MOOON-dispatcher 组件使用指南

一见@2011.8/12

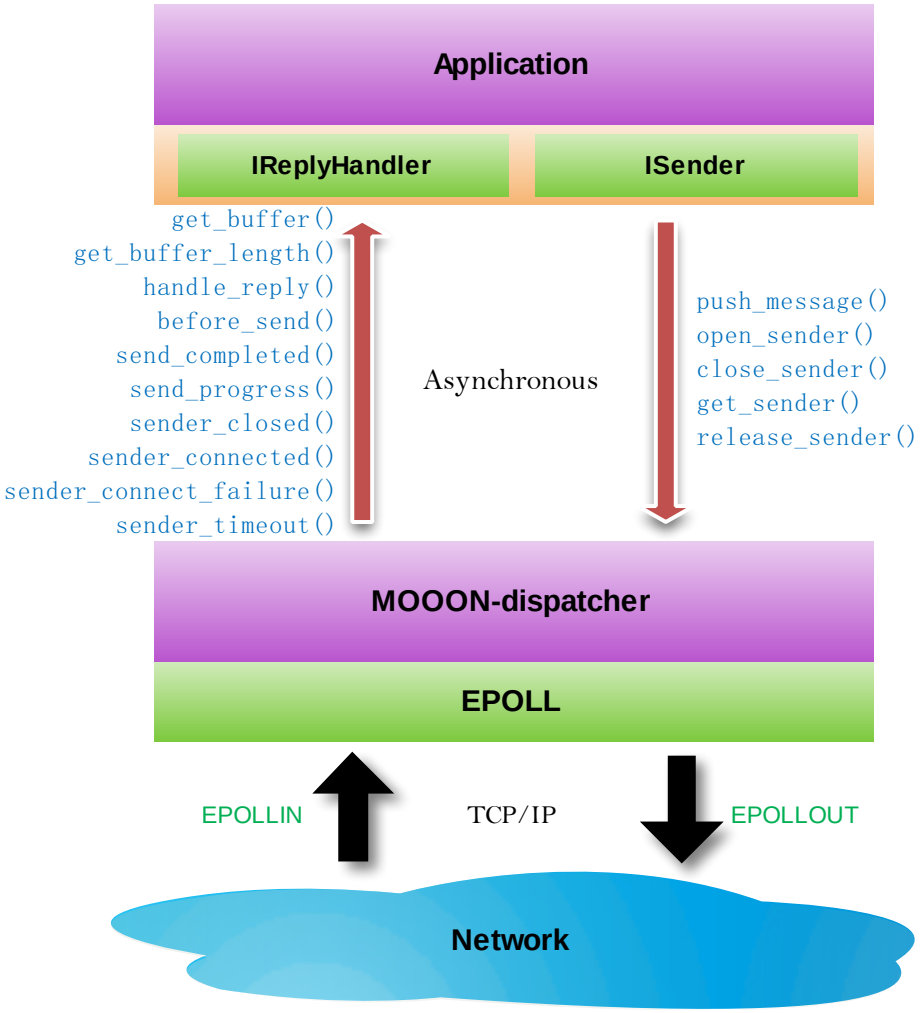
1. 介绍

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

2. 功能

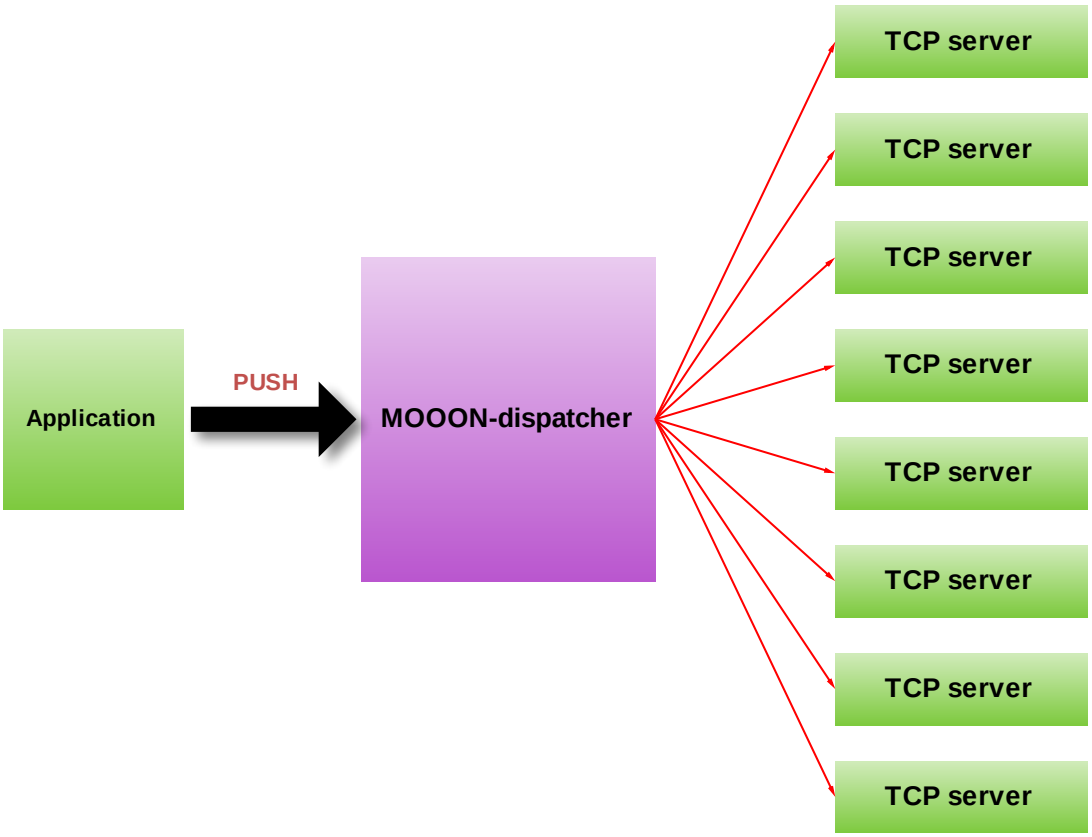
1) 异步收发数据
2) 异步发送文件
3) 长短连接控制
4) 连接超时控制
5) 自动重连接
6) 自动重发送
7) 多线程安全使用

3. 工作原理



MOOON-dispatcher 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-dispatcher 和 Application（应用）之间采用异步回调的方式进行交互。

4. 应用场景



通常情况下，使用 MOOON-dispatcher 来和多个 Server 同时通讯，它们之间所有操作都是异步的，包括异步连接，所以性能很高效。

5. 接口说明

MOOON-dispatcher 的名字空间名为：**dispatcher**。

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

5.1. 组件实例

5.1.1.create

创建 MOOON-dispatcher 组件实例，函数原型如下：

```
/***
  * 创建分发器
  * @thread_count 工作线程个数
  * @return 如果失败则返回 NULL，否则返回非 NULL
```

```
   */
extern IDispatcher* create(uint16_t thread_count);
```

5.1.2.destroy

销毁 MOOON-dispatcher 组件实例，函数原型如下：

```
extern void destroy(IDispatcher* dispatcher);
```

5.2. message

5.2.1.file_message_t

当需要通过 MOOON-dispatcher 发送一个文件时，使得该消息。

```
/***
   * 分发文件类型消息结构
   */
typedef struct
{
      int fd;                 /** 需要发送的文件描述符 */
      off_t offset;        /** 文件偏移，从文件哪个位置开始发送 */
}file_message_t;
```

5.2.2.buffer_message_t

```
/***
   * 分发 Buffer 类型消息结构
   */
typedef struct
{
      char data[0];        /** 需要发送的消息数据 */
}buffer_message_t;
```

5.2.3.create_file_message

创建文件类型的消息，函数原型如下：

```
extern file_message_t* create_file_message(size_t file_size);
```

5.2.4.create_buffer_message

创建 Buffer 类型的消息，函数原型如下：

extern buffer_message_t* create_buffer_message(size_t data_length);

5.3. destroy_file_message

销毁文件类型的消息，函数原型如下：
extern void destroy_file_message(file_message_t* file_messsage);

5.4. destroy_buffer_message

销毁 Buffer 类型的消息，函数原型如下：
extern void destroy_buffer_message(buffer_message_t* buffer_messsage);

5.5. IDispatcher

5.5.1.所在头文件

#include <dispatcher/ dispatcher.h>

5.5.2.接口说明

分发器接口。

5.5.3.接口定义

```
/***
   * 消息分发器接口
   */
class IDispatcher
{
public:
      // 虚析构用于应付编译器
      virtual ~IDispatcher() {}

      virtual IManagedSenderTable* get_managed_sender_table() = 0;
      virtual IUnmanagedSenderTable* get_unmanaged_sender_table() = 0;

      /** 得到发送的线程个数 */
      virtual uint16_t get_thread_number() const = 0;
};
```

5.6. SenderInfo

5.6.1.所在头文件

#include <dispatcher/ dispatcher.h>

5.6.2.接口说明

发送者信息结构，根据该结构来创建 Sender。

5.6.3.接口定义

```
/***
  * 用来创建 Sender 的信息结构
  */
struct SenderInfo
{
    uint16_t key;                    /** Sender 的键值，如果为 Unmanaged 类型的 Sender，则其值忽略 */
    net::ip_node_t ip_node;          /** 需要连接的 IP 节点，包含 IP 地址和端口信息 */
    uint32_t queue_size;             /** 缓存消息队列的大小，其值必须不小于 0 */
    int32_t resend_times;            /** 消息发送失败后自动重发送的次数，如果值小于 0，表示始终自动重发送，直接发送成功 */
    int32_t reconnect_times;         /** 连接断开后，可自动连接的次数，如果值小于 0，表示始终自动重连接，直接连接成功 */
    IReplyHandler* reply_handler; /** 允许为 NULL，如果为 NULL，则所有收到的应答数据丢弃，请注意在 Sender 被销毁时，reply_handler 会一同被删除 */
};
```

5.7. ISender

5.7.1.所在头文件

#include <dispatcher/ dispatcher.h>

5.7.2.接口说明

发送者接口，提供将消息 Push 进 MOOON-dispatcher 的功能。

5.7.3.接口定义

```
/***
  * 发送者接口
  */
class ISender
{
public:
    virtual ~ISender() {}

    /** 转换成可读的字符串信息 */
    virtual std::string str() const = 0;

    /** 得到 Sender 的信息结构 */
    virtual const SenderInfo& get_sender_info() const = 0;

    /***
      * 推送消息
      * @message: 需要推送的消息
      * @milliseconds: 等待推送超时毫秒数，如果为 0 表示不等待立即返回，否则
      *   等待消息可存入队列，直到超时返回
      * @return: 如果消息存入队列，则返回 true，否则返回 false
      */
    virtual bool push_message(file_message_t* message, uint32_t milliseconds=0) = 0;
    virtual bool push_message(buffer_message_t* message, uint32_t milliseconds=0) = 0;
};
```

5.8. ISenderTable

5.8.1.所在头文件

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

5.8.2.接口说明

发送者表基接口。

5.8.3.接口定义

```
class ISenderTable
{
```

```
public:
    virtual ~ISenderTable() {}

    /***
      * 创建一个 Managed 类型的 Sender，并对 Sender 引用计数增一
      * @sender_info 用来创建 Sender 的信息结构
      * @return 如果 Key 对应的 Sender 已经存在，则返回 NULL，否则返回指向新创建好的 Sender 指针
      */
    virtual ISender* open_sender(const SenderInfo& sender_info) = 0;

    /***
      * 关闭 Sender，并对 Sender 引用计数减一
      */
    virtual void close_sender(ISender* sender) = 0;

    /***
      * 对 Sender 引用计数减一
      */
    virtual void release_sender(ISender* sender) = 0;
};
```

5.9. IManagedSenderTable

5.9.1.所在头文件

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

5.9.2.接口说明

可管理的发送者表接口。

5.9.3.接口定义

```
class IManagedSenderTable: public ISenderTable
{
public:
    virtual ~IManagedSenderTable() {}

    /***
      * 获取 Sender，并对 Sender 引用计数增一
```

```
        * @return  如果 Sender 不存在，则返回 NULL，否则返回指向 Sender 的指针
        */
    virtual ISender* get_sender(uint16_t key) = 0;
};
```

5.10. IUnmanagedSenderTable

5.10.1. 所在头文件

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

5.10.2. 接口说明

不可管理的发送者表接口。

5.10.3. 接口定义

```
class IUnmanagedSenderTable: public ISenderTable
{
public:
    virtual ~IUnmanagedSenderTable() {}

    /***
        *  获取 Sender，并对 Sender 引用计数增一
        * @return  如果 Sender 不存在，则返回 NULL，否则返回指向 Sender 的指针
        */
    virtual ISender* get_sender(const net::ip_node_t& ip_node) = 0;
};
```

5.11. IReplyHandler

5.11.1. 所在头文件

```
#include <dispatcher/ reply_handler.h>
```

5.11.2. 接口说明

应答处理器接口。当收到对端发过来的数据后，就会调用该接口的相应方法去接收和处理数据。

5.11.3. 接口定义

```
/***
  * 应答消息处理器，每个 Sender 都会对应一个应答消息处理器
  */
class CALLBACK_INTERFACE IReplyHandler
{
public:
    // 虚析构用于应付编译器
    virtual ~IReplyHandler() {}

    /** 关联到所服务的 Sender */
    virtual void attach(ISender* sender) = 0;

    /** 得到存储应答消息的 buffer */
    virtual char* get_buffer() = 0;

    /** 得到存储应答消息的 buffer 大小 */
    virtual size_t get_buffer_length() const = 0;

    /***
      * 每一个消息被发送前调用
      * @sender: 发送者
      */
    virtual void before_send() {}

    /***
      * 当前消息已经成功发送完成
      * @sender: 发送者
      */
    virtual void send_completed() {}

    /***
      * 数据发送进度
      * @total 总的需要发送的字节数
      * @finished 总的已经发送的字节数
      * @current 当次发送出去的字节数
      */
    virtual void send_progress(size_t total, size_t finished, size_t current) {}

    /***
      * 和目标的连接断开
      * @sender: 发送者
      */
```

```
    virtual void sender_closed() {}

    /***
     * Sender 超时
     * @return  如果返回 true，则 Sender 将会被删除，否则不做处理
     */
    virtual bool sender_timeout() { return true; }

    /***
     *  和目标成功建立连接
     * @sender:  发送者
     */
    virtual void sender_connected() {}

    /***
     *  连接到目标失败
     * @sender:  发送者
     */
    virtual void sender_connect_failure() {}

    /***
     *  收到了应答数据，进行应答处理
     * @sender:  发送者
     * @data_size:  本次收到的数据字节数
     */
    virtual util::handle_result_t handle_reply(size_t data_size) { return util::handle_error; }

    /***
     *  得到状态值
     */
    virtual int get_state() const { return 0; }

    /***
     *  设置状态值
     */
    virtual void set_state(int state) {}
};
```

6. 实例

6.1. WEB-getter

6.1.1.功能说明

该实例实现向一个域名抓取指定网页的功能，如果指定的域名对应多个 IP，则会同时从所有 IP 抓取页面，抓取的结果分别存于不同的文件，文件名格式为：html_域名_Key_IP_端口_网页文件名，如：html_news.163.com_0_60.217.241.33_80_7B4S5OP50001121M.html。

6.1.2.使用方式

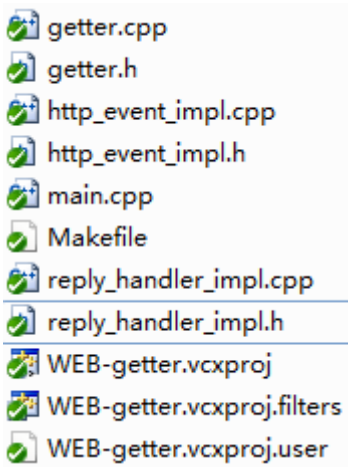
成功编译后，生成单个可执行文件 **web_getter**，使用方式如下：

./**web_getter** --dn=news.163.com --port=80 --url=/11/0811/00/7B4S5OP50001121M.html

1) --dn 参数的值为域名，不支持 IP
2) –port 端口号
3) –url 需要抓取的网页

6.1.3.文件组成

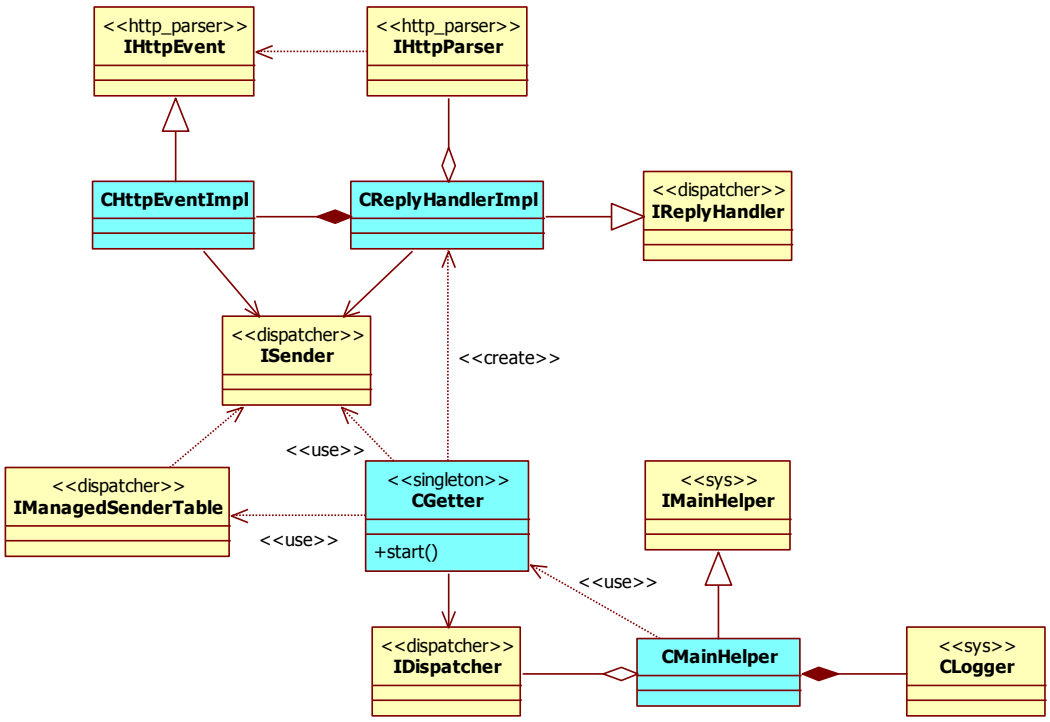
源代码目录下共包含以下几个文件：



对应的 SVN 地址为：

https://mooon.googlecode.com/svn/trunk/common_component/example/MOOON-dispatcher/WEB-getter，可用 SVN 下载或浏览源代码，下载后还可使用 **VC2010** 打开浏览。

6.1.4.类图结构



WEB-getter 本身的类包括：

CMainHelper，CGetter，CHttpEventImpl 和 CReplyHandlerImpl。

1) CMainHelper
   所在文件为 main.cpp。
2) CGetter
   所在文件为 getter.h 和 getter.cpp。
3) CHttpEventImpl
   所在文件为 http_event_impl.h 和 http_event_impl.cpp。
4) CReplyHandlerImpl
   所在文件为 reply_handler_impl.h 和 reply_handler_impl.cpp。

其中 CGetter 为主线类，控制流程的转动。CReplyHandlerImpl 借助 http-parser 的能力来处理来自 WEB server 的响应。