Boost.asio 简介

目录树

Asio/

Asio/detail/

Asio/detail/impl/

Asio/impl/

Asio/ip/

Asio/local/

Asio/posix/

Asio/ssl/

Asio/windows/

# Asio/buffer.hpp

集中写入，分散读取

Asio 库中所有的io操作使用的buffer采用的都是通用的buffer容器(包含buffer的容器)，以便实现集中写入(gather write)和分散读取(scatter read)。

这个头文件中的类和函数都比较简单，就不必啰嗦了。

Buffer不是数据的所有者，使用者应负责释放资源。

唯一值得一提的就是mutable\_buffer，const\_buffer, 都没有获取buffer大小的操作，这样做的好处是避免混淆buffer和BufferSequence的size()操作。

# Asio/io\_service.hpp

# Io\_service实现概况

io\_service 类实现了一个可扩展，类型安全的I/O服务集合。

io\_service::impl\_type 是io\_service 的底层的具体实现，如果是windows平台并且支持iocp则采用detail::win\_iocp\_io\_service，否则则使用detail::task\_io\_service。

io\_service::service 是 io\_service中服务的抽象，所有的io\_service 服务都间接继承自此类。

Io\_service::id 用于标识不同的服务。

Io\_service::service::key 用来在service\_registry中查找特定类型的service。

detail::service\_registry 类用于管理io\_service中注册的各种服务，提供获取和添加服务的接口。

# Service与Service\_registry

Service\_registry 类用于对io\_service内部服务的管理。每个添加到service\_registry的service的key都由service\_registry进行初始化，这样当需要使用service的时候就可以根据特定的key在链表中找到对应的服务。

Service\_registry 实现主要涉及三个文件

asio/detail/service\_registry.hpp;

asio/detail/impl/service\_registry.hpp;

asio/detail/impl/service\_registry.ipp;

构造函数

Service\_registry的定义如下

class service\_registry : private noncopyable

{

public:

// Constructor. Adds the initial service.

template <typename Service, typename Arg>

service\_registry(boost::asio::io\_service& o, Service\* initial\_service, Arg arg);

// Destructor.

BOOST\_ASIO\_DECL ~service\_registry();

// Notify all services of a fork event.

BOOST\_ASIO\_DECL void notify\_fork(boost::asio::io\_service::fork\_event fork\_ev);

// Get the first service object cast to the specified type. Called during

// io\_service construction and so performs no locking or type checking.

template <typename Service>

Service& first\_service();

// Get the service object corresponding to the specified service type. Will

// create a new service object automatically if no such object already

// exists. Ownership of the service object is not transferred to the caller.

template <typename Service>

Service& use\_service();

// Add a service object. Throws on error, in which case ownership of the

// object is retained by the caller.

template <typename Service>

void add\_service(Service\* new\_service);

// Check whether a service object of the specified type already exists.

template <typename Service>

bool has\_service() const;

private:

// Initialise a service's key based on its id.

BOOST\_ASIO\_DECL static void init\_key(

boost::asio::io\_service::service::key& key,

const boost::asio::io\_service::id& id);

#if !defined(BOOST\_ASIO\_NO\_TYPEID)

// Initialise a service's key based on its id.

template <typename Service>

static void init\_key(boost::asio::io\_service::service::key& key,

const boost::asio::detail::service\_id<Service>& /\*id\*/);

#endif // !defined(BOOST\_ASIO\_NO\_TYPEID)

// Check if a service matches the given id.

BOOST\_ASIO\_DECL static bool keys\_match(

const boost::asio::io\_service::service::key& key1,

const boost::asio::io\_service::service::key& key2);

// The type of a factory function used for creating a service instance.

typedef boost::asio::io\_service::service\*

(\*factory\_type)(boost::asio::io\_service&);

// Factory function for creating a service instance.

template <typename Service>

static boost::asio::io\_service::service\* create( boost::asio::io\_service& owner);

// Destroy a service instance.

BOOST\_ASIO\_DECL static void destroy(boost::asio::io\_service::service\* service);

// Helper class to manage service pointers.

struct auto\_service\_ptr;

friend struct auto\_service\_ptr;

struct auto\_service\_ptr

{

boost::asio::io\_service::service\* ptr\_;

~auto\_service\_ptr() { destroy(ptr\_); }

};

// Get the service object corresponding to the specified service key. Will

// create a new service object automatically if no such object already

// exists. Ownership of the service object is not transferred to the caller.

BOOST\_ASIO\_DECL boost::asio::io\_service::service\* do\_use\_service(

const boost::asio::io\_service::service::key& key,

factory\_type factory);

// Add a service object. Throws on error, in which case ownership of the

// object is retained by the caller.

BOOST\_ASIO\_DECL void do\_add\_service(

const boost::asio::io\_service::service::key& key,

boost::asio::io\_service::service\* new\_service);

// Check whether a service object with the specified key already exists.

BOOST\_ASIO\_DECL bool do\_has\_service(

const boost::asio::io\_service::service::key& key) const;

// Mutex to protect access to internal data.

mutable boost::asio::detail::mutex mutex\_;

// The owner of this service registry and the services it contains.

boost::asio::io\_service& owner\_;

// The first service in the list of contained services.

boost::asio::io\_service::service\* first\_service\_;

};

# io\_service实现 task\_io\_service

# epoll\_reactor

# strand

## call\_stack

# deadline\_timer

# signal\_set

# stream\_socket

# stream\_descriptor