IO 操作，它从目标读取数据，放入用户提供的缓存区中，最后通知事件分离者，IO 事情完成；

- 事件分离者通知之前的事件处理者：IO 事件完成；
- 事件处理者，对已存放在缓冲区中的数据进行处理。

ASIO 剖析

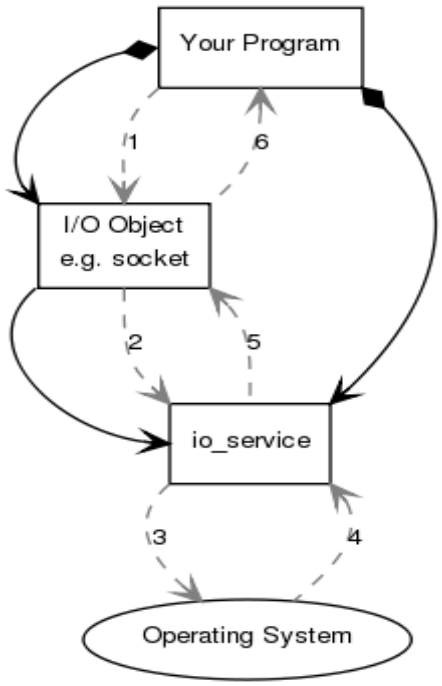
- 下面我们就详细的分析一下 BOOST::ASIO。首先我们从宏观使用的角度来考察 ASIO。作为开始首先从同步 IO 方式开始我们的剖析之旅。

1： BOOST ASIO 下的同步 IO 操作。

由于 asio 库具有在不同平台，不同编译器下工作的特性。其对不同平台和编译器的适应是其一大特色且其具有类似于 ACE 跨平台的特性，相比起 ACE 的复杂庞大，ASIO 则相对较为简单。

1.1：基本概念:

从下图一中我们可以看出：同步方式下的事件发生顺序以如下的方式进行。[以 socket 方式为例]。



（图一）同步方式下的 IO [事件处理流程]

1) 首先，为了能够使用操作系统所提供的服务，程序中至少存在着一个 io_service 对象，其主要的目的是为将 OS 所提供的 IO 服务能够提供给其上层用户 IO 对象，比如：网络套接字，串口等。我们可以使用 boost::asio::io_service io_service; 声明和定义一个 io_service 对象。

2) 将要使用的 IO 对象与一个 io_service 对象进行绑定，以便该 IO 对象能够通过 io_service 与操作系统底层进行通信。 我们以 SOCKET 通信为例，通过下面的代码将 IO 对象与 IO_SERVICE 对象进行绑定：

```
tcp::socket socket(io_service);
```

3) io_service 调用操作系统的服务进行连接。

4) 操作系统将其执行的结果返回给 io_service 对象。

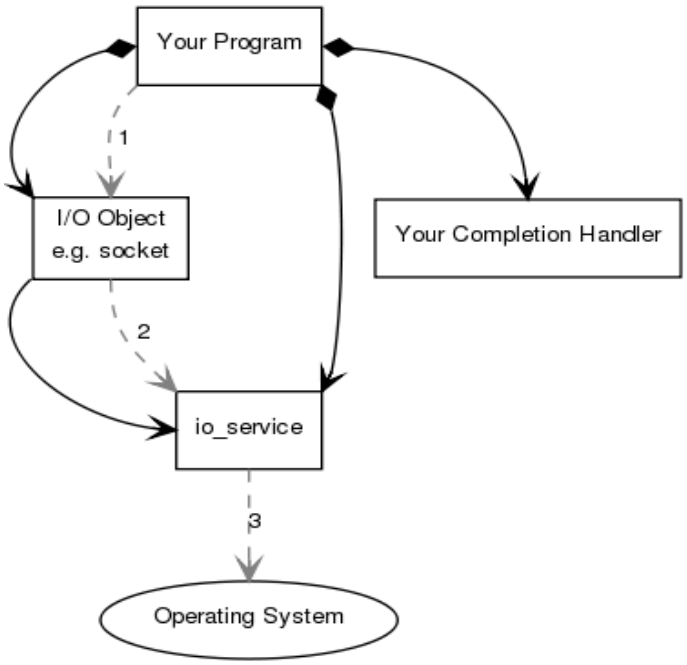
5) io_service 将操作系统返回的错误信息进行相应的转换，将操作系统级的错误代码转换成 boost::system::error_code 形式。然后，将该错误信息发送至 io 对象，此处为 socket 对象。

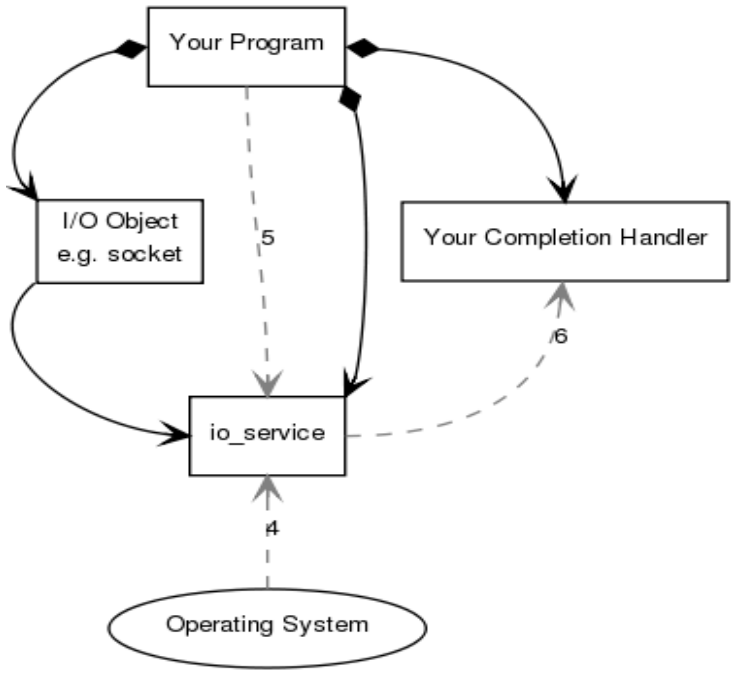
6) IO 对象将该错误信息向上传递至应用程序。

2： BOOST ASIO 下的异步 IO 操作。

2.1:而对于异步方式则与同步方式有所不同，其主要区别如下，其相应执行的流程也有所不同。：

1) 首先由一个 IO 对象，例如：socket，发起一个联接操作。例如：
   socket.async_connect(server_endpoint, your_completion_handler);
   其中 your_completion_handler 为函数指针，其执行联接事件完成后所要执行的服务。
2) IO 对象将其请求信息转发至 io_service。
3) io_service 将通知操作系统，开始一个异步联接请求。
4) 操作系统将联接完成的消息进入完成消息队列，io_service 读取完成的消息由消息队列中的事件消息。
5) 上层应用必须通过调用 io_service.run()函数或其类似的成员函数来获取操作结果。在异步操作未完成时，io_service.run()函数将一直处于阻塞状态。因此，通常是在开始执行异步操作的时候就调用 run 函数。
6) 在 io_service.run ()函数内部，io_service 取出操作结果并将错误代码发送转换成可识别的类型，而后将错误代码转发至 your_completion_handler 函数中，供用户进行相关的错误处理。





（图二）异步方式下的 IO 消息流程

- **以 Proactor 模式的角度来剖析 ASIO [以 socket 为实例]**

从上面对于 Proactor 的分析可知，首先用户以 Initiator 角色发起一个读写事件。Initiator 则通过 stream_socket_service 来进行相关 IO 操作。故而，顺着着这条主线，下面我们就进入 stream_socket_service 的内部来一探究竟。在 stream_socket_service.hpp 文件中我们看到如下一段语句，而该段代码也将会在以后很多的文件中看到。这段代码的主要作用就是：在完成对所运行的平台进行相关的判定后，决定其所使用的具体实现方式。若所运行环境处于 windows 平台下则采用 iocp 的方式来完成异步 IO，若处于 Linux/Unix/MacOS 等环境下则采用 select/poll/epoll 的 Reactor 模式来实现。

- **stream_socket_service 类**

```
#if defined(BOOST_ASIO_HAS_IOCP)
# include <boost/asio/detail/win_iocp_socket_service.hpp>
#else
# include <boost/asio/detail/reactive_socket_service.hpp>
#endif
```

而 BOOST_ASIO_HAS_IOCP 则在 detail/config.h 文件中给出其定义，如下所示：

```
/ Windows: IO Completion Ports.
#if defined(BOOST_WINDOWS) || defined(__CYGWIN__)
# if defined(_WIN32_WINNT) && (_WIN32_WINNT >= 0x0400)
#   if !defined(UNDER_CE)
#     if !defined(BOOST_ASIO_DISABLE_IOCP)
#       define BOOST_ASIO_HAS_IOCP 1
#     endif // !defined(BOOST_ASIO_DISABLE_IOCP)
```

```
#    endif // !defined(UNDER_CE)
# endif // defined(_WIN32_WINNT) && (_WIN32_WINNT >= 0x0400)
#endif // defined(BOOST_WINDOWS) || defined(__CYGWIN__)
// Linux: epoll, eventfd and timerfd.
#if defined(__linux__)
# include <linux/version.h>
# if !defined(BOOST_ASIO_DISABLE_EPOLL)
#    if LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,45)
#      define BOOST_ASIO_HAS_EPOLL 1
#    endif // LINUX_VERSION_CODE >= KERNEL_VERSION(2,5,45)
# endif // !defined(BOOST_ASIO_DISABLE_EVENTFD)
# if !defined(BOOST_ASIO_DISABLE_EVENTFD)
#    if LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,22)
#      define BOOST_ASIO_HAS_EVENTFD 1
#    endif // LINUX_VERSION_CODE >= KERNEL_VERSION(2,6,22)
# endif // !defined(BOOST_ASIO_DISABLE_EVENTFD)
# if defined(BOOST_ASIO_HAS_EPOLL)
#    if (__GLIBC__ > 2) || (__GLIBC__ == 2 && __GLIBC_MINOR__ >= 8)
#      define BOOST_ASIO_HAS_TIMERFD 1
#    endif // (__GLIBC__ > 2) || (__GLIBC__ == 2 && __GLIBC_MINOR__ >= 8)
# endif // defined(BOOST_ASIO_HAS_EPOLL)
#endif // defined(__linux__)
```

看到了这里，我想大家都可以清晰的了解到在许多类的定义中均用到的 BOOST_ASIO_HAS_IOCP 的真实含义了。

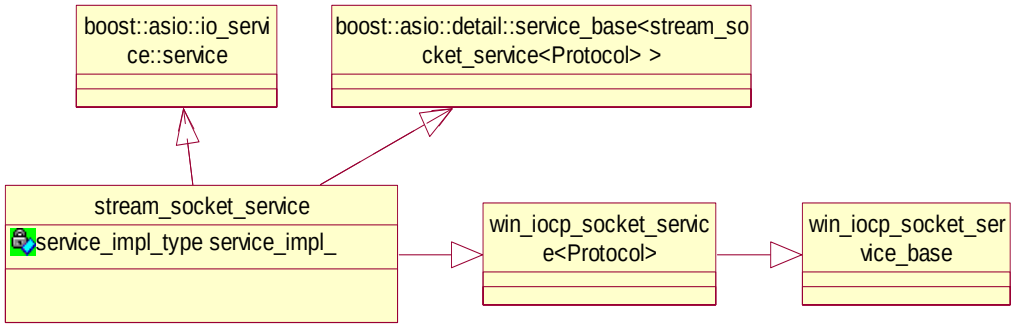
同时，在文件中对于 stream_socket_service 所给出的定义来看，其所继承的基类有两种不同的选择。（如蓝色代码所示）。同样，根据其所处的平台不同，其具体所使用的实现方式也有所不同。

```
template <typename Protocol>
class stream_socket_service
#if defined(GENERATING_DOCUMENTATION)
  : public boost::asio::io_service::service
#else
: public boost::asio::detail::service_base<stream_socket_service<Protocol> >
#endif
{
//而其中关于不同平台上所涉及的实现如下：
//  The type of the platform-specific implementation.
#if defined(BOOST_ASIO_HAS_IOCP)
typedef detail::win_iocp_socket_service<Protocol> service_impl_type; //完成具体的 socket 通信
#else
typedef detail::reactive_socket_service<Protocol> service_impl_type;
#endif
...
service_impl_type service_impl_;
```

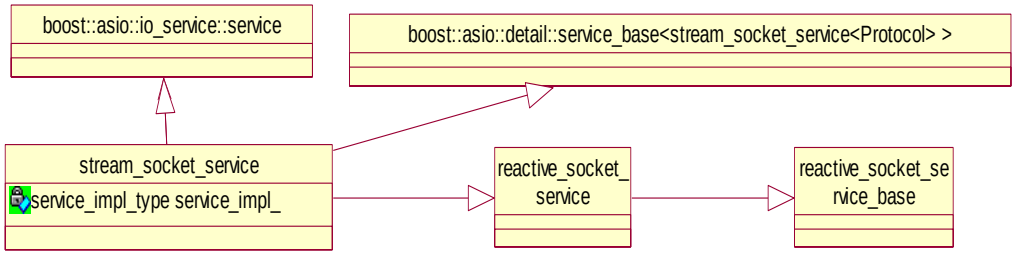
};

而 service_impl_其中之一是便是由 win_iocp_socket_service<Protocol>模板类来给出其定义，或者由 reactive_socket_service<Protocol>来给出。

那么 win_iocp_socket_service<Protocol>类则是来自何方？win_iocp_socket_service 派生于 win_iocp_socket_service_base。相应，对于非 windows 平台的实现：reactive_socket_service 则继承于 reactive_socket_service_base 类。上述的类之间关系可用下图描述：



或者

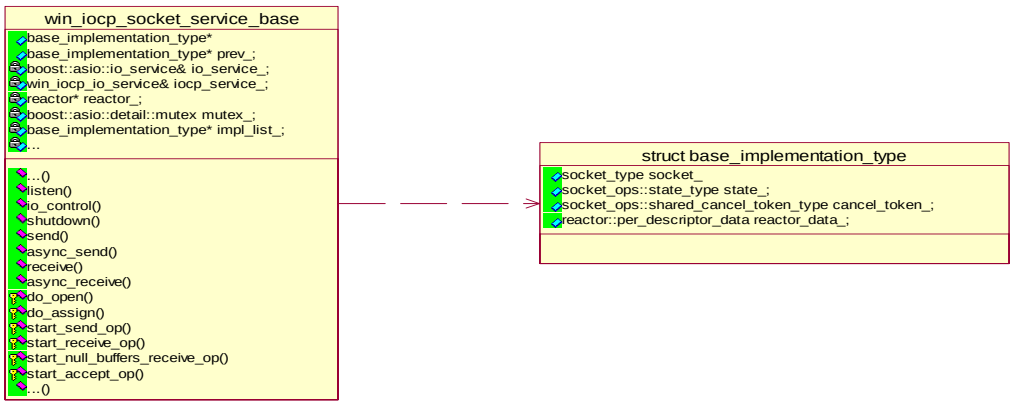


为了更近一步的探索，我们将走进其中之一的 win_iocp_socket_service_base 类，做进一步的剖析。，相应的 Xnix 平台下分析方法类似。

- **win_iocp_socket_server_base**

在 win_iocp_socket_service_base.xpp,, detail/socket_ops.hpp 类中则实现了 windows 平台下的完成端口的主要功能。

在 reactive_socket_service_base, detail/socket_ops.hpp 类中则实现了基础的 IO 操作功能，同样我们在文件中可以看出：相关的实现可以在 xxx.ipp 文件中寻得。其中 xxx 是相应的类名称。win_iocp_socket_server_base 类的主要结构如下：



而在类中的 async_send 或 async_receive 等函数中使用到 win_iocp_socket_send_op 或 win_iocp_socket_recv_op 类，例如我们在 asyn_send 函数中可以看到如下的代码：

```
template <typename ConstBufferSequence, typename Handler>
   void async_send(base_implementation_type& impl,
         const ConstBufferSequence& buffers,
         socket_base::message_flags flags, Handler handler)
{
   typedef win_iocp_socket_send_op<ConstBufferSequence, Handler> op;
      typename op::ptr p = { boost::addressof(handler),
         boost_asio_handler_alloc_helpers::allocate(
            sizeof(op), handler), 0 };
      p.p = new (p.v) op (impl.cancel_token_, buffers, handler);
      buffer_sequence_adapter<boost::asio::const_buffer,
            ConstBufferSequence> bufs(buffers);
      start_send_op (impl, bufs.buffers(), bufs.count(), flags,
            (impl.state_ & socket_ops::stream_oriented) != 0 && bufs.all_empty(),
            p.p);
      p.v = p.p = 0;
}
```

而其中的 start_send_op(

```
      win_iocp_socket_service_base::base_implementation_type& impl,
      WSABUF* buffers, std::size_t buffer_count,
      socket_base::message_flags flags, bool noop, operation* op)
```

函数的定义如下:

```
      {
      update_cancellation_thread_id(impl);
         iocp_service_.work_started();

         if (noop)
            iocp_service_.on_completion(op);
         else if (!is_open(impl))
            iocp_service_.on_completion(op, boost::asio::error::bad_descriptor);
         else
         {
            DWORD bytes_transferred = 0;
            int result = ::WSASend(impl.socket_, buffers,
                  static_cast<DWORD>(buffer_count), &bytes_transferred, flags, op, 0);
            DWORD last_error = ::WSAGetLastError();
            if (last_error == ERROR_PORT_UNREACHABLE)
               last_error = WSAECONNREFUSED;
            if (result != 0 && last_error != WSA_IO_PENDING)
               iocp_service_.on_completion(op, last_error, bytes_transferred);
            else
               iocp_service_.on_pending(op);
         }
```

```
    }
```

从中我们可以看出当系统在通过 WSASend 完成数据的传输后(如蓝色代码所示)，立刻将该完成消息通知并投递到完成消息队列中(如红色代码所示)。而在 win_iocp_io_service::on_completion 的定义如下，其中红色代码表明其对于完成事件在系统层的投递，以及事件进入事件队列中：

```
win_iocp_io_service::on_completion(win_iocp_operation* op,
      DWORD last_error, DWORD bytes_transferred)
{
  // Flag that the operation is ready for invocation.
  op->ready_ = 1;

  // Store results in the OVERLAPPED structure.
  op->Internal = reinterpret_cast<ulong_ptr_t>(
        &boost::asio::error::get_system_category());
  op->Offset = last_error;
  op->OffsetHigh = bytes_transferred;

  // Enqueue the operation on the I/O completion port.
  if (!::PostQueuedCompletionStatus(iocp_.handle,
          0, overlapped_contains_result, op))
  {
    // Out of resources. Put on completed queue instead.
    mutex::scoped_lock lock(dispatch_mutex_);
    completed_ops_.push(op);
    ::InterlockedExchange(&dispatch_required_, 1);
  }
}
```

对于其中的完成事件消息队列的数据结构的详细描述见下文。

- **io_service**

其主要是充当 proactor 角色。我们在 io_service.hpp 中可以看到对于其是否使用 IOCP 模式，其有两种不同的方式方法： windows 平台下，非 windows 平台下。在 windows 平台下使用的是完成端口形式 iocp (IO completion port)，而在非 windows 平台下使用的是 select/poll/epoll 的方式。用以实现 asio 的跨平台的高性能 io 操作。由于 select 模型每次需要内核检查所有 io handle 的状态，导致其效率相对低下，在以后的设计中推荐使用 poll/epoll 方式[Xnix 推荐标准]。而 poll 及 epoll 则在内核获得特定 handle 的状态变化后，通知 poll/epoll 并返回其有效 hanlde，从而无需内核检测所有的 handle，从而进一步的提高了 poll/epoll 的轮询效率。

从以下的 io_service.hpp 文件中我们可以看出对于平台的区别。

```
#if defined(BOOST_ASIO_HAS_IOCP)
# include <boost/asio/detail/win_iocp_io_service_fwd.hpp>   //使用 win 平台下的 iocp 服务。
```

```
#else
# include <boost/asio/detail/task_io_service_fwd.hpp>          //对于非 win 平台。
#endif
#if defined(BOOST_ASIO_HAS_IOCP)
namespace detail { typedef win_iocp_io_service io_service_impl; }
#else
namespace detail { typedef task_io_service io_service_impl; }
#endif
```

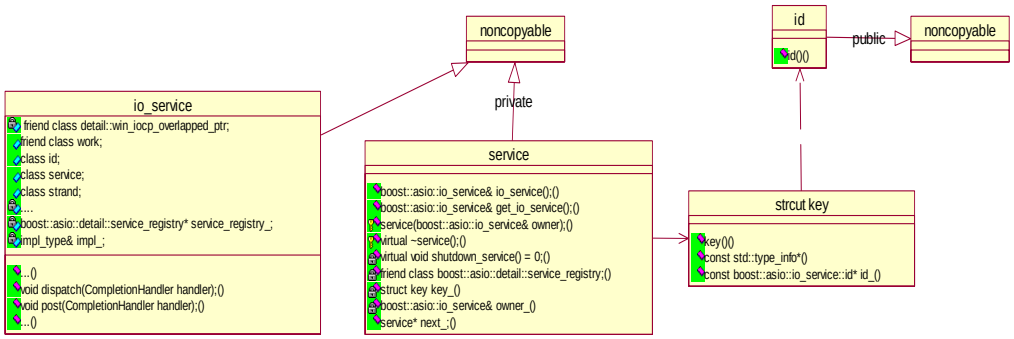
其 io_service 的定义如下，从下面蓝色代码，我们可以看出其完成了完成事件的注册，使得事件完成后所回调的函数与该完成事件能够进行正确的映射：

```
class io_service
  : private noncopyable
{
private:
        typedef detail::io_service_impl impl_type;
        #if defined(BOOST_ASIO_HAS_IOCP)
            friend class detail::win_iocp_overlapped_ptr;
        #endif
public:
  class work;
  friend class work;
  class id;
  class service;
  class strand;
  ....
private:
        #if defined(BOOST_WINDOWS) || defined(__CYGWIN__)
            detail::winsock_init<> init_;
        #elif defined(__sun) || defined(__QNX__) || defined(__hpux) || defined(_AIX) \
          || defined(__osf__)
            detail::signal_init<> init_;
        #endif
  // The service registry.
  boost::asio::detail::service_registry* service_registry_;
  // The implementation.
  impl_type& impl_;
};
```

而其中的 BOOST_ASIO_HAS_IOCP 宏的定义在/asio/detail/config.hpp 文件给出[上文已经给出其详细的说明]。在该文件中给出了不同平台下所应该使用的 IO 方式。如 windows 平台下的完成端口方式，以及 Linux 下的 select/poll/epoll 方式，以及 Mac OS, FreeBSD, NetBSD, OpenBSD 平台下的 kqueue 方式，Solaris 下的/dev/poll 方式。

其中对于 io_service 我们可以追溯到 asio/detail/impl 目录下的 task_io_service.ipp 及

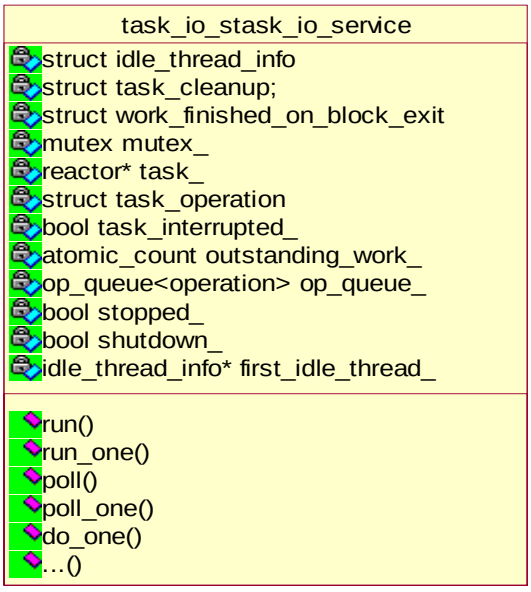
win_iocp_io_service.ipp 这两个文件。其中 task_io_service 为使用 posix 方式完成，而 win_iocp_io_service.ipp 则是在 windows 平台下的完成端口形式。其中的 io_service.run()则提供了上层用户其与 OS 服务之间的桥梁。detail/impl/service_registry.ipp 在该文件中完成了对于异步读写端口函数的注册。用以注册其所服务的函数句柄。



- **win_iocp_io_service**

在 win_iocp_io_service.ipp 文件中的 do_one 函数，我们可以看出 其主要是完成上层用户所提交的操作，如完成端口的 io 操作，定时操作。
而在 task_io_service.ipp 文件中则是 Xnix 下的完成 IO 操作。同样，文件中的 do_one 其与 win_iocp_io_service.ipp 文件中的 do_one 起着相同的作用。

首先，我们来分析一下 Xnix 下的实现，其主要是在 task_io_service.ipp 文件中完成。而 Xnix 下使用的是 reactor 模式来完成。 task_io_service 的结构如下。



idle_thread_info 为空闲线程情况链表，其中存放那些可以进行有效的 IO 读写的空闲线程。
下面系统所使用的几个重要的数据结构的定义：

```
struct task_io_service::idle_thread_info
{
  event wakeup_event;                  //事件描述
```

```
    idle_thread_info* next;                 //下一个对象。
};

struct task_io_service::task_cleanup      //完成任务的清除工作。
{
    ~task_cleanup()
    {
        // Enqueue the completed operations and reinsert the task at the end of
        // the operation queue.
        lock_->lock();
        task_io_service_->task_interrupted_ = true;
        task_io_service_->op_queue_.push(*ops_);
        task_io_service_->op_queue_.push(&task_io_service_->task_operation_);
    }
    task_io_service* task_io_service_;
    mutex::scoped_lock* lock_;
    op_queue<operation>* ops_;
};

struct task_io_service::work_finished_on_block_exit
{
    ~work_finished_on_block_exit()
    {
        task_io_service_->work_finished();
    }
    task_io_service* task_io_service_;
};
```

而其中 run/run_one/poll/poll_one 等函数均调用 do_one 来完成具体的 io 任务。下面是 do_one 的主要描述：

```
std::size_t task_io_service::do_one(mutex::scoped_lock& lock,
                                    task_io_service::idle_thread_info* this_idle_thread)
{
    while (!stopped_)
    {
        if (!op_queue_.empty()) //操作事件队列,为 Proactor 模式中的 Completion Event Queue
        {
            // Prepare to execute first handler from queue.
            operation* o = op_queue_.front();
            op_queue_.pop();
            bool more_handlers = (!op_queue_.empty());
            if (o == &task_operation_) //判定任务队列中是否具有未完成的操作。
            {
```

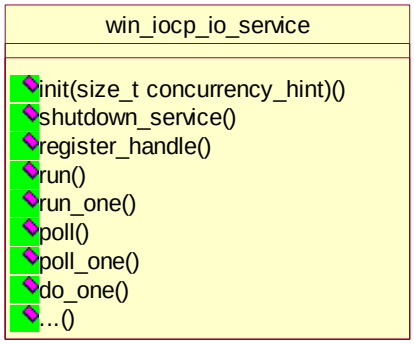
```
        task_interrupted_ = more_handlers || polling;
        // If the task has already run and we're polling then we're done.
        if (task_has_run && polling)
        {
          task_interrupted_ = true;
          op_queue_.push(&task_operation_);
          return 0;
        }
        task_has_run = true;
        if (!more_handlers || !wake_one_idle_thread_and_unlock(lock)) //请求一个空闲的线
程，并对其进行解锁并用来服务该项。
          lock.unlock();
        op_queue<operation> completed_ops;
        task_cleanup c = { this, &lock, &completed_ops };
        (void)c;
        // Run the task. May throw an exception. Only block if the operation
        // queue is empty and we're not polling, otherwise we want to return
        // as soon as possible.
        task_->run(!more_handlers && !polling, completed_ops);
      }
      else
      {
        if (more_handlers)
          wake_one_thread_and_unlock(lock);
        else
          lock.unlock();
        // Ensure the count of outstanding work is decremented on block exit.
        work_finished_on_block_exit on_exit = { this };
        (void)on_exit;
        // Complete the operation. May throw an exception.
        o->complete(*this); // deletes the operation object
        return 1;
      }
    }
    else if (this_idle_thread)
    {
      // Nothing to run right now, so just wait for work to do.
      this_idle_thread->next = first_idle_thread_;
      first_idle_thread_ = this_idle_thread;
      this_idle_thread->wakeup_event.clear(lock);
      this_idle_thread->wakeup_event.wait(lock);
    }
    else
    {
```

```
            return 0;
        }
    }
    return 0;
}
```

在 windows 平台下由 win_iocp_io_service.hpp 及 win_iocp_io_service.ipp 来定义其相关的完成端口操作。在 windows 平台下，我们首先要创建一个 IO 完成端口，在其端口创建成功后，我们需要将我们所需要监视的 IO 端口进行关联注册，使其与一个完成端口相关联。
其相关工作由 CreateIoCompletionPort 函数来完成。而通过函数 GetQueuedCompletionStatus 函数来完成端口上等待下一个 IO Package。而 boost 下的完成端口类型的核心功能由 win_iocp_io_service 类来完成。下面是 win_iocp_io_service.hpp 文件中涉及到的几个重要的数据结构的定义。

```
struct win_iocp_io_service::work_finished_on_block_exit
{
    ~work_finished_on_block_exit()
    {
        io_service_->work_finished();
    }
    win_iocp_io_service* io_service_;
};

struct win_iocp_io_service::timer_thread_function    //由定时器所触发的操作。
{
    void operator()()
    {
        while (::InterlockedExchangeAdd(&io_service_->shutdown_, 0) == 0)
        {
            if (::WaitForSingleObject(io_service_->waitable_timer_.handle, INFINITE) ==
WAIT_OBJECT_0) //等待消息的到来，否则阻塞程序的调用。
            {
                ::InterlockedExchange(&io_service_->dispatch_required_, 1);
                ::PostQueuedCompletionStatus(io_service_->iocp_.handle,
                        0, wake_for_dispatch, 0);
            }
        }
    }

    win_iocp_io_service* io_service_;
```

win_iocp_io_service 类结构如下[略去数据成员]：



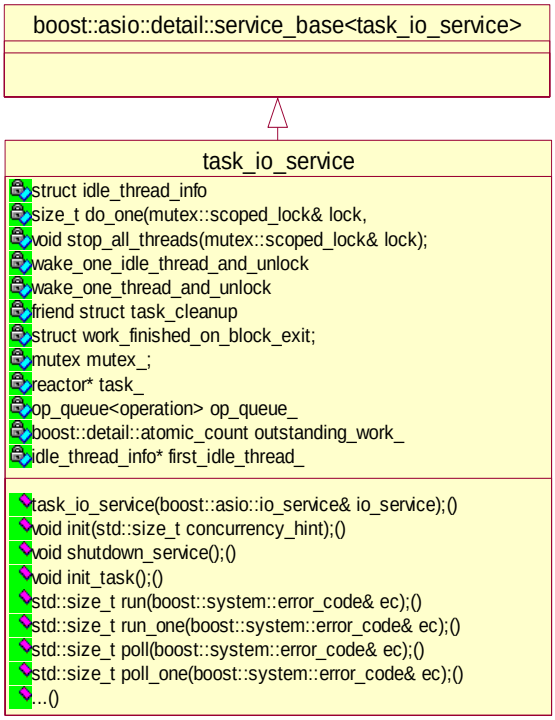
作为该类得核心函数 do_one 的描述如下：首先其轮询所以的待处理的操作或者是定时器，若存在待处理的则调用函数 GetQueuedCompletionStatus 来获得具体的操作及其完成的信息。

```
for (;;)
{
  // Try to acquire responsibility for dispatching timers and completed ops.
  if (::InterlockedCompareExchange(&dispatch_required_, 0, 1) == 1)
  {
    mutex::scoped_lock lock(dispatch_mutex_);
    // Dispatch pending timers and operations.
    op_queue<win_iocp_operation> ops;
    ops.push(completed_ops_);
    timer_queues_.get_ready_timers(ops);
    post_deferred_completions(ops);
    update_timeout();
  }

  // Get the next operation from the queue.
  DWORD bytes_transferred = 0;
  dword_ptr_t completion_key = 0;
  LPOVERLAPPED overlapped = 0;
  ::SetLastError(0);
  BOOL ok = ::GetQueuedCompletionStatus(iocp_.handle, &bytes_transferred,
      &completion_key, &overlapped, block ? gqcs_timeout : 0);
  DWORD last_error = ::GetLastError();

  if (overlapped)
  {
    win_iocp_operation* op = static_cast<win_iocp_operation*>(overlapped);
    boost::system::error_code result_ec(last_error,
        boost::asio::error::get_system_category());

    // We may have been passed the last_error and bytes_transferred in the
    // OVERLAPPED structure itself.
```

```
    if (completion_key == overlapped_contains_result)
    {
      result_ec = boost::system::error_code(static_cast<int>(op->Offset),
          *reinterpret_cast<boost::system::error_category*>(op->Internal));
      bytes_transferred = op->OffsetHigh;
    }
    // Otherwise ensure any result has been saved into the OVERLAPPED
    // structure.
    else
    {
      op->Internal = reinterpret_cast<ulong_ptr_t>(&result_ec.category());
      op->Offset = result_ec.value();
      op->OffsetHigh = bytes_transferred;
    }
    // Dispatch the operation only if ready. The operation may not be ready
    // if the initiating function (e.g. a call to WSARecv) has not yet
    // returned. This is because the initiating function still wants access
    // to the operation's OVERLAPPED structure.
    if (::InterlockedCompareExchange(&op->ready_, 1, 0) == 1)
    {
      // Ensure the count of outstanding work is decremented on block exit.
      work_finished_on_block_exit on_exit = { this };
      (void) on_exit;
      op->complete(*this, result_ec, bytes_transferred);
      ec = boost::system::error_code();
      return 1;
    }
  }
  else if (!ok)
  {
    if (last_error != WAIT_TIMEOUT)
    {
      ec = boost::system::error_code(last_error,
          boost::asio::error::get_system_category());
      return 0;
    }
    // If we're not polling we need to keep going until we get a real handler.
    if (block)
      continue;
    ec = boost::system::error_code();
    return 0;
  }
  else if (completion_key == wake_for_dispatch)
  {
```

```
      // We have been woken up to try to acquire responsibility for dispatching
      // timers and completed operations.
    }
    else
    {
      // The stopped_ flag is always checked to ensure that any leftover
      // interrupts from a previous run invocation are ignored.
      if (::InterlockedExchangeAdd(&stopped_, 0) != 0)
      {
        // Wake up next thread that is blocked on GetQueuedCompletionStatus.
        if (!::PostQueuedCompletionStatus(iocp_.handle, 0, 0, 0))
        {
          last_error = ::GetLastError();
          ec = boost::system::error_code(last_error,
              boost::asio::error::get_system_category());
          return 0;
        }
        ec = boost::system::error_code();
        return 0;
      }
    }
  }
```

至此我们讨论了不同平台下对于完成端口的实现一些细节性的内容，希望能够给大家关于 boost:asio 底层运作机制的一个初步概况，对于非 Windows 平台下所使用的 task_io_service 则可在 asio/detail/及 asio/detail/impl 目录下找到其相关源码。task_io_service 类的结构如下，其中列举了重要的相关数据结构:



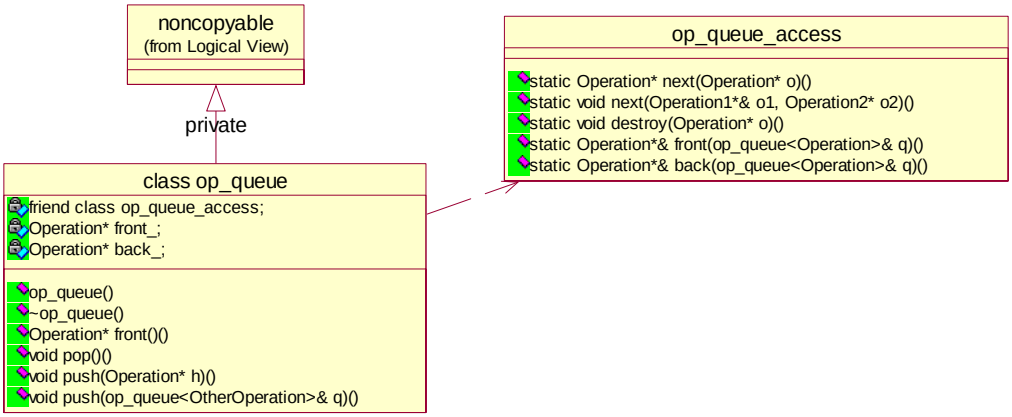
为了使用这些功能 boost 在其上面包装了一个 io_service 壳，从而屏蔽了关于完成端口所实现的一些细节性的描述，而只需要关注 io_service 即可并通过它将 socket 或者是串口等提供 io 服务的一些对象进行关联。这些对于端口的关联均在 task_io_serivce 和 win_icop_io_service 完成。在 io_service.hpp 文件中我们可以看到如下的语句：

```
#if defined(BOOST_ASIO_HAS_IOCP)
namespace detail { typedef win_iocp_io_service io_service_impl; }
#else
namespace detail { typedef task_io_service io_service_impl; }
#endif
….
private:
   typedef detail::io_service_impl impl_type;
…
impl_type& impl_
```

至此，io_service 将提供一个接口，可将 socket handle，以及 serial handle 等与完成端口进行相关联。

- **op_queue**

其中 op_queue 为相关操作的队列，系统所产生或者获得相关操作均由此队列中取得相关操作句柄。而其数据结构的定义如下：



- Endpoint 对象：

下面我们就以一个具体的例子来说明使用 asio 进行同步方式下的网络通信。在详细介绍之前，我们首先要阐述一个概念：服务接入点--- tcp::endpoint。其位于 boost::asio::ip::tcp 的名字空间中。其主要的作用就是：使用一个指定的端口或者 IP 地址构建 endpoint 对象，并将其绑定到 socket 对象上。

tcp::endpoint (tcp::v4(), 13) 使用端口 13 及 ipv4 协议在本地主机上创建一个服务接入点[服务器模式]。tcp::endpoint(“ip.address”,port) 客服端模式。

我们由 endpoint 的实现中可以看出其对于 posix socket 编程中对于 socket 对象中参数设置的封装。

```
endpoint::endpoint()
  : data_()
{
  data_.v4.sin_family = AF_INET;
  data_.v4.sin_port = 0;
  data_.v4.sin_addr.s_addr = INADDR_ANY;
}
endpoint::endpoint(int family, unsigned short port_num)
  : data_()
{
  using namespace std;        // For memcpy.
  if (family == PF_INET)
  {
    data_.v4.sin_family = AF_INET;
    data_.v4.sin_port =
      boost::asio::detail::socket_ops::host_to_network_short(port_num);
    data_.v4.sin_addr.s_addr = INADDR_ANY;
```

```
  }
  else
  {
    data_.v6.sin6_family = AF_INET6;
    data_.v6.sin6_port =
      boost::asio::detail::socket_ops::host_to_network_short(port_num);
    data_.v6.sin6_flowinfo = 0;
    boost::asio::detail::in6_addr_type tmp_addr = IN6ADDR_ANY_INIT;
    data_.v6.sin6_addr = tmp_addr;
    data_.v6.sin6_scope_id = 0;
  }
}
```

上式中 host_to_network_short 其作用是完成主机字节序到网络字节序的转换。其上封装了 posix socket 编程下的 htons(value),这样封装的目的是以更加易于理解的方式进行程序的编写。

其中 data_的数据类型为：

```
  union data_union
  {
    boost::asio::detail::socket_addr_type base;
    boost::asio::detail::sockaddr_storage_type storage;
    boost::asio::detail::sockaddr_in4_type v4;
    boost::asio::detail::sockaddr_in6_type v6;
  } data_;
```

而 data_.v4 的类型为：sockaddr_in 类型[其位于 detail/socket_types.hpp 中 typedef sockaddr_in sockaddr_in4_type;]，由此我们可以看出经过层层的封装，我们使用 endpoint 时候，无需再过度的关注底层的细节设置，而使我们能够更加关注于系统的业务逻辑方面的设计。

- **socket**

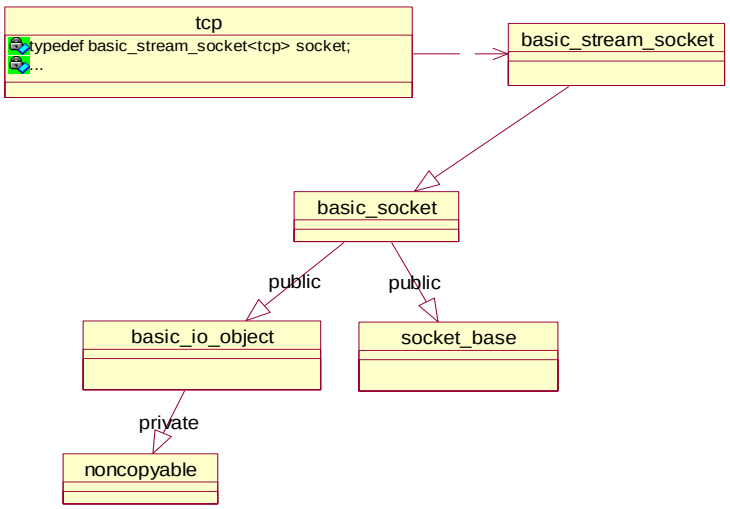
下面我们就对 socket 类进行一些初步的分析。在 ip/tcp.hpp 文件中我们可以看到如下语句：typedef basic_stream_socket<tcp> socket; 可以看出 socket 其真面目为：basic_stream_socket<tcp> 下面就让我们走进 basic_stream_socket<tcp>一探究竟。我们在 basic_stream_socket.hpp 文中我们可以得到 basic_stream_socket 类的定义。其定义如下：

```
template <typename Protocol,
    typename StreamSocketService = stream_socket_service<Protocol> >
class basic_stream_socket
  : public basic_socket<Protocol, StreamSocketService>
```

其继承了 socket 基类 basic_socket，而对 basic_stream_socket 进行初始化之前，首先会完成其基的初始化工作。经过这层由 socket→=basic_stream_socket->basic_socket 的主线，现在我们将目光转向更底层的类：basic_socket，来看看它为我们提供了什么样的服务？在 asio 目录下的 basic_socket.hpp 文件中，我们找到了该类得定义，其形式如下：

```
template <typename Protocol, typename SocketService>
class basic_socket
  : public basic_io_object<SocketService>,
    public socket_base
```

而 basic_io_object<SocketService> 和 socket_base 分别在 asio 目录下的 basic_io_object.hpp 文件和 socket_base.hpp 文件中给出他们的定义。他们之间的关系可以用如下来表示：



这些 socket 通过 namesspace socket_ops 下的函数具体完成 socket 地址的 bind，listen，accept，complete_iocp_accept，connect 等功能的实现。下面我们就到命名空间 socket_ops 中一探究竟。

- **socket_ops**

该命名空间位于 detail/socket_ops.hpp 及 detail/impl/socket_ops.ipp 文件中，其主要的作用就是完成关于 socket 通讯的系统 api 层得操作。例如：我们在上层应用中写了如下语句时:

```
io_service ios;
ip::tcp::endpoint endpoint(tcp::v4(), 80);
ip::tcp::acceptor acceptor(ios, endpoint);
```

在底层所作的工作便有使用 socket_ops 空间内的 bind 函数将一个已设置地址与端口的地址结构 struct sockaddr* addr 与一个 socket 句柄进行绑定。而这些涉及 socket 与 OS 底层发生关系的操作均可以在 socket_ops 命名空间中寻得。关于其空间中所涉及的具体函数在这里不在赘述，可具体参考其源码。

- **boost::bind**

在我们使用 asio 库的时候，经常看见 boost::bind 函数，那么该函数的主要作用是什么呢？我们用 boost::bind 函数又能做哪些工作呢？下面我们就来探究一下该函数。总体来说，其位于 boost 命名空间中，而它的主要的作用描述为：其是标准函数 **std::bind1st** 和 **std::bind2nd** 的泛化形式。其支持任意函数对象，函数，函数指针，成员函数指针。其可以将一个指定的值绑定到任意一个参数上，或者将输入参数重新绑定到参数中的其它任何位置。例如，对于如下的代码：

```
int f(int a, int b)
```

```
{
    return a + b;
}

int g(int a, int b, int c)
{
    return a + b + c;
}
```

bind(f, 1, 2)其作用如同 f(1, 2)一样， bind(g, 1, 2, 3)() 则等价于 g(1, 2, 3).

- **acceptor**

在创建 tcp::acceptor 时候其定义为 在 tcp.hpp 文件中。

typedef basic_socket_acceptor<tcp> acceptor; 而 basic_socket_acceptor 的定义如下：

template <typename Protocol,

typename SocketAcceptorService = socket_acceptor_service<Protocol> > class basic_socket_acceptor

而在对 acceptor 进行实例化时候对其 Protocol 参数为 tcp,而 SocketAcceptorService 为 socket_acceptor_service<Protocol>，因此其为 basic_socket_acceptor 所提供的模板参数为 socket_acceptor_service<tcp>。因此我们在调用 tcp::socket::acceptor 进程对象的初始化时候我们获得的 acceptor 对象其实是使用 basic_acceptor_service<tcp,socket_acceptor_serivce<tcp> > 对对象进行初始化工作。其中 basic_acceptor_service 的定义为 template <typename Protocol,typename SocketAcceptorService = socket_acceptor_service<Protocol> >。

下面我们就对进行分析 basic_acceptor_serivce <tcp, socket_acceptor_serivce<tcp> >。从 socket_acceptor_service.hpp 文件中对于其定义可以看出其具有如下形式的继承方式。

```
template <typename Protocol>
class socket_acceptor_service
#if defined(GENERATING_DOCUMENTATION)
  : public boost::asio::io_service::service
#else
  : public boost::asio::detail::service_base<socket_acceptor_service<Protocol> >
#endif
```

- **综述**

从上述的描述中我们可以看出 boost::asio 对于传统的 posix socket 编程进行了有效的封装。例如：posix socket 编程中所使用的 socket, listen, connect 等函数。 基本的逻辑概念描述：所有的 IO 操作均由 **io_service** 来充当中间层的功能，其主要负责上层程序与 OS 底层所提供的 IO 服务沟通,其 ASIO 中的所有操作均通过 io_service 与操作系统所提的服务进行关联。ASIO 提供了 io 操作的一些核心功能。其所在的 namespace 路径如下

boost::asio::xxx

其下包括了如下的异步 IO 对象。

(1)ip::tcp::socket 网络套接字类,提供 tcp 方式的网络套接字
主要的目的：完成网络套接字对象的初始化工作。

(2)ip::tcp::acceptor 连接接受类，其完成 posix socket api 中的 accept 函数的功能，接受客户端的连接请求。

(3) ip::udp::socket 提供 udp 方式的网络套接字

(4) deadline_time 提供定时操作。为了能够完成网络的超时控制，可以由其中的定时对象来完成异步方式的定时操作。而在 POSIX 下其 timer 所提供的时间控制精度不足，无法满足毫秒或者更为精细的时间控制。为了获得更高的定时精度.我们通常采用 select 等方式。而在 POSIX 下其测试网络连通状态的通常做法是由一个 timer 在指定的时间内发送一个 heartbeat 报文，用以测试网络的状态，我们也可在所发的 heartbeat 报文上打上 timestamp 用以观察网络的延迟。该种方式普遍应用在多种协议中：例如移动的 CMPP,联通的 SGIP,以及银联的 POS 交易协议。

detail/push_options.hpp 中对相关编译器参数的识别，及不同编译器中所使用参数的的配置。
detail/socket_types.hpp 中则重新定义/更名了 Xnix/windows 平台下 socket 编程所有的各种数据结构.例如：微软平台：_MSC_VER， 塞班系统：__SYMBIAN32__，HPUnix: __hpux ,SUN: __sun, 及对于 select 的支持情况。

- **POSIX 下的做法：**

为了满足高性能的 socket io 操作。我们使用 select/poll/epoll 等函数来完成存在着大规模并发 io 的网络通信。

服务端代码：下面是一个传统的 POSIX socket 开发的例子，在完成 socket 对象的建立后，

```
int h_socket= socket (PF_INET, SOCK_STREAM , 0);
…
struct sockaddr_in    serveraddr;
serveraddr.sin_family = AF_INET ;                          //设置好协议族类型
serveraddr.sin_addr.s_addr = inet_addr (szListenIP) ;      //设置监听地址
serveraddr.sin_port    = htons ((short)iListenPort);       //设置监听端口
bind(h_socket,(struct sockaddr*)&serveraddr ,sizeof(serveraddr));   //地址信息的绑定
…
listen (h_socket,1024);                                    //开始监听。
…
```

```
  FD_ZERO (…)
  FD_SET (…)
  Iret= select(1024,&readset,NULL,NULL ,NULL);   //poll, epoll 等高效的复用 i/o
  if (iret) {
   iRecvedSkt_hd = accept   (h_socket,(struct sockaddr*)&csin_addr,&len ) ; //接收客户端请求
     …
  pthread_create (&accptedThread, NULL ,StartBuinessThread , (void*)&iRecvedSkt_hd);//启动
 一个线程对其进行服务。
 }

static void* StartBuinessThread (void* strThreadParams)
{ …}
```

- **参考文件:**
  - **1： Douglas C. Schmidt Proactor Pattern .**
  - **2: Proactor An Object Behavioral Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events**
  - **3: MSDN online: http://msdn.microsoft.com/en-us/library/aa365198.aspx**

- **附录 1：**

一个简单的 Reactor 示例，我们以 IO 操作中的读操作为例。Reactor 具有以下行为方式。
- 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，注册新的事件，然后返还控制权。

相应的 sample code 如下所示：

```
class handler
{
public:
    virtual void onRead() = 0;
    virtual void onWrite() = 0;
```

```
    virtual void onAccept() = 0;
};

class dispatch
{
public:
    void poll()
    {
        // add fd in the set.
        //
        // poll every fd
        int c = select( 0, &read_fd, &write_fd, 0, 0 );
        if( c > 0 )
        {
            for each fd in the read_fd_set
            {     if fd can read
                    _handler->onRead();
                if fd can accept
                    _handler->onAccept();
            }

            for each fd in the write_fd_set
            {
                if fd can write
                    _handler->onWrite();
            }
        }
    }

    void setHandler( handler *_h )
    {
        _handler = _h;
    }

private:
    handler *_handler;
};

/// application
class MyHandler : public handler
{
public:
    void onRead()
    {
    }
```

```
    void onWrite()
    {
    }

    void onAccept()
    { }

};
```

- **附录 2：**

完成端口（IOC）是用来实现高性能服务的一种常用方法，主要是通过操作系统提供的异步调用功能实现 IO 操作，可以通过很少的线程数实现高性能的并发服务。一个完成端口读写的例子，如下：

```
#define _WIN32_WINNT     0x0500
#include <cstdlib>
#include <clocale>
#include <ctime>
#include <iostream>
#include <vector>
#include <algorithm>
#include <winsock2.h>
#include <mswsock.h>

using namespace std;

#pragma comment(lib,"ws2_32.lib")
#pragma comment(lib,"mswsock.lib")

const intMAX_BUFFER_SIZE=1024;
const intPRE_SEND_SIZE=1024;
const intQUIT_TIME_OUT=3000;
const intPRE_DOT_TIMER=QUIT_TIME_OUT/80;

typedef enum{IoTransFile,IoSend,IoRecv,IoQuit} IO_TYPE;

typedef struct
{
    SOCKEThSocket;
    SOCKADDR_IN ClientAddr;
}PRE_SOCKET_DATA,*PPRE_SOCKET_DATA;

typedef struct
{
    OVERLAPPED      oa;
    WSABUF          DataBuf;
```

```
    char        Buffer[MAX_BUFFER_SIZE];
    IO_TYPE     IoType;
}PRE_IO_DATA,*PPRE_IO_DATA;

typedef vector<PPRE_SOCKET_DATA>    SocketDataVector;
typedef vector<PPRE_IO_DATA>        IoDataVector;

SocketDataVector    gSockDataVec;
IoDataVector        gIoDataVec;

CRITICAL_SECTION    csProtection;

char* TimeNow(void)
{
    time_tt=time(NULL);
    tm *localtm=localtime(&t);
    static chartimemsg[512]={0};

    strftime(timemsg,512,"%Z: %B %d %X,%Y",localtm);
    return timemsg;
}

BOOL TransFile(PPRE_IO_DATApIoData,PPRE_SOCKET_DATApSocketData,DWORDdwNameLen)
{
    pIoData->Buffer[dwNameLen-1]='\0';

    HANDLEhFile=CreateFile(pIoData->Buffer,GENERIC_READ,0,NULL,OPEN_EXISTING,0,NULL);
    BOOLbRet=FALSE;
    if(hFile!=INVALID_HANDLE_VALUE)
    {
       cout<<"Transmit File "<<pIoData->Buffer<<" to client"<<endl;
       pIoData->IoType=IoTransFile;
       memset(&pIoData->oa,0,sizeof(OVERLAPPED));
       *reinterpret_cast<HANDLE*>(pIoData->Buffer)=hFile;
       TransmitFile(pSocketData->hSocket,hFile,GetFileSize(hFile,NULL),PRE_SEND_SIZE,reinterpret_ca
st<LPOVERLAPPED>(pIoData),NULL,TF_USE_SYSTEM_THREAD);
       bRet=WSAGetLastError()==WSA_IO_PENDING;
    }
    else
       cout<<"Transmit File "<<"Error:"<<GetLastError()<<endl;
     return bRet;
}

DWORDWINAPI ThreadProc(LPVOIDIocpHandle)
```

```
{
    DWORDdwRecv=0;
    DWORDdwFlags=0;

    HANDLEhIocp=reinterpret_cast<HANDLE>(IocpHandle);
    DWORDdwTransCount=0;
    PPRE_IO_DATApPreIoData=NULL;
    PPRE_SOCKET_DATApPreHandleData=NULL;

    while(TRUE)
    {
      if(GetQueuedCompletionStatus(hIocp,&dwTransCount,
         reinterpret_cast<LPDWORD>(&pPreHandleData),
         reinterpret_cast<LPOVERLAPPED*>(&pPreIoData),INFINITE))
     {
         if(0==dwTransCount&&IoQuit!=pPreIoData->IoType)
         {
            cout<<"Client:"
               <<inet_ntoa(pPreHandleData->ClientAddr.sin_addr)
               <<":"<<ntohs(pPreHandleData->ClientAddr.sin_port)
               <<" is closed"<<endl;

            closesocket(pPreHandleData->hSocket);

            EnterCriticalSection(&csProtection);
              IoDataVector::iteratoritrIoDelete=find(gIoDataVec.begin(),gIoDataVec.end(),pPreIoData);
              SocketDataVector::iteratoritrSockDelete=find(gSockDataVec.begin(),gSockDataVec.end(),
pPreHandleData);
             delete *itrIoDelete;
             delete *itrSockDelete;
             gIoDataVec.erase(itrIoDelete);
             gSockDataVec.erase(itrSockDelete);
             LeaveCriticalSection(&csProtection);
            continue;
         }

         switch(pPreIoData->IoType){
         case IoTransFile:
            cout<<"Client:"
               <<inet_ntoa(pPreHandleData->ClientAddr.sin_addr)
               <<":"<<ntohs(pPreHandleData->ClientAddr.sin_port)
               <<" Transmit finished"<<endl;
            CloseHandle(*reinterpret_cast<HANDLE*>(pPreIoData->Buffer));
            goto LRERECV;
```

```
            case IoSend:
                cout<<"Client:"
                    <<inet_ntoa(pPreHandleData->ClientAddr.sin_addr)
                    <<":"<<ntohs(pPreHandleData->ClientAddr.sin_port)
                    <<" Send finished"<<endl;

    LRERECV:
                pPreIoData->IoType=IoRecv;
                pPreIoData->DataBuf.len=MAX_BUFFER_SIZE;
                memset(&pPreIoData->oa,0,sizeof(OVERLAPPED));
                WSARecv(pPreHandleData->hSocket,&pPreIoData->DataBuf,1,
                    &dwRecv,&dwFlags,
                    reinterpret_cast<LPWSAOVERLAPPED>(pPreIoData),NULL);
                break;
    case IoRecv:
                cout<<"Client:"
                    <<inet_ntoa(pPreHandleData->ClientAddr.sin_addr)
                    <<":"<<ntohs(pPreHandleData->ClientAddr.sin_port)
                    <<" recv finished"<<endl;
                pPreIoData->IoType=IoSend;

                if(!TransFile(pPreIoData,pPreHandleData,dwTransCount))
                {
                    memset(&pPreIoData->oa,0,sizeof(OVERLAPPED));
                    strcpy(pPreIoData->DataBuf.buf,"File transmit error!\r\n");
                    pPreIoData->DataBuf.len=strlen(pPreIoData->DataBuf.buf);

                    WSASend(pPreHandleData->hSocket,&pPreIoData->DataBuf,1,
                        &dwRecv,dwFlags,
                        reinterpret_cast<LPWSAOVERLAPPED>(pPreIoData),NULL);
                }
                break;
            case IoQuit:
                goto LQUIT;
            default:
                ;
            }
        }
    }
    LQUIT:
        return 0;
}
```

```
HANDLEhIocp=NULL;
SOCKEThListen=NULL;

BOOLWINAPI ShutdownHandler(DWORDdwCtrlType)
{
    PRE_SOCKET_DATAPreSockData={0};
    PRE_IO_DATAPreIoData={0};
   PreIoData.IoType=IoQuit;
   if(hIocp)
    {
       PostQueuedCompletionStatus(hIocp,1,
       reinterpret_cast<ULONG_PTR>(&PreSockData),
       reinterpret_cast<LPOVERLAPPED>(&PreIoData));
        cout<<"Shutdown at "<<TimeNow()<<endl<<"wait for a moment please"<<endl;

       for(intt=0;t<80;t+=1)
       {
          Sleep(PRE_DOT_TIMER);
          cout<<".";
       }

       CloseHandle(hIocp);
    }

    inti=0;
     for(;i<gSockDataVec.size();i++)
    {
       PPRE_SOCKET_DATApSockData=gSockDataVec[i];
       closesocket(pSockData->hSocket);
       delete pSockData;
    }
     for(i=0;i<gIoDataVec.size();i++)
    {
       PPRE_IO_DATApIoData=gIoDataVec[i];
       delete pIoData;
    }
     DeleteCriticalSection(&csProtection);
    if(hListen)
       closesocket(hListen);

    WSACleanup();
    exit(0);
    return TRUE;
}
```

```
LONGWINAPI MyExceptionFilter(struct _EXCEPTION_POINTERS *ExceptionInfo)
{
    ShutdownHandler(0);
    return EXCEPTION_EXECUTE_HANDLER;
}

u_shortDefPort=8182;

intmain(intargc,char **argv)
{
    if(argc==2)
       DefPort=atoi(argv[1]);

    InitializeCriticalSection(&csProtection);
    SetUnhandledExceptionFilter(MyExceptionFilter);
    SetConsoleCtrlHandler(ShutdownHandler,TRUE);

    hIocp=CreateIoCompletionPort(INVALID_HANDLE_VALUE,NULL,0,0);

    WSADATA data={0};
    WSAStartup(0x0202,&data);

    hListen=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
    if(INVALID_SOCKET==hListen)
    {
       ShutdownHandler(0);
    }

    SOCKADDR_IN addr={0};
    addr.sin_family=AF_INET;
    addr.sin_port=htons(DefPort);

    if(bind(hListen,reinterpret_cast<PSOCKADDR>(&addr),
       sizeof(addr))==SOCKET_ERROR)
    {
       ShutdownHandler(0);
    }

    if(listen(hListen,256)==SOCKET_ERROR)
       ShutdownHandler(0);

    SYSTEM_INFO si={0};
    GetSystemInfo(&si);
```

```
    si.dwNumberOfProcessors<<=1;

    for(inti=0;i<si.dwNumberOfProcessors;i++)
    {

       QueueUserWorkItem(ThreadProc,hIocp,WT_EXECUTELONGFUNCTION);
    }

    cout<<"Startup at "<<TimeNow()<<endl
       <<"work on port "<<DefPort<<endl
       <<"press CTRL+C to shutdown"<<endl<<endl<<endl;

    while(TRUE)
    {
       intnamelen=sizeof(addr);
       memset(&addr,0,sizeof(addr));
       SOCKEThAccept=accept(hListen,reinterpret_cast<PSOCKADDR>(&addr),&namelen);
        if(hAccept!=INVALID_SOCKET)
       {
          cout<<"accept a client:"<<inet_ntoa(addr.sin_addr)<<":"<<ntohs(addr.sin_port)<<endl;
           PPRE_SOCKET_DATApPreHandleData=newPRE_SOCKET_DATA;
          pPreHandleData->hSocket=hAccept;
          memcpy(&pPreHandleData->ClientAddr,&addr,sizeof(addr));
          CreateIoCompletionPort(reinterpret_cast<HANDLE>(hAccept),hIocp,reinterpret_cast<DWOR
D>(pPreHandleData),0);
          PPRE_IO_DATApPreIoData=new(nothrow) PRE_IO_DATA;
           if(pPreIoData)
          {
             EnterCriticalSection(&csProtection);
             gSockDataVec.push_back(pPreHandleData);
             gIoDataVec.push_back(pPreIoData);
             LeaveCriticalSection(&csProtection);
             memset(pPreIoData,0,sizeof(PRE_IO_DATA));
             pPreIoData->IoType=IoRecv;
             pPreIoData->DataBuf.len=MAX_BUFFER_SIZE;
             pPreIoData->DataBuf.buf=pPreIoData->Buffer;
             DWORDdwRecv=0;
             DWORDdwFlags=0;
             WSARecv(hAccept,&pPreIoData->DataBuf,1,&dwRecv,&dwFlags,reinterpret_cast<WSAOVE
RLAPPED*>(pPreIoData),NULL);
          }
          else
          {
             delete pPreHandleData;
```

```
            closesocket(hAccept);
          }
        }
    }
    return 0;
}
```

- **附录 3：**

一个 proactor 模式的示例。

```
class AsyIOProcessor
{
public:
      void do_read()
      {
            //send read operation to OS
            // read io finished.and dispatch notification
            _proactor->dispatch_read();
      }

private:
      Proactor *_proactor;
};

class Proactor
{
public:
      void dispatch_read()
      {
            _handlerMgr->onRead();
      }

private:
      HandlerManager *_handlerMgr;
};

class HandlerManager
{
public:
      typedef std::list<Handler*> HandlerList;
public:
      void onRead()
      {
            // notify all the handlers.
            std::for_each( _handlers.begin(), _handlers.end(), onRead );
```

```
    }
private:
    HandlerList *_handlers;
};

class Handler
{
public:
    virtual void onRead() = 0;
};

// application level handler.
class MyHandler : public Handler
{
public:
    void onRead()
    {
    }
};
```