EasyNet-Framework 网络应用程序开发框架

刘勇进
xmulyj@gmail.com

目录

一、简介

"天下武功，为快不破"。

在当前的互联网中，很多公司都使用快速迭代、小步快跑的开发方式，以 2 周甚至更短的周期来不断推出自己的应用产品，为的是能够持续的吸引用户。这种快速开发的效果，对用户来说最直接的是体现在客户端上。然而现在的互联网产品很多都是需要通过网络与服务器通信的，通过后台服务器提供产品的功能。实际上，服务器端的开发实际上制约着互联网产品的开发速度。

在客户端开发中，比较经典的如 MFC，通过向导就能够快速创建一个应用程序的框架，开发者只需要在该框架中实现自己的业务逻辑便能很快开发出自己的应用产品。而对于后台服务的开发，特别是网络应用程序的开发，到目前为止并没有一套完整实用的类似客户端开发的框架可以支持服务端的快速开发。网络服务器的开发，需要从监听端口、绑定 socket、接收链接、监听链接上的数据请求、业务逻辑层处理数据、回复数据等一步一步设计编写程序。实际上开发者关心的只有业务逻辑层这部分的开发，但是却需要花费大量的时间和精力来处理底层的网络数据的接发送，严重制约开发的速度。

本文介绍 Easyet-Framework 框架来提高服务器端的开发速度。

EasyNet-Framework 是一个基于 Linux 的 C/C++网络应用程序（服务器）开发框架，能够让开发者从底层的网络编程细节中解脱出来，从而能够把精力集中于业务逻辑层的开发。其对网络层（如 socket、IO 多路复用等）、链接的创建和接收、协议数据的序列化和反序列化、以及数据的发送和接收进行了封装并进行了合理的组织，为应用层提供了统一的调用接口，从而为网络应用程序的开发提供了快速高效的方法。同时框架提供了 easynet 工具，可以用来生成应用程序的框架代码。开发者可以在生成的服务器框架类代码中实现框架的接口方法以及实现自己的业务逻辑代码从而完成应用产品的开发。

文本的组织结构如下：

第二章和第三章介绍了框架的安装和使用。

第四章介绍了框架的基本原理和工作流程，包括连接的接收和管理、数据的接收、数据的发送。

第五章对核心模块做了详细的介绍，包括对各种数据格式（结构化的二进制协议和文本数据）的支持。

第六章介绍了框架的扩展能力，包括对多线程的支持和内存优化方面。

第七章介绍使用框架搭建的单线程和多线程的 HTTP 服务器例子。

第八章列出了框架的主要组建及相关的介绍。

二、框架安装

下载 EasyNet-Framework 的源码并解压:tar zxvf EasyNetFramework.tar.gz,EasyNet 的目录结构如下:

```
EasyNetFramework
    |---doc                 //本文档所在目录
    |---eaxmples            //框架提供的所有例子的目录
```

```
|---src                    //框架源代码
    |---common
    |---framework
    |---lib                //编译后生成框架库的目录
    |---objects            //编译时生成临时文件的目录
    |---tools              //框架代码生成的工具
```

进入 src 子目录,在命令行中输入 make 编译代码, make install 安装框架（可能需要 root 权限)

三、框架的使用

安装好 EasyNet 后，可以使用 easynet 工具来辅助生成框架代码。当然用户可以自己手动编写相应的代码，但是比较麻烦一点。

3.1 获取帮助

在命令行中直接输入 easynet：

```
easynet -i | -p | -a ClassName [-m]
    -i : Init environment.
    -p : Generate ProtocolFactory Class.
    -a : Generate Application Instance Class.
    -m : Optional. Generate server main framework.
```

3.2 初始化目录

在命令行中输入 :easynet -i将在当前目录中创建框架需要的一些子目录和相应的配置文件 :其中 bin 用来存放生成可执行文件，config 用来存放配置文件，log 用来存放日志文件；log4cplus.conf 是 log4cplus 的配置文件。

```
[tim@localhost test_dir]$ easynet -i
Init environment ...
create dir ./bin ...
create dir ./config ...
create dir ./log ...
copy log config to ./config/log4cplus.conf ...
Init environment ... [OK]
[tim@localhost test_dir]$ ll
total 12
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 bin
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 config
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 log
[tim@localhost test_dir]$
```

图 3.1 目录初始化

3.3 生成代码框架

在命令行输入：easynet -a TestServer -m 将生成应用程序框架代码类。其中-m 参数用来生成应用程序运行的主文件，其中包含 main 函数调用TestServer的实例代码。生成应用程序框架之后需要实现其中的接口方法，详细见框架代码。

```
[tim@localhost test_dir]$ easynet -a TestServer -m
Generate Application instance class file: TestServer.h ...
Generate Application instance class file: TestServer.cpp ...
Generate ApplicationMain class file: TestServerMain.cpp ...

Compile Using:
        -I/usr/include/easynet -leasynet -lpthread -llog4cplus

See temp makefile: Makefile.tmp
[tim@localhost test_dir]$ ll
total 28
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 bin
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 config
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 log
-rw-rw-r--. 1 tim tim   68 Jul  5 11:10 Makefile.tmp
-rw-r--r--. 1 tim tim 1318 Jul  5 11:10 TestServer.cpp
-rw-r--r--. 1 tim tim 2555 Jul  5 11:10 TestServer.h
-rw-r--r--. 1 tim tim  246 Jul  5 11:10 TestServerMain.cpp
```

图 3.2 生成主代码框架

3.4 生成协议工厂

框架默认使用来 KVDataProtocolFactory 协议工厂来控制数据的序列化和反序列化，如果应用程序需要创建自己的协议格式的话，可以使用-p 参数来生成协议工厂类，并实现自己的协议格式。使用时需要在框架类中创建自己的协议工厂实例并重写框架的GetProtocolFactory 方法，返回自己的协议工厂实例：

```
[tim@localhost test_dir]$ easynet -p TestProtocolFactory
Generate ProtocolFactory class file:
        TestProtocolFactory.h ...
        TestProtocolFactory.cpp ...
Generate ProtocolFactory ... [OK]
[tim@localhost test_dir]$ ll
total 36
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 bin
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 config
drwxrwxr-x. 2 tim tim 4096 Jul  5 11:04 log
-rw-rw-r--. 1 tim tim   68 Jul  5 11:10 Makefile.tmp
-rw-r--r--. 1 tim tim  833 Jul  5 11:17 TestProtocolFactory.cpp
-rw-r--r--. 1 tim tim 1111 Jul  5 11:17 TestProtocolFactory.h
-rw-r--r--. 1 tim tim 1318 Jul  5 11:10 TestServer.cpp
-rw-r--r--. 1 tim tim 2555 Jul  5 11:10 TestServer.h
-rw-r--r--. 1 tim tim  246 Jul  5 11:10 TestServerMain.cpp
```

图 3.3 生成协议工厂

具体怎么使用使用框架和协议工厂，可以参考 HttpEchoServer 例子。

四、框架原理

框架使用 EventServer 来监控所有 socket fd 的事件（如可读、可写、超时、错误、关闭连接等）。并将收集到的事件分派给注册的 EventHandler 事件处理接口类。另外 EventServer 还提供了时钟功能，当超过注册的超时时间时，将触发时钟的超时事件。

下面先从链接的接收、数据的接收、数据的发送三个方面介绍框架的实现原理。其中对数据序列化和反序列化的介绍见第五部的 IprotocolFactory 模块一节。

4.1 接收链接

链接的接收流程如图 4.1 所示：

（1）AppInterface 实例通过调用 Listen 方法，监听指定的端口，并调用 EventServer 的 SetEvent 方法向 EventServer 注册 listen_fd 的可读事件。其中 *ET_PER_RD* 表示持续可读事件，listen_handler 为事件处理接口类，idle_timeout 为链接空闲超时时间，当链接超过该时间未发生时间时将发生超时事件。监听链接一般设置为-1 表示永不超时。

（2）EventServer 的 DispatchEvents 方法将会收集注册到其上的 socket 的所有事件，并调用对应的 EventHandler 处理事件。当 Client 向服务器发送 connect 请求时，将触发 listen_fd 上的可读事件，DispatchEvents 调用 ListenHandler 的 OnEventRead 接口响应该可读事件。

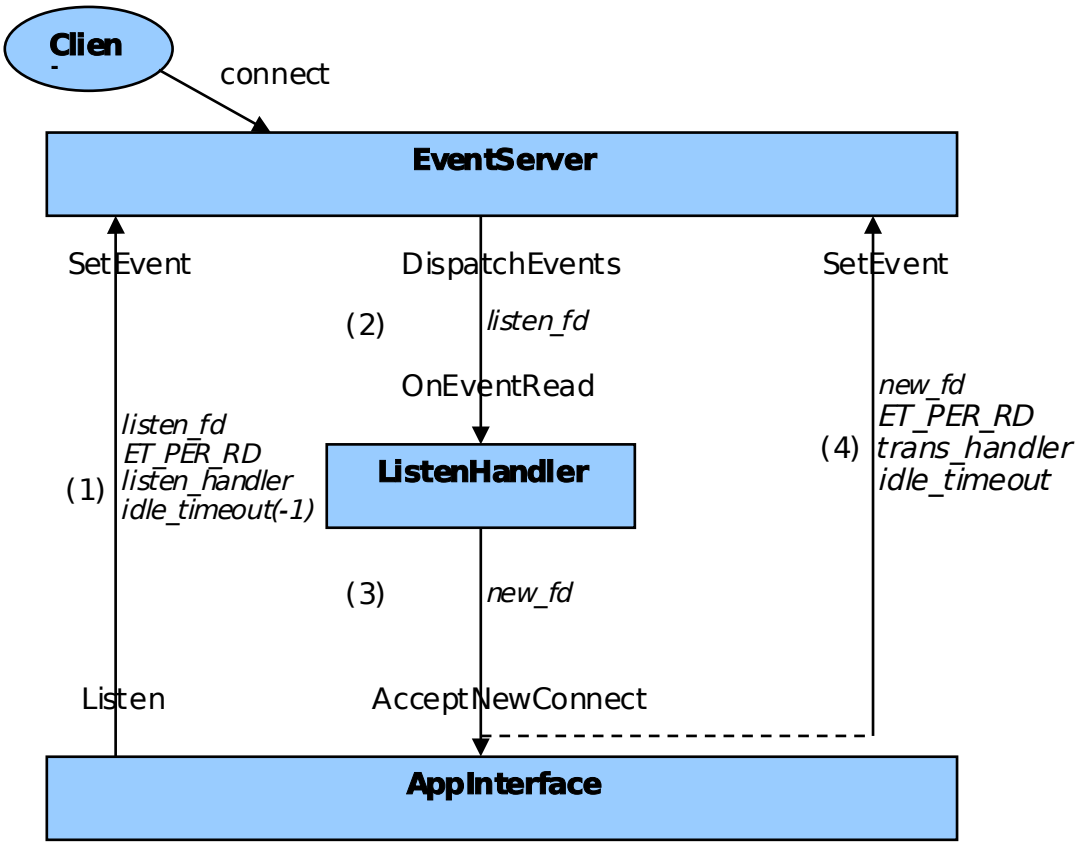

图 4.1 链接接收流程

（3）OnEventRead 接口调用 AppInterface 的 AcceptNewConnect 接口将接收到的 client 的链接 new_fd 传递给 AppInterface。

（4）AcceptNewConnect 将收到的 new_fd 注册到 EventServer。其中 trans_handler 为 new_fd 的事件处理接口类。

4.2 接收数据

数据的接收过程如图 4.2 所示：

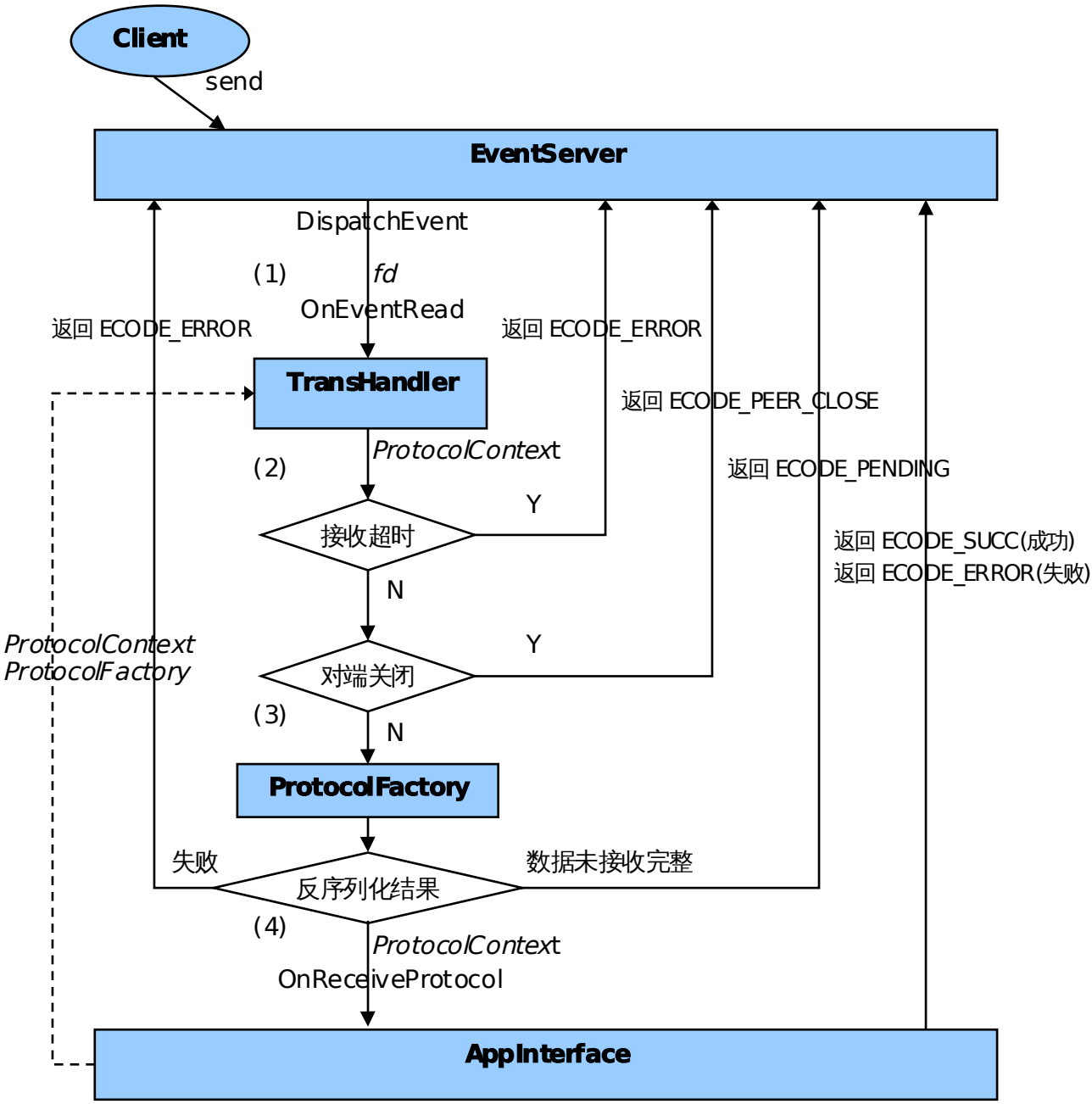

图 4.2 数据接收流程

（1）当 Client 发送数据时（send），EventServer 触发可读事件并调用 TransHandler 的 OnEventRead 接口分派事件。

（2）OnEventRead 接口首先判断 fd 上是否存在对应的 protocolcontext，如果没有则通过 AppInterface 创建一个（表示读数据的开始部分），如果有则直接使用该 protocolcotext

（表示之前发生过可读事件，但是数据未接收完整）。然后检查是否接收数据超时（当上次读取数据未完整，返回 ECODE_PENDING错误码，protocol_context处于等待数据状态），如果超时返回 ECODE_ERROR 错误码。如果未超时则继续接收数据，如果发现对端关闭掉链接，返回 ECODE_PEER_CLOSE 错误码。

（3）当接收到数据并保存到 protocolcontext 后，调用ProtocolFactory 对数据进行反序列化。如果数据有问题，反序列化失败返回 ECODE_ERROR 错误码；如果数据未完整接收（对端未发送完整）返回 ECODE_PENDING 错误码，等待下次可读事件触发时继续接收数据；如果反序列化成功，将调用 AppInterface 的 OnReceiveProtocl 接口，该接口需要由应用程序实现，对 protocolcontext 进行处理，完成实际的业务逻辑。

（4）最后，根据应用程序返回的值，返回相应的 ECODE_SUCC 或者 ECODE_ERROR，表示处理成功或失败。

4.3 发送数据

数据的发送过程如图 4.3 所示：

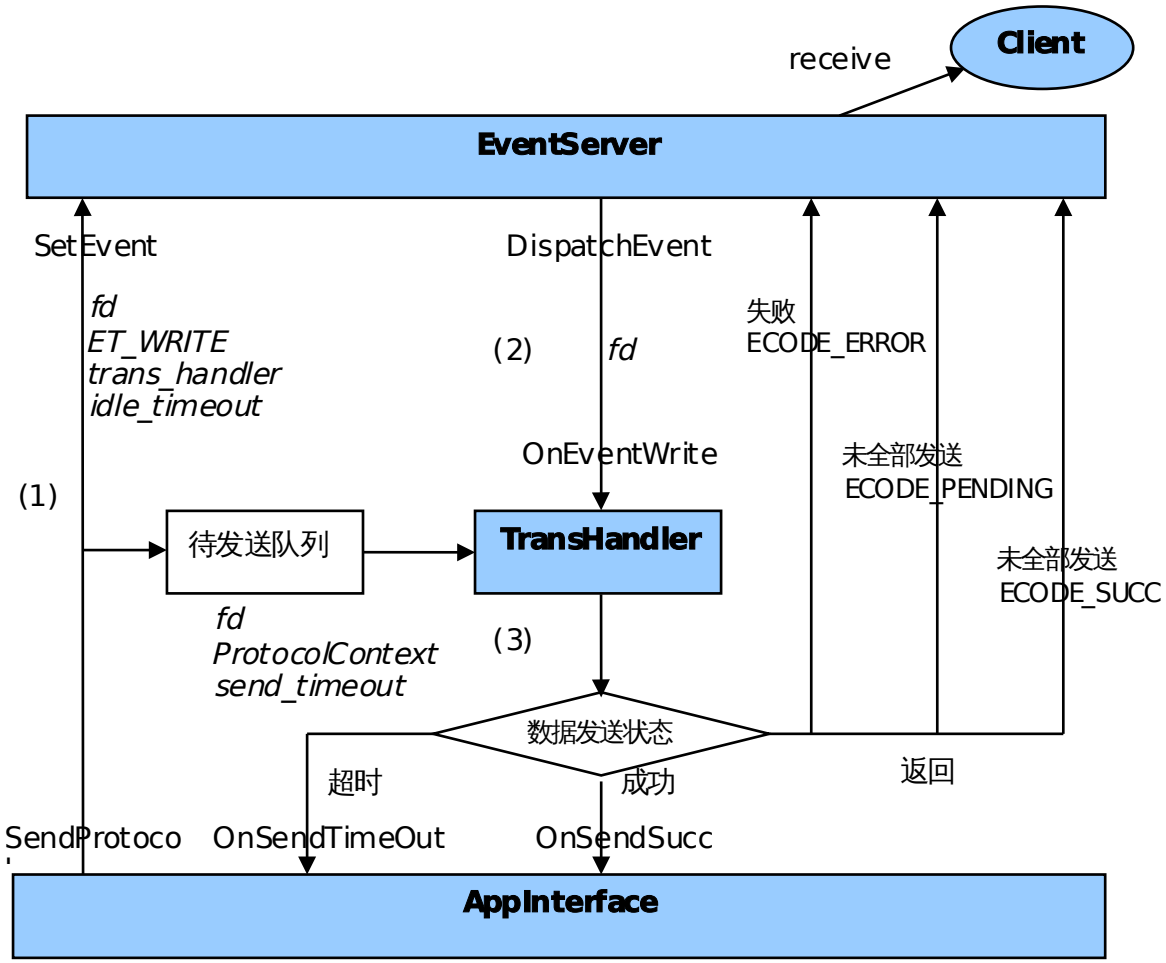

图 4.3 数据发送流程

（1）当应用程序通过调用 AppInterface 的 SendProtocol 方法发送数据。SendProtocol 将待发送的数据 protocol_context 添加到待发送队列，并设置其发送的超时时间

send_timeout，当超过该时间数据未发送完毕，将发生超时事件；同时向 EventServer 注册可写事件 ET_WRITE、链接空闲超时时间 idletimeout、事件处理接口类 trans_handler。

（2）当 socket fd 可写时，将触发可写事件。EventServer 的 DispatchEvents 方法通过调用 TransHandler 的 OnEventWrite 接口处理该事件。

（3）OnEventWrite 接口首先判断在 fd 上是否有未发送完成的数据 ProtocolContext，如果有则继续发送该数据；如果没有则从 AppInterface 的待发送队列获取一个新的待发送数据进行发送。在发送数据时：

a. 先判断数据是否发送超时，如果超时则调用 AppInterface 的 OnSendTimeout 接口（应用层需要实现该接口方法，处理数据的超时事件）；

b. 如果数据发送成功则调用 AppInterface 的 OnSendSucc 接口（应用层需要实现该接口方法，处理发送完成的数据）；

c. 如果数据未完全发送则返回 ECODE_PENDING 错误码，等待下次可写事件发生继续发送数据；

d. 如果数据发送失败（比如对端已经关闭链接等），则返回 ECODE_ERROR，由 EventServer 触发错误事件调用 EventHandler 的 OnEventError 接口来处理错误事件。

e. 当数据发送完成后返回 ECODE_SUCC 表示本轮处理正常。

五、框架模块介绍

EasyNet 框架由两大部分组成：一部分是 common 库，包含一些基础类，比如 ByteBuffer、Socket、Thread、Heap、IMemory、EventServer 等；一部分是 framework 库，是 EasyNet 的框架组织，包含 IAppInterface 应用实例接口类、ListenHandler 监听端口事件处理接口、TransHandler 数据传输事件处理接口、IProtocolFactory 协议数据工厂及框架提供的默认 KVDataProtocolFactory Key-Value 键值协议格式工厂 5 个模块。

5.1 EventServer 模块

EventServer 模块包含在 common 库中，是对 I/O 多路复用的一个抽象，除了支持监听 socket 的事件外，还提供了时钟超时功能。

5.1.1 I/O 事件触发

1. EventServer 定义了如下基本 I/O 事件类型 EventType：

（1）ET_READ：单次可读事件，当完成读事件时，EventServer 将移除该事件的注册。如果该事件与 ET_PERSIST 组合时，则在完成读事件后，不移除该事件。

（2）ET_WRITE:单次可读事件，当完成写事件时，EventServer 将移除该事件的注册。

（3）ET_PERSIST：永久性标记。对 ET_READ 事件有效。

2. EventServer 定义的事件处理接口类 IEventHandler 主要有如下几个接口方法：

```
virtual void OnTimeout(uint64_t nowtime_ms)=0;//时钟超时
virtual void OnEventError(int32_t fd, uint64_t nowtime_ms, ERROR_CODE code)=0;//错误事件
```

```
virtual ERROR_CODE OnEventRead(int32_t fd, uint64_t nowtime_ms)=0;//可读事件
virtual ERROR_CODE OnEventWrite(int32_t fd, uint64_t nowtime_ms)=0;//可写事件
```

3. ERROR_CODE 的取值如下：
   （1）ECODE_TIMEOUT：超时；
   （2）ECODE_PEER_CLOSE：对端关闭；
   （3）ECODE_ACTIVE_CLOSE：主动关闭；
   （4）ECODE_ERROR：错误；
   （5）ECODE_PENDING：读/写数据未完整；
   （6）ECODE_SUCC：成功；
4. EventServer 的 I/O 事件触发功能如图 5.1 所示：
   （1）应用程序 App 调用 EventServer 的 SetEvent 方法将事件、事件处理接口类、空闲超时时间注册到 EventServer 中。
   （2）EventServer 的 DispathchEvents 方法先收集时钟和 I/O 的超时事件，如果有 I/O 超时事件产生，则移除其注册的所有事件。
   （3）接着收集并处理 I/O 事件：
      a. 如果 socket 产生错误，则调用 DeleteEvent 方法移除所有注册的事件，并调用 IEventHandler 的 OnEventError 接口处理错误事件。
      b. 如果没产生错误并且产生可读事件，则调用 IEventHandler 的 OnEventRead 接口处理可读事件。如果返回 ECODE_ERROR（表示产生错误）或者 ECODE_PEER_CLOSE（表示对端关闭链接）错误码，则调用 DeleteEvent 方法移除所有注册的事件，并调用 OnEventError 处理错误事件；如果返回 ECODE_PENDING（表示数据未读完整）错误码，保留可读事件继续等待下次读事件发生并读取剩下的数据；如果返回 ECODE_SUCC 表示处理成功，此时如果注册的时候没有设置 ET_PERSIST 持续性标记，则调用 DeleteEvent 方法移除可读事件，此后不再监听该 socket 的可读事件除非重新注册。
      c. 如果没产生错误并且发生可写事件，则调用 IEventHandler 的 OnEventWrite 接口处理可写事件。 如果返回 ECODE_ERROR（表示产生错误）或者 ECODE_PEER_CLOSE（表示对端关闭链接）错误码，则调用 DeleteEvent 方法移除所有注册的事件，并调用 OnEventError 处理错误事件；如果返回 ECODE_PENDING（表示数据未发送完整）错误码，保留可写事件继续等待下次可写事件发生并发送剩下的数据；如果返回 ECODE_SUCC 表示处理成功。
      d. 处理完 socket 的 I/O 事件后，需要更新 socket 的超时时间，主要是根据设置的超时时间重新更新超时的时间点。时钟及 IO 的超时事件主要由最小堆来维护。
   （4）处理完 I/O 事件，接着处理之前收集的超时事件。如果是 I/O 超时，则调用 OnEventError 处理 ET_TIMEOUT 超时事件；如果是时钟超时，则调用 OnTimeout 接口处理时钟的超时事件，返回后如果注册的时钟事件设置了 ET_PERSIST 持续性标记，则重新设置时钟的下次超时事件，否则删除时钟。

5.1.2 时钟管理

时钟管理由最小堆来实现，主要用来管理时钟和 IO 的超时事件。EventServer 分派事件时，从最小堆中取出超时时间点小于/等于当前时间的对象，并存放到队列中，等待处理完 IO 事件后再处理超时

事件。之所以不立即处理超时事件，主要是考虑优先处理 IO 事件。可参考 5.1.1 节的介绍。

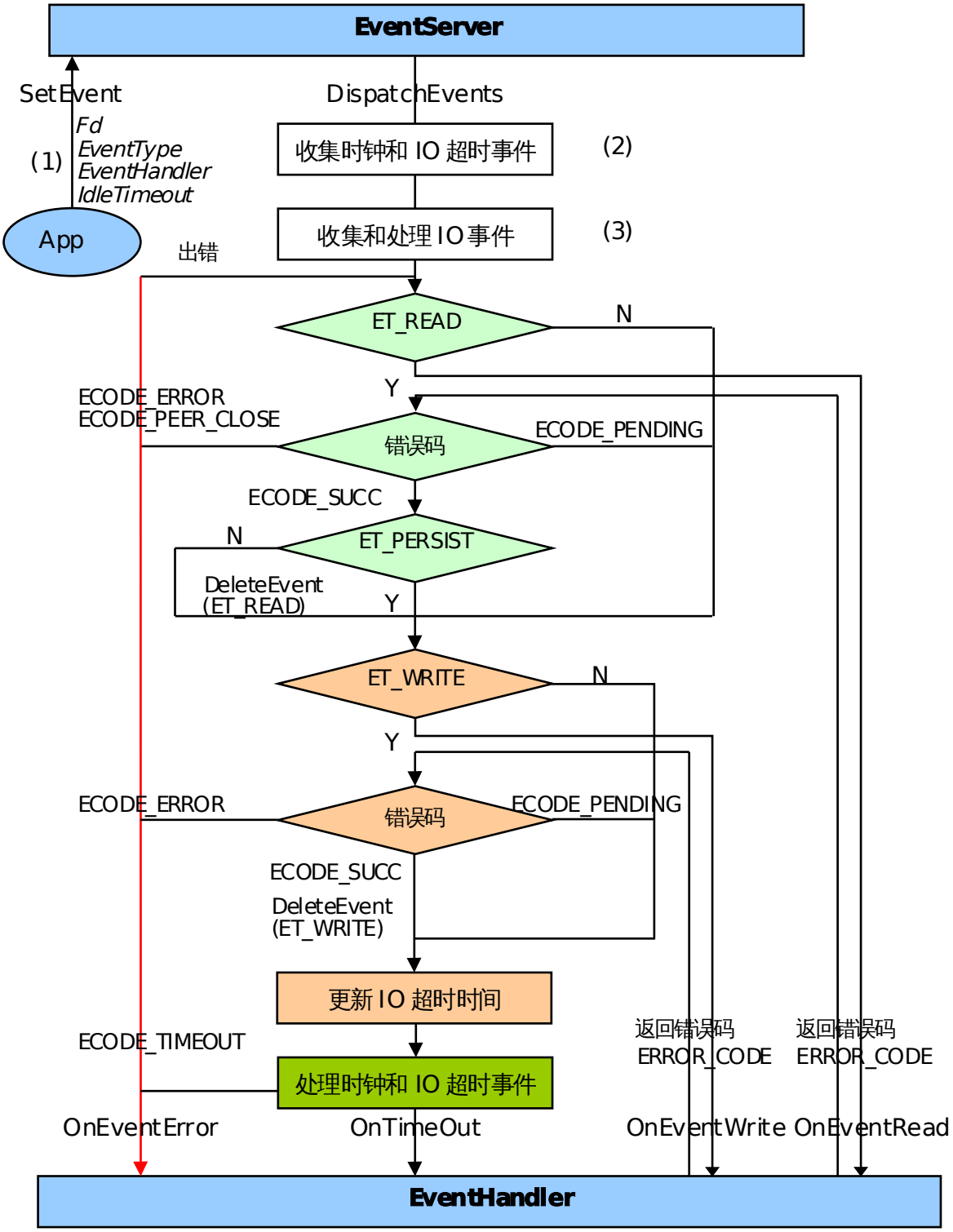

图 5.1 EventServer 事件处理流程

5.1.3 EventServer 的实现

EventServerEpoll 是 EventServer 的一个实现，其使用 epoll 来进行 I/O 多路复用。

5.2 IAppInterface 模块

IAppInterface 是 Easyet 框架的组织者，将各个模块有机的整合起来，完成网络的链接、数据的接收、数据的发送等基础逻辑；同时也是应用程序的抽象，其为应用程序定义来一系列的接口方法，应用程序通过继承本接口类并实现这些接口方法实现自己的业务逻辑。

IAppInterface 包含了 EventServer、EventHandler(ListenHandler 和 TransHandler）、IProtocolFactoy、IMemory 等模块。在第四部分框架原理中已经介绍过框架接收链接请求、数据接收和发送等流程，展示了 EventServer 和 EventHandler 如何触发响应事件，并最后调用 IAppInterface 框架接口的，然而该部分只是对正常流程的介绍，未提及对错误事件的处理。本节将对错误事件的处理进行描述，过程如图 5.2 所示。

（1）当事件处理出错（错误、对端关闭、链接空闲超时）时，EventHandler 的 OnEventError 接口方法被 EventServer 调用。

（2）OnEventError 在处理出错事件时，调用框架的 DeleteProtocolContext 方法删除正在接收的数据（如果有的话）。

（3）调用框架的 OnSendError 接口处理正在发送的数据和在等待队列中的数据（如果有的话）。之所以需要该接口方法，是因为只有应用程序知道怎么处理这些数据，而框架不知道。

（4）最后调用 OnSocketFinished 接口通知应用程序该链接已经完成所有任务。应用程序一般需要关闭该链接。

（5）另外，应用程序在处理任务时，可能通过调用 NotifySocketFinish 方法来主动结束链接的使用，该方法首先调用 DeleteEvent 方法删除注册到 EventServer 中的监听事件，然后调用 OnEventError 来处理该结束事件（ECODE_ACTIVE_CLOSE）。

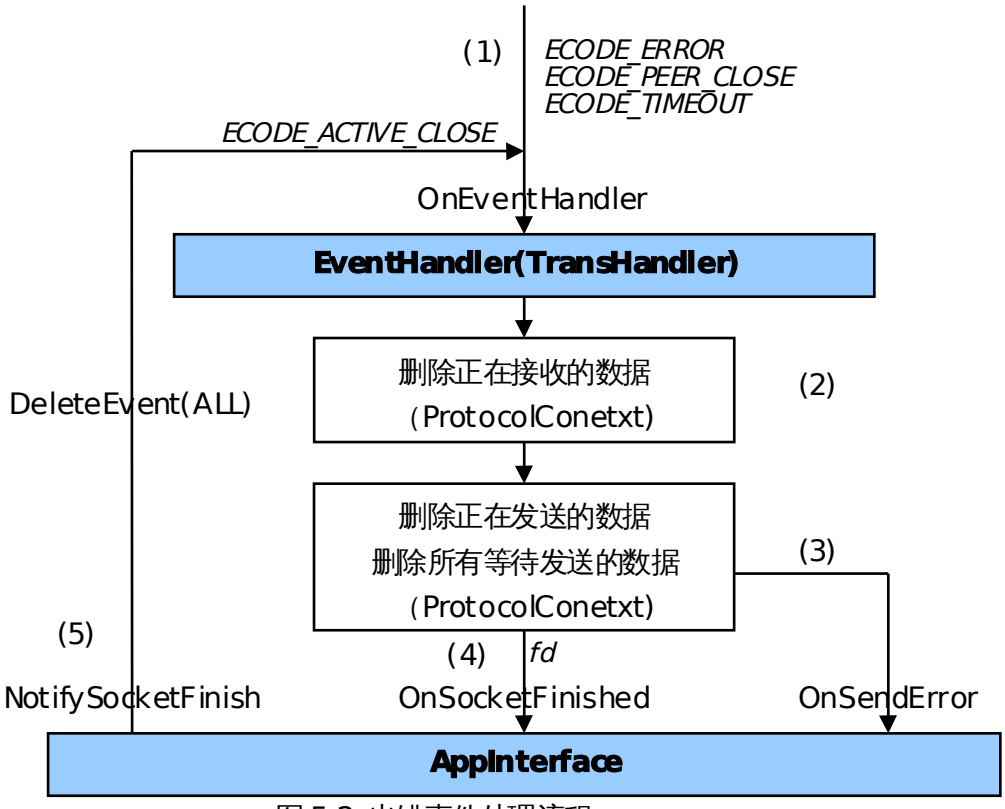

图 5.2 出错事件处理流程

5.3 IprotocolFactory 模块

5.3.1 数据类型

应用程序对数据的处理大体可以分为数据的接收、数据的处理、数据的发送 3 个部分。框架需要尽量完成数据的接收和发送这两部分的工作（同事还需要处理超时、错误等），从而让应用程序集中在对数据的处理上。这也是整个框架设计的目标。另外，我们将应用程序处理的数据格式概括为两大类：

（1）二进制协议：指在传输过程中包含包头和包体两部分数据。其中包头有固定长度，并指定包体长度（或整个数据包的长度），这样在读数据时能够指定读多少字节能够读到完整的数据包。这类格式的数据在连续发送时能够正确的分包。

（2）文本协议；指在传输过程中没有明显进行编码的数据，读数据的时候无法指定读多少数据才算读完一个完整的数据，而是需要不断读数据直到某些特殊字节才结束，比如 HTTP 协议、以 NULL 结尾的字符串等等。

框架应该同时支持这两种类数据的收发才能满足应用程序的需求。

为了将数据和应用程序的逻辑解耦出来，框架提供了 IProtocolFactory 接口类，用来对数据进行序列化和反序列化。应用程序通过继承该类实现自己的数据格式(二进制、文本或者同时支持两种格式)，并通过 IAppInterface 的 GetProtocolFactory 接口返回给框架使用。另外 IProtocolFactory 还定义了 ProtocolContext 类，用来接收和发送数据。ProtocolContext 继承 ByteBuffer 类，能够方便的管理内存的分配和释放。通过提供自己的 IMemory 类实现，特别是对于小内存的分配和释放，还能够优化内存的管理，避免内存碎片从而提高系统的性能。

5.3.2 使用 IProtocolFactory

IProtocolFactory 在框架中的使用过程如图 5.3 所示：

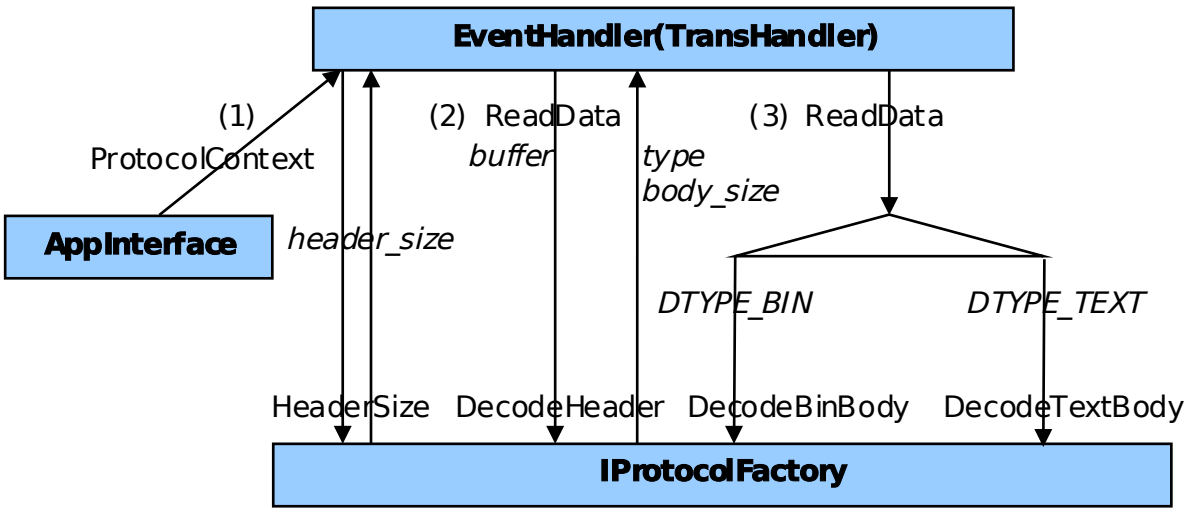

图 5.3 数据反序列化流程

（1）开始读数据时，IAppInterface 分配一个 protocol_context 并初始化为 DTYPE_INVALID 表示未知数据格式，同时调用 IProtocolFactory 的 HeaderSize 接口获取能够区别二进制和文本协议的最小头部长度 header_size。

（2）如果数据格式未知，则读取 header_size 个字节的头部数据并调用 IProtocolFactory 的 DecodeHeader 对头部数据进行解码，获取数据的格式类型 type 和协议体长度 body_size；如果数据不够返回 ECODE_PENDING 错误码，等待下次可读事件发生继续处理。

（3）如果数据格式已知，尽量读取 body_size 个字节，根据 type 的类型进行解码协议体数据。如果 type 是 DTYPE_BIN 二进制数据，调用 DecodeBinBody 解码二进制数据；如果 type 是 DTYPE_TEXT 文本数据，调用 DecodeTextBody 解码文本数据；如果数据不够返回 ECODE_PENDING 错误码，等待下次可读事件发生继续处理。

六、框架的扩展

6.1 多线程

到目前为止，我们对框架的介绍都只是在单线程的背景下进行的，基于单线程的应用程序在现实的开发中使用有限，价值较小。因此具有多线程的能力更为重要，更有现实意义。

EasyNet 框架的核心思想是以 EventServer 为基础，通过往其注册事件处理类 IEventHandler 来处理我们关心的 I/O、时钟等事件。而 EventServer 目前的实现并不具备多线程安全的能力，实际上也不大需要这样的能力，因为在需要 EventServer 的时候我们可以单独为每个线程创建自己的 EventServer 实例。

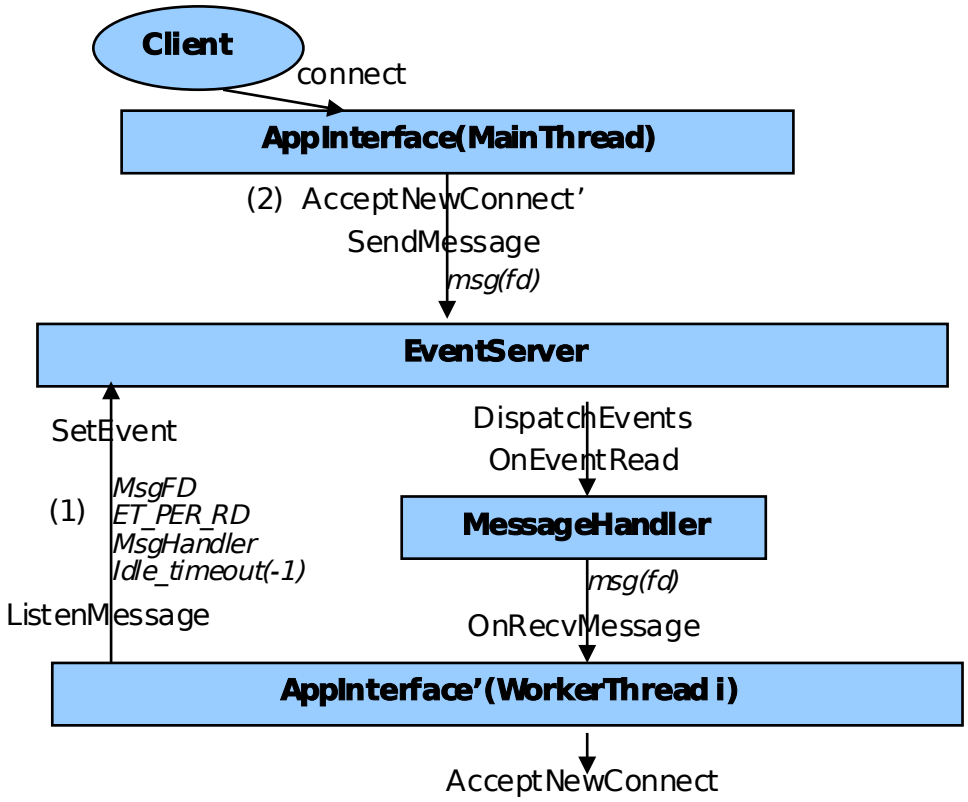

图 6.1 链接分发流程

框架提供了相应的接口，以便更方便更快捷地支持多线程，比如将新的链接 socket fd 分配到某个线程并由该线程独立处理该链接上的读写事件（实际上也可以由应用程序自己来处理）。具体见图 6.1：

（1）每个工作线程 WorkerThread（AppInterface'的实例）调用 ListenMessage 方法注册监听消息事件；

（2）Client 请求链接，主线程（AppInterface 的实例）接收到链接，经过自己的 EventServer 触发监听事件并分派事件，最后调用 AcceptNewConnect'方法（具体见图 4.1 链接接收流程）。应用程序重写该方法，调用工作 WorkerThread（AppInterface'的实例）的 SendMessage 方法，将新的链接 fd 分配给该工作线程；

（3）工作线程的 EventServer 触发事件并调用 MessageHandler 的 OnEventRead 接口处理事件；

（4）OnEventRead 调用 AppInterface'实例的 OnRecvMessage 接口接收分配到的链接 fd，其默认实现是调用 AcceptNewConnect 接口将 fd 注册到 EventServer 准备接收/发送数据（具体见图 4.1）；

具体的例子见 7.2 节的多线程 Http 服务器；

6.2 内存池

内存的操作(分配和释放) 在程序中是一件和平常又很频繁的事情，一般都是由系统或者库来完成，开发者直接调用 API 就可以了，并不用太过关心内存的管理。但是在开发服务器端时，有一个问题值得特别注意：内存碎片。如果在程序中频繁的分配和释放小内存的话，有可能导致内存碎片，严重影响系统/程序的性能。在框架这一层，由于面向的是各种应用程序的实例，也即是面临各种大小的内存分配，因此更加需要避免出现内存碎片的产生。

为了避免内存碎片的产生，EasyNet 提供了一个 IMemory 内存管理的实现版本：MemoryPool（内存池）。 MemoryPool 的基本原理是：

（1）通过参数设置来控制是否线程安全；

（2）按内存大小分为若干个内存区间，每个区间维护一定数量的内存槽（slab)；

（3）根据请求分配的内存大小找到对应的内存区间，如果该区间的空闲列表中有空闲的内存块，直接返回空闲块；如果没有则从该区间的 slab 中获取内存；

（4）释放内存时，将待释放的内存保存到对应区间的空闲列表中；

MemoryPool 的结构图如 6.2：

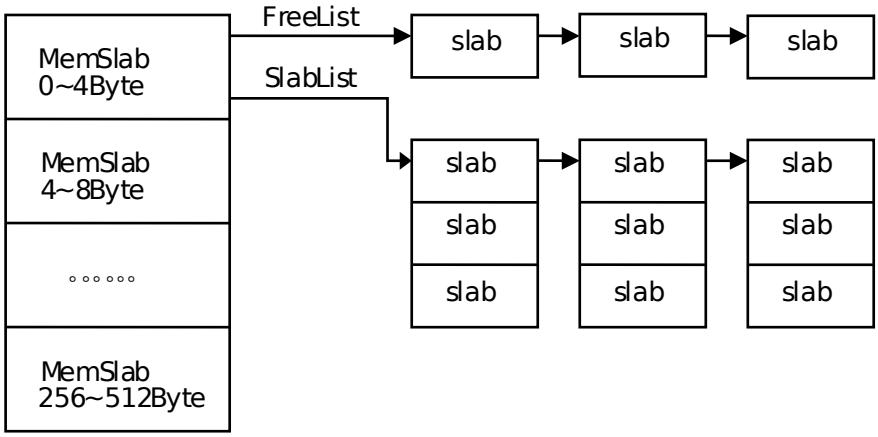

图 6.2 MemoryPool 结构

七、例子

本节我们介绍如果利用框架快速开发应用程序的例子。首先我们介绍一个简易的 HTTP 服务器，该服务器只接受 GET 请求，然后把相应的 html 文件发送给请求者。所有例子的代码见 examples 子目录。

7.1 简易 HTTP Server

7.1.1 生成框架代码

按顺序执行下面的命令生成框架代码：

（1）mkdir HttpEchoServer；cd HttpEchoServer

（2）easynet -i

（3）easynet -a HttpEchoServer -m

（4）easynet -p HttpReqProtocolFactory

第 1 步创建目录并进入到目录 HttpEchoServer 中；

第 2 步初始化目录结构，创建了 bin，conf，log 等子目录和 log 的配置文件等；

第 3 步生成框架代码 HttpEchoServer.h，HttpEchoServer.cpp HttpEchoServerMain.cpp；

第 4 步生成协议代码 HttpReqProtocolFactory.h，HttpReqProtocolFactory.cpp

自此所有的框架代码已经生成，接下来只要集中精力实现我们的 Http 服务器应用程序的具体业务。

7.1.2 实现 Http 协议

我们实现第 4 步生成的 HttpReqProtocolFactory 协议工厂，定义接收的数据类型，解析 Http 请求命令等；HTTP 的请求行格式为 Method Url Version\r\n 其它\r\n\r\n；Method 主要有 Get，POST，HEAD 等；当出现连续两个”\r\n”表示请求结束；

（1）因此我们定义协议头最小长度为 4，用来识别 Method 请求方法，框架在反序列化数据的时候调用 *HeaderSize* 接口获取最小头部长度，并且至少要读取该长度才会开始解码数据：

```
uint32_t HttpReqProtocolFactory::HeaderSize()
{
    ////Add Your Code Here
    return 4;
}
```

（2）接着实现解码头部数据，设置数据类型和 body_size 大小：

```
DecodeResult HttpReqProtocolFactory::DecodeHeader(const char *buffer
    , DataType &type
    , uint32_t &body_size)
{
    ////Add Your Code Here
    if(strncasecmp(buffer, "GET ", 4) != 0)
        return DECODE_ERROR;
    type = DTYPE_TEXT;
    body_size = 1024; //http请求串最大长度
    return DECODE_SUCC;
}
```

这边我们只是简单的处理 GET 请求，其他请求方法暂不支持，如果不是 GET 请求直接返回错误码；数据类型 type 设置为 DTYPE_TEXT 表示接收到的数据是文本类型的；body_size 设置为 1024 表示服务器接受的请求长度最大为 1024 字节；当解码成功后，框架将接续接收剩下的数据并解析协议体数据；

（3）由于我们只接收文本类型的数据，于是只需要实现 *DecodeTextBody* 接口（如果同时接收二进制数据的话，同时需要实现 *DecodeBinBody* 接口）：

```
DecodeResult HttpReqProtocolFactory::DecodeTextBody(ProtocolContext *context)
{
    ////Add Your Code Here
    char *data = context->Buffer+context->Size;
    data -= 4;
    if(strncmp(data, "\r\n\r\n", 4) != 0)
        return DECODE_DATA;
    HttpRequest *req = (HttpRequest*)m_Memory->Alloc(sizeof(HttpRequest));
    assert(req != NULL);
    req->Init();
    if(HttpParser::ParseRequest(*req, context->Buffer, context->Size) == false)
        return DECODE_ERROR;
    context->protocol = (void*)req;
    return DECODE_SUCC;
}
```

接口对接收到数据的最后四个字节进行检查，判断是否是两个连续的”\r\n“，如果不是表示 http 请求数据未完全接收到，返回 DECODE_DATA 表示数据不够；否则生成一个 HttpRequest 对象表示请求数据，然后解析 http 请求命令：解析 method 方法，url 和 http version；具体见 HttpParser.h 文件；

（4）最后当响应结束后，框架会通过调用 DeleteProtocolContext 删除协议上下文，其中又通过调用 DeleteProtocol 来删除生成的 ProtocolContext->Protocol 数据，即上面设置的 HttpRequest 对象：

```
void HttpReqProtocolFactory::DeleteProtocol(uint32_t protocol_type, void *protocol)
{
    ////Add Your Code Here
    m_Memory->Free(protocol, sizeof(HttpRequest));
    return;
}
```

至此我们已经实现了 Http 协议工厂需要使用的接口方法，其他接口方法使用默认的即可。

7.1.3 实现 Http Server

实现 Http 协议工厂后，我们开始正式的实现我们的 Http Server：

（1）首先在构造方法中创建 *HttpReqProtocolFactory* 实例，然后通过重写 *GetProtocolFactory* 方法将协议工厂提供给框架使用：

```
HttpEchoServer::HttpEchoServer()
{
    m_ProtocolFactory = (IProtocolFactory*)new HttpReqProtocolFactory(GetMemory());
}
IProtocolFactory* HttpEchoServer::GetProtocolFactory()
{
    return m_ProtocolFactory;
}
```

（2）实现 *OnReceiveProtocol* 接口处理接收到的 http 请求。具体实现见 HttpEchoServer 代码。主要是分析 *HttpRequest* 的 Url ，打开请求的 html 文件并将数据保存到 *send_context* ，然后调用 *SendProtocol* 将数据发送回客户端；

（3）当框架将数据成功发送给客户端后调用 *HttpEchoServer::OnSendSucc* 处理，如果客户端发送了 keep-alive ，则将连接设置为长连接（实际设置链接空闲超时时长为 20s），否则直接调用 *NotifySocketFinish* 方法通知框架链接结束（最后框架回调用 OnSocketFinished）；OnSendError 和 OnSendTimeout 用来响应发送错误和超时，将数据删除掉并通知结束链接；

（4）在 Start 方法中监听 8080 端口，并启动 EventServer 事件监听对象：

```
bool HttpEchoServer::Start()
{
    //Add Your Code Here
    if(Listen(8080) == false)
    {
        LOG_ERROR(logger, "listen on port=8080 error.");
        return false;
    }
    LOG_INFO(logger, "server listen on port=8080 succ.");
    IEventServer*event_server = GetEventServer();
    event_server->RunLoop();
    return true;
}
```

（5）其他一些方法的重写比如链接空闲超时时间，最大连接数等见代码；

7.1.4 运行结果

我们事先在 bin 目录下准备了一个 html 子目录，其中放置了几个 html 文件作为我们的服务内容；然后编译后进入 bin 目录运行服务器： ./HttpEchoServer

（1）在浏览器中输入：http://127.0.0.1:8080/ 默认输出浏览器请求的内容：

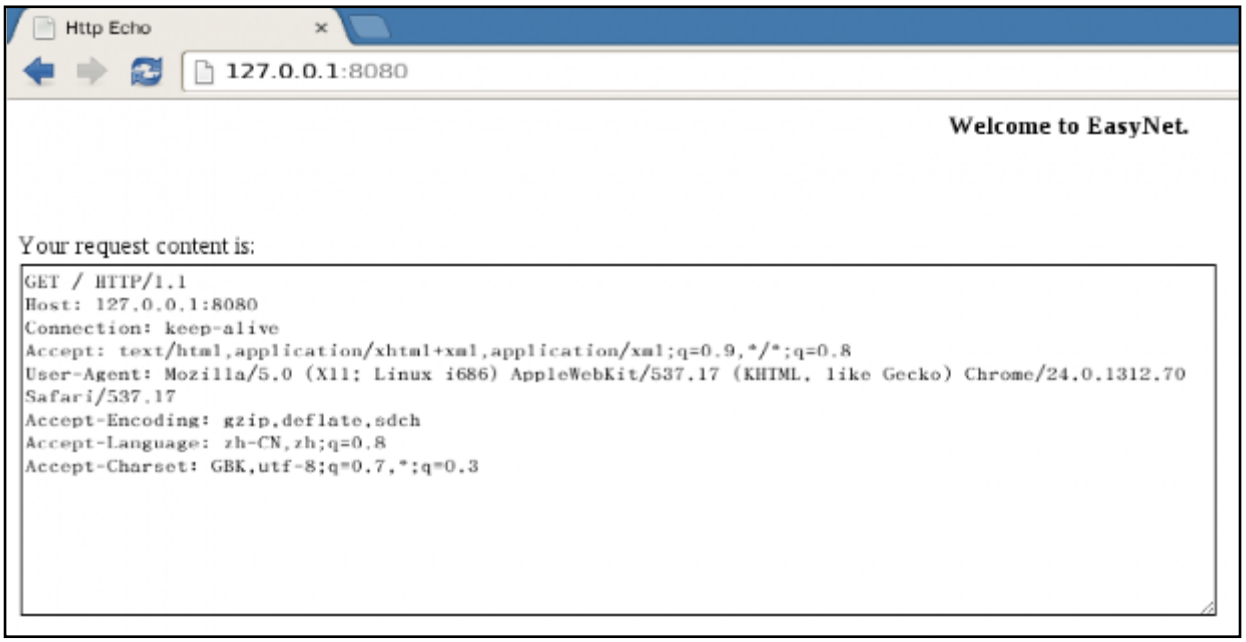

图 7.1 默认页面

（2）输入:http://127.0.0.1:8080/test.html 结果为：

图 7.2 test 页面

（3）点击 A Page 页面

图 7.3 a.html 页面

（4）点击 NotFound Page 链接

图 7.3 404 页面

OK，我们的 Http 服务器完成了...

7.2 多线程 Http Server

7.2.1 生成框架代码

按顺序执行下面的命令生成框架代码：

（1）mkdir HttpEchoServerThreads；cd HttpEchoServerThreads

（2）easynet -i

（3）easynet -a HttpEchoServer

（4）easynet -p HttpReqProtocolFactory

（5）easynet -a HttpEchoInterface -m

和 7.1 节的步骤类似，只是其中第 3 步的 HttpEchoServer 不是 Main 主框架（少了-m 参数），另外多了第 5 步，创建主工作线程框架类；第 4 步生成的 http 协议工厂类和 7.1 节一样，在此不再介绍；

7.2.2 实现多线程 Http Server

HttpEchoInterface 作为主线程，用来接入客户端的链接请求，通过重写 AcceptNewConnect 将新链接分派到工作线程 HttpEchoServer 中去：

```
boolHttpInterface::AcceptNewConnect(int32_tfd)
{
    LOG_INFO(logger, "send fd="<<fd<<"to Index="<<m_Index);
    m_HttpEchoServer[m_Index].SendMessage(fd);
    m_Index= (m_Index+1)%2;
    returntrue;
}
```

另外在 Start 方法中除了监听 8080 端口外，还需要创建工作线程：

```
bool HttpInterface::Start()
{
    if(Listen(8080) == false)
    {
        LOG_ERROR(logger, "listen on port=8080 error.");
        returnfalse;
    }
    LOG_INFO(logger, "server listen on port=8080 succ.");
    //启动线程
    m_Index= 0;
    Thread*thread = NULL;
    thread = &m_HttpEchoServer[0];
    thread->Start();
    thread = &m_HttpEchoServer[1];
    thread->Start();
    IEventServer*event_server = GetEventServer();
    event_server->RunLoop();
    returntrue;
}
```

HttpEchoServer 是工作线程，真正处理链接上的数据接收/发送工作。HttpEchoServer 除了包含 7.1 节介绍的内容外，还实现了线程类 Thread 的 *DoRun* 接口用来启动线程实例：

```
voidHttpEchoServer::DoRun()
{
    Start();
}
```

在 Start 方法中注册消息监听接口：

```
bool HttpEchoServer::Start()
{
    //Add Your Code Here
    if(ListenMessage() == false)
    {
        LOG_ERROR(logger, "listen message error.");
        returnfalse;
    }
    LOG_INFO(logger, "server listen message succ.");
    IEventServer*event_server = GetEventServer();
    event_server->RunLoop();
    returntrue;
}
```

八、Appendix

8.1 ByteBuffer.h

是对字符串缓冲区的一个封装，主要处理缓冲区内存的分配，同时调用者能够直接使用缓冲区，为使用者带来很大的方便。使用者可以提供自己的IMemory内存分配器对内存的分配进行管理（比如减少内存碎片的产生），以便提高性能。

8.2 CondQueue.h

条件队列模板类，可以设置操作的等待时间等。

8.3 ConfigReader.h

配置文件读取工具类。

8.4 EventServer.h

I/O事件和时钟超时事件管理模块。

8.5 EventServerEpoll.h

EventServer的epoll实现版本。

8.6 Heap.h

最小堆排序，用来进行时钟的管理。

8.7 Imemory.h

内存分配器接口。

8.8 KVData.h

键-值数据打包/解包工具类，可用于数据的序列化/反序列化。该类的特点是使用方便，支持大数据操作。

8.8.1 序列化数据

```
//key值定义
#define KEY0 0
#define KEY1 1
KVData kvdata(true); //使用网络字节序
//KVData kvdata(false); //不使用网络字节序
int8_t v8=8;
int64_t v64=64;
//设置值
kvdata.SetValue(KEY0, v8);
kvdata.SetValue(KEY1, v64);
```

```
uint32_tsize = kvdata.Size();
//序列化
char*buffer = (char*)malloc(size);
kvdata.Serialize(buffer);
//use buffer...
```

8.8.2 反序列化数据

```
KVData kvdata(true); //使用网络字节序
//KVData kvdata(false); //不使用网络字节序
int8_t v8;
int64_t v64;
if(kvdata.UnSerialize(buffer, szie)==false)
    return-1; //反序列化错误
if(kvdata.GetValue(KEY0, v8) == false)
    return-1; //没有对应类型的数据
if(kvdata.GetValue(KEY1, v64) == false)
    return-1; //没有对应类型的数据
//use v8,v64...
```

8.8.3 嵌套使用 KVData

```
KVData sub_kv(true);
//KVData sub_kv(false);
//use sub_kv...
kvdata.SetValue(KEY_KV, &sub_kv);
kvdata.Serialize(buffer);
//Get KVData
kvdata.UnSerialize(buffer, len);
if(kvdata.GetValue(KEY_KV, sub_kv) == false)
    return-1;
//use sub_kv...
```

8.8.4 高级功能——大数据的读写

当需要写入大数据时(比如几 M的数据),数据可能从某个地方读入(第一次copy)到缓冲区data，然后再使用 KVData 将缓冲区 data 的数据序列化到缓冲区 buffer(第二次 copy),最后再使用 buffer(比如发送到网络)。KVData 提供了一种方法来减少 copy 的次数 :直接使用buffer 作为第一次 copy 的缓冲区，从而省掉第一次的 copy。

```
KVBuffer kvbuffer = kvdata.BeginWrite(buffer, KEY2);
char*data = kvbuffer.second;
uint32_t len = fread(data, 1, 1024*1024, fp); //从文件 fp中读入 1M 的数据
uint32_t data_len = kvdata.EndWrite(kvbuffer, len); //设置数据实际长度.返回总共占用字节数
//sendbuffer...
```

8.9 MemoryPool.h

IMemory 的一个实现类，用来管理不同大小的内存，防止内存碎片产生。线程安全。

8.10 Socket.h

对底层 socket 的封装。

8.11 Thread.h

线程接口，方便用于创建线程。

8.12 ThreadPool.h

线程池接口，方便创建线程池。

8.13 IappInterface.h

应用程序接口类，对 EventServer，EventHander，IMemory，IProtocolFactory 等模块的组织；为应用层提供了统一的接口，方便用户开发。

8.14 IprotocolFactory.h

协议数据工厂类，用于数据的序列化和反序列化。

8.15 KVDataProtocolFactory.h

用 KVData 实现的数据工厂类。

8.16 ListenHandler.h

用于处理监听端口的事件处理类。

8.17 TransHandler.h

用于数据传输的事件处理类。