MOOON-server 组件使用指南

一见@2011.8/11

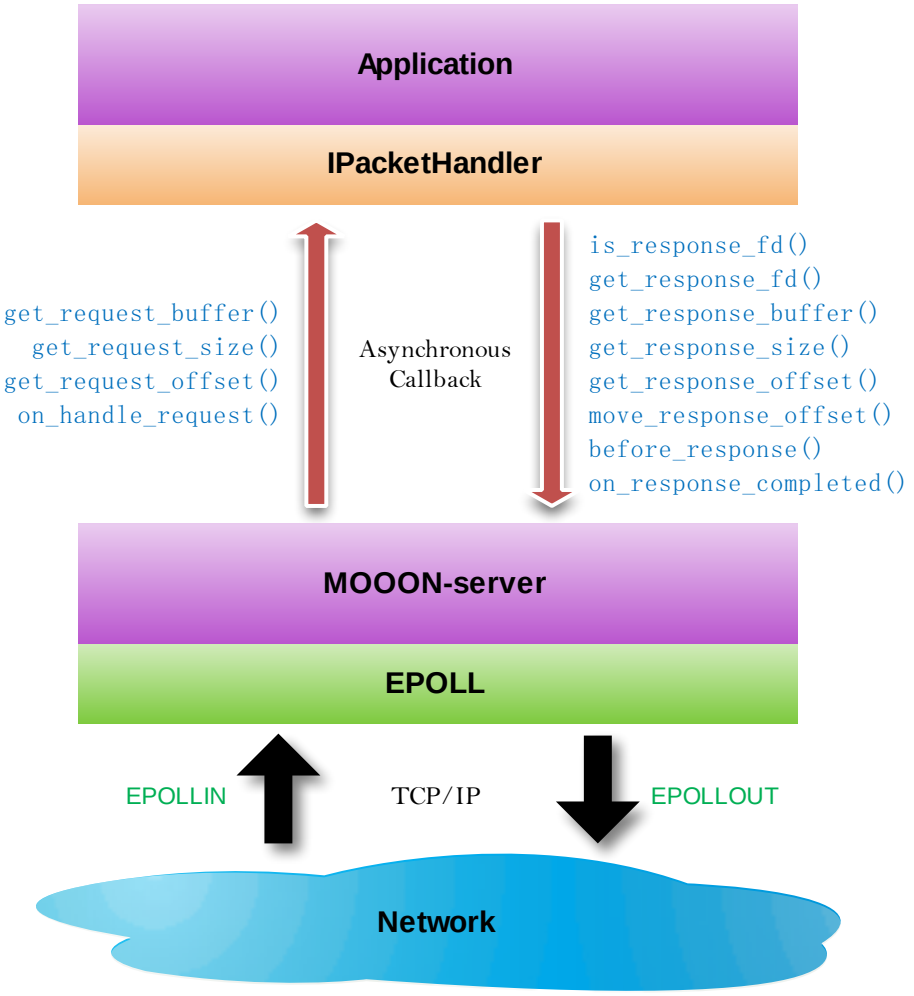
1. 介绍

MOOON-server 是一个 TCP 服务端公共组件，提供收发数据和发送文件的功能。

2. 功能

1) 异步收发数据
2) 异步发送文件
3) 长短连接控制
4) 连接超时控制
5) 线程切换-可控制一个连接从一个线程切换到另一个线程

3. 工作原理

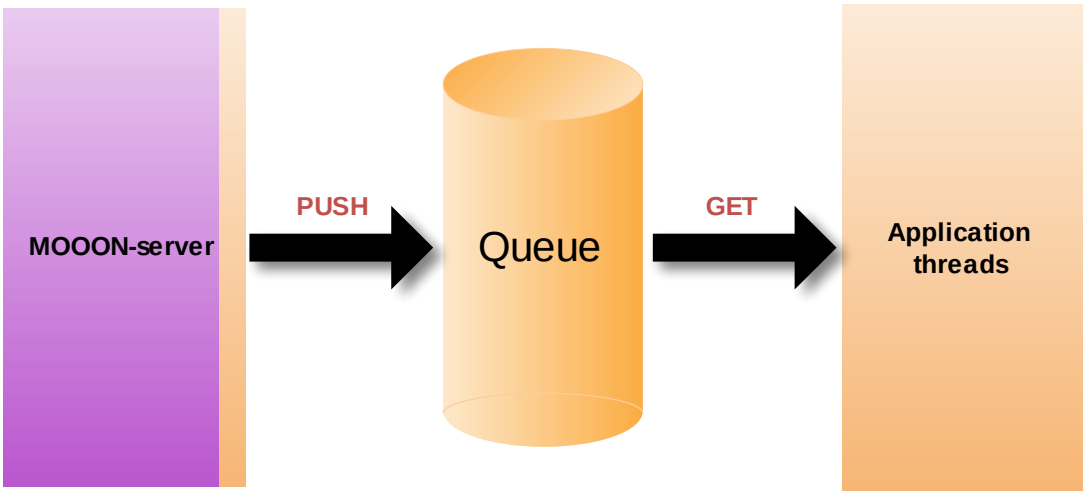


MOOON-server 的工作原理如上图所示，基于 EPOLL，以提供高性能的大并发处理能力。MOOON-server 和 Application（应用）之间采用异步回调的方式进行交互，当有数据可接收或可发送数据时，都会调用 IPacketHandler 的相应方法。数据的收和发，都是在 MOOON-server 的线程中完成。MOOON-server 提供由一组线程来接受连接请求和数据的收发，这一组线程构造一个线程池，线程个数是可以根据需要指定的，但是不能动态变化。

MOOON-server 本身并不维护和提供任何数据缓冲区，所以数据的收和发，都必须由 Application 提供缓冲区，及必要的缓冲区信息。

4. 应用场景

4.1. 消息队列



使用消息队列，这是最常见的应用场景。基于 MOOON-server 做一层简单的包装，目的是为接收一个完整的消息，当消息接收完整之后，就将消息 PUSH 进 Queue（队列）中。一旦消息进入队列中，则会激活 Application threads（应用线程），Application threads 从 Queue 取出消息进行处理。

4.2. FTP 服务器

要支持文件的上传和下载，消息队列方式就不适合了，因为文件一大或一多，如果再读入内存，然后放入消息队列，就会导致内存被撑爆。这种场景下，所有工作都可在 MOOON-server 线程中完成，不需要额外的 Application 线程。

4.3. 线程切换

如果在下载某个文件时，不管是什么时候，或者来自哪儿的下载请求，都必须由同一个线程服务时，上述两种方式就都不合适了，比如要能容忍某块磁盘挂起故障（即调用 read/write 等磁盘操作的线程会被挂住不动）。

这个时候，接收请求信息的线程和服务的线程需要存在切换，即接收线程根据已经接收到的信息，找到服务线程，然后将连接转交给服务线程。

请注意，此功能要求连接池的大小为 0，也就是不能使用连接池功能，性能会略有下降。

5. 接口说明

MOOON-server 的名字空间名为：**server**。

需要引用的头文件为：**#include <server/server.h>**，它会包含所有其它需要使用到的头文件。

5.1. server

5.1.1.所在头文件

#include <server/server.h>

5.1.2.server_t

MOOON-server 的类型，请总是直接使用 server_t，因为它的具体定义将来可能变化，当前的定义为：typedef void* server_t;，但将来 server_t 可能变成接口 IServer。

5.1.3.logger

MOOON-server 组件的日志器，默认为 NULL，日志将直接通过标准输出和标准出错输出。

✧ **为何 MOOON-server 需要外部传递 logger？**

答：目的是方便 MOOON-server 的日志风格和使用者的日志风格统一，以避免出现不同日志风格和产生多个不同的日志文件。

5.1.4.create 函数

用来创建和启动一个 MOOON-server 组件实例，函数原型为：

```
extern server_t create(IConfig* config, IFactory* factory);
```

其中 IConfig 是 MOOON-server 组件需要用的配置接口，IFactory 是 MOOON-server 组件需要用来创建用户对象的工厂。

5.1.5.destroy 函数

用来销毁指定的 MOOON-server 组件实例，函数原型为：

```
extern void destroy(server_t server);
```

5.2. IConfig

5.2.1.所在头文件

#include <server/config.h>

5.2.2.接口说明

定义 MOOON-server 需要用到的配置项。

✧ **为何不采用配置文件的方式？**

MOOON-server 只所有没有直接采用配置文件，是为方便使用者统一配置文件格式和风格。

5.2.3.接口定义

```
/***
  * 配置回调接口
  */
class CALLBACK_INTERFACE IConfig
{
public:
    /** 空虚拟析构函数，以屏蔽编译器告警 */
    virtual ~IConfig() {}

    /** 得到 epoll 大小 */
    virtual uint32_t get_epoll_size() const { return 10000; }

    /** 得到框架的工作线程个数 */
    virtual uint16_t get_thread_number() const { return 1; }

    /** 得到每个线程的连接池大小 */
    virtual uint32_t get_connection_pool_size() const { return 10000; }

    /** 连接超时秒数 */
    virtual uint32_t get_connection_timeout_seconds() const { return 10; }

    /** 得到 epool 等待超时毫秒数 */
    virtual uint32_t get_epoll_timeout_milliseconds() const { return 2000; }

    /** 得到监听参数 */
    virtual const net::ip_port_pair_array_t& get_listen_parameter() const = 0;

    /** 得到每个线程的接管队列的大小 */
    virtual uint32_t get_takeover_queue_size() const { return 1000; }
};
```

5.3. IConnection

5.3.1.所在头文件

#include <server/connection.h>

5.3.2.接口说明

提供 IConnection，是为方便使用者获取连接相关的信息。

5.3.3.接口定义

```
/***
   * 网络连接
   */
class IConnection
{
public:
      virtual ~IConnection() {}

      /** 得到字符串格式的标识 */
      virtual std::string str() const = 0;

      /** 得到本端的端口号 */
      virtual net::port_t self_port() const = 0;

      /** 得到对端的端口号 */
      virtual net::port_t peer_port() const = 0;

      /** 得到本端的 IP 地址 */
      virtual const net::ip_address_t& self_ip() const = 0;

      /** 得到对端的 IP 地址 */
      virtual const net::ip_address_t& peer_ip() const = 0;

      /** 得到所在线程的顺序号 */
      virtual uint16_t get_thread_index() const = 0;
};
```

5.4. IFactory

5.4.1.所在头文件

#include <server/factory.h>

5.4.2.接口说明

工厂接口，用来创建需要使用者实现的回调对象。

5.4.3.接口定义

```
/***
   * 工厂回调接口，用来创建报文解析器和报文处理器
   */
class CALLBACK_INTERFACE IFactory
{
public:
      /** 空虚拟析构函数，以屏蔽编译器告警 */
      virtual ~IFactory() {}

      /** 创建线程伙伴 */
      virtual IThreadFollower* create_thread_follower(uint16_t index) { return NULL; }

      /** 创建包处理器 */
      virtual IPacketHandler* create_packet_handler(IConnection* connection) = 0;
};
```

5.5. IPacketHandler

5.5.1.所在头文件

#include <server/packet_handler.h>

5.5.2.接口说明

包处理器接口，这是 MOOON-server 中需要使用者实现的最核心的接口。整个接口的

定义主要由三部分组成：

1) 与请求相关的，用以提供接收数据的必要信息，如数据往哪儿收
2) 与响应相关的，用以提供发送数据的必要信息，如发送多大的数据
3) 与网络连接相关的，如连接被关闭

5.5.3.接口定义

```
/***
  * 下一步动作指标器
  */
struct Indicator
{
    bool reset;                  /** 是否复位状态 */
    uint16_t thread_index;   /** 下一步跳到的线程顺序号 */
    uint32_t epoll_events;   /** 下一步注册的 epoll 事件，可取值 EPOLLIN 或 EPOLLOUT，或 EPOLLIN|EPOLLOUT */
};

/***
  * 包处理器，包括对请求和响应的处理
  */
class CALLBACK_INTERFACE IPacketHandler
{
public:
    /** 空虚拟析构函数，以屏蔽编译器告警 */
    virtual ~IPacketHandler() {}

    /***
      * 复位解析状态
      */
    virtual void reset() = 0;

    /***
      * 连接被关闭
      */
    virtual void on_connection_closed() { }

    /***
      * 得到用来接收数据的 Buffer
      */
    virtual char* get_request_buffer() = 0;

    /***
      * 得到用来接收数据的 Buffer 大小
```

```
    */
    virtual size_t get_request_size() const = 0;

    /***
     * 得到从哪个位置开始将接收到的数据存储到 Buffer
     */
    virtual size_t get_request_offset() const = 0;

    /***
     * 对收到的数据进行解析
     * @param indicator.reset 默认值为 false
     *              indicator.thread_index 默认值为当前线程顺序号
     *              indicator.epoll_events 默认值为 EPOLLIN
     * @data_size: 新收到的数据大小
     */
    virtual util::handle_result_t on_handle_request(size_t data_size, Indicator& indicator) = 0;

    /***
     * 是否发送一个文件
     */
    virtual bool is_response_fd() const { return false; }

    /***
     * 得到文件句柄
     */
    virtual int get_response_fd() const { return -1; }

    /***
     * 得到需要发送的数据
     */
    virtual const char* get_response_buffer() const { return NULL; }

    /***
     * 得到需要发送的大小
     */
    virtual size_t get_response_size() const { return 0; }

    /***
     * 得到从哪偏移开始发送
     */
    virtual size_t get_response_offset() const { return 0; }

    /***
     * 移动偏移
```

```
        * @offset:  本次发送的字节数
        */
    virtual void move_response_offset(size_t offset) {}

    /***
        *  开始响应前的事件
        */
    virtual void before_response() {}

    /***
        *  包发送完后被回调
        * @param indicator.reset  默认值为 true
        *              indicator.thread_index  默认值为当前线程顺序号
        *              indicator.epoll_events  默认值为 EPOLLOUT
        * @return  如果返回 util::handle_continue 表示不关闭连接继续使用；
        *              如果返回 util::handle_release 表示需要移交控制权，
        *              返回其它值则关闭连接
        */
    virtual util::handle_result_t on_response_completed(Indicator& indicator) { return util::handle_finish; }
};
```

5.6. IThreadFollower

5.6.1.所在头文件

#include <server/thread_follower.h>

5.6.2.接口说明

提供执行和线程相关的机会，如线程进入工作状态前的处理。

5.6.3.接口定义

```
/***
  *  线程伙计
  */
class IThreadFollower
{
public:
    virtual ~IThreadFollower() {}
```

```
    /***
      * 线程 run 之前被调用
      * @return  如果返回 true，则会进入 run 过程，否则线程绕过 run 而退出
      */
    virtual bool before_run() { return true; }

    /***
      * 线程 run 之后被调用
      */
    virtual void after_run() {}
};
```

6. 使用步骤

1) 实现以下接口
IConfig、IFactory 和 IPacketHandler，可选实现 IThreadFollower。
2) 创建配置实例
3) 创建工厂实例
4) 创建和启动 MOOON-server 实例

7. 实例

7.1. ECHO-server

7.1.1.什么是 ECHO-server

ECHO-server 是一个回显服务器，即对端发送什么，ECHO-server 就原原本本回送什么。

7.1.2.示例运行方式

可执行程序文件名为 echod，为单个可执行文件，可不带任何参数，也可指定一个端口号参数，如果不指定端口号，则默认端口号为 **2012**。

7.1.3.需要实现的接口

ECHO-server 共需要实现 3 个 MOOON-server 接口，分别为：
1) server::IConfig，对应的实现为 CConfigImpl，所在文件为 config_imp.h 和 config_imp.cpp

2) server::IFactory，对应的实现为 CFactoryImpl，所在文件为 factory_impl.h 和 factory_impl.cpp
3) server::IPacketHandler，对应的实现为 CPakcetHandlerImpl，所在文件为 packet_handler_impl.h 和 packet_handler_impl.cpp

7.1.4.所有文件

Makefile
main.cpp
config_impl.h
config_impl.cpp
factory_impl.h
factory_impl.cpp
packet_handler_impl.h
packet_handler_impl.cpp

7.1.5.config_imp.h

```
#include <server/server.h>

class CConfigImpl: public server::IConfig
{
public:
	bool init(uint16_t port);

private:
	virtual const net::ip_port_pair_array_t& get_listen_parameter() const;

private:
	net::ip_port_pair_array_t _ip_port_pair_array;
};
```

7.1.6.config_imp.cpp

```
#include <net/util.h>
#include "config_impl.h"

bool CConfigImpl::init(uint16_t port)
{
	try
	{
		net::eth_ip_array_t eth_ip_array;
```

```
            net::CUtil::get_ethx_ip(eth_ip_array);

            // 设置默认的监听端口
            if (0 == port)
            {
                port = 2012;
            }

            // 取得网卡上所有的 IP 地址
            for (int i=0; i<(int)eth_ip_array.size(); ++i)
            {
                net::ip_port_pair_t ip_port_pair;
                ip_port_pair.first = eth_ip_array[i].second.c_str();
                ip_port_pair.second = port;

                _ip_port_pair_array.push_back(ip_port_pair);
            }

            return !_ip_port_pair_array.empty();
        }
        catch (sys::CSyscallException& ex)
        {
            fprintf(stderr, "Get IP error: %s.\n", ex.to_string().c_str());
            return false;
        }
}

const net::ip_port_pair_array_t& CConfigImpl::get_listen_parameter() const
{
    return _ip_port_pair_array;
}
```

7.1.7.factory_impl.h

```
#include <server/server.h>

class CFactoryImpl: public server::IFactory
{
private:
    virtual server::IPacketHandler* create_packet_handler(server::IConnection* connection);
};
```

7.1.8.factory_impl.cpp

```
#include "factory_impl.h"
#include "packet_handler_impl.h"

server::IPacketHandler* CFactoryImpl::create_packet_handler(server::IConnection* connection)
{
	return new CPakcetHandlerImpl(connection);
}
```

7.1.9.packet_handler_impl.h

```
#include <server/server.h>

class CPakcetHandlerImpl: public server::IPacketHandler
{
public:
	CPakcetHandlerImpl(server::IConnection* connection);
	~CPakcetHandlerImpl();

private:
	virtual void reset();

	virtual char* get_request_buffer();
	virtual size_t get_request_size() const;
	virtual util::handle_result_t on_handle_request(size_t data_size, server::Indicator&
indicator);

	virtual const char* get_response_buffer() const;
	virtual size_t get_response_size() const;
	virtual size_t get_response_offset() const;
	virtual void move_response_offset(size_t offset);
	virtual util::handle_result_t on_response_completed(server::Indicator& indicator);

private:
	server::IConnection* _connection; // 建立的连接
	size_t _request_size;	// 接收请求缓冲区的最大字节数
	size_t _response_size;	// 需要发送的响应数据字节数
	size_t _response_offset; // 当前已经发送的响应数据字节数
	char* _request_buffer;	// 用来存放请求数据的缓冲区
};
```

7.1.10. packet_handler_impl.cpp

```
#include <sys/util.h>
#include "packet_handler_impl.h"

CPakcetHandlerImpl::CPakcetHandlerImpl(server::IConnection* connection)
	:_connection(connection)
{
	_response_size = 0;
	_response_offset = 0;
	_request_size = sys::CUtil::get_page_size();
	_request_buffer = new char[_request_size];
}

CPakcetHandlerImpl::~CPakcetHandlerImpl()
{
	delete []_request_buffer;
}

void CPakcetHandlerImpl::reset()
{
	_response_size = 0;
	_response_offset = 0;
}

char* CPakcetHandlerImpl::get_request_buffer()
{
	return _request_buffer;
}

size_t CPakcetHandlerImpl::get_request_size() const
{
	return _request_size;
}

util::handle_result_t CPakcetHandlerImpl::on_handle_request(size_t data_size, server::Indicator&
indicator)
{
	_response_size = data_size;
	return util::handle_finish; /** finish 表示请求已经接收完成，可进入响应过程 */
}

const char* CPakcetHandlerImpl::get_response_buffer() const
{
```

```
    return _request_buffer;
}

size_t CPakcetHandlerImpl::get_response_size() const
{
    return _response_size;
}

size_t CPakcetHandlerImpl::get_response_offset() const
{
    return _response_offset;
}

void CPakcetHandlerImpl::move_response_offset(size_t offset)
{
    _response_offset += offset;
}

util::handle_result_t CPakcetHandlerImpl::on_response_completed(server::Indicator& indicator)
{
    // 如果收到 quit 指令，则关闭连接
    return 0 == strncmp(_request_buffer, "quit", sizeof("quit")-1)
        ? util::handle_finish /** finish 表示可关闭连接 */
        : util::handle_continue; /** continue 表示连接保持，不要关闭 */
}
```

7.1.11. main.cpp

```
#include <util/string_util.h>
#include <sys/main_template.h>
#include "config_impl.h"
#include "factory_impl.h"

/***
  * 使用 main 函数模板，需要实现 sys::IMainHelper 接口
  */
class CMainHelper: public sys::IMainHelper
{
public:
    CMainHelper();

private:
    virtual bool init(int argc, char* argv[]);
    virtual void fini();
```

```
    virtual int get_exit_signal() const { return SIGUSR1; }

private:
    uint16_t get_listen_port(int argc, char* argv[]);

private:
    server::server_t _server;
    CConfigImpl _config_impl;
    CFactoryImpl _factory_impl;
};

/***
  * 可带一个端口号参数，也可不带任何参数，默认端口号为 2012
  */
int main(int argc, char* argv[])
{
    CMainHelper main_helper;
    return sys::main_template(&main_helper, argc, argv);
}

CMainHelper::CMainHelper()
    :_server(NULL)
{
}

bool CMainHelper::init(int argc, char* argv[])
{
    uint16_t port = get_listen_port(argc, argv);
    _config_impl.init(port);

    // 创建一个 MOOON-server 组件实例
    _server = server::create(&_config_impl, &_factory_impl);
    return _server != NULL;
}

void CMainHelper::fini()
{
    if (_server != NULL)
    {
        // 销毁 MOOON-server 组件实例
        server::destroy(_server);
        _server = NULL;
    }
}
```

```
uint16_t CMainHelper::get_listen_port(int argc, char* argv[])
{
	uint16_t port = 0;

	if (argc > 1)
	{
		(void)util::CStringUtil::string2int(argv[1], port);
	}

	return port;
}
```

7.1.12. Makefile

```
#
# 默认认为 mooon 安装在${HOME}/mooon 目录下，可根据实际进行修改
# 编译成功后，生成的可执行程序名为 echod，可带一个端口参数，
# 也可不带任何参数运行，默认端口号为 2012
#
MOOON=${HOME}/mooon
MOOON_LIB=$(MOOON)/lib/libserver.a $(MOOON)/lib/libnet.a $(MOOON)/lib/libsys.a $(MOOON)/lib/libutil.a
MOOON_INCLUDE=-I$(MOOON)/include

echod: *.cpp
	g++ -g -o $@ *.cpp -lrt -pthread $(MOOON_INCLUDE) $(MOOON_LIB)

clean:
	rm -f *.o
	rm -f echod
```