 http://www.oschina.net/p/supersocket

# [[置顶] C#高性能大容量SOCKET并发（一）：IOCP完成端口例子介绍](http://blog.csdn.net/sqldebug_fan/article/details/17556353)

标签： [C#](http://www.csdn.net/tag/C%23)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[IOCP](http://www.csdn.net/tag/IOCP)[SocketAsyncEventArgs](http://www.csdn.net/tag/SocketAsyncEventArgs)

2014-01-22 15:49 29264人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/17556353#comments)(60) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/17556353#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

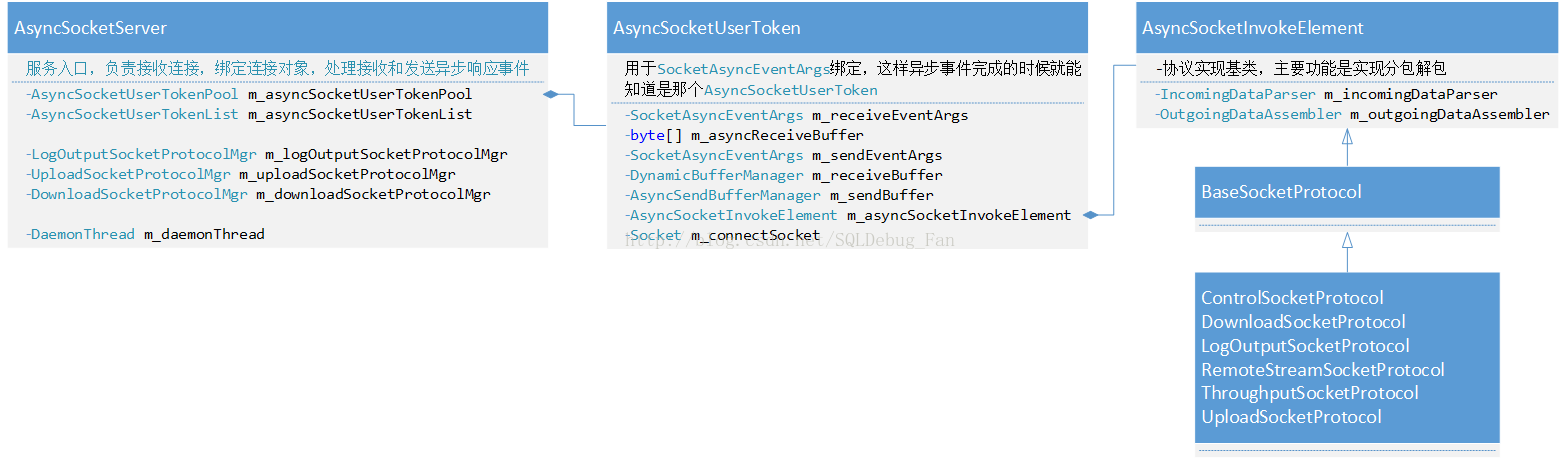
例子主要包括SocketAsyncEventArgs通讯封装、服务端实现日志查看、SCOKET列表、上传、下载、远程文件流、吞吐量协议，用于[**测试**](http://lib.csdn.net/base/softwaretest)SocketAsyncEventArgs的性能和压力，最大连接数支持65535个长连接，最高命令交互速度达到250MB/S（使用的是127.0.0.1的方式，相当于千兆网卡1Gb=125MB/S两倍的吞吐量）。服务端用C#编写，并使用log4net作为日志模块；客户端使用DELPHI编写，和以前的DELPHI版完成端口共用一个客户端，提供C#版的上传客户端。

**C#版完成端口具有以下特点：**

* 连接在线管理（提供在线连接维护，连接会话管理，数据接收，连接断开等相关事件跟踪）；
* 发送数据智能合并（组件会根据资源使用情况，对多个同时发送向同一连接的多个消息数据进行合并写入缓冲区；
* 内存池管理（提供一系列的接收和发送buffer可复用池）；
* 数据读写封装；
* 通讯协议封装；

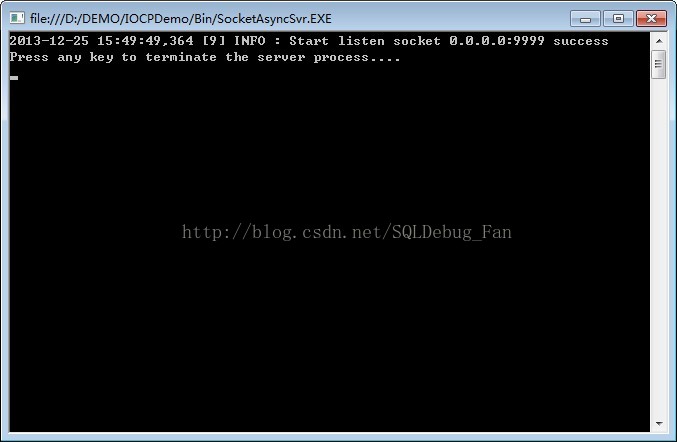
由不同的类负责实现，比MSDN的例子代码多了不少逻辑，为了方便大家阅读和理解，把整体类框架设计整体如下。

**类结构图**



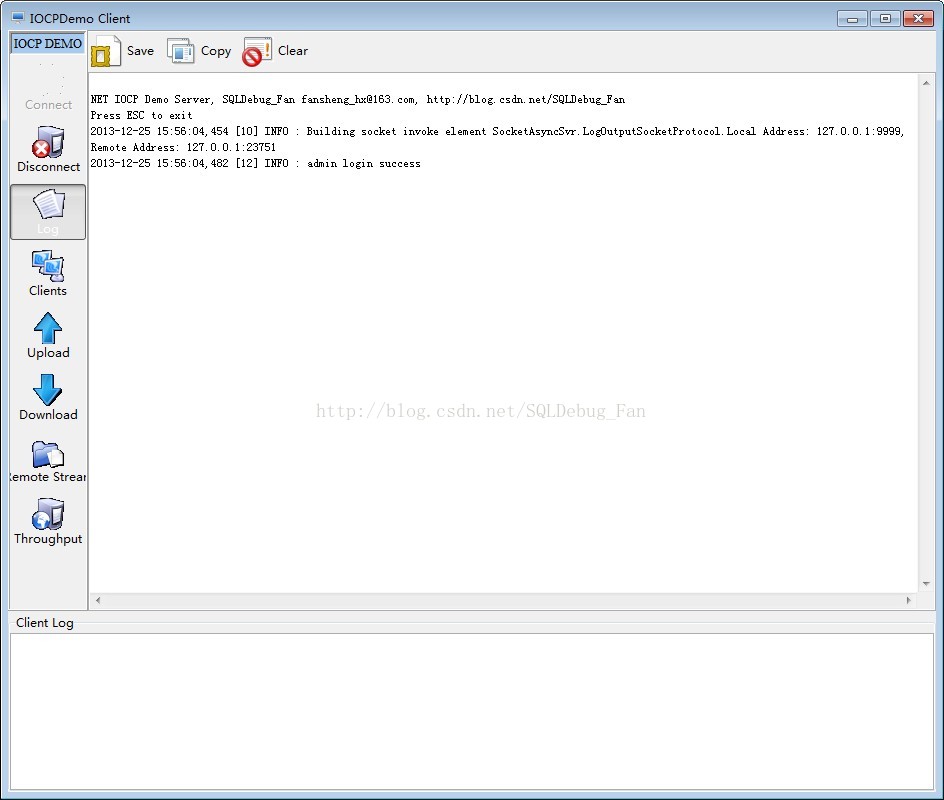
**服务端**

服务端是一个控制台程序，支持分颜色显示日志，界面截图如下：



**客户端**

用于对服务端进行性能测试，实现了吞吐量测试协议，还有日志、查看客户端、上传、下载、远程文件流五个应用层协议，可以作为要实现其他协议的例子参考，其中远程文件流可以作为分布式系统的底层数据交互模块，界面截图如下：



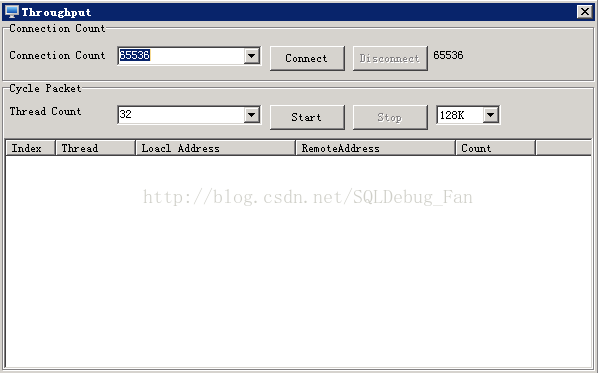
暂不提供NET的客户端，NET的界面编程还在学习中，非常抱歉。

**性能测试结果**

支持65535个长连接，为了支持65535个连接，需要把程序用64位进行编译。

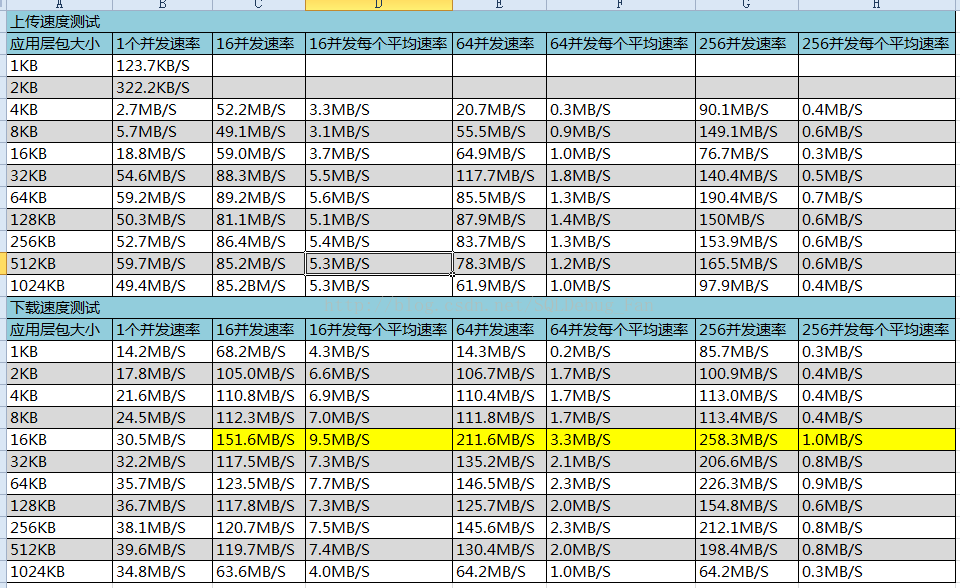
循环发包网络流量可以达到250MB/S，受限于是网卡流量，千兆网卡最大1Gb=125MB/S，为了测试最大吞吐量，我们需要把客户端和服务器放在同一台电脑上，使用127.0.0.1这个IP测试，127.0.0.1是本机回送地址（Loopback Address），即主机IP堆栈内部的IP地址，主要用于网络[**软件测试**](http://lib.csdn.net/base/softwaretest)以及本地机进程间通信，无论什么程序，一旦使用回送地址发送数据，协议软件立即返回，不进行任何网络传输。

其中截图如下：





按文件上传下载对服务器进行应用层速率测试，分为1、16、64、256个并发进行测试，发包大小分1、2、4、8、16、32、64、128、256、512、1024KB，测试结果汇总如下，其中最高速率达到250MB/S，综合最优发包大小为16、32KB，其中速率计算有可能存在偏差，因为传到最后，并发数会变小，造成整个的速率上升，可以作为性能测试的参考。



DEMO下载地址：[http://download.csdn](http://download.csdn.net/detail/sqldebug_fan/7467745)[**.NET**](http://lib.csdn.net/base/dotnet)/detail/sqldebug\_fan/7467745

免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

# [[置顶] C#高性能大容量SOCKET并发（一）：IOCP完成端口例子介绍](http://blog.csdn.net/sqldebug_fan/article/details/17556353)

标签： [C#](http://www.csdn.net/tag/C%23)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[IOCP](http://www.csdn.net/tag/IOCP)[SocketAsyncEventArgs](http://www.csdn.net/tag/SocketAsyncEventArgs)

2014-01-22 15:49 29265人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/17556353#comments)(60) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/17556353#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

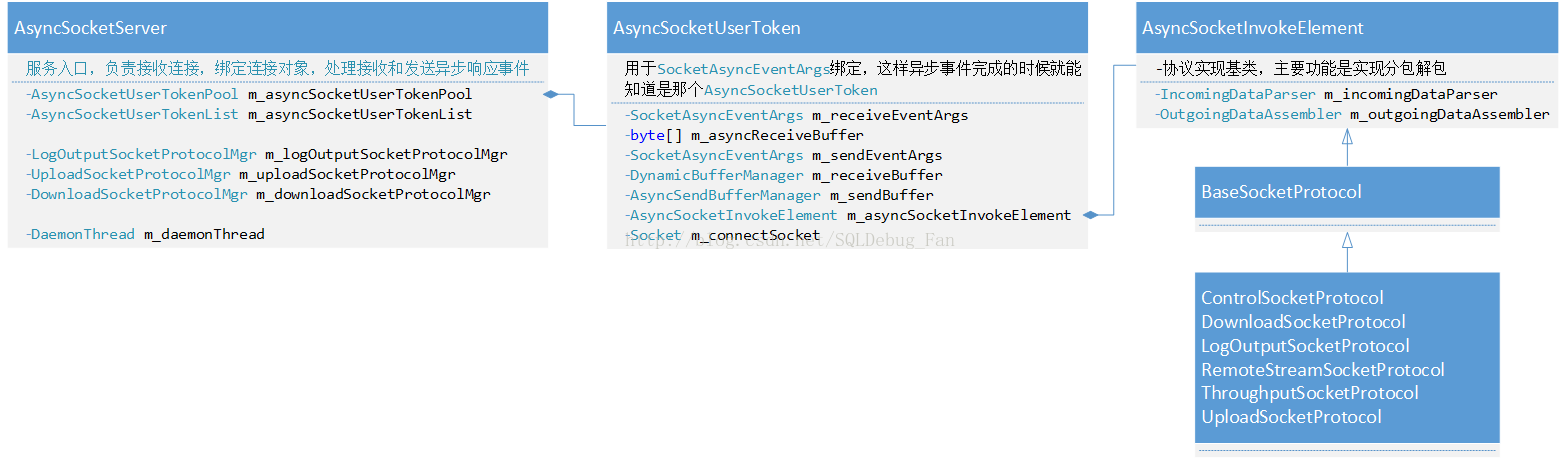
例子主要包括SocketAsyncEventArgs通讯封装、服务端实现日志查看、SCOKET列表、上传、下载、远程文件流、吞吐量协议，用于[**测试**](http://lib.csdn.net/base/softwaretest)SocketAsyncEventArgs的性能和压力，最大连接数支持65535个长连接，最高命令交互速度达到250MB/S（使用的是127.0.0.1的方式，相当于千兆网卡1Gb=125MB/S两倍的吞吐量）。服务端用C#编写，并使用log4net作为日志模块；客户端使用DELPHI编写，和以前的DELPHI版完成端口共用一个客户端，提供C#版的上传客户端。

**C#版完成端口具有以下特点：**

* 连接在线管理（提供在线连接维护，连接会话管理，数据接收，连接断开等相关事件跟踪）；
* 发送数据智能合并（组件会根据资源使用情况，对多个同时发送向同一连接的多个消息数据进行合并写入缓冲区；
* 内存池管理（提供一系列的接收和发送buffer可复用池）；
* 数据读写封装；
* 通讯协议封装；

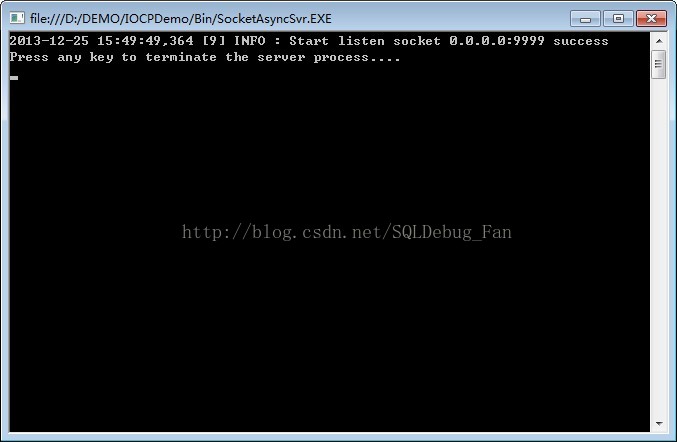
由不同的类负责实现，比MSDN的例子代码多了不少逻辑，为了方便大家阅读和理解，把整体类框架设计整体如下。

**类结构图**



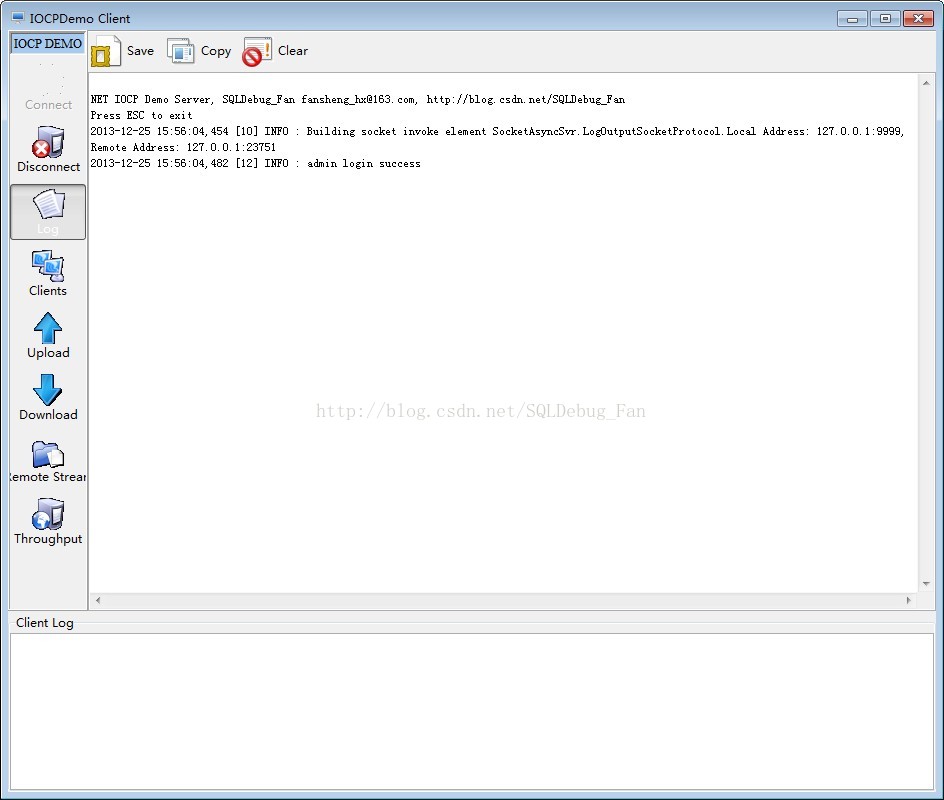
**服务端**

服务端是一个控制台程序，支持分颜色显示日志，界面截图如下：



**客户端**

用于对服务端进行性能测试，实现了吞吐量测试协议，还有日志、查看客户端、上传、下载、远程文件流五个应用层协议，可以作为要实现其他协议的例子参考，其中远程文件流可以作为分布式系统的底层数据交互模块，界面截图如下：



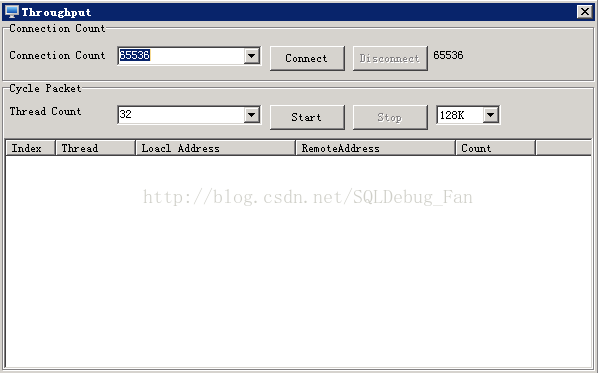
暂不提供NET的客户端，NET的界面编程还在学习中，非常抱歉。

**性能测试结果**

支持65535个长连接，为了支持65535个连接，需要把程序用64位进行编译。

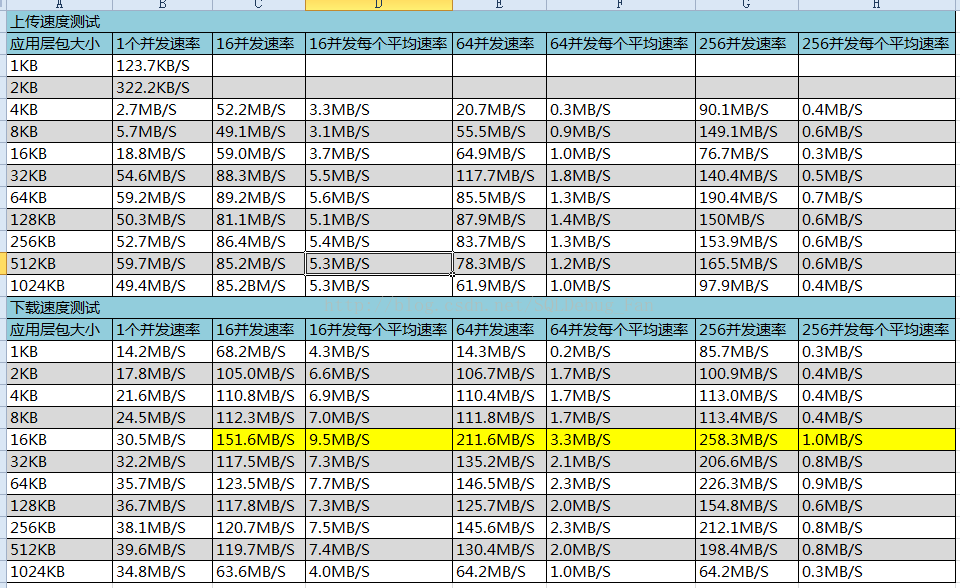
循环发包网络流量可以达到250MB/S，受限于是网卡流量，千兆网卡最大1Gb=125MB/S，为了测试最大吞吐量，我们需要把客户端和服务器放在同一台电脑上，使用127.0.0.1这个IP测试，127.0.0.1是本机回送地址（Loopback Address），即主机IP堆栈内部的IP地址，主要用于网络[**软件测试**](http://lib.csdn.net/base/softwaretest)以及本地机进程间通信，无论什么程序，一旦使用回送地址发送数据，协议软件立即返回，不进行任何网络传输。

其中截图如下：





按文件上传下载对服务器进行应用层速率测试，分为1、16、64、256个并发进行测试，发包大小分1、2、4、8、16、32、64、128、256、512、1024KB，测试结果汇总如下，其中最高速率达到250MB/S，综合最优发包大小为16、32KB，其中速率计算有可能存在偏差，因为传到最后，并发数会变小，造成整个的速率上升，可以作为性能测试的参考。



DEMO下载地址：[http://download.csdn](http://download.csdn.net/detail/sqldebug_fan/7467745)[**.NET**](http://lib.csdn.net/base/dotnet)/detail/sqldebug\_fan/7467745

免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

19

踩

4

* 上一篇[DELPHI高性能大容量SOCKET并发（十）：IOCP完成端口性能优化](http://blog.csdn.net/sqldebug_fan/article/details/9043719)
* 下一篇[C#高性能大容量SOCKET并发（二）：SocketAsyncEventArgs封装](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[置顶] [C#高性能大容量SOCKET并发（二）：SocketAsyncEventArgs封装](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[SocketAsyncEventArgs](http://www.csdn.net/tag/SocketAsyncEventArgs)[IOCP](http://www.csdn.net/tag/IOCP)

2014-03-04 15:28 31926人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/17557341#comments)(33) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/17557341#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

1、SocketAsyncEventArgs介绍

SocketAsyncEventArgs是微软提供的高性能异步Socket实现类，主要为高性能网络服务器应用程序而设计，主要是为了避免在在异步套接字 I/O 量非常大时发生重复的对象分配和同步。使用此类执行异步套接字操作的模式包含以下步骤：  
1.分配一个新的 SocketAsyncEventArgs 上下文对象，或者从应用程序池中获取一个空闲的此类对象。  
2.将该上下文对象的属性设置为要执行的操作（例如，完成回调方法、数据缓冲区、缓冲区偏移量以及要传输的最[**大数据**](http://lib.csdn.net/base/spark)量）。  
3.调用适当的套接字方法 (xxxAsync) 以启动异步操作。  
4.如果异步套接字方法 (xxxAsync) 返回 true，则在回调中查询上下文属性来获取完成状态。  
5.如果异步套接字方法 (xxxAsync) 返回 false，则说明操作是同步完成的。 可以查询上下文属性来获取操作结果。  
6.将该上下文重用于另一个操作，将它放回到应用程序池中，或者将它丢弃。

2、SocketAsyncEventArgs封装

使用SocketAsyncEventArgs之前需要先建立一个Socket监听对象，使用如下代码：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **public** **void** Start(IPEndPoint localEndPoint)
2. {
3. listenSocket = **new** Socket(localEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
4. listenSocket.Bind(localEndPoint);
5. listenSocket.Listen(m\_numConnections);
6. Program.Logger.InfoFormat("Start listen socket {0} success", localEndPoint.ToString());
7. //for (int i = 0; i < 64; i++) //不能循环投递多次AcceptAsync，会造成只接收8000连接后不接收连接了
8. StartAccept(**null**);
9. m\_daemonThread = **new** DaemonThread(**this**);
10. }

然后开始接受连接，SocketAsyncEventArgs有连接时会通过Completed事件通知外面，所以接受连接的代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **public** **void** StartAccept(SocketAsyncEventArgs acceptEventArgs)
2. {
3. **if** (acceptEventArgs == **null**)
4. {
5. acceptEventArgs = **new** SocketAsyncEventArgs();
6. acceptEventArgs.Completed += **new** EventHandler<SocketAsyncEventArgs>(AcceptEventArg\_Completed);
7. }
8. **else**
9. {
10. acceptEventArgs.AcceptSocket = **null**; //释放上次绑定的Socket，等待下一个Socket连接
11. }
13. m\_maxNumberAcceptedClients.WaitOne(); //获取信号量
14. **bool** willRaiseEvent = listenSocket.AcceptAsync(acceptEventArgs);
15. **if** (!willRaiseEvent)
16. {
17. ProcessAccept(acceptEventArgs);
18. }
19. }

接受连接响应事件代码：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **void** AcceptEventArg\_Completed(**object** sender, SocketAsyncEventArgs acceptEventArgs)
2. {
3. **try**
4. {
5. ProcessAccept(acceptEventArgs);
6. }
7. **catch** (Exception E)
8. {
9. Program.Logger.ErrorFormat("Accept client {0} error, message: {1}", acceptEventArgs.AcceptSocket, E.Message);
10. Program.Logger.Error(E.StackTrace);
11. }
12. }

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **private** **void** ProcessAccept(SocketAsyncEventArgs acceptEventArgs)
2. {
3. Program.Logger.InfoFormat("Client connection accepted. Local Address: {0}, Remote Address: {1}",
4. acceptEventArgs.AcceptSocket.LocalEndPoint, acceptEventArgs.AcceptSocket.RemoteEndPoint);
6. AsyncSocketUserToken userToken = m\_asyncSocketUserTokenPool.Pop();
7. m\_asyncSocketUserTokenList.Add(userToken); //添加到正在连接列表
8. userToken.ConnectSocket = acceptEventArgs.AcceptSocket;
9. userToken.ConnectDateTime = DateTime.Now;
11. **try**
12. {
13. **bool** willRaiseEvent = userToken.ConnectSocket.ReceiveAsync(userToken.ReceiveEventArgs); //投递接收请求
14. **if** (!willRaiseEvent)
15. {
16. **lock** (userToken)
17. {
18. ProcessReceive(userToken.ReceiveEventArgs);
19. }
20. }
21. }
22. **catch** (Exception E)
23. {
24. Program.Logger.ErrorFormat("Accept client {0} error, message: {1}", userToken.ConnectSocket, E.Message);
25. Program.Logger.Error(E.StackTrace);
26. }
28. StartAccept(acceptEventArgs); //把当前异步事件释放，等待下次连接
29. }

接受连接后，从当前Socket缓冲池AsyncSocketUserTokenPool中获取一个用户对象AsyncSocketUserToken，AsyncSocketUserToken包含一个接收异步事件m\_receiveEventArgs，一个发送异步事件m\_sendEventArgs，接收数据缓冲区m\_receiveBuffer，发送数据缓冲区m\_sendBuffer，协议逻辑调用对象m\_asyncSocketInvokeElement，建立服务对象后，需要实现接收和发送的事件响应函数：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **void** IO\_Completed(**object** sender, SocketAsyncEventArgs asyncEventArgs)
2. {
3. AsyncSocketUserToken userToken = asyncEventArgs.UserToken **as** AsyncSocketUserToken;
4. userToken.ActiveDateTime = DateTime.Now;
5. **try**
6. {
7. **lock** (userToken)
8. {
9. **if** (asyncEventArgs.LastOperation == SocketAsyncOperation.Receive)
10. ProcessReceive(asyncEventArgs);
11. **else** **if** (asyncEventArgs.LastOperation == SocketAsyncOperation.Send)
12. ProcessSend(asyncEventArgs);
13. **else**
14. **throw** **new** ArgumentException("The last operation completed on the socket was not a receive or send");
15. }
16. }
17. **catch** (Exception E)
18. {
19. Program.Logger.ErrorFormat("IO\_Completed {0} error, message: {1}", userToken.ConnectSocket, E.Message);
20. Program.Logger.Error(E.StackTrace);
21. }
22. }

在Completed事件中需要处理发送和接收的具体逻辑代码，其中接收的逻辑实现如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **private** **void** ProcessReceive(SocketAsyncEventArgs receiveEventArgs)
2. {
3. AsyncSocketUserToken userToken = receiveEventArgs.UserToken **as** AsyncSocketUserToken;
4. **if** (userToken.ConnectSocket == **null**)
5. **return**;
6. userToken.ActiveDateTime = DateTime.Now;
7. **if** (userToken.ReceiveEventArgs.BytesTransferred > 0 && userToken.ReceiveEventArgs.SocketError == SocketError.Success)
8. {
9. **int** offset = userToken.ReceiveEventArgs.Offset;
10. **int** count = userToken.ReceiveEventArgs.BytesTransferred;
11. **if** ((userToken.AsyncSocketInvokeElement == **null**) & (userToken.ConnectSocket != **null**)) //存在Socket对象，并且没有绑定协议对象，则进行协议对象绑定
12. {
13. BuildingSocketInvokeElement(userToken);
14. offset = offset + 1;
15. count = count - 1;
16. }
17. **if** (userToken.AsyncSocketInvokeElement == **null**) //如果没有解析对象，提示非法连接并关闭连接
18. {
19. Program.Logger.WarnFormat("Illegal client connection. Local Address: {0}, Remote Address: {1}", userToken.ConnectSocket.LocalEndPoint,
20. userToken.ConnectSocket.RemoteEndPoint);
21. CloseClientSocket(userToken);
22. }
23. **else**
24. {
25. **if** (count > 0) //处理接收数据
26. {
27. **if** (!userToken.AsyncSocketInvokeElement.ProcessReceive(userToken.ReceiveEventArgs.Buffer, offset, count))
28. { //如果处理数据返回失败，则断开连接
29. CloseClientSocket(userToken);
30. }
31. **else** //否则投递下次介绍数据请求
32. {
33. **bool** willRaiseEvent = userToken.ConnectSocket.ReceiveAsync(userToken.ReceiveEventArgs); //投递接收请求
34. **if** (!willRaiseEvent)
35. ProcessReceive(userToken.ReceiveEventArgs);
36. }
37. }
38. **else**
39. {
40. **bool** willRaiseEvent = userToken.ConnectSocket.ReceiveAsync(userToken.ReceiveEventArgs); //投递接收请求
41. **if** (!willRaiseEvent)
42. ProcessReceive(userToken.ReceiveEventArgs);
43. }
44. }
45. }
46. **else**
47. {
48. CloseClientSocket(userToken);
49. }
50. }

由于我们制定的协议第一个字节是协议标识，因此在接收到第一个字节的时候需要绑定协议解析对象，具体代码实现如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **private** **void** BuildingSocketInvokeElement(AsyncSocketUserToken userToken)
2. {
3. **byte** flag = userToken.ReceiveEventArgs.Buffer[userToken.ReceiveEventArgs.Offset];
4. **if** (flag == (**byte**)SocketFlag.Upload)
5. userToken.AsyncSocketInvokeElement = **new** UploadSocketProtocol(**this**, userToken);
6. **else** **if** (flag == (**byte**)SocketFlag.Download)
7. userToken.AsyncSocketInvokeElement = **new** DownloadSocketProtocol(**this**, userToken);
8. **else** **if** (flag == (**byte**)SocketFlag.RemoteStream)
9. userToken.AsyncSocketInvokeElement = **new** RemoteStreamSocketProtocol(**this**, userToken);
10. **else** **if** (flag == (**byte**)SocketFlag.Throughput)
11. userToken.AsyncSocketInvokeElement = **new** ThroughputSocketProtocol(**this**, userToken);
12. **else** **if** (flag == (**byte**)SocketFlag.Control)
13. userToken.AsyncSocketInvokeElement = **new** ControlSocketProtocol(**this**, userToken);
14. **else** **if** (flag == (**byte**)SocketFlag.LogOutput)
15. userToken.AsyncSocketInvokeElement = **new** LogOutputSocketProtocol(**this**, userToken);
16. **if** (userToken.AsyncSocketInvokeElement != **null**)
17. {
18. Program.Logger.InfoFormat("Building socket invoke element {0}.Local Address: {1}, Remote Address: {2}",
19. userToken.AsyncSocketInvokeElement, userToken.ConnectSocket.LocalEndPoint, userToken.ConnectSocket.RemoteEndPoint);
20. }
21. }

发送响应函数实现需要注意，我们是把发送数据放到一个列表中，当上一个发送事件完成响应Completed事件，这时我们需要检测发送队列中是否存在未发送的数据，如果存在则继续发送。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **private** **bool** ProcessSend(SocketAsyncEventArgs sendEventArgs)
2. {
3. AsyncSocketUserToken userToken = sendEventArgs.UserToken **as** AsyncSocketUserToken;
4. **if** (userToken.AsyncSocketInvokeElement == **null**)
5. **return** **false**;
6. userToken.ActiveDateTime = DateTime.Now;
7. **if** (sendEventArgs.SocketError == SocketError.Success)
8. **return** userToken.AsyncSocketInvokeElement.SendCompleted(); //调用子类回调函数
9. **else**
10. {
11. CloseClientSocket(userToken);
12. **return** **false**;
13. }
14. }

SendCompleted用于回调下次需要发送的数据，具体实现过程如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **public** **virtual** **bool** SendCompleted()
2. {
3. m\_activeDT = DateTime.UtcNow;
4. m\_sendAsync = **false**;
5. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
6. asyncSendBufferManager.ClearFirstPacket(); //清除已发送的包
7. **int** offset = 0;
8. **int** count = 0;
9. **if** (asyncSendBufferManager.GetFirstPacket(**ref** offset, **ref** count))
10. {
11. m\_sendAsync = **true**;
12. **return** m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
13. asyncSendBufferManager.DynamicBufferManager.Buffer, offset, count);
14. }
15. **else**
16. **return** SendCallback();
17. }
19. //发送回调函数，用于连续下发数据
20. **public** **virtual** **bool** SendCallback()
21. {
22. **return** **true**;
23. }

当一个SocketAsyncEventArgs断开后，我们需要断开对应的Socket连接，并释放对应资源，具体实现函数如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17557341) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17557341)

[在CODE上查看代码片](https://code.csdn.net/snippets/216730)

1. **public** **void** CloseClientSocket(AsyncSocketUserToken userToken)
2. {
3. **if** (userToken.ConnectSocket == **null**)
4. **return**;
5. **string** socketInfo = **string**.Format("Local Address: {0} Remote Address: {1}", userToken.ConnectSocket.LocalEndPoint,
6. userToken.ConnectSocket.RemoteEndPoint);
7. Program.Logger.InfoFormat("Client connection disconnected. {0}", socketInfo);
8. **try**
9. {
10. userToken.ConnectSocket.Shutdown(SocketShutdown.Both);
11. }
12. **catch** (Exception E)
13. {
14. Program.Logger.ErrorFormat("CloseClientSocket Disconnect client {0} error, message: {1}", socketInfo, E.Message);
15. }
16. userToken.ConnectSocket.Close();
17. userToken.ConnectSocket = **null**; //释放引用，并清理缓存，包括释放协议对象等资源
19. m\_maxNumberAcceptedClients.Release();
20. m\_asyncSocketUserTokenPool.Push(userToken);
21. m\_asyncSocketUserTokenList.Remove(userToken);
22. }

3、SocketAsyncEventArgs封装和MSDN的不同点

MSDN在http://msdn.microsoft.com/zh-cn/library/system[**.NET**](http://lib.csdn.net/base/dotnet).sockets.socketasynceventargs(v=vs.110).aspx实现了示例代码，并实现了初步的池化处理，我们是在它的基础上扩展实现了接收数据缓冲，发送数据队列，并把发送SocketAsyncEventArgs和接收SocketAsyncEventArgs分开，并实现了协议解析单元，这样做的好处是方便后续逻辑实现文件的上传，下载和日志输出。

DEMO下载地址：[http://download.csdn](http://download.csdn.net/detail/sqldebug_fan/7467745)[**.Net**](http://lib.csdn.net/base/dotnet)/detail/sqldebug\_fan/7467745  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.c

* 上一篇[C#高性能大容量SOCKET并发（一）：IOCP完成端口例子介绍](http://blog.csdn.net/sqldebug_fan/article/details/17556353)
* 下一篇[C#高性能大容量SOCKET并发（三）：接收、发送](http://blog.csdn.net/sqldebug_fan/article/details/17558071)

[[置顶] C#高性能大容量SOCKET并发（三）：接收、发送](http://blog.csdn.net/sqldebug_fan/article/details/17558071)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[socket](http://www.csdn.net/tag/socket)[IOCP](http://www.csdn.net/tag/IOCP)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)

2014-03-04 16:57 16422人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/17558071#comments)(40) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/17558071#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

异步数据接收有可能收到的数据不是一个完整包，或者接收到的数据超过一个包的大小，因此我们需要把接收的数据进行缓存。异步发送我们也需要把每个发送的包加入到一个队列，然后通过队列逐个发送出去，如果每个都实时发送，有可能造成上一个数据包未发送完成，这时再调用SendAsync会抛出异常，提示SocketAsyncEventArgs正在进行异步操作，因此我们需要建立接收缓存和发送缓存。

**接收**

通过Completed事件响应后调用AsyncSocketInvokeElement.ProcessReceive，在ProcessReceive中，我们把收到数据先写入一个缓存，然后进行分包，分包后压给包处理函数ProcessPacket，ProcessPacket函数然后调用ProcessCommand处理具体的命令，也是各个协议实现业务逻辑的地方，具体代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558071) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558071)

[在CODE上查看代码片](https://code.csdn.net/snippets/216972)

1. **public** **virtual** **bool** ProcessReceive(**byte**[] buffer, **int** offset, **int** count) //接收异步事件返回的数据，用于对数据进行缓存和分包
2. {
3. m\_activeDT = DateTime.UtcNow;
4. DynamicBufferManager receiveBuffer = m\_asyncSocketUserToken.ReceiveBuffer;
6. receiveBuffer.WriteBuffer(buffer, offset, count);
7. **if** (receiveBuffer.DataCount > **sizeof**(**int**))
8. {
9. //按照长度分包
10. **int** packetLength = BitConverter.ToInt32(receiveBuffer.Buffer, 0); //获取包长度
11. **if** (NetByteOrder)
12. packetLength = System.Net.IPAddress.NetworkToHostOrder(packetLength); //把网络字节顺序转为本地字节顺序

15. **if** ((packetLength > 10 \* 1024 \* 1024) | (receiveBuffer.DataCount > 10 \* 1024 \* 1024)) //最大Buffer异常保护
16. **return** **false**;
18. **if** ((receiveBuffer.DataCount - **sizeof**(**int**)) >= packetLength) //收到的数据达到包长度
19. {
20. **bool** result = ProcessPacket(receiveBuffer.Buffer, **sizeof**(**int**), packetLength);
21. **if** (result)
22. receiveBuffer.Clear(packetLength + **sizeof**(**int**)); //从缓存中清理
23. **return** result;
24. }
25. **else**
26. {
27. **return** **true**;
28. }
29. }
30. **else**
31. {
32. **return** **true**;
33. }
34. }
36. **public** **virtual** **bool** ProcessPacket(**byte**[] buffer, **int** offset, **int** count) //处理分完包后的数据，把命令和数据分开，并对命令进行解析
37. {
38. **if** (count < **sizeof**(**int**))
39. **return** **false**;
40. **int** commandLen = BitConverter.ToInt32(buffer, offset); //取出命令长度
41. **string** tmpStr = Encoding.UTF8.GetString(buffer, offset + **sizeof**(**int**), commandLen);
42. **if** (!m\_incomingDataParser.DecodeProtocolText(tmpStr)) //解析命令
43. **return** **false**;
45. **return** ProcessCommand(buffer, offset + **sizeof**(**int**) + commandLen, count - **sizeof**(**int**) - commandLen); //处理命令
46. }
48. **public** **virtual** **bool** ProcessCommand(**byte**[] buffer, **int** offset, **int** count) //处理具体命令，子类从这个方法继承，buffer是收到的数据
49. {
50. **return** **true**;
51. }

**发送**

通过Completed事件响应后调用AsyncSocketInvokeElement.SendCompleted，在SendCompleted中我们需要在队列中清除已发送的包，并检测是否还有剩余需要发送的数据包，如果有，则继续发送，具体实现如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558071) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558071)

[在CODE上查看代码片](https://code.csdn.net/snippets/216972)

1. **public** **virtual** **bool** SendCompleted()
2. {
3. m\_activeDT = DateTime.UtcNow;
4. m\_sendAsync = **false**;
5. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
6. asyncSendBufferManager.ClearFirstPacket(); //清除已发送的包
7. **int** offset = 0;
8. **int** count = 0;
9. **if** (asyncSendBufferManager.GetFirstPacket(**ref** offset, **ref** count))
10. {
11. m\_sendAsync = **true**;
12. **return** m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
13. asyncSendBufferManager.DynamicBufferManager.Buffer, offset, count);
14. }
15. **else**
16. **return** SendCallback();
17. }
19. //发送回调函数，用于连续下发数据
20. **public** **virtual** **bool** SendCallback()
21. {
22. **return** **true**;
23. }

在AsyncSocketInvokeElement中提供函数给子类发送数据，业务逻辑是把当前数据包写入缓存，并检测当前是否正在发送包，如果正在发送，则等待回调，如果没有正在发送的数据包，则投递发送请求。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558071) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558071)

[在CODE上查看代码片](https://code.csdn.net/snippets/216972)

1. **public** **bool** DoSendResult()
2. {
3. **string** commandText = m\_outgoingDataAssembler.GetProtocolText();
4. **byte**[] bufferUTF8 = Encoding.UTF8.GetBytes(commandText);
5. **int** totalLength = **sizeof**(**int**) + bufferUTF8.Length; //获取总大小
6. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
7. asyncSendBufferManager.StartPacket();
8. asyncSendBufferManager.DynamicBufferManager.WriteInt(totalLength, **false**); //写入总大小
9. asyncSendBufferManager.DynamicBufferManager.WriteInt(bufferUTF8.Length, **false**); //写入命令大小
10. asyncSendBufferManager.DynamicBufferManager.WriteBuffer(bufferUTF8); //写入命令内容
11. asyncSendBufferManager.EndPacket();
13. **bool** result = **true**;
14. **if** (!m\_sendAsync)
15. {
16. **int** packetOffset = 0;
17. **int** packetCount = 0;
18. **if** (asyncSendBufferManager.GetFirstPacket(**ref** packetOffset, **ref** packetCount))
19. {
20. m\_sendAsync = **true**;
21. result = m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
22. asyncSendBufferManager.DynamicBufferManager.Buffer, packetOffset, packetCount);
23. }
24. }
25. **return** result;
26. }
28. **public** **bool** DoSendResult(**byte**[] buffer, **int** offset, **int** count)
29. {
30. **string** commandText = m\_outgoingDataAssembler.GetProtocolText();
31. **byte**[] bufferUTF8 = Encoding.UTF8.GetBytes(commandText);
32. **int** totalLength = **sizeof**(**int**) + bufferUTF8.Length + count; //获取总大小
33. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
34. asyncSendBufferManager.StartPacket();
35. asyncSendBufferManager.DynamicBufferManager.WriteInt(totalLength, **false**); //写入总大小
36. asyncSendBufferManager.DynamicBufferManager.WriteInt(bufferUTF8.Length, **false**); //写入命令大小
37. asyncSendBufferManager.DynamicBufferManager.WriteBuffer(bufferUTF8); //写入命令内容
38. asyncSendBufferManager.DynamicBufferManager.WriteBuffer(buffer, offset, count); //写入二进制数据
39. asyncSendBufferManager.EndPacket();
41. **bool** result = **true**;
42. **if** (!m\_sendAsync)
43. {
44. **int** packetOffset = 0;
45. **int** packetCount = 0;
46. **if** (asyncSendBufferManager.GetFirstPacket(**ref** packetOffset, **ref** packetCount))
47. {
48. m\_sendAsync = **true**;
49. result = m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
50. asyncSendBufferManager.DynamicBufferManager.Buffer, packetOffset, packetCount);
51. }
52. }
53. **return** result;
54. }
56. **public** **bool** DoSendBuffer(**byte**[] buffer, **int** offset, **int** count)
57. {
58. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
59. asyncSendBufferManager.StartPacket();
60. asyncSendBufferManager.DynamicBufferManager.WriteBuffer(buffer, offset, count);
61. asyncSendBufferManager.EndPacket();
63. **bool** result = **true**;
64. **if** (!m\_sendAsync)
65. {
66. **int** packetOffset = 0;
67. **int** packetCount = 0;
68. **if** (asyncSendBufferManager.GetFirstPacket(**ref** packetOffset, **ref** packetCount))
69. {
70. m\_sendAsync = **true**;
71. result = m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
72. asyncSendBufferManager.DynamicBufferManager.Buffer, packetOffset, packetCount);
73. }
74. }
75. **return** result;
76. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（二）：SocketAsyncEventArgs封装](http://blog.csdn.net/sqldebug_fan/article/details/17557341)
* 下一篇[C#高性能大容量SOCKET并发（四）：缓存设计](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

[[置顶] C#高性能大容量SOCKET并发（四）：缓存设计](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

标签： [C#](http://www.csdn.net/tag/C%23)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[SocketAsyncEventArgs](http://www.csdn.net/tag/SocketAsyncEventArgs)[socket](http://www.csdn.net/tag/socket)

2014-03-05 17:30 11393人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/17558027#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/17558027#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

在编写服务端大并发的应用程序，需要非常注意缓存设计，缓存的设计是一个折衷的结果，需要通过并发[**测试**](http://lib.csdn.net/base/softwaretest)反复验证。有很多服务程序是在启动时申请足够的内存空间，避免在运行期间再申请空间，这种是固定空间申请。还有一种是在运行期间动态增长的缓存设计，随着运行动态申请内存，这种事动态空间申请。这两种机制各有优劣，固定空间申请优点是效率高，运行稳定，缺点是对应用场景具有限制；动态空间申请优点是能适应更好的应用场景，缺点是效率相对低一些，并发数降一些；这种性能下降不是太明显，毕竟申请释放内存的效率NET是有优化的，具体需要根据应用场景设计。

在C#版IOCP中我们结合了固定缓存设计和动态缓存设计，其中服务端支持连接数使用了固定缓存设计（AsyncSocketUserTokenPool），根据程序启动时设置的最大连接数申请固定个数的对象。其中接收数据缓存（DynamicBufferManager m\_receiveBuffer）、发送数据列表（AsyncSendBufferManager m\_sendBuffer）是随着接收数据大小动态增长。

**固定缓存设计**

固定缓存设计我们需要建立一个列表进行，并在初始化的时候加入到列表中，实现非常简单，列出代码供参考。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558027) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

[在CODE上查看代码片](https://code.csdn.net/snippets/219155)

1. **public** **class** AsyncSocketUserTokenPool
2. {
3. **private** Stack<AsyncSocketUserToken> m\_pool;
5. **public** AsyncSocketUserTokenPool(**int** capacity)
6. {
7. m\_pool = **new** Stack<AsyncSocketUserToken>(capacity);
8. }
10. **public** **void** Push(AsyncSocketUserToken item)
11. {
12. **if** (item == **null**)
13. {
14. **throw** **new** ArgumentException("Items added to a AsyncSocketUserToken cannot be null");
15. }
16. **lock** (m\_pool)
17. {
18. m\_pool.Push(item);
19. }
20. }
22. **public** AsyncSocketUserToken Pop()
23. {
24. **lock** (m\_pool)
25. {
26. **return** m\_pool.Pop();
27. }
28. }
30. **public** **int** Count
31. {
32. **get** { **return** m\_pool.Count; }
33. }
34. }

初始化加入列表的代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558027) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

[在CODE上查看代码片](https://code.csdn.net/snippets/219155)

1. **public** **void** Init()
2. {
3. AsyncSocketUserToken userToken;
4. **for** (**int** i = 0; i < m\_numConnections; i++) //按照连接数建立读写对象
5. {
6. userToken = **new** AsyncSocketUserToken(m\_receiveBufferSize);
7. userToken.ReceiveEventArgs.Completed += **new** EventHandler<SocketAsyncEventArgs>(IO\_Completed);
8. userToken.SendEventArgs.Completed += **new** EventHandler<SocketAsyncEventArgs>(IO\_Completed);
9. m\_asyncSocketUserTokenPool.Push(userToken);
10. }
11. }

**动态缓存设计**

动态缓存是随着数据量大小动态增长，申请的内存在运行过程中重复利用，不释放，这样对内存只进行读写，不进行申请和释放，整体性能较高，因为内存申请释放比读写的效率低很多，因为申请释放内存需要进行加锁，进行系统内核和用户切换，因此使用动态缓存可以降低内核和用户态切换，提高性能。动态缓存的代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558027) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

[在CODE上查看代码片](https://code.csdn.net/snippets/219155)

1. **public** **class** DynamicBufferManager
2. {
3. **public** **byte**[] Buffer { **get**; **set**; } //存放内存的数组
4. **public** **int** DataCount { **get**; **set**; } //写入数据大小
6. **public** DynamicBufferManager(**int** bufferSize)
7. {
8. DataCount = 0;
9. Buffer = **new** **byte**[bufferSize];
10. }
12. **public** **int** GetDataCount() //获得当前写入的字节数
13. {
14. **return** DataCount;
15. }
17. **public** **int** GetReserveCount() //获得剩余的字节数
18. {
19. **return** Buffer.Length - DataCount;
20. }
22. **public** **void** Clear(**int** count) //清理指定大小的数据
23. {
24. **if** (count >= DataCount) //如果需要清理的数据大于现有数据大小，则全部清理
25. {
26. DataCount = 0;
27. }
28. **else**
29. {
30. **for** (**int** i = 0; i < DataCount - count; i++) //否则后面的数据往前移
31. {
32. Buffer[i] = Buffer[count + i];
33. }
34. DataCount = DataCount - count;
35. }
36. }
38. **public** **void** WriteBuffer(**byte**[] buffer, **int** offset, **int** count)
39. {
40. **if** (GetReserveCount() >= count) //缓冲区空间够，不需要申请
41. {
42. Array.Copy(buffer, offset, Buffer, DataCount, count);
43. DataCount = DataCount + count;
44. }
45. **else** //缓冲区空间不够，需要申请更大的内存，并进行移位
46. {
47. **int** totalSize = Buffer.Length + count - GetReserveCount(); //总大小-空余大小
48. **byte**[] tmpBuffer = **new** **byte**[totalSize];
49. Array.Copy(Buffer, 0, tmpBuffer, 0, DataCount); //复制以前的数据
50. Array.Copy(buffer, offset, tmpBuffer, DataCount, count); //复制新写入的数据
51. DataCount = DataCount + count;
52. Buffer = tmpBuffer; //替换
53. }
54. }
56. **public** **void** WriteBuffer(**byte**[] buffer)
57. {
58. WriteBuffer(buffer, 0, buffer.Length);
59. }
61. **public** **void** WriteShort(**short** value, **bool** convert)
62. {
63. **if** (convert)
64. {
65. value = System.Net.IPAddress.HostToNetworkOrder(value); //NET是小头结构，网络字节是大头结构，需要客户端和服务器约定好
66. }
67. **byte**[] tmpBuffer = BitConverter.GetBytes(value);
68. WriteBuffer(tmpBuffer);
69. }
71. **public** **void** WriteInt(**int** value, **bool** convert)
72. {
73. **if** (convert)
74. {
75. value = System.Net.IPAddress.HostToNetworkOrder(value); //NET是小头结构，网络字节是大头结构，需要客户端和服务器约定好
76. }
77. **byte**[] tmpBuffer = BitConverter.GetBytes(value);
78. WriteBuffer(tmpBuffer);
79. }
81. **public** **void** WriteLong(**long** value, **bool** convert)
82. {
83. **if** (convert)
84. {
85. value = System.Net.IPAddress.HostToNetworkOrder(value); //NET是小头结构，网络字节是大头结构，需要客户端和服务器约定好
86. }
87. **byte**[] tmpBuffer = BitConverter.GetBytes(value);
88. WriteBuffer(tmpBuffer);
89. }
91. **public** **void** WriteString(**string** value) //文本全部转成UTF8，UTF8兼容性好
92. {
93. **byte**[] tmpBuffer = Encoding.UTF8.GetBytes(value);
94. WriteBuffer(tmpBuffer);
95. }
96. }

**异步发送列表**

异步发送列表是在动态缓存的基础上加了一个列表管理，记录每个包的位置信息，并提供管理函数，代码示例如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/17558027) [copy](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

[在CODE上查看代码片](https://code.csdn.net/snippets/219155)

1. **namespace** SocketAsyncSvr
2. {
3. **struct** SendBufferPacket
4. {
5. **public** **int** Offset;
6. **public** **int** Count;
7. }
9. //由于是异步发送，有可能接收到两个命令，写入了两次返回，发送需要等待上一次回调才发下一次的响应
10. **public** **class** AsyncSendBufferManager
11. {
12. **private** DynamicBufferManager m\_dynamicBufferManager;
13. **public** DynamicBufferManager DynamicBufferManager { **get** { **return** m\_dynamicBufferManager; } }
14. **private** List<SendBufferPacket> m\_sendBufferList;
15. **private** SendBufferPacket m\_sendBufferPacket;
17. **public** AsyncSendBufferManager(**int** bufferSize)
18. {
19. m\_dynamicBufferManager = **new** DynamicBufferManager(bufferSize);
20. m\_sendBufferList = **new** List<SendBufferPacket>();
21. m\_sendBufferPacket.Offset = 0;
22. m\_sendBufferPacket.Count = 0;
23. }
25. **public** **void** StartPacket()
26. {
27. m\_sendBufferPacket.Offset = m\_dynamicBufferManager.DataCount;
28. m\_sendBufferPacket.Count = 0;
29. }
31. **public** **void** EndPacket()
32. {
33. m\_sendBufferPacket.Count = m\_dynamicBufferManager.DataCount - m\_sendBufferPacket.Offset;
34. m\_sendBufferList.Add(m\_sendBufferPacket);
35. }
37. **public** **bool** GetFirstPacket(**ref** **int** offset, **ref** **int** count)
38. {
39. **if** (m\_sendBufferList.Count <= 0)
40. **return** **false**;
41. offset = m\_sendBufferList[0].Offset;
42. count = m\_sendBufferList[0].Count;
43. **return** **true**;
44. }
46. **public** **bool** ClearFirstPacket()
47. {
48. **if** (m\_sendBufferList.Count <= 0)
49. **return** **false**;
50. **int** count = m\_sendBufferList[0].Count;
51. m\_dynamicBufferManager.Clear(count);
52. m\_sendBufferList.RemoveAt(0);
53. **return** **true**;
54. }
56. **public** **void** ClearPacket()
57. {
58. m\_sendBufferList.Clear();
59. m\_dynamicBufferManager.Clear(m\_dynamicBufferManager.DataCount);
60. }
61. }
62. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

* 上一篇[C#高性能大容量SOCKET并发（三）：接收、发送](http://blog.csdn.net/sqldebug_fan/article/details/17558071)
* 下一篇[C#高性能大容量SOCKET并发（五）：粘包、分包、解包](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

[[置顶] C#高性能大容量SOCKET并发（五）：粘包、分包、解包](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[粘包分包解包](http://www.csdn.net/tag/%e7%b2%98%e5%8c%85%e5%88%86%e5%8c%85%e8%a7%a3%e5%8c%85)

2014-03-07 11:40 18120人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20465455#comments)(10) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20465455#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

粘包

使用TCP长连接就会引入粘包的问题，粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾。粘包可能由发送方造成，也可能由接收方造成。TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据，造成多个数据包的粘连。如果接收进程不及时接收数据，已收到的数据就放在系统接收缓冲区，用户进程读取数据时就可能同时读到多个数据包。

粘包一般的解决办法是制定通讯协议，由协议来规定如何分包解包。

分包

在NETIOCPDemo例子程序中，我们分包的逻辑是先发一个长度，然后紧接着是数据包内容，这样就可以把每个包分开。

应用层数据包格式如下：

|  |  |
| --- | --- |
| 应用层数据包格式 |  |
| 数据包长度Len：Cardinal（4字节无符号整数） | 数据包内容，长度为Len |

AsyncSocketInvokeElement分包处理主要代码，我们收到的数据都是在ProcessReceive方法中处理，处理的方法是把收到的数据存到缓冲区数组中，然后取前4个字节为长度，如果剩下的字节数大于等于长度，则取到一个完整包，进行后续逻辑处理，如果取到的不够一个包，则不处理，等待后续包接收，具体代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465455) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

[在CODE上查看代码片](https://code.csdn.net/snippets/222632)

1. **public** **virtual** **bool** ProcessReceive(**byte**[] buffer, **int** offset, **int** count) //接收异步事件返回的数据，用于对数据进行缓存和分包
2. {
3. m\_activeDT = DateTime.UtcNow;
4. DynamicBufferManager receiveBuffer = m\_asyncSocketUserToken.ReceiveBuffer;
6. receiveBuffer.WriteBuffer(buffer, offset, count);
7. **if** (receiveBuffer.DataCount > **sizeof**(**int**))
8. {
9. //按照长度分包
10. **int** packetLength = BitConverter.ToInt32(receiveBuffer.Buffer, 0); //获取包长度
11. **if** (NetByteOrder)
12. packetLength = System.Net.IPAddress.NetworkToHostOrder(packetLength); //把网络字节顺序转为本地字节顺序

15. **if** ((packetLength > 10 \* 1024 \* 1024) | (receiveBuffer.DataCount > 10 \* 1024 \* 1024)) //最大Buffer异常保护
16. **return** **false**;
18. **if** ((receiveBuffer.DataCount - **sizeof**(**int**)) >= packetLength) //收到的数据达到包长度
19. {
20. **bool** result = ProcessPacket(receiveBuffer.Buffer, **sizeof**(**int**), packetLength);
21. **if** (result)
22. receiveBuffer.Clear(packetLength + **sizeof**(**int**)); //从缓存中清理
23. **return** result;
24. }
25. **else**
26. {
27. **return** **true**;
28. }
29. }
30. **else**
31. {
32. **return** **true**;
33. }
34. }

解包

由于我们应用层数据包既可以传命令也可以传数据，因而针对每个包我们进行解包，分出命令和数据分别处理，因而每个Socket服务对象都需要解包，我们解包的逻辑是放在ProcessPacket中，命令和数据的包格式为：

|  |  |  |
| --- | --- | --- |
| 命令长度Len：Cardinal（4字节无符号整数） | 命令 | 数据 |

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465455) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

[在CODE上查看代码片](https://code.csdn.net/snippets/222632)

1. **public** **virtual** **bool** ProcessPacket(**byte**[] buffer, **int** offset, **int** count) //处理分完包后的数据，把命令和数据分开，并对命令进行解析
2. {
3. **if** (count < **sizeof**(**int**))
4. **return** **false**;
5. **int** commandLen = BitConverter.ToInt32(buffer, offset); //取出命令长度
6. **string** tmpStr = Encoding.UTF8.GetString(buffer, offset + **sizeof**(**int**), commandLen);
7. **if** (!m\_incomingDataParser.DecodeProtocolText(tmpStr)) //解析命令
8. **return** **false**;
10. **return** ProcessCommand(buffer, offset + **sizeof**(**int**) + commandLen, count - **sizeof**(**int**) - commandLen); //处理命令
11. }

每个包中包含多个协议关键字，每个协议关键字用回车换行分开，因此我们需要调用文本分开函数，然后针对每条命令解析出关键字和值，具体代码在IncomingDataParser.DecodeProtocolText如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465455) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

[在CODE上查看代码片](https://code.csdn.net/snippets/222632)

1. **public** **bool** DecodeProtocolText(**string** protocolText)
2. {
3. m\_header = "";
4. m\_names.Clear();
5. m\_values.Clear();
6. **int** speIndex = protocolText.IndexOf(ProtocolKey.ReturnWrap);
7. **if** (speIndex < 0)
8. {
9. **return** **false**;
10. }
11. **else**
12. {
13. **string**[] tmpNameValues = protocolText.Split(**new** **string**[] { ProtocolKey.ReturnWrap }, StringSplitOptions.RemoveEmptyEntries);
14. **if** (tmpNameValues.Length < 2) //每次命令至少包括两行
15. **return** **false**;
16. **for** (**int** i = 0; i < tmpNameValues.Length; i++)
17. {
18. **string**[] tmpStr = tmpNameValues[i].Split(**new** **string**[] { ProtocolKey.EqualSign }, StringSplitOptions.None);
19. **if** (tmpStr.Length > 1) //存在等号
20. {
21. **if** (tmpStr.Length > 2) //超过两个等号，返回失败
22. **return** **false**;
23. **if** (tmpStr[0].Equals(ProtocolKey.Command, StringComparison.CurrentCultureIgnoreCase))
24. {
25. m\_command = tmpStr[1];
26. }
27. **else**
28. {
29. m\_names.Add(tmpStr[0].ToLower());
30. m\_values.Add(tmpStr[1]);
31. }
32. }
33. }
34. **return** **true**;
35. }
36. }

处理命令

解析出命令后，需要对每个命令进行处理，各个协议实现类从AsyncSocketInvokeElement.ProcessCommand继承，然后编写各自协议处理逻辑，如吞吐量的[**测试**](http://lib.csdn.net/base/softwaretest)协议逻辑实现代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465455) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465455)

[在CODE上查看代码片](https://code.csdn.net/snippets/222632)

1. **namespace** SocketAsyncSvr
2. {
3. **class** ThroughputSocketProtocol : BaseSocketProtocol
4. {
5. **public** ThroughputSocketProtocol(AsyncSocketServer asyncSocketServer, AsyncSocketUserToken asyncSocketUserToken)
6. : **base**(asyncSocketServer, asyncSocketUserToken)
7. {
8. m\_socketFlag = "Throughput";
9. }
11. **public** **override** **void** Close()
12. {
13. **base**.Close();
14. }
16. **public** **override** **bool** ProcessCommand(**byte**[] buffer, **int** offset, **int** count) //处理分完包的数据，子类从这个方法继承
17. {
18. ThroughputSocketCommand command = StrToCommand(m\_incomingDataParser.Command);
19. m\_outgoingDataAssembler.Clear();
20. m\_outgoingDataAssembler.AddResponse();
21. m\_outgoingDataAssembler.AddCommand(m\_incomingDataParser.Command);
22. **if** (command == ThroughputSocketCommand.CyclePacket)
23. **return** DoCyclePacket(buffer, offset, count);
24. **else**
25. {
26. Program.Logger.Error("Unknow command: " + m\_incomingDataParser.Command);
27. **return** **false**;
28. }
29. }
31. **public** ThroughputSocketCommand StrToCommand(**string** command)
32. {
33. **if** (command.Equals(ProtocolKey.CyclePacket, StringComparison.CurrentCultureIgnoreCase))
34. **return** ThroughputSocketCommand.CyclePacket;
35. **else**
36. **return** ThroughputSocketCommand.None;
37. }
39. **public** **bool** DoCyclePacket(**byte**[] buffer, **int** offset, **int** count)
40. {
41. **int** cycleCount = 0;
42. **if** (m\_incomingDataParser.GetValue(ProtocolKey.Count, **ref** cycleCount))
43. {
44. m\_outgoingDataAssembler.AddSuccess();
45. cycleCount = cycleCount + 1;
46. m\_outgoingDataAssembler.AddValue(ProtocolKey.Count, cycleCount);
47. }
48. **else**
49. m\_outgoingDataAssembler.AddFailure(ProtocolCode.ParameterError, "");
50. **return** DoSendResult(buffer, offset, count);
51. }
52. }
53. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

上一篇[C#高性能大容量SOCKET并发（四）：缓存设计](http://blog.csdn.net/sqldebug_fan/article/details/17558027)

* 下一篇[C#高性能大容量SOCKET并发（六）：超时Socket断开（守护线程）和心跳包](http://blog.csdn.net/sqldebug_fan/article/details/20465195)

[[置顶] C#高性能大容量SOCKET并发（六）：超时Socket断开（守护线程）和心跳包](http://blog.csdn.net/sqldebug_fan/article/details/20465195)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[socket](http://www.csdn.net/tag/socket)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)

2014-03-07 15:06 12005人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20465195#comments)(28) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20465195#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**守护线程**

在服务端版Socket编程需要处理长时间没有发送数据的Socket，需要在超时多长时间后断开连接，我们需要独立一个线程（DaemonThread）来轮询，在执行断开时，需要把Socket对象锁定，并调用CloseClientSocket来断开连接，具体处理代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465195) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465195)

[在CODE上查看代码片](https://code.csdn.net/snippets/222997)

1. **namespace** SocketAsyncSvr
2. {
3. **class** DaemonThread : Object
4. {
5. **private** Thread m\_thread;
6. **private** AsyncSocketServer m\_asyncSocketServer;
8. **public** DaemonThread(AsyncSocketServer asyncSocketServer)
9. {
10. m\_asyncSocketServer = asyncSocketServer;
11. m\_thread = **new** Thread(DaemonThreadStart);
12. m\_thread.Start();
13. }
15. **public** **void** DaemonThreadStart()
16. {
17. **while** (m\_thread.IsAlive)
18. {
19. AsyncSocketUserToken[] userTokenArray = **null**;
20. m\_asyncSocketServer.AsyncSocketUserTokenList.CopyList(**ref** userTokenArray);
21. **for** (**int** i = 0; i < userTokenArray.Length; i++)
22. {
23. **if** (!m\_thread.IsAlive)
24. **break**;
25. **try**
26. {
27. **if** ((DateTime.Now - userTokenArray[i].ActiveDateTime).Milliseconds > m\_asyncSocketServer.SocketTimeOutMS) //超时Socket断开
28. {
29. **lock** (userTokenArray[i])
30. {
31. m\_asyncSocketServer.CloseClientSocket(userTokenArray[i]);
32. }
33. }
34. }
35. **catch** (Exception E)
36. {
37. Program.Logger.ErrorFormat("Daemon thread check timeout socket error, message: {0}", E.Message);
38. Program.Logger.Error(E.StackTrace);
39. }
40. }
42. **for** (**int** i = 0; i < 60 \* 1000 / 10; i++) //每分钟检测一次
43. {
44. **if** (!m\_thread.IsAlive)
45. **break**;
46. Thread.Sleep(10);
47. }
48. }
49. }
51. **public** **void** Close()
52. {
53. m\_thread.Abort();
54. m\_thread.Join();
55. }
56. }
57. }

**心跳包**

有超时连接，相对应的需要设计心跳包，心跳包用来检测连接和维护连接状态，心跳包的原理是客户端发送一个包给服务器，服务器收到后发一个响应包给客户端，通过检测是否有返回来判断连接是否正常，心跳包实现放在BaseSocketProtocol.DoActive方法中。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465195) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465195)

[在CODE上查看代码片](https://code.csdn.net/snippets/222997)

1. **public** **bool** DoActive()
2. {
3. m\_outgoingDataAssembler.AddSuccess();
4. **return** DoSendResult();
5. }

具体由各个协议是否决定调用，如控制协议ControlSocketProtocol实现心跳协议如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465195) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465195)

[在CODE上查看代码片](https://code.csdn.net/snippets/222997)

1. **public** **override** **bool** ProcessCommand(**byte**[] buffer, **int** offset, **int** count) //处理分完包的数据，子类从这个方法继承
2. {
3. ControlSocketCommand command = StrToCommand(m\_incomingDataParser.Command);
4. m\_outgoingDataAssembler.Clear();
5. m\_outgoingDataAssembler.AddResponse();
6. m\_outgoingDataAssembler.AddCommand(m\_incomingDataParser.Command);
7. **if** (!CheckLogined(command)) //检测登录
8. {
9. m\_outgoingDataAssembler.AddFailure(ProtocolCode.UserHasLogined, "");
10. **return** DoSendResult();
11. }
12. **if** (command == ControlSocketCommand.Login)
13. **return** DoLogin();
14. **else** **if** (command == ControlSocketCommand.Active)
15. **return** DoActive();
16. **else** **if** (command == ControlSocketCommand.GetClients)
17. **return** DoGetClients();
18. **else**
19. {
20. Program.Logger.Error("Unknow command: " + m\_incomingDataParser.Command);
21. **return** **false**;
22. }
23. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（五）：粘包、分包、解包](http://blog.csdn.net/sqldebug_fan/article/details/20465455)
* 下一篇[C#高性能大容量SOCKET并发（七）：协议字符集](http://blog.csdn.net/sqldebug_fan/article/details/20465297)

[[置顶] C#高性能大容量SOCKET并发（七）：协议字符集](http://blog.csdn.net/sqldebug_fan/article/details/20465297)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[协议字符集](http://www.csdn.net/tag/%e5%8d%8f%e8%ae%ae%e5%ad%97%e7%ac%a6%e9%9b%86)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)

2014-03-07 15:34 5806人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20465297#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20465297#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**UTF-8**

UTF-8是UNICODE的一种变长字符编码又称万国码，由Ken Thompson于1992年创建。现在已经标准化为RFC 3629。UTF-8用1到6个字节编码UNICODE字符。用在网页上可以同一页面显示中文简体繁体及其它语言（如日文，韩文）。

使用UTF-8的好处是现在一些手机平台都是使用UTF-8，另外在一些[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)平台，如果不支持中文，只支持英文，可以不转换，UTF-8就可以识别。

NET默认支持的编码是Unicode，为了支持UTF-8，需要转换下，为了编程简便性，我们只在发包和解包的地方进行UTF-8转换。

**收包把UTF-8转为UNICODE**

收包我们需要在分包完成后，把内存数组是UTF-8编码转换为Unicode，即为C#的string，后续的处理就都可以基于string进行处理，比较方便。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465297) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465297)

[在CODE上查看代码片](https://code.csdn.net/snippets/223071)

1. **public** **virtual** **bool** ProcessPacket(**byte**[] buffer, **int** offset, **int** count) //处理分完包后的数据，把命令和数据分开，并对命令进行解析
2. {
3. **if** (count < **sizeof**(**int**))
4. **return** **false**;
5. **int** commandLen = BitConverter.ToInt32(buffer, offset); //取出命令长度
6. <span style="color:#ff6666;">**string** tmpStr = Encoding.UTF8.GetString(buffer, offset + **sizeof**(**int**), commandLen);</span>
7. **if** (!m\_incomingDataParser.DecodeProtocolText(tmpStr)) //解析命令
8. **return** **false**;
10. **return** ProcessCommand(buffer, offset + **sizeof**(**int**) + commandLen, count - **sizeof**(**int**) - commandLen); //处理命令
11. }

**发包把UNICODE转为UTF-8**

发包我们主要调用DoSendResult，从发送缓冲中获取协议文本后，转换为UTF-8，然后写入发送列表中。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465297) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465297)

[在CODE上查看代码片](https://code.csdn.net/snippets/223071)

1. **public** **bool** DoSendResult()
2. {
3. **string** commandText = m\_outgoingDataAssembler.GetProtocolText();
4. <span style="color:#ff6666;">**byte**[] bufferUTF8 = Encoding.UTF8.GetBytes(commandText);</span>
5. **int** totalLength = **sizeof**(**int**) + bufferUTF8.Length; //获取总大小
6. AsyncSendBufferManager asyncSendBufferManager = m\_asyncSocketUserToken.SendBuffer;
7. asyncSendBufferManager.StartPacket();
8. asyncSendBufferManager.DynamicBufferManager.WriteInt(totalLength, **false**); //写入总大小
9. asyncSendBufferManager.DynamicBufferManager.WriteInt(bufferUTF8.Length, **false**); //写入命令大小
10. asyncSendBufferManager.DynamicBufferManager.WriteBuffer(bufferUTF8); //写入命令内容
11. asyncSendBufferManager.EndPacket();
13. **bool** result = **true**;
14. **if** (!m\_sendAsync)
15. {
16. **int** packetOffset = 0;
17. **int** packetCount = 0;
18. **if** (asyncSendBufferManager.GetFirstPacket(**ref** packetOffset, **ref** packetCount))
19. {
20. m\_sendAsync = **true**;
21. result = m\_asyncSocketServer.SendAsyncEvent(m\_asyncSocketUserToken.ConnectSocket, m\_asyncSocketUserToken.SendEventArgs,
22. asyncSendBufferManager.DynamicBufferManager.Buffer, packetOffset, packetCount);
23. }
24. }
25. **return** result;
26. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（六）：超时Socket断开（守护线程）和心跳包](http://blog.csdn.net/sqldebug_fan/article/details/20465195)
* 下一篇[C#高性能大容量SOCKET并发（八）：通讯协议](http://blog.csdn.net/sqldebug_fan/article/details/20465639)

[[置顶] C#高性能大容量SOCKET并发（八）：通讯协议](http://blog.csdn.net/sqldebug_fan/article/details/20465639)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[通讯协议](http://www.csdn.net/tag/%e9%80%9a%e8%ae%af%e5%8d%8f%e8%ae%ae)

2014-03-07 15:54 7107人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20465639#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20465639#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**协议种类**

开发Socket程序有两种协议类型，一种是用文本描述的，类似HTTP协议，定义字符集，好处是兼容性和调试方便，缺点是解析文本会损耗一些性能；一种是用Code加结构体，定义字节顺序，好处是性能高，缺点是兼容性和调试不方便。这个可以根据应用场景灵活选择，如果您的应用相对稳定，需求变化少，性能要求高，则可以使用Code加结构体的方式。如果您的应用需要不停的扩充功能，但是对性能要求不苛刻，则可以使用文本解析的方式。这两种协议有两个比较典型的应用场景，Code加结构体更多应用在中间件上，因为协议的封装都是透明的，不需要联调，而且性能要求较高；文本解析则更多应用在外部交互上，如和设备、手机通讯，需要联调，但是性能要求没那么高。

我们Demo是采用文本解析的方式，具体可以根据应用灵活选择。

定义协议有以下注意点（方便不同平台接入）。

**字节顺序**

不同硬件平台或[**操作系统**](http://lib.csdn.net/base/operatingsystem)下，字节顺序是不一致的，有的是高位在前，低位在后，有的则是低位在前。Windows是低位在前，高位在后，每个平台下都有函数实现字节转换。TCP/IP定义的字节顺序是高位在前、低位在后，可以使用IPAddress类的

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465639) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465639)

[在CODE上查看代码片](https://code.csdn.net/snippets/223117)

1. //
2. // 摘要:
3. //     将整数值由网络字节顺序转换为主机字节顺序。
4. //
5. // 参数:
6. //   network:
7. //     以网络字节顺序表示的要转换的数字。
8. //
9. // 返回结果:
10. //     以主机字节顺序表示的整数值。
11. [TargetedPatchingOptOut("Performance critical to inline this type of method across NGen image boundaries")]
12. **public** **static** **int** NetworkToHostOrder(**int** network);
13. //
14. // 摘要:
15. //     将长值由网络字节顺序转换为主机字节顺序。
16. //
17. // 参数:
18. //   network:
19. //     以网络字节顺序表示的要转换的数字。
20. //
21. // 返回结果:
22. //     以主机字节顺序表示的长值。
23. [TargetedPatchingOptOut("Performance critical to inline this type of method across NGen image boundaries")]
24. **public** **static** **long** NetworkToHostOrder(**long** network);
25. //
26. // 摘要:
27. //     将短值由网络字节顺序转换为主机字节顺序。
28. //
29. // 参数:
30. //   network:
31. //     以网络字节顺序表示的要转换的数字。
32. //
33. // 返回结果:
34. //     以主机字节顺序表示的短值。
35. [TargetedPatchingOptOut("Performance critical to inline this type of method across NGen image boundaries")]
36. **public** **static** **short** NetworkToHostOrder(**short** network);

来实现网络字节顺序转为本地字节顺序，反之可以调用

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465639) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465639)

[在CODE上查看代码片](https://code.csdn.net/snippets/223117)

1. //
2. // 摘要:
3. //     将整数值由主机字节顺序转换为网络字节顺序。
4. //
5. // 参数:
6. //   host:
7. //     以主机字节顺序表示的要转换的数字。
8. //
9. // 返回结果:
10. //     以网络字节顺序表示的整数值。
11. **public** **static** **int** HostToNetworkOrder(**int** host);
12. //
13. // 摘要:
14. //     将长值由主机字节顺序转换为网络字节顺序。
15. //
16. // 参数:
17. //   host:
18. //     以主机字节顺序表示的要转换的数字。
19. //
20. // 返回结果:
21. //     以网络字节顺序表示的长值。
22. **public** **static** **long** HostToNetworkOrder(**long** host);
23. //
24. // 摘要:
25. //     将短值由主机字节顺序转换为网络字节顺序。
26. //
27. // 参数:
28. //   host:
29. //     以主机字节顺序表示的要转换的数字。
30. //
31. // 返回结果:
32. //     以网络字节顺序表示的短值。
33. **public** **static** **short** HostToNetworkOrder(**short** host);

来实现本地字节顺序转为网络字节顺序。

更流行的做法是使用网络字节顺序，这样规范统一。我们这里使用Windows字节顺序，即低位在前、高位在后，和网络字节顺序刚好相反。

**字符集**

字符集最合适的就是使用UTF-8，这个编码是免费的，对于某些不支持中文的[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)系统，全英文可以不需要转换就是UTF-8格式，对于跨平台具有优势。

**数据包格式**

Code加结构体、文本解析两种协议风格都是统一使用相同的数据包格式，即先发一个4字节的长度，后面跟着是内容，下一个也是先发一个4字节长度，接着是内容。结构体可以定义为文件流结构体，采用字节对齐。

**协议样本**

Active：检测连接（心跳包）

客户端->服务器  
{  
[Request]  
Command=Active  
}  
服务器->客户端  
{  
[Response]  
Command= Active  
Code= Error Code#错误码  
Message=Message#如果出错，返回错误描述信息  
}

DEMO下载地址：[http://download.csdn](http://download.csdn.net/detail/sqldebug_fan/7467745)[**.NET**](http://lib.csdn.net/base/dotnet)/detail/sqldebug\_fan/7467745  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（七）：协议字符集](http://blog.csdn.net/sqldebug_fan/article/details/20465297)
* 下一篇[C#高性能大容量SOCKET并发（九）：断点续传](http://blog.csdn.net/sqldebug_fan/article/details/20465731)

[[置顶] C#高性能大容量SOCKET并发（九）：断点续传](http://blog.csdn.net/sqldebug_fan/article/details/20465731)

标签： [C#](http://www.csdn.net/tag/C%23)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[断点续传](http://www.csdn.net/tag/%e6%96%ad%e7%82%b9%e7%bb%ad%e4%bc%a0)

2014-03-07 16:28 7727人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20465731#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20465731#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**上传断点续传**

断点续传主要是用在上传或下载文件，一般做法是开始上传的时候，服务器返回上次已经上传的大小，如果上传完成，则返回-1；下载开始的时候，由客户端上报本地已经下载大小，服务器根据位置信息下发数据，因此上传下载协议都需要带Size大小，例如我们协议格式。

**上传开始：**

**客户端->服务器**

{

[Request]

Command=Upload

Dir=Dir                        #目录，全路径名

FileName=FileName      #文件名（不包括路径）

}

**服务器->客户端**

{

[Response]

Command=Upload

*Code= Error Code                #错误码*

*Message=Message                 #如果出错，返回错误描述信息*

FileSize=FileSize                 #已上传文件的大小，用于续传

}

因此在接收客户端上传请求时需要下发服务器上次接收到文件地址：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465731) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465731)

[在CODE上查看代码片](https://code.csdn.net/snippets/223197)

1. **public** **bool** DoUpload()
2. {
3. **string** dirName = "";
4. **string** fileName = "";
5. **if** (m\_incomingDataParser.GetValue(ProtocolKey.DirName, **ref** dirName) & m\_incomingDataParser.GetValue(ProtocolKey.FileName, **ref** fileName))
6. {
7. **if** (dirName == "")
8. dirName = Program.FileDirectory;
9. **else**
10. dirName = Path.Combine(Program.FileDirectory, dirName);
11. fileName = Path.Combine(dirName, fileName);
12. Program.Logger.Info("Start upload file: " + fileName);
13. **if** (m\_fileStream != **null**) //关闭上次传输的文件
14. {
15. m\_fileStream.Close();
16. m\_fileStream = **null**;
17. m\_fileName = "";
18. }
19. **if** (File.Exists(fileName))
20. {
21. **if** (!CheckFileInUse(fileName)) //检测文件是否正在使用中
22. {
23. m\_fileName = fileName;
24. m\_fileStream = **new** FileStream(fileName, FileMode.Open, FileAccess.ReadWrite);
25. m\_fileStream.Position = m\_fileStream.Length; //文件移到末尾
26. m\_outgoingDataAssembler.AddSuccess();
27. m\_outgoingDataAssembler.AddValue(ProtocolKey.FileSize, m\_fileStream.Length);
28. }
29. **else**
30. {
31. m\_outgoingDataAssembler.AddFailure(ProtocolCode.FileIsInUse, "");
32. Program.Logger.Error("Start upload file error, file is in use: " + fileName);
33. }
34. }
35. **else**
36. {
37. m\_fileName = fileName;
38. m\_fileStream = **new** FileStream(fileName, FileMode.OpenOrCreate, FileAccess.ReadWrite);
39. m\_fileStream.Position = m\_fileStream.Length; //文件移到末尾
40. m\_outgoingDataAssembler.AddSuccess();
41. m\_outgoingDataAssembler.AddValue(ProtocolKey.FileSize, m\_fileStream.Length);
42. }
43. }
44. **else**
45. m\_outgoingDataAssembler.AddFailure(ProtocolCode.ParameterError, "");
46. **return** DoSendResult();
47. }

**下载断点续传**

**下载开始：**

**客户端->服务器**

{

[Request]

Command=Download

Dir=Dir                        #目录，全路径名

FileName=FileName      #文件名（不包括路径）

FileSize=FileSize          #客户端本地文件大小，用于断点续传

PacketSize=PacketSize   #下发数据包大小，单位为KB，用于速度[**测试**](http://lib.csdn.net/base/softwaretest)

}

**服务器->客户端**

{

[Response]

Command= Download

*Code= Error Code                #错误码*

*Message=Message                 #如果出错，返回错误描述信息*

}

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20465731) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20465731)

[在CODE上查看代码片](https://code.csdn.net/snippets/223197)

1. **public** **bool** DoDownload()
2. {
3. **string** dirName = "";
4. **string** fileName = "";
5. Int64 fileSize = 0;
6. **int** packetSize = 0;
7. **if** (m\_incomingDataParser.GetValue(ProtocolKey.DirName, **ref** dirName) & m\_incomingDataParser.GetValue(ProtocolKey.FileName, **ref** fileName)
8. & m\_incomingDataParser.GetValue(ProtocolKey.FileSize, **ref** fileSize) & m\_incomingDataParser.GetValue(ProtocolKey.PacketSize, **ref** packetSize))
9. {
10. **if** (dirName == "")
11. dirName = Program.FileDirectory;
12. **else**
13. dirName = Path.Combine(Program.FileDirectory, dirName);
14. fileName = Path.Combine(dirName, fileName);
15. Program.Logger.Info("Start download file: " + fileName);
16. **if** (m\_fileStream != **null**) //关闭上次传输的文件
17. {
18. m\_fileStream.Close();
19. m\_fileStream = **null**;
20. m\_fileName = "";
21. m\_sendFile = **false**;
22. }
23. **if** (File.Exists(fileName))
24. {
25. **if** (!CheckFileInUse(fileName)) //检测文件是否正在使用中
26. {
27. m\_fileName = fileName;
28. m\_fileStream = **new** FileStream(fileName, FileMode.Open, FileAccess.ReadWrite);
29. m\_fileStream.Position = fileSize; //文件移到上次下载位置
30. m\_outgoingDataAssembler.AddSuccess();
31. m\_sendFile = **true**;
32. m\_packetSize = packetSize;
33. }
34. **else**
35. {
36. m\_outgoingDataAssembler.AddFailure(ProtocolCode.FileIsInUse, "");
37. Program.Logger.Error("Start download file error, file is in use: " + fileName);
38. }
39. }
40. **else**
41. {
42. m\_outgoingDataAssembler.AddFailure(ProtocolCode.FileNotExist, "");
43. }
44. }
45. **else**
46. m\_outgoingDataAssembler.AddFailure(ProtocolCode.ParameterError, "");
47. **return** DoSendResult();
48. }

**多线程并发下载**

断点续传的一个应用就是并发下载，做法是客户端起多个线程并发请求同一个文件，每个线程下文件的一部分，全部下载完成后，把每个数据块合并为一个文件。这个服务端和客户端协议都不需要修改，只是需要做下载逻辑的更改。

**多线程并发上传**

这个需要定义通讯来支持这个逻辑，主要是服务器要提供合并多个数据文件为一个文件的协议逻辑。

DEMO下载地址：[http://download.csdn](http://download.csdn.net/detail/sqldebug_fan/7467745)[**.NET**](http://lib.csdn.net/base/dotnet)/detail/sqldebug\_fan/7467745  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（八）：通讯协议](http://blog.csdn.net/sqldebug_fan/article/details/20465639)
* 下一篇[C#高性能大容量SOCKET并发（十）：SocketAsyncEventArgs线程模型](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[[置顶] C#高性能大容量SOCKET并发（十）：SocketAsyncEventArgs线程模型](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

标签： [完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[多线程](http://www.csdn.net/tag/%e5%a4%9a%e7%ba%bf%e7%a8%8b)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[IOCP](http://www.csdn.net/tag/IOCP)

2014-03-07 17:18 8095人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/20707035#comments)(5) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/20707035#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**线程模型**

SocketAsyncEventArgs编程模式不支持设置同时工作线程个数，使用的NET的IO线程，由NET底层提供，这点和直接使用完成端口API编程不同。NET底层IO线程也是每个异步事件都是由不同的线程返回到Completed事件，因此在Completed事件需要对用户对象进行加锁，避免同一个用户对象同时触发两个Completed事件。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20707035) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[在CODE上查看代码片](https://code.csdn.net/snippets/223337)

1. **void** IO\_Completed(**object** sender, SocketAsyncEventArgs asyncEventArgs)
2. {
3. AsyncSocketUserToken userToken = asyncEventArgs.UserToken **as** AsyncSocketUserToken;
4. userToken.ActiveDateTime = DateTime.Now;
5. **try**
6. {
7. **lock** (userToken) //避免同一个userToken同时有多个线程操作
8. {
9. **if** (asyncEventArgs.LastOperation == SocketAsyncOperation.Receive)
10. ProcessReceive(asyncEventArgs);
11. **else** **if** (asyncEventArgs.LastOperation == SocketAsyncOperation.Send)
12. ProcessSend(asyncEventArgs);
13. **else**
14. **throw** **new** ArgumentException("The last operation completed on the socket was not a receive or send");
15. }
16. }
17. **catch** (Exception E)
18. {
19. Program.Logger.ErrorFormat("IO\_Completed {0} error, message: {1}", userToken.ConnectSocket, E.Message);
20. Program.Logger.Error(E.StackTrace);
21. }
22. }

使用ProceXP可以看到服务端在比较忙的时候，服务的线程会越多。在Completed事件加锁有好处是后续逻辑处理都是串行的，可以不用考虑线程同步。还有一个地方需要注意的是断开超时连接，由于超时连接会调用Shutdown函数来强行中断SOCKET，因此在守护线程调用时，也需要锁住userToken对象。

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20707035) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[在CODE上查看代码片](https://code.csdn.net/snippets/223337)

1. **public** **void** DaemonThreadStart()
2. {
3. **while** (m\_thread.IsAlive)
4. {
5. AsyncSocketUserToken[] userTokenArray = **null**;
6. m\_asyncSocketServer.AsyncSocketUserTokenList.CopyList(**ref** userTokenArray);
7. **for** (**int** i = 0; i < userTokenArray.Length; i++)
8. {
9. **if** (!m\_thread.IsAlive)
10. **break**;
11. **try**
12. {
13. **if** ((DateTime.Now - userTokenArray[i].ActiveDateTime).Milliseconds > m\_asyncSocketServer.SocketTimeOutMS) //超时Socket断开
14. {
15. **lock** (userTokenArray[i])
16. {
17. m\_asyncSocketServer.CloseClientSocket(userTokenArray[i]);
18. }
19. }
20. }
21. **catch** (Exception E)
22. {
23. Program.Logger.ErrorFormat("Daemon thread check timeout socket error, message: {0}", E.Message);
24. Program.Logger.Error(E.StackTrace);
25. }
26. }
28. **for** (**int** i = 0; i < 60 \* 1000 / 10; i++) //每分钟检测一次
29. {
30. **if** (!m\_thread.IsAlive)
31. **break**;
32. Thread.Sleep(10);
33. }
34. }
35. }

在CloseClientSocket方法中，对m\_asyncSocketUserTokenPool和m\_asyncSocketUserTokenList进行处理的时候都有加锁，代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20707035) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[在CODE上查看代码片](https://code.csdn.net/snippets/223337)

1. **public** **void** CloseClientSocket(AsyncSocketUserToken userToken)
2. {
3. **if** (userToken.ConnectSocket == **null**)
4. **return**;
5. **string** socketInfo = **string**.Format("Local Address: {0} Remote Address: {1}", userToken.ConnectSocket.LocalEndPoint,
6. userToken.ConnectSocket.RemoteEndPoint);
7. Program.Logger.InfoFormat("Client connection disconnected. {0}", socketInfo);
8. **try**
9. {
10. userToken.ConnectSocket.Shutdown(SocketShutdown.Both);
11. }
12. **catch** (Exception E)
13. {
14. Program.Logger.ErrorFormat("CloseClientSocket Disconnect client {0} error, message: {1}", socketInfo, E.Message);
15. }
16. userToken.ConnectSocket.Close();
17. userToken.ConnectSocket = **null**; //释放引用，并清理缓存，包括释放协议对象等资源
19. m\_maxNumberAcceptedClients.Release();
20. m\_asyncSocketUserTokenPool.Push(userToken);
21. m\_asyncSocketUserTokenList.Remove(userToken);
22. }

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20707035) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[在CODE上查看代码片](https://code.csdn.net/snippets/223337)

1. **public** **void** Push(AsyncSocketUserToken item)
2. {
3. **if** (item == **null**)
4. {
5. **throw** **new** ArgumentException("Items added to a AsyncSocketUserToken cannot be null");
6. }
7. **lock** (m\_pool)
8. {
9. m\_pool.Push(item);
10. }
11. }

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/20707035) [copy](http://blog.csdn.net/sqldebug_fan/article/details/20707035)

[在CODE上查看代码片](https://code.csdn.net/snippets/223337)

1. **public** **void** Remove(AsyncSocketUserToken userToken)
2. {
3. **lock** (m\_list)
4. {
5. m\_list.Remove(userToken);
6. }
7. }

在有些性能要求更高的系统，特别是在一些C++写的完成端口中，会使用原子操作来代替锁，这样做的好处是不用进行系统内核和用户态切换，性能会高。不过技术比较偏门，不易维护，而且实际表现需要进行多方面测试，这类优化更建议优化业务逻辑，并尽量减少内存分配和释放。

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

1

踩

0

* 上一篇[C#高性能大容量SOCKET并发（九）：断点续传](http://blog.csdn.net/sqldebug_fan/article/details/20465731)
* 下一篇[C#高性能大容量SOCKET并发（零）：代码结构说明](http://blog.csdn.net/sqldebug_fan/article/details/23092673)

[[置顶] C#高性能大容量SOCKET并发（零）：代码结构说明](http://blog.csdn.net/sqldebug_fan/article/details/23092673)

标签： [完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)[C#](http://www.csdn.net/tag/C%23)[socket](http://www.csdn.net/tag/socket)[SocketAsyncEventArgs](http://www.csdn.net/tag/SocketAsyncEventArgs)[结构](http://www.csdn.net/tag/%e7%bb%93%e6%9e%84)

2014-04-07 22:35 12081人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/23092673#comments)(23) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/23092673#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

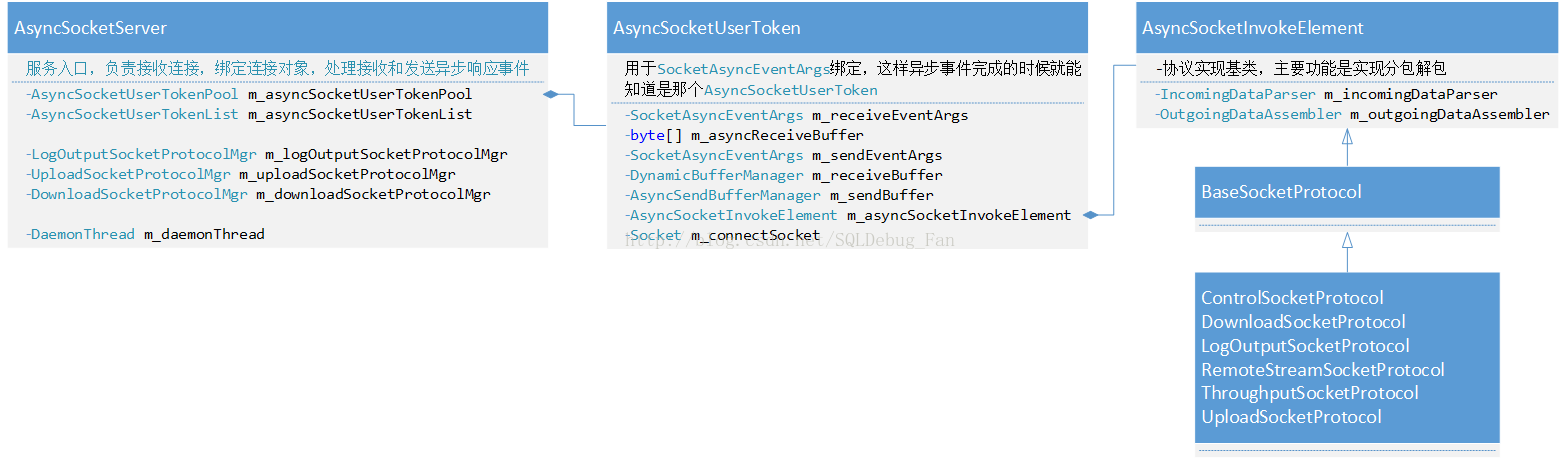
版权声明：本文为博主原创文章，未经博主允许不得转载。

C#版完成端口具有以下特点：

* 连接在线管理（提供在线连接维护，连接会话管理，数据接收，连接断开等相关事件跟踪）；
* 发送数据智能合并（组件会根据资源使用情况，对多个同时发送向同一连接的多个消息数据进行合并写入缓冲区；
* 内存池管理（提供一系列的接收和发送buffer可复用池）；
* 数据读写封装；
* 通讯协议封装；

由不同的类负责实现，比MSDN的例子代码多了不少逻辑，为了方便大家阅读和理解，把整体类框架设计整体如下。

**类结构图**



**类说明**

1. AsyncSocketServer：服务入口，建立Socket监听，负责接收连接，绑定连接对象，处理异步事件返回的接收和发送事件；m\_asyncSocketUserTokenPool是管理所有空闲的AsyncSocketUserToken，采用栈的管理方式，后进先出；m\_asyncSocketUserTokenList是管理所有正在执行的AsyncSocketUserToken，是一个列表；m\_logOutputSocketProtocolMgr是LogOutputSocketProtocol的管理对象；m\_uploadSocketProtocolMgr是UploadSocketProtocol的管理对象，用于检测是否同时上传同一个文件；m\_downloadSocketProtocolMgr是DownloadSocketProtocol的管理对象；m\_daemonThread是守护进程，用于关闭超时连接。
2. AsyncSocketUserToken：用于SocketAsyncEventArgs绑定，保存每个Socket服务对象，包括：m\_receiveEventArgs接收数据异步事件；m\_asyncReceiveBuffer接收数据异步事件使用的缓存；m\_sendEventArgs发送数据异步事件；m\_receiveBuffer接收异步事件返回的数据存放缓存，用于后续的分包；m\_sendBuffer用于保存发送的数据缓存；m\_asyncSocketInvokeElement是用于协议调用的基类，主要实现分包，并发发送的包加到发送列表中，发送完成回调时继续发送下一个包；m\_connectSocket是连接的Socket对象。
3. AsyncSocketInvokeElement：用于协议调用的基类，主要实现分包，并发发送的包加到发送列表中，发送完成回调时继续发送下一个包，包括：m\_incomingDataParser用于分析接收到的数据包，把协议解析出命令、关键字、数据，并存储到列表中；m\_outgoingDataAssembler用于组装需要发送的数据，把命令、关键字、数据组成符合协议格式的数据，并存储在列表中。
4. BaseSocketProtocol是所有协议的基类，把一些公共的方法放在这里，后续的ControlSocketProtocol、DownloadSocketProtocol、LogOutputSocketProtocol、RemoteStreamSocketProtocol、ThroughputSocketProtocol、UploadSocketProtocol都从这里继承。

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

* 上一篇[C#高性能大容量SOCKET并发（十）：SocketAsyncEventArgs线程模型](http://blog.csdn.net/sqldebug_fan/article/details/20707035)
* 下一篇[C#高性能大容量SOCKET并发（十一）：编写上传客户端](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[[置顶] C#高性能大容量SOCKET并发（十一）：编写上传客户端](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

标签： [高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[C#](http://www.csdn.net/tag/C%23)[socket](http://www.csdn.net/tag/socket)[完成端口](http://www.csdn.net/tag/%e5%ae%8c%e6%88%90%e7%ab%af%e5%8f%a3)

2014-06-08 21:02 7436人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/29391551#comments)(24) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/29391551#report)

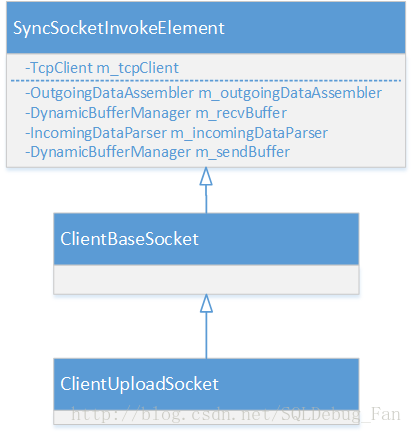
http://static.blog.csdn.net/images/category_icon.jpg 分类：

C#完成端口SOCKET开发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg 高性能大容量SOCKET并发（11） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

**客户端封装整体框架**

客户端编程基于阻塞同步模式，只有数据正常发送或接收才返回，如果发生错误则抛出异常，基于TcpClient进行封装，主要类结构如下图：



TcpClient：NET系统封装，实现了底层Socket操作，提供了阻塞和非阻塞调用；

OutgoingDataAssembler m\_outgoingDataAssembler：协议组装器，用来组装往外发送的命令，主要用于组装协议格式；

DynamicBufferManager m\_sendBuffer：用于把命令和数据同时写入到缓存中，调用一次发送，这样服务器就只会产生一次IOCP回调，可以提高性能；

IncomingDataParser m\_incomingDataParser：收到数据的解析器，用于解析返回的内容，主要是解析文本格式；

protected DynamicBufferManager m\_recvBuffer：接收数据的缓存，数据存到缓存中后，可以解析命令和数据；

**TcpClient说明，阻塞和非阻塞**

TcpClient封装了NET的底层Socket操作，基于TCP协议，提供了阻塞和非阻塞模式调用，具体是设置m\_tcpClient.Client.Blocking = true表示使用阻塞模式，反之则使用非阻塞模式。阻塞模式表示接收完指定长度的数据才返回，非阻塞模式表示收到一点数据就返回。

如我们调用m\_tcpClient.Client.Receive(m\_recvBuffer.Buffer, sizeof(int), packetLength, SocketFlags.None)，假设传入的长度为1024，阻塞模式一点要等到数据达到1024长度才返回，否则一直等待Socket超时或者链路断了，非阻塞模式则不同，加入收到8字节了，则返回调用者，调用者使用循环继续接受1024-8=1016的数据。

**发送命令**

发送数据和服务端相同，主要是对数据进行组包，然后调用发送函数发送，具体代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/29391551) [copy](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[在CODE上查看代码片](https://code.csdn.net/snippets/382981)

1. **public** **void** SendCommand(**byte**[] buffer, **int** offset, **int** count)
2. {
3. **string** commandText = m\_outgoingDataAssembler.GetProtocolText();
4. **byte**[] bufferUTF8 = Encoding.UTF8.GetBytes(commandText);
5. **int** totalLength = **sizeof**(**int**) + bufferUTF8.Length + count; //获取总大小
6. m\_sendBuffer.Clear();
7. m\_sendBuffer.WriteInt(totalLength, **false**); //写入总大小
8. m\_sendBuffer.WriteInt(bufferUTF8.Length, **false**); //写入命令大小
9. m\_sendBuffer.WriteBuffer(bufferUTF8); //写入命令内容
10. m\_sendBuffer.WriteBuffer(buffer, offset, count); //写入二进制数据
11. m\_tcpClient.Client.Send(m\_sendBuffer.Buffer, 0, m\_sendBuffer.DataCount, SocketFlags.None);
12. }

**接收命令**

接收命令和发送相反，先接收长度，然后接收内容，然后对数据进行解包，具体代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/29391551) [copy](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[在CODE上查看代码片](https://code.csdn.net/snippets/382981)

1. **public** **bool** RecvCommand(**out** **byte**[] buffer, **out** **int** offset, **out** **int** size)
2. {
3. m\_recvBuffer.Clear();
4. m\_tcpClient.Client.Receive(m\_recvBuffer.Buffer, **sizeof**(**int**), SocketFlags.None);
5. **int** packetLength = BitConverter.ToInt32(m\_recvBuffer.Buffer, 0); //获取包长度
6. **if** (NetByteOrder)
7. packetLength = System.Net.IPAddress.NetworkToHostOrder(packetLength); //把网络字节顺序转为本地字节顺序
8. m\_recvBuffer.SetBufferSize(**sizeof**(**int**) + packetLength); //保证接收有足够的空间
9. m\_tcpClient.Client.Receive(m\_recvBuffer.Buffer, **sizeof**(**int**), packetLength, SocketFlags.None);
10. **int** commandLen = BitConverter.ToInt32(m\_recvBuffer.Buffer, **sizeof**(**int**)); //取出命令长度
11. **string** tmpStr = Encoding.UTF8.GetString(m\_recvBuffer.Buffer, **sizeof**(**int**) + **sizeof**(**int**), commandLen);
12. **if** (!m\_incomingDataParser.DecodeProtocolText(tmpStr)) //解析命令
13. {
14. buffer = **null**;
15. offset = 0;
16. size = 0;
17. **return** **false**;
18. }
19. **else**
20. {
21. buffer = m\_recvBuffer.Buffer;
22. offset = commandLen + **sizeof**(**int**) + **sizeof**(**int**);
23. size = packetLength - offset;
24. **return** **true**;
25. }
26. }

**命令交互**

封装了底层Socket操作和协议解析后，实现一个命令交互如登录代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/29391551) [copy](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[在CODE上查看代码片](https://code.csdn.net/snippets/382981)

1. **public** **bool** DoLogin(**string** userName, **string** password)
2. {
3. **try**
4. {
5. m\_outgoingDataAssembler.Clear();
6. m\_outgoingDataAssembler.AddRequest();
7. m\_outgoingDataAssembler.AddCommand(AsyncSocketServer.ProtocolKey.Login);
8. m\_outgoingDataAssembler.AddValue(AsyncSocketServer.ProtocolKey.UserName, userName);
9. m\_outgoingDataAssembler.AddValue(AsyncSocketServer.ProtocolKey.Password, AsyncSocketServer.BasicFunc.MD5String(password));
10. SendCommand();
11. **bool** bSuccess = RecvCommand();
12. **if** (bSuccess)
13. {
14. bSuccess = CheckErrorCode();
15. **if** (bSuccess)
16. {
17. m\_userName = userName;
18. m\_password = password;
19. }
20. **return** bSuccess;
21. }
22. **else**
23. **return** **false**;
24. }
25. **catch** (Exception E)
26. {
27. //记录日志
28. m\_errorString = E.Message;
29. **return** **false**;
30. }
31. }

**上传协议**

上传协议主要分为三个命令，第一个是Upload，向服务器请求上传的文件，如果服务器有相同的文件，则返回是否传完，如果未传完，返回需要续传的文件位置，然后客户端则从上一个位置开始传输，传输数据服务器只接收，不应答，客户端传输完后，发完成（EOF）命令。因此三个命令封装代码如下：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/29391551) [copy](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[在CODE上查看代码片](https://code.csdn.net/snippets/382981)

1. **public** **bool** DoUpload(**string** dirName, **string** fileName, **ref** **long** fileSize)
2. {
3. **bool** bConnect = ReConnectAndLogin(); //检测连接是否还在，如果断开则重连并登录
4. **if** (!bConnect)
5. **return** bConnect;
6. **try**
7. {
8. m\_outgoingDataAssembler.Clear();
9. m\_outgoingDataAssembler.AddRequest();
10. m\_outgoingDataAssembler.AddCommand(AsyncSocketServer.ProtocolKey.Upload);
11. m\_outgoingDataAssembler.AddValue(AsyncSocketServer.ProtocolKey.DirName, dirName);
12. m\_outgoingDataAssembler.AddValue(AsyncSocketServer.ProtocolKey.FileName, fileName);
13. SendCommand();
14. **bool** bSuccess = RecvCommand();
15. **if** (bSuccess)
16. {
17. bSuccess = CheckErrorCode();
18. **if** (bSuccess)
19. {
20. bSuccess = m\_incomingDataParser.GetValue(AsyncSocketServer.ProtocolKey.FileSize, **ref** fileSize);
21. }
22. **return** bSuccess;
23. }
24. **else**
25. **return** **false**;
26. }
27. **catch** (Exception E)
28. {
29. //记录日志
30. m\_errorString = E.Message;
31. **return** **false**;
32. }
33. }
35. **public** **bool** DoData(**byte**[] buffer, **int** offset, **int** count)
36. {
37. **try**
38. {
39. m\_outgoingDataAssembler.Clear();
40. m\_outgoingDataAssembler.AddRequest();
41. m\_outgoingDataAssembler.AddCommand(AsyncSocketServer.ProtocolKey.Data);
42. SendCommand(buffer, offset, count);
43. **return** **true**;
44. }
45. **catch** (Exception E)
46. {
47. //记录日志
48. m\_errorString = E.Message;
49. **return** **false**;
50. }
51. }
53. **public** **bool** DoEof(Int64 fileSize)
54. {
55. **try**
56. {
57. m\_outgoingDataAssembler.Clear();
58. m\_outgoingDataAssembler.AddRequest();
59. m\_outgoingDataAssembler.AddCommand(AsyncSocketServer.ProtocolKey.Eof);
60. SendCommand();
61. **bool** bSuccess = RecvCommand();
62. **if** (bSuccess)
63. **return** CheckErrorCode();
64. **else**
65. **return** **false**;
66. }
67. **catch** (Exception E)
68. {
69. //记录日志
70. m\_errorString = E.Message;
71. **return** **false**;
72. }
73. }

调用过程：

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/29391551) [copy](http://blog.csdn.net/sqldebug_fan/article/details/29391551)

[在CODE上查看代码片](https://code.csdn.net/snippets/382981)

1. **protected** **static** **bool** SendFile(**string** fileName, ClientUploadSocket uploadSocket)
2. {
3. FileStream fileStream = **new** FileStream(fileName, FileMode.Open, FileAccess.ReadWrite);
4. **try**
5. {
6. **try**
7. {
8. **long** fileSize = 0;
9. **if** (!uploadSocket.DoUpload("", Path.GetFileName(fileName), **ref** fileSize))
10. **throw** **new** Exception(uploadSocket.ErrorString);
11. fileStream.Position = fileSize;
12. **byte**[] readBuffer = **new** **byte**[PacketSize];
13. **while** (fileStream.Position < fileStream.Length)
14. {
15. **int** count = fileStream.Read(readBuffer, 0, PacketSize);
16. **if** (!uploadSocket.DoData(readBuffer, 0, count))
17. **throw** **new** Exception(uploadSocket.ErrorString);
18. }
19. **if** (!uploadSocket.DoEof(fileStream.Length))
20. **throw** **new** Exception(uploadSocket.ErrorString);
21. **return** **true**;
22. }
23. **catch** (Exception E)
24. {
25. Console.WriteLine("Upload File Error: " + E.Message);
26. **return** **false**;
27. }
28. }
29. **finally**
30. {
31. fileStream.Close();
32. }
33. }

DEMO下载地址：<http://download.csdn.net/detail/sqldebug_fan/7467745>  
免责声明：此代码只是为了演示C#完成端口编程，仅用于学习和研究，切勿用于商业用途。水平有限，C#也属于初学，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

8

踩

0

* 上一篇[C#高性能大容量SOCKET并发（零）：代码结构说明](http://blog.csdn.net/sqldebug_fan/article/details/23092673)
* 下一篇[Netty高性能大容量Socket并发（一）：Netty性能测试](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

[[置顶] Netty高性能大容量Socket并发（一）：Netty性能测试](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

标签： [Netty](http://www.csdn.net/tag/Netty)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)[java](http://www.csdn.net/tag/java)[性能测试](http://www.csdn.net/tag/%e6%80%a7%e8%83%bd%e6%b5%8b%e8%af%95)

2015-11-12 17:35 5643人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/49801259#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/49801259#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Netty（1） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

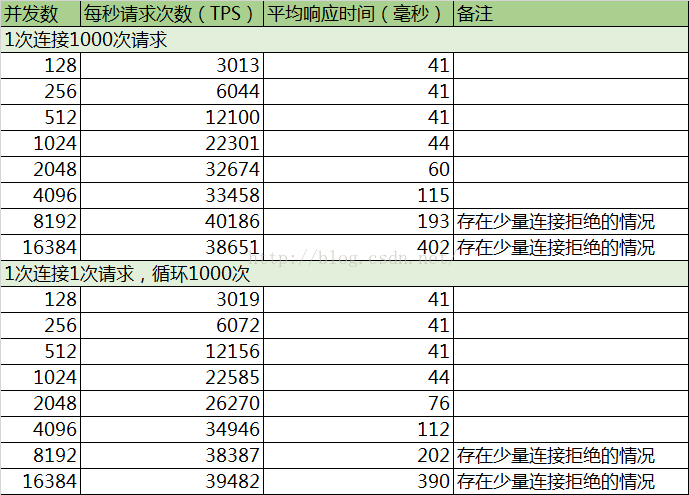
版权声明：本文为博主原创文章，未经博主允许不得转载。

目录[(?)[+]](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

**Netty性能测试**

Netty是由JBOSS提供的一个[**Java**](http://lib.csdn.net/base/javaee)开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。Netty 是一个基于NIO的客户，服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户，服务端应用。Netty相当简化和流线化了网络应用的编程开发过程，例如，TCP和UDP的socket服务开发。

Netty简化socket编程，并没有使写出来的Socket效率变低，我写了一个所有都使用默认配置的服务端，模拟最常用的RPC方案，传输协议使用4字节长度加json串的方式来传输，[**测试**](http://lib.csdn.net/base/softwaretest)服务端的通讯效率，测试结果如下。



**从测试结果看，Netty性能是非常高的，在所有使用默认配置的情况下，单台服务器能够达到4万次请求解析，作为RPC框架是足够用的。还有一个有趣的现象是每次都创建连接和重用连接的差别不大，性能损耗对应用层几乎没影响，但是大家如果在应用环境中使用每次新建的情况，一定要进行压测，确认没影响后再使用。**

**测试用例说明**

1. 部署1台Netty服务器，部署8台并发测试客户端，每个客户端跑1024个并发；
2. 分为1次连接请求1000次数据和1次连接请求1次数据循环1000次；
3. Netty服务器为Cent OS 6.5 64位，阿里云8核16G内存，JVM使用默认配置，没有进行任何调优；
4. 并发客户端为Windows Server 2008 R2  64位企业版，阿里云8核16G内存，NET Framework 4.5运行环境，没有进行任何调优；
5. 通讯数据格式为：4字节长度+JSON串，服务器负责接收JSON串，进行解析和返回；

**Netty服务端代码**

主程序入口

**[java]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/49801259) [copy](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

1. **public** **class** AppNettyServer {
3. **public** **static** **void** main(String[] args) **throws** Exception {
4. System.out.print("Hello Netty\r\n");
6. **int** port = 9090;
7. **if** (args != **null** && args.length > 0) {
8. **try** {
9. port = Integer.valueOf(args[0]);
10. } **catch** (NumberFormatException e) {
11. System.out.print("Invalid Port, Start Default Port: 9090");
12. }
13. }
14. **try** {
15. NettyCommandServer commandServer = **new** NettyCommandServer();
16. System.out.print("Start listen socket, port " + port + "\r\n");
17. commandServer.bind(port);
18. } **catch** (Exception e) {
19. System.out.print(e.getMessage());
20. }
21. }
22. }

Netty服务端初始化

**[java]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/49801259) [copy](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

1. **public** **class** NettyCommandServer {
3. **public** **static** InternalLogger logger = InternalLoggerFactory.getInstance(NettyCommandServer.**class**);
5. **public** **void** bind(**int** port) **throws** Exception {
6. EventLoopGroup bossGroup = **new** NioEventLoopGroup();
7. EventLoopGroup workerGroup = **new** NioEventLoopGroup();
8. **try** {
9. ServerBootstrap serverBootstrap = **new** ServerBootstrap();
10. serverBootstrap.group(bossGroup, workerGroup);
11. serverBootstrap.channel(NioServerSocketChannel.**class**);
12. serverBootstrap.option(ChannelOption.SO\_BACKLOG, 1024);
13. serverBootstrap.handler(**new** LoggingHandler());
14. serverBootstrap.childHandler(**new** NettyChannelHandler());
15. ChannelFuture channelFuture = serverBootstrap.bind(port).sync();
16. channelFuture.channel().closeFuture().sync();
17. } **finally** {
18. bossGroup.shutdownGracefully();
19. workerGroup.shutdownGracefully();
20. }
21. }
23. **private** **class** NettyChannelHandler **extends** ChannelInitializer<SocketChannel> {
25. @Override
26. **protected** **void** initChannel(SocketChannel socketChannel)
27. **throws** Exception {
28. socketChannel.pipeline().addLast(**new** LengthFieldBasedFrameDecoder(ByteOrder.BIG\_ENDIAN, 64 \* 1024, 0, 4, 0, 4, **true**));
29. socketChannel.pipeline().addLast(**new** StringDecoder(Charset.forName("UTF-8")));
30. socketChannel.pipeline().addLast(**new** NettyCommandHandler());
31. }
32. }
33. }

通讯协议解析数据

**[java]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/49801259) [copy](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

1. **public** **class** NettyCommandHandler **extends** ChannelHandlerAdapter {
2. **private** **int** counter = 0;
4. @Override
5. **public** **void** channelRead(ChannelHandlerContext ctx, Object msg) {
6. **try** {
7. String body = (String) msg;
8. JsonDataObject request = JsonUtil.fromJson(body, JsonDataObject.**class**);
10. counter = counter + 1;
11. JsonDataObject response = **new** JsonDataObject();
12. response.setCode(0);
13. response.setMsg("Success");
14. response.setData(counter+"");
15. String respJson = JsonUtil.toJson(response);
17. **byte**[] respUtf8 = respJson.getBytes("UTF-8");
18. **int** respLength = respUtf8.length;
19. ByteBuf respLengthBuf = PooledByteBufAllocator.DEFAULT.buffer(4);
20. respLengthBuf.writeInt(respLength);
21. respLengthBuf.order(ByteOrder.BIG\_ENDIAN);
22. ctx.write(respLengthBuf);
23. ByteBuf resp = PooledByteBufAllocator.DEFAULT.buffer(respUtf8.length);
24. resp.writeBytes(respUtf8);
25. ctx.write(resp);
26. } **catch** (Exception e) {
27. NettyCommandServer.logger.error(e.getMessage() + "\r\n");
28. StringWriter sw = **new** StringWriter();
29. PrintWriter pw = **new** PrintWriter(sw);
30. e.printStackTrace(pw);
31. pw.flush();
32. sw.flush();
33. NettyCommandServer.logger.error(sw.toString());
34. }
35. }
37. @Override
38. **public** **void** channelReadComplete(ChannelHandlerContext ctx) {
39. ctx.flush();
40. }
42. @Override
43. **public** **void** exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
44. ctx.close();
45. }
46. }

C#客户端代码

**[csharp]** [view plain](http://blog.csdn.net/sqldebug_fan/article/details/49801259) [copy](http://blog.csdn.net/sqldebug_fan/article/details/49801259)

1. **public** **class** SocketClient : Object
2. {
3. **private** TcpClient tcpClient;
5. **public** SocketClient()
6. {
7. tcpClient = **new** TcpClient();
8. tcpClient.Client.Blocking = **true**;
9. }
11. **public** **void** Connect(**string** host, **int** port)
12. {
13. tcpClient.Connect(host, port);
14. }
16. **public** **void** Disconnect()
17. {
18. tcpClient.Close();
19. }
21. **public** **string** SendJson(**string** json)
22. {
23. **byte**[] bufferUTF8 = Encoding.UTF8.GetBytes(json);
24. **int** jsonLength = System.Net.IPAddress.HostToNetworkOrder(bufferUTF8.Length); //转换为网络字节顺序，大头结构
25. **byte**[] bufferLength = BitConverter.GetBytes(jsonLength);
26. tcpClient.Client.Send(bufferLength, 0, bufferLength.Length, SocketFlags.None); //发送4字节长度
27. tcpClient.Client.Send(bufferUTF8, 0, bufferUTF8.Length, SocketFlags.None); //发送Json串内容
29. **byte**[] bufferRecvLength = **new** **byte**[**sizeof**(**int**)];
30. tcpClient.Client.Receive(bufferRecvLength, **sizeof**(**int**), SocketFlags.None); //获取长度
31. **int** recvLength = BitConverter.ToInt32(bufferRecvLength, 0);
32. recvLength = System.Net.IPAddress.NetworkToHostOrder(recvLength); //转为本地字节顺序
33. **byte**[] bufferRecvUtf8 = **new** **byte**[recvLength];
34. tcpClient.Client.Receive(bufferRecvUtf8, recvLength, SocketFlags.None);
35. **string** recvCommand = Encoding.UTF8.GetString(bufferRecvUtf8, 0, recvLength);
36. **return** recvCommand;
37. }
38. }

测试DEMO下载地址：[下载地址](http://download.csdn.net/detail/sqldebug_fan/9264311)

免责声明：此代码只是为了演示Netty性能测试，仅用于学习和研究，切勿用于商业用途。水平有限，错误在所难免，欢迎指正和指导。邮箱地址：fansheng\_hx@163.com。

顶

2

踩

0

* 上一篇[C#高性能大容量SOCKET并发（十一）：编写上传客户端](http://blog.csdn.net/sqldebug_fan/article/details/29391551)
* 下一篇[Netty高性能大容量Socket并发（二）：Netty简介](http://blog.csdn.net/sqldebug_fan/article/details/49820891)

# [[置顶] Netty高性能大容量Socket并发（二）：Netty简介](http://blog.csdn.net/sqldebug_fan/article/details/49820891)

标签： [java](http://www.csdn.net/tag/java)[netty](http://www.csdn.net/tag/netty)[socket](http://www.csdn.net/tag/socket)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)[并发](http://www.csdn.net/tag/%e5%b9%b6%e5%8f%91)

2015-11-16 14:41 1542人阅读 [评论](http://blog.csdn.net/sqldebug_fan/article/details/49820891#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/sqldebug_fan/article/details/49820891#report)

http://static.blog.csdn.net/images/category_icon.jpg 分类：

Netty（1） http://static.blog.csdn.net/images/arrow_triangle%20_down.jpg

版权声明：本文为博主原创文章，未经博主允许不得转载。

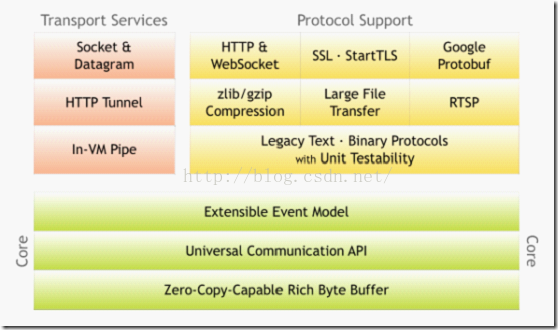
目录[(?)[+]](http://blog.csdn.net/sqldebug_fan/article/details/49820891)

## Netty简介

Netty是由JBOSS提供的一个[**Java**](http://lib.csdn.net/base/javaee)开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。和传统BIO不同，NIO是非阻塞的，和传统BIO不同，NIO相当于是线程池方式的BIO，相比BIO一个连接一个线程，NIO能支持更多的连接。BIO适合流量很高的应用，如文件传输，NIO适合流量不高，但是并发很高的应用，如聊天、推送。如果既要高并发，又要高流量的可以选择AIO。

Netty随着最近几年的发展，特别是跟着[**云计算**](http://lib.csdn.net/base/docker)、[**大数据**](http://lib.csdn.net/base/spark)的流行，很多开源分布式框架都使用了Netty作为通讯框架，如：[**Hadoop**](http://lib.csdn.net/base/hadoop)、Storm、[**Spark**](http://lib.csdn.net/base/spark)、Facebook、Twitter、阿里巴巴都在使用的Java高性能NIO通信框架，Netty经过多年打磨，其稳定性和性能达到了非常高的程度。

## Netty架构



Netty之所以能够提供高并发和高性能的通讯，主要是由以下原因形成的：

1. 零拷贝内存池；
2. 异步非阻塞通讯模型；
3. 高效的Reactor线程模型；
4. 无锁化的串行设计；
5. 高效的并发编程；
6. 高性能的序列化框架；

## Netty三个核心组件：Buffer（缓存）、Channel（通道）和ChannelEvent（事件模型）

Netty的缓存分为堆内存（HeapByteBuf）和直接内存（DirectByteBuf）缓冲区。堆内存缓冲区的特点是分配和回收速度快，可以被JVM自动回收，缺点是进行Socket的IO读写，需要额外进行一次内存复制，将堆内存对应的缓冲区复制到内核中，性能会有一定程度的下降；直接内存缓冲区的特点是非堆内存，，它在堆外进行内存分配，相比于堆内存，它的分配和回收速度会慢一些，但是将它写入或者从Socket Channel中读取时，由于少了一次内存复制，速度比堆内存快。

Channel包括但不限于网络的读、写，客户端发起连接，主动关闭连接，链路关闭，获取通信双方的网络地址等，封装了Socket的操作。当有Socket操作发生时，会触发事件响应操作，主要操作包括channelRead、channelReadComplete、exceptionCaught等方法，还包括连接建立和断开等事件。

Netty正是通过Buffer、Channel、ChannelEvent等的封装，实现了高效率，简单易用的Socket并发方案，其编程简单，默认使用的类都具有很好的效率。

* 上一篇[Netty高性能大容量Socket并发（一）：Netty性能测试](http://blog.csdn.net/sqldebug_fan/article/details/49801259)