InIOCP（小旋风服务套件）

技　术　要　点

（版本：2.5.35.1245）

适用版本：Delphi 7、Delphi 2007、
Delphi XE、Delphi XE5-XE10

作　　者：高凉新农，QQ 群：365531817

下载途径：csdn.net，2ccc.com 的 ftp 资源
github：https://github.com/cnwneumann/InIOCP

进入正题之前，先简单谈谈开发这套软件的缘由，作者本人是 Delphi 使用者，从事数据库开发较多，偶尔也用 Indy、RO、RTC 等组件弄一下网络开发，其中 RTC 用得最多，总的来说，感觉都不理想。本人接触 IOCP 不久，学习它后产生了亲自写一套组件的想法，于是按设想慢慢摸索，没想到停不下来了，最终 InIOCP 得以成品发布。

再顺便说一下本软件的名称，以前用“In-IOCP 服务组件库”，有人说很拗口，现在稍作改动，叫“InIOCP 服务套件”，顺便起了个中文名“小旋风”，全称为“InIOCP 小旋风服务套件”，名字嘛，不同人的喜好不同，如果你喜欢，也可以另起一个。

InIOCP 服务套件首发已经一年多，作了多次更新，但一直没写过什么使用说明或技术文档，主要原因是写文档与写代码的思路不同、要组织敲打文字，比较费时，还有就是旧版本的 C/S 模式的相关代码比较乱，传输机制也不理想等。

1.6 版之后，开始着手改进 C/S 模式的传输机制，采用与 HTTP 服务类似的方法，经过几个月的改造，总算完工。

新版重新规划了 C/S 模式代码，质量有了较大提升，服务端 Socket 的内存占用减半，而使用方法与旧版的基本一样，只是个别过程、方法或事件名称有了调整，删除了旧版的子任务功能，增加了新特性，使用起来更为灵活方便了；HTTP 服务部分代码基本不变。

本软件没什么高深的技术，对熟悉 Delphi 的开发者来说，理解代码是很容易的事，但为了方便初学者学习交流，有必要写个简单的文档，简明扼要地解释一下。

组织文字和写代码的思维完全不同，要讲清楚也不容易，只能抓重点了，先从简单的开始。

一、基础知识

一、消息封装

1.字段。用过数据库组件的都知道，数据集 DataSet 里面有字段 Field，InIOCP 的消息包相当于 DataSet，也有 Field，详见单元 iocp_msgPacks:

```
TVarField = class(TObject)
... ...
```

2.消息包。为了更方便直观使用消息，对其进行封装是有必要的，每个消息就包含一个或多个上面说的字段 Field:

```
TBasePack = class(TInList)
... ...
```

从 TBasePack 继承出带协议头的消息类 THeaderPack，再从 THeaderPack 继承出发送、接收用的消息类（收发型消息），这里不作细述。

3.消息组成。InIOCP 的收发型消息分两部分，即**主体和附件**，主体指 As...系列的变量型内容,附件指附加到消息上的内容,也就是文件或数据流。

这是 InIOCP 消息的一大特点，让消息使用更灵活，大大方便了各种数据的传输。

InIOCP 消息还支持压缩和校验码，压缩类型为 Zip,校验码类型有 MD5、MurmurHash 两种。

InIOCP 消息有两个重要的过程（procedure），一个是 **SaveToStream**，作用是把 As 系列变量表转换为内存流，这是发送前必须的；一个是 **Initialize**，作用是把内存流解析为 As 系列变量表，方便消息的变量/字段读取。

C/S 模式的每一消息都带协议头 TMsgHead，协议头是对当前消息的整体描述，记录其重要信息，详见单元 iocp_msgPacks。

现版本支持 WebSocket 协议，也做了简单的 JSON 封装，详见第三部分。

二、数据发送器

服务端和客户端的数据发送器有所不同，服务端用 WSASend 发送数据，

客户端用 Send，但都用 AddTask 系列方法加入发送任务，详见单元 iocp_senders。

三、数据接收器

服务端接收数据与客户端发送数据有一对一关系，接收方法相对简单，客户端除了接收服务端的反馈数据,也可能同时收到其他客户端的推送消息，其方法稍微复杂一点。

客户端每完整接收一个消息，立刻把它加到投放线程的消息队列，投放线程的任务是把消息逐一取出、送入应用层，详见单元 iocp_receivers。

四、IOCP 相关

此部分比较复杂，很难几句话说得清楚，建议搜索相关资料进行学习，详见单元 iocp_server 的类:

1.IOCP 引擎: TIOCPEngine

2.IOCP 服务器: TInIOCPServer

3.套接字预设管理: TAcceptManager

4.工作线程: TWorkThread

5.工作线程管理: TWorkThreadPool

五、超时检查

最初版本的 InIOCP 不设 TCP 心跳包，建一个专用线程检查超时，一旦超时，立刻推算一个消息给服务端 Socket（见: TBaseSocket.PostEvent），消息发出后即断开连接，只要不是特殊情况，客户端都会收到超时消息。

现版本做了调整，当 TInIOCPServer.TimeOut 为 0 时，自动增加心跳，这种情况下超时断开时不会向客户端发送消息。详见单元 iocp_server:

TTimeoutThread = class(TBaseThread)

... ...

六、套接字的关闭

服务端建一个专用线程关闭 Socket,详见单元 iocp_server:

TCloseSocketThread = class(TCycleThread)

... ...

七、业务线程

它就是执行业务工作的线程 TBusiThread，名称比“逻辑线程”的叫法更直观，业务线程有多个，用管理器 TBusiWorkManager 集中管理，服务端收到的各种请求都被加到 TBusiWorkManager 的任务列表，TBusiThread 则不断循环从任务列表中取出、执行。详见单元 iocp_threads:

```
TBusiThread = class(TCycleThread)
... ...
TBusiWorkManager = class(TObject)
... ...
```

八、推送线程

顾名思义，它是专门负责推送消息的线程，类名为 TPushThread，推送线程也有多个，用管理器 TPushMsgManager 集中管理，要推送的消息被加到 TPushMsgManager 的消息池 TMsgPushPool 中，后者定期把消息投放到推送列表，TPushThread 则不断循环从消息列表中取出、发送。详见单元 iocp_threads:

```
TPushThread = class(TCycleThread)
... ...
TMsgPushPool = class(TBaseThread)
... ...
TPushMsgManager = class(TObject)
... ...
```

九、服务端套接字

服务端套接字类都继承于 TBaseSocket，根据不同协议继承出相应的子类并在其中实现相应的行为。连接时，服务端根据不同的协议建相应的 Socket 对象，默认用 THttpSocekt 类对象，详见单元 iocp_sockets。

十、对象池和列表

单元 iocp_objPools 定义了对象池 TObjectPool 及其子类；单元 iocp_lists 定义了几种列表，，TInStringList 在最初版本的 HTTP 服务的响应使用中起了很大作用，但效率稍低，经优化后加快了响应报头的写入。

二、业务流程

这里只讲业务流程大概，提及代码的关键位置。

一个客户端连接含有三个重要对象，一是发送线程，一是接收线程，一是投放线程。接收线程里面有一个数据接收器，后者每完整接收一个消息，立刻把它塞入投放线程的消息列表，投放线程的任务是把消息逐一取出、送入应用层。

客户端的消息类从 TBaseMessage 继承，一个消息的发送流程如下（括号内为关键代码位置，用图形比较麻烦）：

1.消息

2.Post 操作（TMessagePack.Post 和 TInBaseClient.InternalPost）

3.加入发送线程的消息列表（TSendThread.AddWork）

4.发送线程从列表取消息（TSendThread.GetWork）

5.发送（TSendThread.SendMessage）

为方便访问消息字段和隐藏一些属性，发送代码在消息类 TClientParams 中实现，发送时先准备数据流，接着发送消息描述、主体和附件（类似于 HTTP 协议），详见: TClientParams.InternalSend。

消息发送出去了，下面看看服务端的消息流程，不了解 IOCP 或线程的赶快补一下这方面的知识。

IOCP 不管监测到任何活动，都立刻触发工作线程的 Execute 方法，这里执行的是 TWorkThread.ExecuteWork，记住这两点:

1.客户端连接进入，执行 TInIOCPServer.AcceptExClient（初始化）

2.有数据进出、断开或异常，执行 TWorkThread.HandleIOData

收到数据时把对应的服务端 Socket 加到业务线程管理器的任务列表，业务线程不断从列表中取任务、执行和反馈结果，此流程大概如下:

1.IOCP 激活工作线程（TWorkThread.ExecuteWork）

2.工作线程根据重叠结构状态，调用 TWorkThread.HandleIOData

3.把服务端 Socket 加到任务列表（TBusiWorkManager.AddWork）

4.激活业务线程，业务线程开始工作

服务端 Socket 对象的 FRecvBuf 是客户端发送来的数据，要处理的正是它，现在轮到业务线程工作了:

1.业务线程被激活（TCycleThread.Activate）

2.从任务列表取 Socket(TBusiThread.DoMethod)

3.执行业务（TBaseSocket.DoWork，是虚函数）

此时确实执行了业务？不一定，FRecvBuf 的容量有限，数据未必接收完毕，只有接收完毕才真正执行，看这里（以 C/S 模式为例）:

TReceiveSocket.ExecuteWork

TReceiveSocket 类有个数据接收器 FReceiver，它不断接收数据，接收完毕且校验无误后把主体数据流解析成变量表，调用 TIOCPSocket.HandleDataPack，如果没其他异常则真正进入应用层，见虚函数 TBaseWorker.Execute（Http 服务是 TBaseWorker.HttpExecute）。

服务端应用层对消息的处理方法和客户端的是基本一样的,TIOCPSocket 的 FResult 是 TReturnResult 的类对象，是返回给客户端的消息，当执行业务、设置了待返回结果后，在这里发送结果给客户端:

TReturnResult.ReturnResult

到此，服务端把结果发送出去了，现在轮到客户端的数据接收器开始工作了，它做法和服务端的相似，只不过要考虑推送消息的接收，稍微复杂一点，看这里:

TRecvThread.HandleDataPacket

这里把收到的数据不断送到数据接收器，后者不断分离消息，每完整接收一个消息，立刻把它加到投放线程的消息列表，步骤:

1.接收首数据包（带协议头，TClientReceiver.Prepare）

2.接收后续数据（TClientReceiver.Receive）

3.一个消息接收完毕，投放（TClientReceiver.InterPostResult）

4.继续接收（调用 WSARecv，传递回调函数 WorkerRoutine）

收到的消息被加到投放线程 TPostThread 的消息列表中，投放线程被激活，逐一取出消息，执行应用层操作，见:

```
TPostThread = class(TCycleThread)
... ...
```

到此，一个消息的流程就走完了。

三、WebSocket 协议

既然支持 HTTP 协议，如果不支持 WebSocket 就美中不足了，现版本已支持 WebSocket 并作了扩展，用 InIOCP-JSON 扩展消息封装，同样支持消息推送。

InIOCP-JSON 消息基类为 TBaseJSON，为了方便对消息进行操作、文件传输等，预设了一些属性，增加对数据集传输的支持，详见单元 iocp_WsJSON。

WebSocket 的使用比较简单，详细的使用方法请参考系统软件包的三个相关例子。

四、消息推送

现在结合自己的认识理解谈谈消息推送，开始我把它简单理解为“把消息发给别人”，直接在当前线程中发送出去，后经仔细思考，这样的做法非常错误，不能把消息推送理解为一个从属、地位低于业务线程的行为，应该提高它的地位，它和业务线程的地位是平等的。

基于上述思想，给 InIOCP 设置独立的推送线程，它和业务线程共同竞争服务端 Socket，只有 Socket 处于空闲状态且任务结束（数据接收、业务处理和结果反馈均完成）才对 Socket 进行加锁发送数据，这样保证服务端的消息有序发送，客户端收到的消息自然也是有序的（TCP 的数据收发有序）。

理论上，推送线程的工作原理与业务线程的一致，既然业务线程能很好地工作，它当然也行，所以说，新版的推送方法是可行的——这是我个人的一点点见解。

此推送方法不能说非常好，但确实简单可行，新版在处理超时、拒绝服务和删除客户端时都使用它，这对提高稳定性有很大帮助。

本文到此为止，作者技术有限，如有不足之处敬请包涵。

谢谢各位拜读，祝一切顺利！

声　明

本软件属开源产品，作者只保留版权，在法律范围内，任何组织或个人均可自由复制、使用和交换，作者尽能力维护软件本身的问题，不承诺解决用户在使用过程中出现的任何问题。

鸣　谢

感谢 InIOCP 群（365531817）的全体网友!

感谢在 InIOCP 开发过程中给予支持、指导和测试的其他网友!