[**[置顶] C#网络编程系列文章(一)之Socket实现异步TCP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

2015-03-14 13:43 4490人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44258719#comments)(10) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44258719#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  计算机网络（2）

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/zhujunxxxxx/article/details/44258719)

**原创性声明**

本文作者：小竹zz 本文地址<http://blog.csdn.net/zhujunxxxxx/article/details/44258719> 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**代码下载地址**

<http://download.csdn.net/detail/zhujunxxxxx/8510991>

**开篇**

本人因为对于网络编程的喜爱，经常性的使用c#编写各类服务器(e.g TCP服务器，UDP服务器)，但是基本上都是搞着玩，网上也有很多讲c#网络编程的文章，当然我也参考了很多作者写的文章。看了这篇文章以后再也不用导出找资料了。

本系列文章会依次介绍使用Socket实现的异步TCP服务器、同步TCP服务器、异步UDP服务器、同步UDP服务器 and 使用TcpListener和UdpClient实现的异步TCP服务器、同步TCP服务器、异步UDP服务器、同步UDP服务器。

**Socket异步TCP服务器**

相信搞过网络编程的人来说这个TCP一点也不陌生吧，在C#中微软已经帮我们封装过了一个TcpListener和TcpClient这两个类了，实现了对于套接字的封装，但是呢实际上还是不怎么好用，所以我们用Socket来实现一个异步的TCP服务器。

在本文中我只给出服务器端代码，客户端代码自己可以找找别处，毕竟我只是为了写出一个好的服务器端

下面是代码

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44258719) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
8. **namespace** NetFrame.Net.TCP.Sock.Asynchronous
9. {
10. /// <summary>
11. /// Socket实现的异步TCP服务器
12. /// </summary>
13. **public** **class** AsyncSocketTCPServer : IDisposable
14. {
15. #region Fields
16. /// <summary>
17. /// 服务器程序允许的最大客户端连接数
18. /// </summary>
19. **private** **int** \_maxClient;
21. /// <summary>
22. /// 当前的连接的客户端数
23. /// </summary>
24. **private** **int** \_clientCount;
26. /// <summary>
27. /// 服务器使用的异步socket
28. /// </summary>
29. **private** Socket \_serverSock;
31. /// <summary>
32. /// 客户端会话列表
33. /// </summary>
34. **private** List<AsyncSocketState> \_clients;
36. **private** **bool** disposed = **false**;
38. #endregion
40. #region Properties
42. /// <summary>
43. /// 服务器是否正在运行
44. /// </summary>
45. **public** **bool** IsRunning { **get**; **private** **set**; }
46. /// <summary>
47. /// 监听的IP地址
48. /// </summary>
49. **public** IPAddress Address { **get**; **private** **set**; }
50. /// <summary>
51. /// 监听的端口
52. /// </summary>
53. **public** **int** Port { **get**; **private** **set**; }
54. /// <summary>
55. /// 通信使用的编码
56. /// </summary>
57. **public** Encoding Encoding { **get**; **set**; }

60. #endregion
62. #region 构造函数
64. /// <summary>
65. /// 异步Socket TCP服务器
66. /// </summary>
67. /// <param name="listenPort">监听的端口</param>
68. **public** AsyncSocketTCPServer(**int** listenPort)
69. : **this**(IPAddress.Any, listenPort,1024)
70. {
71. }
73. /// <summary>
74. /// 异步Socket TCP服务器
75. /// </summary>
76. /// <param name="localEP">监听的终结点</param>
77. **public** AsyncSocketTCPServer(IPEndPoint localEP)
78. : **this**(localEP.Address, localEP.Port,1024)
79. {
80. }
82. /// <summary>
83. /// 异步Socket TCP服务器
84. /// </summary>
85. /// <param name="localIPAddress">监听的IP地址</param>
86. /// <param name="listenPort">监听的端口</param>
87. /// <param name="maxClient">最大客户端数量</param>
88. **public** AsyncSocketTCPServer(IPAddress localIPAddress, **int** listenPort,**int** maxClient)
89. {
90. **this**.Address = localIPAddress;
91. **this**.Port = listenPort;
92. **this**.Encoding = Encoding.Default;
94. \_maxClient = maxClient;
95. \_clients = **new** List<AsyncSocketState>();
96. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
97. }
99. #endregion
101. #region Method
103. /// <summary>
104. /// 启动服务器
105. /// </summary>
106. **public** **void** Start()
107. {
108. **if** (!IsRunning)
109. {
110. IsRunning = **true**;
111. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
112. \_serverSock.Listen(1024);
113. \_serverSock.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), \_serverSock);
114. }
115. }
117. /// <summary>
118. /// 启动服务器
119. /// </summary>
120. /// <param name="backlog">
121. /// 服务器所允许的挂起连接序列的最大长度
122. /// </param>
123. **public** **void** Start(**int** backlog)
124. {
125. **if** (!IsRunning)
126. {
127. IsRunning = **true**;
128. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
129. \_serverSock.Listen(backlog);
130. \_serverSock.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), \_serverSock);
131. }
132. }
134. /// <summary>
135. /// 停止服务器
136. /// </summary>
137. **public** **void** Stop()
138. {
139. **if** (IsRunning)
140. {
141. IsRunning = **false**;
142. \_serverSock.Close();
143. //TODO 关闭对所有客户端的连接
145. }
146. }
148. /// <summary>
149. /// 处理客户端连接
150. /// </summary>
151. /// <param name="ar"></param>
152. **private** **void** HandleAcceptConnected(IAsyncResult ar)
153. {
154. **if** (IsRunning)
155. {
156. Socket server = (Socket)ar.AsyncState;
157. Socket client = server.EndAccept(ar);
159. //检查是否达到最大的允许的客户端数目
160. **if** (\_clientCount >= \_maxClient)
161. {
162. //C-TODO 触发事件
163. RaiseOtherException(**null**);
164. }
165. **else**
166. {
167. AsyncSocketState state = **new** AsyncSocketState(client);
168. **lock** (\_clients)
169. {
170. \_clients.Add(state);
171. \_clientCount++;
172. RaiseClientConnected(state); //触发客户端连接事件
173. }
174. state.RecvDataBuffer = **new** **byte**[client.ReceiveBufferSize];
175. //开始接受来自该客户端的数据
176. client.BeginReceive(state.RecvDataBuffer, 0, state.RecvDataBuffer.Length, SocketFlags.None,
177. **new** AsyncCallback(HandleDataReceived), state);
178. }
179. //接受下一个请求
180. server.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), ar.AsyncState);
181. }
182. }
183. /// <summary>
184. /// 处理客户端数据
185. /// </summary>
186. /// <param name="ar"></param>
187. **private** **void** HandleDataReceived(IAsyncResult ar)
188. {
189. **if** (IsRunning)
190. {
191. AsyncSocketState state = (AsyncSocketState)ar.AsyncState;
192. Socket client = state.ClientSocket;
193. **try**
194. {
195. //如果两次开始了异步的接收,所以当客户端退出的时候
196. //会两次执行EndReceive
197. **int** recv = client.EndReceive(ar);
198. **if** (recv == 0)
199. {
200. //C- TODO 触发事件 (关闭客户端)
201. Close(state);
202. RaiseNetError(state);
203. **return**;
204. }
205. //TODO 处理已经读取的数据 ps:数据在state的RecvDataBuffer中
207. //C- TODO 触发数据接收事件
208. RaiseDataReceived(state);
209. }
210. **catch** (SocketException)
211. {
212. //C- TODO 异常处理
213. RaiseNetError(state);
214. }
215. **finally**
216. {
217. //继续接收来自来客户端的数据
218. client.BeginReceive(state.RecvDataBuffer, 0, state.RecvDataBuffer.Length, SocketFlags.None,
219. **new** AsyncCallback(HandleDataReceived), state);
220. }
221. }
222. }
224. /// <summary>
225. /// 发送数据
226. /// </summary>
227. /// <param name="state">接收数据的客户端会话</param>
228. /// <param name="data">数据报文</param>
229. **public** **void** Send(AsyncSocketState state, **byte**[] data)
230. {
231. RaisePrepareSend(state);
232. Send(state.ClientSocket, data);
233. }
235. /// <summary>
236. /// 异步发送数据至指定的客户端
237. /// </summary>
238. /// <param name="client">客户端</param>
239. /// <param name="data">报文</param>
240. **public** **void** Send(Socket client, **byte**[] data)
241. {
242. **if** (!IsRunning)
243. **throw** **new** InvalidProgramException("This TCP Scoket server has not been started.");
245. **if** (client == **null**)
246. **throw** **new** ArgumentNullException("client");
248. **if** (data == **null**)
249. **throw** **new** ArgumentNullException("data");
250. client.BeginSend(data, 0, data.Length, SocketFlags.None,
251. **new** AsyncCallback(SendDataEnd), client);
252. }
254. /// <summary>
255. /// 发送数据完成处理函数
256. /// </summary>
257. /// <param name="ar">目标客户端Socket</param>
258. **private** **void** SendDataEnd(IAsyncResult ar)
259. {
260. ((Socket)ar.AsyncState).EndSend(ar);
261. RaiseCompletedSend(**null**);
262. }
263. #endregion
265. #region 事件
267. /// <summary>
268. /// 与客户端的连接已建立事件
269. /// </summary>
270. **public** **event** EventHandler<AsyncSocketEventArgs> ClientConnected;
271. /// <summary>
272. /// 与客户端的连接已断开事件
273. /// </summary>
274. **public** **event** EventHandler<AsyncSocketEventArgs> ClientDisconnected;
276. /// <summary>
277. /// 触发客户端连接事件
278. /// </summary>
279. /// <param name="state"></param>
280. **private** **void** RaiseClientConnected(AsyncSocketState state)
281. {
282. **if** (ClientConnected != **null**)
283. {
284. ClientConnected(**this**, **new** AsyncSocketEventArgs(state));
285. }
286. }
287. /// <summary>
288. /// 触发客户端连接断开事件
289. /// </summary>
290. /// <param name="client"></param>
291. **private** **void** RaiseClientDisconnected(Socket client)
292. {
293. **if** (ClientDisconnected != **null**)
294. {
295. ClientDisconnected(**this**, **new** AsyncSocketEventArgs("连接断开"));
296. }
297. }
299. /// <summary>
300. /// 接收到数据事件
301. /// </summary>
302. **public** **event** EventHandler<AsyncSocketEventArgs> DataReceived;
304. **private** **void** RaiseDataReceived(AsyncSocketState state)
305. {
306. **if** (DataReceived != **null**)
307. {
308. DataReceived(**this**, **new** AsyncSocketEventArgs(state));
309. }
310. }
312. /// <summary>
313. /// 发送数据前的事件
314. /// </summary>
315. **public** **event** EventHandler<AsyncSocketEventArgs> PrepareSend;
317. /// <summary>
318. /// 触发发送数据前的事件
319. /// </summary>
320. /// <param name="state"></param>
321. **private** **void** RaisePrepareSend(AsyncSocketState state)
322. {
323. **if** (PrepareSend != **null**)
324. {
325. PrepareSend(**this**, **new** AsyncSocketEventArgs(state));
326. }
327. }
329. /// <summary>
330. /// 数据发送完毕事件
331. /// </summary>
332. **public** **event** EventHandler<AsyncSocketEventArgs> CompletedSend;
334. /// <summary>
335. /// 触发数据发送完毕的事件
336. /// </summary>
337. /// <param name="state"></param>
338. **private** **void** RaiseCompletedSend(AsyncSocketState state)
339. {
340. **if** (CompletedSend != **null**)
341. {
342. CompletedSend(**this**, **new** AsyncSocketEventArgs(state));
343. }
344. }
346. /// <summary>
347. /// 网络错误事件
348. /// </summary>
349. **public** **event** EventHandler<AsyncSocketEventArgs> NetError;
350. /// <summary>
351. /// 触发网络错误事件
352. /// </summary>
353. /// <param name="state"></param>
354. **private** **void** RaiseNetError(AsyncSocketState state)
355. {
356. **if** (NetError != **null**)
357. {
358. NetError(**this**, **new** AsyncSocketEventArgs(state));
359. }
360. }
362. /// <summary>
363. /// 异常事件
364. /// </summary>
365. **public** **event** EventHandler<AsyncSocketEventArgs> OtherException;
366. /// <summary>
367. /// 触发异常事件
368. /// </summary>
369. /// <param name="state"></param>
370. **private** **void** RaiseOtherException(AsyncSocketState state, **string** descrip)
371. {
372. **if** (OtherException != **null**)
373. {
374. OtherException(**this**, **new** AsyncSocketEventArgs(descrip, state));
375. }
376. }
377. **private** **void** RaiseOtherException(AsyncSocketState state)
378. {
379. RaiseOtherException(state, "");
380. }
381. #endregion
383. #region Close
384. /// <summary>
385. /// 关闭一个与客户端之间的会话
386. /// </summary>
387. /// <param name="state">需要关闭的客户端会话对象</param>
388. **public** **void** Close(AsyncSocketState state)
389. {
390. **if** (state != **null**)
391. {
392. state.Datagram = **null**;
393. state.RecvDataBuffer = **null**;
395. \_clients.Remove(state);
396. \_clientCount--;
397. //TODO 触发关闭事件
398. state.Close();
399. }
400. }
401. /// <summary>
402. /// 关闭所有的客户端会话,与所有的客户端连接会断开
403. /// </summary>
404. **public** **void** CloseAllClient()
405. {
406. **foreach** (AsyncSocketState client **in** \_clients)
407. {
408. Close(client);
409. }
410. \_clientCount = 0;
411. \_clients.Clear();
412. }
413. #endregion
415. #region 释放
416. /// <summary>
417. /// Performs application-defined tasks associated with freeing,
418. /// releasing, or resetting unmanaged resources.
419. /// </summary>
420. **public** **void** Dispose()
421. {
422. Dispose(**true**);
423. GC.SuppressFinalize(**this**);
424. }
426. /// <summary>
427. /// Releases unmanaged and - optionally - managed resources
428. /// </summary>
429. /// <param name="disposing"><c>true</c> to release
430. /// both managed and unmanaged resources; <c>false</c>
431. /// to release only unmanaged resources.</param>
432. **protected** **virtual** **void** Dispose(**bool** disposing)
433. {
434. **if** (!**this**.disposed)
435. {
436. **if** (disposing)
437. {
438. **try**
439. {
440. Stop();
441. **if** (\_serverSock != **null**)
442. {
443. \_serverSock = **null**;
444. }
445. }
446. **catch** (SocketException)
447. {
448. //TODO
449. RaiseOtherException(**null**);
450. }
451. }
452. disposed = **true**;
453. }
454. }
455. #endregion
456. }
457. }

事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44258719) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.TCP.Sock.Asynchronous
7. {
8. /// <summary>
9. /// 异步Socket TCP事件参数类
10. /// </summary>
11. **public** **class** AsyncSocketEventArgs:EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** AsyncSocketState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** AsyncSocketEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** AsyncSocketEventArgs(AsyncSocketState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** AsyncSocketEventArgs(**string** msg, AsyncSocketState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

用户状态封装

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44258719) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
7. **namespace** NetFrame.Net.TCP.Sock.Asynchronous
8. {
9. /// <summary>
10. /// 异步SOCKET TCP 中用来存储客户端状态信息的类
11. /// </summary>
12. **public** **class** AsyncSocketState
13. {
14. #region 字段
15. /// <summary>
16. /// 接收数据缓冲区
17. /// </summary>
18. **private** **byte**[] \_recvBuffer;
20. /// <summary>
21. /// 客户端发送到服务器的报文
22. /// 注意:在有些情况下报文可能只是报文的片断而不完整
23. /// </summary>
24. **private** **string** \_datagram;
26. /// <summary>
27. /// 客户端的Socket
28. /// </summary>
29. **private** Socket \_clientSock;
31. #endregion
33. #region 属性
35. /// <summary>
36. /// 接收数据缓冲区
37. /// </summary>
38. **public** **byte**[] RecvDataBuffer
39. {
40. **get**
41. {
42. **return** \_recvBuffer;
43. }
44. **set**
45. {
46. \_recvBuffer = value;
47. }
48. }
50. /// <summary>
51. /// 存取会话的报文
52. /// </summary>
53. **public** **string** Datagram
54. {
55. **get**
56. {
57. **return** \_datagram;
58. }
59. **set**
60. {
61. \_datagram = value;
62. }
63. }
65. /// <summary>
66. /// 获得与客户端会话关联的Socket对象
67. /// </summary>
68. **public** Socket ClientSocket
69. {
70. **get**
71. {
72. **return** \_clientSock;
74. }
75. }

78. #endregion
80. /// <summary>
81. /// 构造函数
82. /// </summary>
83. /// <param name="cliSock">会话使用的Socket连接</param>
84. **public** AsyncSocketState(Socket cliSock)
85. {
86. \_clientSock = cliSock;
87. }
89. /// <summary>
90. /// 初始化数据缓冲区
91. /// </summary>
92. **public** **void** InitBuffer()
93. {
94. **if** (\_recvBuffer == **null**&&\_clientSock!=**null**)
95. {
96. \_recvBuffer=**new** **byte**[\_clientSock.ReceiveBufferSize];
97. }
98. }
100. /// <summary>
101. /// 关闭会话
102. /// </summary>
103. **public** **void** Close()
104. {
106. //关闭数据的接受和发送
107. \_clientSock.Shutdown(SocketShutdown.Both);
109. //清理资源
110. \_clientSock.Close();
111. }
112. }
113. }

本文作者：小竹zz 本文地址<http://blog.csdn.net/zhujunxxxxx/article/details/44258719> 转载请注明出处

顶

1

踩

0

* 上一篇[基于C#的socket编程的TCP异步实现](http://blog.csdn.net/zhujunxxxxx/article/details/44239743)
* 下一篇[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[**[置顶] C#网络编程系列文章(二)之Socket实现同步TCP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

2015-03-14 17:54 3225人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44261375#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44261375#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

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/zhujunxxxxx/article/details/44261375)

**原创性声明**

本文作者:小竹zz  本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

在上一篇博客中我说了，我将会介绍c#中使用Socket和TcpListener和UdpClient实现各种同步和异步的TCP和UDP服务器，这些都是是我自己花了很多天的时间来总结的，这样一来相信刚接触c#网络编程的朋友们不会像以前的我一样到处出找资料，到处调试。本次我介绍的是使用Socket来实现的同步的TCP服务器，同步的TCP服务器和第一篇里面介绍的异步TCP服务器的区别就是，在Socket调用Accept的时候是否会阻塞。

同步的TCP服务器在接受到一个客户端的请求的时候一般是开启一个线程去处理和这个客户端的通信工作，这种方式的不好的地方就是资源消耗大，但是假如使用线程池的话，事先分配一定数量的线程，性能还是可以。

但是我之所以给出很多方法实现的服务器，主要目的就是想测试哪种情况性能比较不错，就目前来看，异步的模式比较NICE，但是其实还有一种IOCP模式性能是最好的。。。

**Socket同步TCP服务器**

服务端代码

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261375) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **using** System.Threading;
9. **namespace** NetFrame.Net.TCP.Sock.Synchronous
10. {
11. /// <summary>
12. /// 基于socket实现的同步TCP服务器
13. /// </summary>
14. **public** **class** SocketTCPServer
15. {
16. #region Fields
17. /// <summary>
18. /// 服务器程序允许的最大客户端连接数
19. /// </summary>
20. **private** **int** \_maxClient;
22. /// <summary>
23. /// 当前的连接的客户端数
24. /// </summary>
25. **private** **int** \_clientCount;
27. /// <summary>
28. /// 服务器使用的异步socket
29. /// </summary>
30. **private** Socket \_serverSock;
32. /// <summary>
33. /// 客户端会话列表
34. /// </summary>
35. **private** List<SocketClientHandle> \_clients;
37. **private** **bool** disposed = **false**;
39. #endregion
41. #region Properties
43. /// <summary>
44. /// 服务器是否正在运行
45. /// </summary>
46. **public** **bool** IsRunning { **get**; **private** **set**; }
47. /// <summary>
48. /// 监听的IP地址
49. /// </summary>
50. **public** IPAddress Address { **get**; **private** **set**; }
51. /// <summary>
52. /// 监听的端口
53. /// </summary>
54. **public** **int** Port { **get**; **private** **set**; }
55. /// <summary>
56. /// 通信使用的编码
57. /// </summary>
58. **public** Encoding Encoding { **get**; **set**; }

61. #endregion
63. #region 构造函数
64. /// <summary>
65. /// 同步Socket TCP服务器
66. /// </summary>
67. /// <param name="listenPort">监听的端口</param>
68. **public** SocketTCPServer(**int** listenPort)
69. : **this**(IPAddress.Any, listenPort, 1024)
70. {
71. }
73. /// <summary>
74. /// 同步Socket TCP服务器
75. /// </summary>
76. /// <param name="localEP">监听的终结点</param>
77. **public** SocketTCPServer(IPEndPoint localEP)
78. : **this**(localEP.Address, localEP.Port, 1024)
79. {
80. }
82. /// <summary>
83. /// 同步Socket TCP服务器
84. /// </summary>
85. /// <param name="localIPAddress">监听的IP地址</param>
86. /// <param name="listenPort">监听的端口</param>
87. /// <param name="maxClient">最大客户端数量</param>
88. **public** SocketTCPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
89. {
90. **this**.Address = localIPAddress;
91. **this**.Port = listenPort;
92. **this**.Encoding = Encoding.Default;
94. \_maxClient = maxClient;
95. \_clients = **new** List<SocketClientHandle>();
96. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
97. }
99. #endregion
101. #region Method
102. /// <summary>
103. /// 启动服务器
104. /// </summary>
105. **public** **void** Start()
106. {
107. **if** (!IsRunning)
108. {
109. IsRunning = **true**;
110. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
111. Thread thread = **new** Thread(StartListen);
112. thread.Start();
114. }
115. }
116. /// <summary>
117. /// 开始进行监听
118. /// </summary>
119. **private** **void** StartListen()
120. {
121. \_serverSock.Listen(1024);
122. SocketClientHandle handle;
123. **while** (IsRunning)
124. {
125. **if** (\_clientCount >= \_maxClient)
126. {
127. //TODO 客户端过多异常
128. RaiseOtherException(**null**);
129. }
130. **else**
131. {
132. Socket clientSock = \_serverSock.Accept();
133. \_clientCount++;
134. //TODO 创建一个处理客户端的线程并启动
135. handle = **new** SocketClientHandle(clientSock);
136. \_clients.Add(handle);
137. //使用线程池来操作
138. ThreadPool.QueueUserWorkItem(**new** WaitCallback(handle.RecevieData));
140. //Thread pthread;
141. //pthread = new Thread(new ThreadStart(client.RecevieData));
142. //pthread.Start();
143. //这里应该使用线程池来进行
144. }
145. }
147. }
148. /// <summary>
149. /// 停止服务器
150. /// </summary>
151. **public** **void** Stop()
152. {
153. **if** (IsRunning)
154. {
155. IsRunning = **false**;
156. \_serverSock.Close();
157. //TODO 关闭对所有客户端的连接
159. }
160. }
161. /// <summary>
162. /// 发送函数
163. /// </summary>
164. **public** **void** Send(**string** msg, SocketClientHandle client)
165. {
166. //TODO
167. }
169. /// <summary>
170. /// 关闭一个与客户端之间的会话
171. /// </summary>
172. /// <param name="handle">需要关闭的客户端会话对象</param>
173. **public** **void** Close(SocketClientHandle handle)
174. {
175. **if** (handle != **null**)
176. {
177. \_clients.Remove(handle);
178. handle.Dispose();
179. \_clientCount--;
180. //TODO 触发关闭事件
182. }
183. }
184. /// <summary>
185. /// 关闭所有的客户端会话,与所有的客户端连接会断开
186. /// </summary>
187. **public** **void** CloseAllClient()
188. {
189. **foreach** (SocketClientHandle handle **in** \_clients)
190. {
191. Close(handle);
192. }
193. \_clientCount = 0;
194. \_clients.Clear();
195. }
197. #endregion
199. #region 事件
201. /// <summary>
202. /// 与客户端的连接已建立事件
203. /// </summary>
204. **public** **event** EventHandler<SocketEventArgs> ClientConnected;
205. /// <summary>
206. /// 与客户端的连接已断开事件
207. /// </summary>
208. **public** **event** EventHandler<SocketEventArgs> ClientDisconnected;
210. /// <summary>
211. /// 触发客户端连接事件
212. /// </summary>
213. /// <param name="state"></param>
214. **private** **void** RaiseClientConnected(SocketClientHandle handle)
215. {
216. **if** (ClientConnected != **null**)
217. {
218. ClientConnected(**this**, **new** SocketEventArgs(handle));
219. }
220. }
221. /// <summary>
222. /// 触发客户端连接断开事件
223. /// </summary>
224. /// <param name="client"></param>
225. **private** **void** RaiseClientDisconnected(Socket client)
226. {
227. **if** (ClientDisconnected != **null**)
228. {
229. ClientDisconnected(**this**, **new** SocketEventArgs("连接断开"));
230. }
231. }
233. /// <summary>
234. /// 接收到数据事件
235. /// </summary>
236. **public** **event** EventHandler<SocketEventArgs> DataReceived;
238. **private** **void** RaiseDataReceived(SocketClientHandle handle)
239. {
240. **if** (DataReceived != **null**)
241. {
242. DataReceived(**this**, **new** SocketEventArgs(handle));
243. }
244. }
246. /// <summary>
247. /// 数据发送事件
248. /// </summary>
249. **public** **event** EventHandler<SocketEventArgs> CompletedSend;
251. /// <summary>
252. /// 触发数据发送事件
253. /// </summary>
254. /// <param name="state"></param>
255. **private** **void** RaiseCompletedSend(SocketClientHandle handle)
256. {
257. **if** (CompletedSend != **null**)
258. {
259. CompletedSend(**this**, **new** SocketEventArgs(handle));
260. }
261. }

264. /// <summary>
265. /// 网络错误事件
266. /// </summary>
267. **public** **event** EventHandler<SocketEventArgs> NetError;
268. /// <summary>
269. /// 触发网络错误事件
270. /// </summary>
271. /// <param name="state"></param>
272. **private** **void** RaiseNetError(SocketClientHandle handle)
273. {
274. **if** (NetError != **null**)
275. {
276. NetError(**this**, **new** SocketEventArgs(handle));
277. }
278. }
280. /// <summary>
281. /// 异常事件
282. /// </summary>
283. **public** **event** EventHandler<SocketEventArgs> OtherException;
284. /// <summary>
285. /// 触发异常事件
286. /// </summary>
287. /// <param name="state"></param>
288. **private** **void** RaiseOtherException(SocketClientHandle handle, **string** descrip)
289. {
290. **if** (OtherException != **null**)
291. {
292. OtherException(**this**, **new** SocketEventArgs(descrip, handle));
293. }
294. }
295. **private** **void** RaiseOtherException(SocketClientHandle handle)
296. {
297. RaiseOtherException(handle, "");
298. }
300. #endregion
302. #region Close 未实现
303. #endregion
305. #region 释放
306. /// <summary>
307. /// Performs application-defined tasks associated with freeing,
308. /// releasing, or resetting unmanaged resources.
309. /// </summary>
310. **public** **void** Dispose()
311. {
312. Dispose(**true**);
313. GC.SuppressFinalize(**this**);
314. }
316. /// <summary>
317. /// Releases unmanaged and - optionally - managed resources
318. /// </summary>
319. /// <param name="disposing"><c>true</c> to release
320. /// both managed and unmanaged resources; <c>false</c>
321. /// to release only unmanaged resources.</param>
322. **protected** **virtual** **void** Dispose(**bool** disposing)
323. {
324. **if** (!**this**.disposed)
325. {
326. **if** (disposing)
327. {
328. **try**
329. {
330. Stop();
331. **if** (\_serverSock != **null**)
332. {
333. \_serverSock = **null**;
334. }
335. }
336. **catch** (SocketException)
337. {
338. //TODO 异常
339. }
340. }
341. disposed = **true**;
342. }
343. }
344. #endregion
345. }
346. }

对客户端操作封装的Handle类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261375) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
7. **namespace** NetFrame.Net.TCP.Sock.Synchronous
8. {
9. /// <summary>
10. /// Socket 服务器用于处理客户端连接封装的客户端处理类
11. /// </summary>
12. **public** **class** SocketClientHandle:IDisposable
13. {
14. /// <summary>
15. /// 与客户端相关联的socket
16. /// </summary>
17. **private** Socket \_client;
19. /// <summary>
20. /// 标识是否与客户端相连接
21. /// </summary>
22. **private** **bool** \_is\_connect;
23. **public** **bool** IsConnect
24. {
25. **get** { **return** \_is\_connect; }
26. **set** { \_is\_connect = value; }
27. }
29. /// <summary>
30. /// 数据接受缓冲区
31. /// </summary>
32. **private** **byte**[] \_recvBuffer;
34. **public** SocketClientHandle(Socket client)
35. {
36. **this**.\_client = client;
37. \_is\_connect = **true**;
38. \_recvBuffer = **new** **byte**[1024 \* 1024 \* 2];
39. }
41. #region Method
42. /// <summary>
43. /// 接受来自客户端发来的数据
44. /// </summary>
45. **public** **void** RecevieData(Object state)
46. {
47. **int** len = -1;
48. **while** (\_is\_connect)
49. {
50. **try**
51. {
52. len = \_client.Receive(\_recvBuffer);
53. }
54. **catch** (Exception)
55. {
56. //TODO
57. }
58. }
59. }
61. /// <summary>
62. /// 向客户端发送数据
63. /// </summary>
64. **public** **void** SendData(**string** msg)
65. {
66. **byte**[] data = Encoding.Default.GetBytes(msg);
67. **try**
68. {
69. //有一种比较好的写法
70. \_client.Send(data);
71. }
72. **catch** (Exception)
73. {
74. //TODO 处理异常
75. }
76. }
78. #endregion

81. #region 事件

84. //TODO 消息发送事件
85. //TODO 数据收到事件
86. //TODO 异常处理事件
88. #endregion
90. #region 释放
91. /// <summary>
92. /// Performs application-defined tasks associated with freeing,
93. /// releasing, or resetting unmanaged resources.
94. /// </summary>
95. **public** **void** Dispose()
96. {
97. \_is\_connect = **false**;
98. **if** (\_client != **null**)
99. {
100. \_client.Close();
101. \_client = **null**;
102. }
103. GC.SuppressFinalize(**this**);
104. }
106. #endregion
107. }
108. }

Socket同步TCP服务器的时间参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261375) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.TCP.Sock.Synchronous
7. {
8. /// <summary>
9. /// 同步Socket TCP服务器事件类
10. /// </summary>
11. **public** **class** SocketEventArgs : EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** SocketClientHandle \_handle;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** SocketEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** SocketEventArgs(SocketClientHandle handle)
34. {
35. **this**.\_handle = handle;
36. IsHandled = **false**;
37. }
38. **public** SocketEventArgs(**string** msg, SocketClientHandle handle)
39. {
40. **this**.\_msg = msg;
41. **this**.\_handle = handle;
42. IsHandled = **false**;
43. }
44. }
45. }

本文作者:小竹zz  本文地址http://blog.csdn.net/zhujunxxxxx/article/details/44258719 转载请注明出处

顶

1

踩

0

* 上一篇[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)
* 下一篇[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[**[置顶] C#网络编程系列文章(三)之TcpListener实现异步TCP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

2015-03-14 18:02 3573人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44261497#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44261497#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

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/zhujunxxxxx/article/details/44261497)

**原创性声明**

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

TcpListener 类提供一些简单方法，用于在阻止同步模式下侦听和接受传入连接请求。 可使用 TcpClient 或 Socket 来连接 TcpListener。 可使用 IPEndPoint、本地 IP 地址及端口号或者仅使用端口号，来创建 TcpListener。 可以将本地 IP 地址指定为 Any，将本地端口号指定为 0（如果希望基础服务提供程序为您分配这些值）。 如果您选择这样做，可在连接套接字后使用 LocalEndpoint 属性来标识已指定的信息。使用 Start 方法，可开始侦听传入的连接请求。 Start 将对传入连接进行排队，直至您调用 Stop 方法或它已经完成 MaxConnections 排队为止。 可使用 AcceptSocket 或 AcceptTcpClient 从传入连接请求队列提取连接。 这两种方法将阻止。 如果要避免阻止，可首先使用 Pending 方法来确定队列中是否有可用的连接请求。

虽然TcpListener已经封装的比较不错了，我们于是就使用它在构造一个比较不错的异步TCP服务器，这里依然和前两章一样，给出服务器中的代码，代码中注释很详细，我也会给出相关的封装类。

**TcpListener异步TCP服务器**

服务器代码

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261497) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **namespace** NetFrame.Net.TCP.Listener.Asynchronous
8. {
9. /// <summary>
10. /// TcpListener实现异步TCP服务器
11. /// </summary>
12. **public** **class** AsyncTCPServer : IDisposable
13. {
14. #region Fields
15. /// <summary>
16. /// 服务器程序允许的最大客户端连接数
17. /// </summary>
18. **private** **int** \_maxClient;
20. /// <summary>
21. /// 当前的连接的客户端数
22. /// </summary>
23. **private** **int** \_clientCount;
25. /// <summary>
26. /// 服务器使用的异步TcpListener
27. /// </summary>
28. **private** TcpListener \_listener;
30. /// <summary>
31. /// 客户端会话列表
32. /// </summary>
33. **private** List<Object> \_clients;
35. **private** **bool** disposed = **false**;
37. #endregion
39. #region Properties
41. /// <summary>
42. /// 服务器是否正在运行
43. /// </summary>
44. **public** **bool** IsRunning { **get**; **private** **set**; }
45. /// <summary>
46. /// 监听的IP地址
47. /// </summary>
48. **public** IPAddress Address { **get**; **private** **set**; }
49. /// <summary>
50. /// 监听的端口
51. /// </summary>
52. **public** **int** Port { **get**; **private** **set**; }
53. /// <summary>
54. /// 通信使用的编码
55. /// </summary>
56. **public** Encoding Encoding { **get**; **set**; }

59. #endregion
61. #region 构造函数
63. /// <summary>
64. /// 异步TCP服务器
65. /// </summary>
66. /// <param name="listenPort">监听的端口</param>
67. **public** AsyncTCPServer(**int** listenPort)
68. : **this**(IPAddress.Any, listenPort)
69. {
70. }
72. /// <summary>
73. /// 异步TCP服务器
74. /// </summary>
75. /// <param name="localEP">监听的终结点</param>
76. **public** AsyncTCPServer(IPEndPoint localEP)
77. : **this**(localEP.Address, localEP.Port)
78. {
79. }
81. /// <summary>
82. /// 异步TCP服务器
83. /// </summary>
84. /// <param name="localIPAddress">监听的IP地址</param>
85. /// <param name="listenPort">监听的端口</param>
86. **public** AsyncTCPServer(IPAddress localIPAddress, **int** listenPort)
87. {
88. Address = localIPAddress;
89. Port = listenPort;
90. **this**.Encoding = Encoding.Default;
92. \_clients = **new** List<Object>();
94. \_listener = **new** TcpListener(Address, Port);
95. \_listener.AllowNatTraversal(**true**);
96. }
98. #endregion
100. #region Method
102. /// <summary>
103. /// 启动服务器
104. /// </summary>
105. **public** **void** Start()
106. {
107. **if** (!IsRunning)
108. {
109. IsRunning = **true**;
110. \_listener.Start();
111. \_listener.BeginAcceptTcpClient(
112. **new** AsyncCallback(HandleTcpClientAccepted), \_listener);
113. }
114. }

117. /// <summary>
118. /// 启动服务器
119. /// </summary>
120. /// <param name="backlog">
121. /// 服务器所允许的挂起连接序列的最大长度
122. /// </param>
123. **public** **void** Start(**int** backlog)
124. {
125. **if** (!IsRunning)
126. {
127. IsRunning = **true**;
128. \_listener.Start(backlog);
129. \_listener.BeginAcceptTcpClient(
130. **new** AsyncCallback(HandleTcpClientAccepted), \_listener);
131. }
132. }
134. /// <summary>
135. /// 停止服务器
136. /// </summary>
137. **public** **void** Stop()
138. {
139. **if** (IsRunning)
140. {
141. IsRunning = **false**;
142. \_listener.Stop();
143. **lock** (\_clients)
144. {
145. //关闭所有客户端连接
146. CloseAllClient();
147. }
148. }
149. }
151. /// <summary>
152. /// 处理客户端连接的函数
153. /// </summary>
154. /// <param name="ar"></param>
155. **private** **void** HandleTcpClientAccepted(IAsyncResult ar)
156. {
157. **if** (IsRunning)
158. {
159. //TcpListener tcpListener = (TcpListener)ar.AsyncState;
161. TcpClient client = \_listener.EndAcceptTcpClient(ar);
162. **byte**[] buffer = **new** **byte**[client.ReceiveBufferSize];
164. TCPClientState state
165. = **new** TCPClientState(client, buffer);
166. **lock** (\_clients)
167. {
168. \_clients.Add(state);
169. RaiseClientConnected(state);
170. }
172. NetworkStream stream = state.NetworkStream;
173. //开始异步读取数据
174. stream.BeginRead(state.Buffer, 0, state.Buffer.Length, HandleDataReceived, state);
176. \_listener.BeginAcceptTcpClient(
177. **new** AsyncCallback(HandleTcpClientAccepted), ar.AsyncState);
178. }
179. }
180. /// <summary>
181. /// 数据接受回调函数
182. /// </summary>
183. /// <param name="ar"></param>
184. **private** **void** HandleDataReceived(IAsyncResult ar)
185. {
186. **if** (IsRunning)
187. {
188. TCPClientState state = (TCPClientState)ar.AsyncState;
189. NetworkStream stream = state.NetworkStream;
191. **int** recv = 0;
192. **try**
193. {
194. recv = stream.EndRead(ar);
195. }
196. **catch**
197. {
198. recv = 0;
199. }
201. **if** (recv == 0)
202. {
203. // connection has been closed
204. **lock** (\_clients)
205. {
206. \_clients.Remove(state);
207. //触发客户端连接断开事件
208. RaiseClientDisconnected(state);
209. **return**;
210. }
211. }
213. // received byte and trigger event notification
214. **byte**[] buff = **new** **byte**[recv];
215. Buffer.BlockCopy(state.Buffer, 0, buff, 0, recv);
216. //触发数据收到事件
217. RaiseDataReceived(state);
219. // continue listening for tcp datagram packets
220. stream.BeginRead(state.Buffer, 0, state.Buffer.Length, HandleDataReceived, state);
221. }
222. }
224. /// <summary>
225. /// 发送数据
226. /// </summary>
227. /// <param name="state">接收数据的客户端会话</param>
228. /// <param name="data">数据报文</param>
229. **public** **void** Send(TCPClientState state, **byte**[] data)
230. {
231. RaisePrepareSend(state);
232. Send(state.TcpClient, data);
233. }
235. /// <summary>
236. /// 异步发送数据至指定的客户端
237. /// </summary>
238. /// <param name="client">客户端</param>
239. /// <param name="data">报文</param>
240. **public** **void** Send(TcpClient client, **byte**[] data)
241. {
242. **if** (!IsRunning)
243. **throw** **new** InvalidProgramException("This TCP Scoket server has not been started.");
245. **if** (client == **null**)
246. **throw** **new** ArgumentNullException("client");
248. **if** (data == **null**)
249. **throw** **new** ArgumentNullException("data");
250. client.GetStream().BeginWrite(data, 0, data.Length, SendDataEnd, client);
251. }
253. /// <summary>
254. /// 发送数据完成处理函数
255. /// </summary>
256. /// <param name="ar">目标客户端Socket</param>
257. **private** **void** SendDataEnd(IAsyncResult ar)
258. {
259. ((TcpClient)ar.AsyncState).GetStream().EndWrite(ar);
260. RaiseCompletedSend(**null**);
261. }
262. #endregion
264. #region 事件
266. /// <summary>
267. /// 与客户端的连接已建立事件
268. /// </summary>
269. **public** **event** EventHandler<AsyncEventArgs> ClientConnected;
270. /// <summary>
271. /// 与客户端的连接已断开事件
272. /// </summary>
273. **public** **event** EventHandler<AsyncEventArgs> ClientDisconnected;

276. /// <summary>
277. /// 触发客户端连接事件
278. /// </summary>
279. /// <param name="state"></param>
280. **private** **void** RaiseClientConnected(TCPClientState state)
281. {
282. **if** (ClientConnected != **null**)
283. {
284. ClientConnected(**this**, **new** AsyncEventArgs(state));
285. }
286. }
287. /// <summary>
288. /// 触发客户端连接断开事件
289. /// </summary>
290. /// <param name="client"></param>
291. **private** **void** RaiseClientDisconnected(TCPClientState state)
292. {
293. **if** (ClientDisconnected != **null**)
294. {
295. ClientDisconnected(**this**, **new** AsyncEventArgs("连接断开"));
296. }
297. }
299. /// <summary>
300. /// 接收到数据事件
301. /// </summary>
302. **public** **event** EventHandler<AsyncEventArgs> DataReceived;
304. **private** **void** RaiseDataReceived(TCPClientState state)
305. {
306. **if** (DataReceived != **null**)
307. {
308. DataReceived(**this**, **new** AsyncEventArgs(state));
309. }
310. }
312. /// <summary>
313. /// 发送数据前的事件
314. /// </summary>
315. **public** **event** EventHandler<AsyncEventArgs> PrepareSend;
317. /// <summary>
318. /// 触发发送数据前的事件
319. /// </summary>
320. /// <param name="state"></param>
321. **private** **void** RaisePrepareSend(TCPClientState state)
322. {
323. **if** (PrepareSend != **null**)
324. {
325. PrepareSend(**this**, **new** AsyncEventArgs(state));
326. }
327. }
329. /// <summary>
330. /// 数据发送完毕事件
331. /// </summary>
332. **public** **event** EventHandler<AsyncEventArgs> CompletedSend;
334. /// <summary>
335. /// 触发数据发送完毕的事件
336. /// </summary>
337. /// <param name="state"></param>
338. **private** **void** RaiseCompletedSend(TCPClientState state)
339. {
340. **if** (CompletedSend != **null**)
341. {
342. CompletedSend(**this**, **new** AsyncEventArgs(state));
343. }
344. }
346. /// <summary>
347. /// 网络错误事件
348. /// </summary>
349. **public** **event** EventHandler<AsyncEventArgs> NetError;
350. /// <summary>
351. /// 触发网络错误事件
352. /// </summary>
353. /// <param name="state"></param>
354. **private** **void** RaiseNetError(TCPClientState state)
355. {
356. **if** (NetError != **null**)
357. {
358. NetError(**this**, **new** AsyncEventArgs(state));
359. }
360. }
362. /// <summary>
363. /// 异常事件
364. /// </summary>
365. **public** **event** EventHandler<AsyncEventArgs> OtherException;
366. /// <summary>
367. /// 触发异常事件
368. /// </summary>
369. /// <param name="state"></param>
370. **private** **void** RaiseOtherException(TCPClientState state, **string** descrip)
371. {
372. **if** (OtherException != **null**)
373. {
374. OtherException(**this**, **new** AsyncEventArgs(descrip, state));
375. }
376. }
377. **private** **void** RaiseOtherException(TCPClientState state)
378. {
379. RaiseOtherException(state, "");
380. }
382. #endregion
384. #region Close
385. /// <summary>
386. /// 关闭一个与客户端之间的会话
387. /// </summary>
388. /// <param name="state">需要关闭的客户端会话对象</param>
389. **public** **void** Close(TCPClientState state)
390. {
391. **if** (state != **null**)
392. {
393. state.Close();
394. \_clients.Remove(state);
395. \_clientCount--;
396. //TODO 触发关闭事件
397. }
398. }
399. /// <summary>
400. /// 关闭所有的客户端会话,与所有的客户端连接会断开
401. /// </summary>
402. **public** **void** CloseAllClient()
403. {
404. **foreach** (TCPClientState client **in** \_clients)
405. {
406. Close(client);
407. }
408. \_clientCount = 0;
409. \_clients.Clear();
410. }
411. #endregion
413. #region 释放
414. /// <summary>
415. /// Performs application-defined tasks associated with freeing,
416. /// releasing, or resetting unmanaged resources.
417. /// </summary>
418. **public** **void** Dispose()
419. {
420. Dispose(**true**);
421. GC.SuppressFinalize(**this**);
422. }
424. /// <summary>
425. /// Releases unmanaged and - optionally - managed resources
426. /// </summary>
427. /// <param name="disposing"><c>true</c> to release
428. /// both managed and unmanaged resources; <c>false</c>
429. /// to release only unmanaged resources.</param>
430. **protected** **virtual** **void** Dispose(**bool** disposing)
431. {
432. **if** (!**this**.disposed)
433. {
434. **if** (disposing)
435. {
436. **try**
437. {
438. Stop();
439. **if** (\_listener != **null**)
440. {
441. \_listener = **null**;
442. }
443. }
444. **catch** (SocketException)
445. {
446. //TODO
447. RaiseOtherException(**null**);
448. }
449. }
450. disposed = **true**;
451. }
452. }
453. #endregion
454. }
455. }

客户端处理封装类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261497) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
7. **namespace** NetFrame.Net.TCP.Listener.Asynchronous
8. {
9. **public** **class** TCPClientState
10. {
11. /// <summary>
12. /// 与客户端相关的TcpClient
13. /// </summary>
14. **public** TcpClient TcpClient { **get**; **private** **set**; }
16. /// <summary>
17. /// 获取缓冲区
18. /// </summary>
19. **public** **byte**[] Buffer { **get**; **private** **set**; }
21. /// <summary>
22. /// 获取网络流
23. /// </summary>
24. **public** NetworkStream NetworkStream
25. {
26. **get** { **return** TcpClient.GetStream(); }
27. }
29. **public** TCPClientState(TcpClient tcpClient, **byte**[] buffer)
30. {
31. **if** (tcpClient == **null**)
32. **throw** **new** ArgumentNullException("tcpClient");
33. **if** (buffer == **null**)
34. **throw** **new** ArgumentNullException("buffer");
36. **this**.TcpClient = tcpClient;
37. **this**.Buffer = buffer;
38. }
39. /// <summary>
40. /// 关闭
41. /// </summary>
42. **public** **void** Close()
43. {
44. //关闭数据的接受和发送
45. TcpClient.Close();
46. Buffer = **null**;
47. }
48. }
49. }

服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44261497) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.TCP.Listener.Asynchronous
7. {
8. /// <summary>
9. /// 异步TcpListener TCP服务器事件参数类
10. /// </summary>
11. **public** **class** AsyncEventArgs:EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** TCPClientState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** AsyncEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** AsyncEventArgs(TCPClientState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** AsyncEventArgs(**string** msg, TCPClientState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

本文作者：小竹zz 本文地址http://blog.csdn.net/zhujunxxxxx/article/details/44258719 转载请注明出处

顶

3

踩

0

* 上一篇[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)
* 下一篇[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[**[置顶] C#网络编程系列文章(四)之TcpListener实现同步TCP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

2015-03-15 18:03 1125人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44278861#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44278861#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

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/zhujunxxxxx/article/details/44278861)

**原创性声明**

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

TcpListener 类提供一些简单方法，用于在阻止同步模式下侦听和接受传入连接请求。 可使用 TcpClient 或 Socket 来连接 TcpListener。 可使用 IPEndPoint、本地 IP 地址及端口号或者仅使用端口号，来创建 TcpListener。 可以将本地 IP 地址指定为 Any，将本地端口号指定为 0（如果希望基础服务提供程序为您分配这些值）。 如果您选择这样做，可在连接套接字后使用 LocalEndpoint 属性来标识已指定的信息。使用 Start 方法，可开始侦听传入的连接请求。 Start 将对传入连接进行排队，直至您调用 Stop 方法或它已经完成 MaxConnections 排队为止。 可使用 AcceptSocket 或 AcceptTcpClient 从传入连接请求队列提取连接。 这两种方法将阻止。 如果要避免阻止，可首先使用 Pending 方法来确定队列中是否有可用的连接请求。  
虽然TcpListener已经封装的比较不错了，我们于是就使用它在构造一个比较不错的同步TCP服务器，这里依然和前两章一样，给出服务器中的代码，代码中注释很详细，我也会给出相关的封装类。

**TcpListener同步TCP服务器**

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44278861) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **using** System.Threading;
9. **namespace** NetFrame.Net.TCP.Listener.Synchronous
10. {
11. /// <summary>
12. /// TcpListener实现同步TCP服务器
13. /// </summary>
14. **public** **class** TCPServer
15. {
16. #region Fields
17. /// <summary>
18. /// 服务器程序允许的最大客户端连接数
19. /// </summary>
20. **private** **int** \_maxClient;
22. /// <summary>
23. /// 当前的连接的客户端数
24. /// </summary>
25. **private** **int** \_clientCount;
27. /// <summary>
28. /// 服务器使用的异步TcpListener
29. /// </summary>
30. **private** TcpListener \_listener;
32. /// <summary>
33. /// 客户端会话列表
34. /// </summary>
35. **private** List<TCPClientHandle> \_clients;
37. **private** **bool** disposed = **false**;
39. #endregion
41. #region Properties
43. /// <summary>
44. /// 服务器是否正在运行
45. /// </summary>
46. **public** **bool** IsRunning { **get**; **private** **set**; }
47. /// <summary>
48. /// 监听的IP地址
49. /// </summary>
50. **public** IPAddress Address { **get**; **private** **set**; }
51. /// <summary>
52. /// 监听的端口
53. /// </summary>
54. **public** **int** Port { **get**; **private** **set**; }
55. /// <summary>
56. /// 通信使用的编码
57. /// </summary>
58. **public** Encoding Encoding { **get**; **set**; }
60. #endregion
62. #region 构造器
63. /// <summary>
64. /// 同步TCP服务器
65. /// </summary>
66. /// <param name="listenPort">监听的端口</param>
67. **public** TCPServer(**int** listenPort)
68. : **this**(IPAddress.Any, listenPort, 1024)
69. {
70. }
72. /// <summary>
73. /// 同步TCP服务器
74. /// </summary>
75. /// <param name="localEP">监听的终结点</param>
76. **public** TCPServer(IPEndPoint localEP)
77. : **this**(localEP.Address, localEP.Port, 1024)
78. {
79. }
81. /// <summary>
82. /// 同步TCP服务器
83. /// </summary>
84. /// <param name="localIPAddress">监听的IP地址</param>
85. /// <param name="listenPort">监听的端口</param>
86. /// <param name="maxClient">最大客户端数量</param>
87. **public** TCPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
88. {
89. **this**.Address = localIPAddress;
90. **this**.Port = listenPort;
91. **this**.Encoding = Encoding.Default;
93. \_maxClient = maxClient;
94. \_clients = **new** List<TCPClientHandle>();
95. \_listener = **new** TcpListener(**new** IPEndPoint(**this**.Address, **this**.Port));
96. }
98. #endregion
100. #region Method
101. /// <summary>
102. /// 启动服务器
103. /// </summary>
104. **public** **void** Start()
105. {
106. **if** (!IsRunning)
107. {
108. IsRunning = **true**;
109. \_listener.Start();
110. Thread thread = **new** Thread(Accept);
111. thread.Start();
112. }
113. }
114. /// <summary>
115. /// 开始进行监听
116. /// </summary>
117. **private** **void** Accept()
118. {
119. TCPClientHandle handle;
120. **while** (IsRunning)
121. {
122. TcpClient client = \_listener.AcceptTcpClient();
123. **if** (\_clientCount >= \_maxClient)
124. {
125. //TODO 触发事件
126. }
127. **else**
128. {
129. handle = **new** TCPClientHandle(client);
130. \_clientCount++;
131. \_clients.Add(handle);
133. //TODO 创建一个处理客户端的线程并启动
134. //使用线程池来操作
135. **new** Thread(**new** ThreadStart(handle.RecevieData)).Start();
136. }
137. }
139. }
140. /// <summary>
141. /// 停止服务器
142. /// </summary>
143. **public** **void** Stop()
144. {
145. **if** (IsRunning)
146. {
147. IsRunning = **false**;
148. \_listener.Stop();
149. //TODO 关闭对所有客户端的连接
150. }
151. }
152. /// <summary>
153. /// 发送函数
154. /// </summary>
155. **public** **void** Send(**string** msg, TcpClient client)
156. {
157. //TODO
158. }
160. #endregion
162. #region 事件
164. /// <summary>
165. /// 与客户端的连接已建立事件
166. /// </summary>
167. **public** **event** EventHandler<TCPEventArgs> ClientConnected;
168. /// <summary>
169. /// 与客户端的连接已断开事件
170. /// </summary>
171. **public** **event** EventHandler<TCPEventArgs> ClientDisconnected;
173. /// <summary>
174. /// 触发客户端连接事件
175. /// </summary>
176. /// <param name="state"></param>
177. **private** **void** RaiseClientConnected(TCPClientHandle handle)
178. {
179. **if** (ClientConnected != **null**)
180. {
181. ClientConnected(**this**, **new** TCPEventArgs(handle));
182. }
183. }
184. /// <summary>
185. /// 触发客户端连接断开事件
186. /// </summary>
187. /// <param name="client"></param>
188. **private** **void** RaiseClientDisconnected(Socket client)
189. {
190. **if** (ClientDisconnected != **null**)
191. {
192. ClientDisconnected(**this**, **new** TCPEventArgs("连接断开"));
193. }
194. }
196. /// <summary>
197. /// 接收到数据事件
198. /// </summary>
199. **public** **event** EventHandler<TCPEventArgs> DataReceived;
201. **private** **void** RaiseDataReceived(TCPClientHandle handle)
202. {
203. **if** (DataReceived != **null**)
204. {
205. DataReceived(**this**, **new** TCPEventArgs(handle));
206. }
207. }
209. /// <summary>
210. /// 数据发送事件
211. /// </summary>
212. **public** **event** EventHandler<TCPEventArgs> CompletedSend;
214. /// <summary>
215. /// 触发数据发送事件
216. /// </summary>
217. /// <param name="state"></param>
218. **private** **void** RaiseCompletedSend(TCPClientHandle handle)
219. {
220. **if** (CompletedSend != **null**)
221. {
222. CompletedSend(**this**, **new** TCPEventArgs(handle));
223. }
224. }

227. /// <summary>
228. /// 网络错误事件
229. /// </summary>
230. **public** **event** EventHandler<TCPEventArgs> NetError;
231. /// <summary>
232. /// 触发网络错误事件
233. /// </summary>
234. /// <param name="state"></param>
235. **private** **void** RaiseNetError(TCPClientHandle handle)
236. {
237. **if** (NetError != **null**)
238. {
239. NetError(**this**, **new** TCPEventArgs(handle));
240. }
241. }
243. /// <summary>
244. /// 异常事件
245. /// </summary>
246. **public** **event** EventHandler<TCPEventArgs> OtherException;
247. /// <summary>
248. /// 触发异常事件
249. /// </summary>
250. /// <param name="state"></param>
251. **private** **void** RaiseOtherException(TCPClientHandle handle, **string** descrip)
252. {
253. **if** (OtherException != **null**)
254. {
255. OtherException(**this**, **new** TCPEventArgs(descrip, handle));
256. }
257. }
258. **private** **void** RaiseOtherException(TCPClientHandle handle)
259. {
260. RaiseOtherException(handle, "");
261. }
263. #endregion
265. #region Close
267. /// <summary>
268. /// 关闭一个与客户端之间的会话
269. /// </summary>
270. /// <param name="handle">需要关闭的客户端会话对象</param>
271. **public** **void** Close(TCPClientHandle handle)
272. {
273. **if** (handle != **null**)
274. {
275. \_clients.Remove(handle);
276. handle.Dispose();
277. \_clientCount--;
278. //TODO 触发关闭事件
280. }
281. }
282. /// <summary>
283. /// 关闭所有的客户端会话,与所有的客户端连接会断开
284. /// </summary>
285. **public** **void** CloseAllClient()
286. {
287. **foreach** (TCPClientHandle handle **in** \_clients)
288. {
289. Close(handle);
290. }
291. \_clientCount = 0;
292. \_clients.Clear();
293. }
294. #endregion
296. #region 释放
297. /// <summary>
298. /// Performs application-defined tasks associated with freeing,
299. /// releasing, or resetting unmanaged resources.
300. /// </summary>
301. **public** **void** Dispose()
302. {
303. Dispose(**true**);
304. GC.SuppressFinalize(**this**);
305. }
307. /// <summary>
308. /// Releases unmanaged and - optionally - managed resources
309. /// </summary>
310. /// <param name="disposing"><c>true</c> to release
311. /// both managed and unmanaged resources; <c>false</c>
312. /// to release only unmanaged resources.</param>
313. **protected** **virtual** **void** Dispose(**bool** disposing)
314. {
315. **if** (!**this**.disposed)
316. {
317. **if** (disposing)
318. {
319. **try**
320. {
321. Stop();
322. **if** (\_listener != **null**)
323. {
324. \_listener = **null**;
325. }
326. }
327. **catch** (SocketException)
328. {
329. //TODO 异常
330. }
331. }
332. disposed = **true**;
333. }
334. }
335. #endregion
336. }
337. }

客户端处理封装类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44278861) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.IO;
8. **namespace** NetFrame.Net.TCP.Listener.Synchronous
9. {
10. /// <summary>
11. /// TcpListener实现同步TCP服务器 的客户端连接处理类
12. /// </summary>
13. **public** **class** TCPClientHandle
14. {
15. **private** TcpClient \_tcpclient;
17. **private** BinaryReader rs;
19. **private** BinaryWriter ws;
21. /// <summary>
22. /// 标识是否与客户端相连接
23. /// </summary>
24. **private** **bool** \_is\_connect;
25. **public** **bool** IsConnect
26. {
27. **get** { **return** \_is\_connect; }
28. **set** { \_is\_connect = value; }
29. }
31. /// <summary>
32. /// 数据接受缓冲区
33. /// </summary>
34. **private** **byte**[] \_recvBuffer;
36. **public** TCPClientHandle(TcpClient client)
37. {
38. \_tcpclient = client;
39. rs = **new** BinaryReader(client.GetStream());
40. ws = **new** BinaryWriter(client.GetStream());
41. // NetworkStream ns = tmpTcpClient.GetStream();
42. // if(ns.CanRead&&ns.CanWrite)
43. \_recvBuffer=**new** **byte**[client.ReceiveBufferSize];
44. }
46. /// <summary>
47. /// 接受数据
48. /// </summary>
49. **public** **void** RecevieData()
50. {
51. **int** len = 0;
52. **while** (\_is\_connect)
53. {
54. **try**
55. {
56. len = rs.Read(\_recvBuffer, 0, \_recvBuffer.Length);
57. }
58. **catch** (Exception)
59. {
60. **break**;
61. }
62. **if** (len == 0)
63. {
64. //the client has disconnected from server
65. **break**;
66. }
67. //TODO 处理收到的数据
69. }
70. }
71. /// <summary>
72. /// 向客户端发送数据
73. /// </summary>
74. /// <param name="msg"></param>
75. **public** **void** SendData(**string** msg)
76. {
77. **byte**[] data = Encoding.Default.GetBytes(msg);
78. **try**
79. {
80. ws.Write(data, 0, data.Length);
81. ws.Flush();
82. }
83. **catch** (Exception)
84. {
85. //TODO 处理异常
86. }
87. }
89. #region 事件

92. //TODO 消息发送事件
93. //TODO 数据收到事件
94. //TODO 异常处理事件
96. #endregion
98. #region 释放
99. /// <summary>
100. /// Performs application-defined tasks associated with freeing,
101. /// releasing, or resetting unmanaged resources.
102. /// </summary>
103. **public** **void** Dispose()
104. {
105. \_is\_connect = **false**;
106. **if** (\_tcpclient != **null**)
107. {
108. \_tcpclient.Close();
109. \_tcpclient = **null**;
110. }
111. GC.SuppressFinalize(**this**);
112. }
114. #endregion
115. }
116. }

Tcp服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44278861) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.TCP.Listener.Synchronous
7. {
8. /// <summary>
9. /// 同步TcpListener TCP服务器事件类
10. /// </summary>
11. **public** **class** TCPEventArgs : EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** TCPClientHandle \_handle;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** TCPEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** TCPEventArgs(TCPClientHandle handle)
34. {
35. **this**.\_handle = handle;
36. IsHandled = **false**;
37. }
38. **public** TCPEventArgs(**string** msg, TCPClientHandle handle)
39. {
40. **this**.\_msg = msg;
41. **this**.\_handle = handle;
42. IsHandled = **false**;
43. }
44. }
45. }

顶

0

踩

0

* 上一篇[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)
* 下一篇[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[**[置顶] C#网络编程系列文章(五)之Socket实现异步UDP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

2015-03-15 21:20 3232人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44280959#comments)(3) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44280959#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

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/zhujunxxxxx/article/details/44280959)

**原创性声明**

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

在.Net中，System.Net.Sockets 命名空间为需要严密控制网络访问的开发人员提供了 Windows Sockets (Winsock) 接口的托管实现。System.Net 命名空间中的所有其他网络访问类都建立在该套接字Socket实现之上，如TCPClient、TCPListener 和 UDPClient 类封装有关创建到 Internet 的 TCP 和 UDP 连接的详细信息；NetworkStream类则提供用于网络访问的基础数据流等，常见的许多Internet服务都可以见到Socket的踪影，如Telnet、Http、Email、Echo等，这些服务尽管通讯协议Protocol的定义不同，但是其基础的传输都是采用的Socket。 其实，Socket可以象流Stream一样被视为一个数据通道，这个通道架设在应用程序端（客户端）和远程服务器端之间，而后，数据的读取（接收）和写入（发送）均针对这个通道来进行。   
可见，在应用程序端或者服务器端创建了Socket对象之后，就可以使用Send/SentTo方法将数据发送到连接的Socket,或者使用Receive/ReceiveFrom方法接收来自连接Socket的数据；   
针对Socket编程，.NET 框架的 Socket 类是 Winsock32 API 提供的套接字服务的托管代码版本。其中为实现网络编程提供了大量的方法，大多数情况下，Socket 类方法只是将数据封送到它们的本机 Win32 副本中并处理任何必要的安全检查。如果你熟悉Winsock API函数，那么用Socket类编写网络程序会非常容易，当然，如果你不曾接触过，也不会太困难，跟随下面的解说，你会发觉使用Socket类开发windows 网络应用程序原来有规可寻，它们在大多数情况下遵循大致相同的步骤。

本节介绍使用Socket来实现一个高性能的异步UDP服务器，实际上UDP是不分客户机和服务器的，但是我们有的时候和服务器通讯就是使用UDP来进行的。

**Socket异步UDP服务器**

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44280959) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net;
6. **using** System.Net.Sockets;
8. **namespace** NetFrame.Net.UDP.Sock.Asynchronous
9. {
10. /// <summary>
11. /// SOCKET实现异步UDP服务器
12. /// </summary>
13. **public** **class** AsyncSocketUDPServer
14. {
15. #region Fields
16. /// <summary>
17. /// 服务器程序允许的最大客户端连接数
18. /// </summary>
19. **private** **int** \_maxClient;
21. /// <summary>
22. /// 当前的连接的客户端数
23. /// </summary>
24. //private int \_clientCount;
26. /// <summary>
27. /// 服务器使用的同步socket
28. /// </summary>
29. **private** Socket \_serverSock;
31. /// <summary>
32. /// 客户端会话列表
33. /// </summary>
34. //private List<AsyncUDPSocketState> \_clients;
36. **private** **bool** disposed = **false**;
38. /// <summary>
39. /// 数据接受缓冲区
40. /// </summary>
41. **private** **byte**[] \_recvBuffer;
43. #endregion
45. #region Properties
47. /// <summary>
48. /// 服务器是否正在运行
49. /// </summary>
50. **public** **bool** IsRunning { **get**; **private** **set**; }
51. /// <summary>
52. /// 监听的IP地址
53. /// </summary>
54. **public** IPAddress Address { **get**; **private** **set**; }
55. /// <summary>
56. /// 监听的端口
57. /// </summary>
58. **public** **int** Port { **get**; **private** **set**; }
59. /// <summary>
60. /// 通信使用的编码
61. /// </summary>
62. **public** Encoding Encoding { **get**; **set**; }
64. #endregion
66. #region 构造函数
68. /// <summary>
69. /// 异步Socket UDP服务器
70. /// </summary>
71. /// <param name="listenPort">监听的端口</param>
72. **public** AsyncSocketUDPServer(**int** listenPort)
73. : **this**(IPAddress.Any, listenPort,1024)
74. {
75. }
77. /// <summary>
78. /// 异步Socket UDP服务器
79. /// </summary>
80. /// <param name="localEP">监听的终结点</param>
81. **public** AsyncSocketUDPServer(IPEndPoint localEP)
82. : **this**(localEP.Address, localEP.Port,1024)
83. {
84. }
86. /// <summary>
87. /// 异步Socket UDP服务器
88. /// </summary>
89. /// <param name="localIPAddress">监听的IP地址</param>
90. /// <param name="listenPort">监听的端口</param>
91. /// <param name="maxClient">最大客户端数量</param>
92. **public** AsyncSocketUDPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
93. {
94. **this**.Address = localIPAddress;
95. **this**.Port = listenPort;
96. **this**.Encoding = Encoding.Default;
98. \_maxClient = maxClient;
99. //\_clients = new List<AsyncUDPSocketState>();
100. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Dgram, ProtocolType.Udp);
102. \_recvBuffer=**new** **byte**[\_serverSock.ReceiveBufferSize];
103. }
105. #endregion
107. #region Method
108. /// <summary>
109. /// 启动服务器
110. /// </summary>
111. /// <returns>异步TCP服务器</returns>
112. **public** **void** Start()
113. {
114. **if** (!IsRunning)
115. {
116. IsRunning = **true**;
117. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
118. //\_serverSock.Connect(new IPEndPoint(IPAddress.Any, 0));
120. AsyncSocketUDPState so = **new** AsyncSocketUDPState();
121. so.workSocket = \_serverSock;
123. \_serverSock.BeginReceiveFrom(so.buffer, 0, so.buffer.Length, SocketFlags.None,
124. **ref** so.remote, **new** AsyncCallback(ReceiveDataAsync), **null**);

127. //EndPoint sender = new IPEndPoint(IPAddress.Any, 0);
129. //\_serverSock.BeginReceiveFrom(\_recvBuffer, 0, \_recvBuffer.Length, SocketFlags.None,
130. //    ref sender, new AsyncCallback(ReceiveDataAsync), sender);
132. //BeginReceive 和 BeginReceiveFrom的区别是什么
133. /\*\_serverSock.BeginReceive(\_recvBuffer, 0, \_recvBuffer.Length, SocketFlags.None,
134. new AsyncCallback(ReceiveDataAsync), null);\*/
135. }
136. }
138. /// <summary>
139. /// 停止服务器
140. /// </summary>
141. **public** **void** Stop()
142. {
143. **if** (IsRunning)
144. {
145. IsRunning = **false**;
146. \_serverSock.Close();
147. //TODO 关闭对所有客户端的连接
149. }
150. }
152. /// <summary>
153. /// 接收数据的方法
154. /// </summary>
155. /// <param name="ar"></param>
156. **private** **void** ReceiveDataAsync(IAsyncResult ar)
157. {
158. AsyncSocketUDPState so = ar.AsyncState **as** AsyncSocketUDPState;
159. //EndPoint sender = new IPEndPoint(IPAddress.Any, 0);
160. **int** len = -1;
161. **try**
162. {
163. len = \_serverSock.EndReceiveFrom(ar, **ref** so.remote);
165. //len = \_serverSock.EndReceiveFrom(ar, ref sender);
167. //EndReceiveFrom 和 EndReceive区别
168. //len = \_serverSock.EndReceive(ar);
169. //TODO 处理数据
171. //触发数据收到事件
172. RaiseDataReceived(so);
173. }
174. **catch** (Exception)
175. {
176. //TODO 处理异常
177. RaiseOtherException(so);
178. }
179. **finally**
180. {
181. **if** (IsRunning && \_serverSock != **null**)
182. \_serverSock.BeginReceiveFrom(so.buffer, 0, so.buffer.Length, SocketFlags.None,
183. **ref** so.remote, **new** AsyncCallback(ReceiveDataAsync), so);
184. }
185. }
186. /// <summary>
187. /// 发送数据
188. /// </summary>
189. /// <param name="msg"></param>
190. /// <param name="remote"></param>
191. **public** **void** Send(**string** msg,EndPoint remote)
192. {
193. **byte**[] data = Encoding.Default.GetBytes(msg);
194. **try**
195. {
196. RaisePrepareSend(**null**);
197. \_serverSock.BeginSendTo(data, 0, data.Length, SocketFlags.None, remote, **new** AsyncCallback(SendDataEnd), \_serverSock);
198. }
199. **catch** (Exception)
200. {
201. //TODO 异常处理
202. RaiseOtherException(**null**);
203. }
204. }
206. **private** **void** SendDataEnd(IAsyncResult ar)
207. {
208. ((Socket)ar.AsyncState).EndSendTo(ar);
209. RaiseCompletedSend(**null**);
210. }
212. #endregion
214. #region 事件
215. /// <summary>
216. /// 接收到数据事件
217. /// </summary>
218. **public** **event** EventHandler<AsyncSocketUDPEventArgs> DataReceived;
220. **private** **void** RaiseDataReceived(AsyncSocketUDPState state)
221. {
222. **if** (DataReceived != **null**)
223. {
224. DataReceived(**this**, **new** AsyncSocketUDPEventArgs(state));
225. }
226. }
228. /// <summary>
229. /// 发送数据前的事件
230. /// </summary>
231. **public** **event** EventHandler<AsyncSocketUDPEventArgs> PrepareSend;
233. /// <summary>
234. /// 触发发送数据前的事件
235. /// </summary>
236. /// <param name="state"></param>
237. **private** **void** RaisePrepareSend(AsyncSocketUDPState state)
238. {
239. **if** (PrepareSend != **null**)
240. {
241. PrepareSend(**this**, **new** AsyncSocketUDPEventArgs(state));
242. }
243. }
245. /// <summary>
246. /// 数据发送完毕事件
247. /// </summary>
248. **public** **event** EventHandler<AsyncSocketUDPEventArgs> CompletedSend;
250. /// <summary>
251. /// 触发数据发送完毕的事件
252. /// </summary>
253. /// <param name="state"></param>
254. **private** **void** RaiseCompletedSend(AsyncSocketUDPState state)
255. {
256. **if** (CompletedSend != **null**)
257. {
258. CompletedSend(**this**, **new** AsyncSocketUDPEventArgs(state));
259. }
260. }
262. /// <summary>
263. /// 网络错误事件
264. /// </summary>
265. **public** **event** EventHandler<AsyncSocketUDPEventArgs> NetError;
266. /// <summary>
267. /// 触发网络错误事件
268. /// </summary>
269. /// <param name="state"></param>
270. **private** **void** RaiseNetError(AsyncSocketUDPState state)
271. {
272. **if** (NetError != **null**)
273. {
274. NetError(**this**, **new** AsyncSocketUDPEventArgs(state));
275. }
276. }
278. /// <summary>
279. /// 异常事件
280. /// </summary>
281. **public** **event** EventHandler<AsyncSocketUDPEventArgs> OtherException;
282. /// <summary>
283. /// 触发异常事件
284. /// </summary>
285. /// <param name="state"></param>
286. **private** **void** RaiseOtherException(AsyncSocketUDPState state, **string** descrip)
287. {
288. **if** (OtherException != **null**)
289. {
290. OtherException(**this**, **new** AsyncSocketUDPEventArgs(descrip, state));
291. }
292. }
293. **private** **void** RaiseOtherException(AsyncSocketUDPState state)
294. {
295. RaiseOtherException(state, "");
296. }
297. #endregion
299. #region Close
300. /// <summary>
301. /// 关闭一个与客户端之间的会话
302. /// </summary>
303. /// <param name="state">需要关闭的客户端会话对象</param>
304. **public** **void** Close(AsyncSocketUDPState state)
305. {
306. **if** (state != **null**)
307. {
308. //\_clients.Remove(state);
309. //\_clientCount--;
310. //TODO 触发关闭事件
311. }
312. }
313. /// <summary>
314. /// 关闭所有的客户端会话,与所有的客户端连接会断开
315. /// </summary>
316. **public** **void** CloseAllClient()
317. {
318. //foreach (AsyncUDPSocketState client in \_clients)
319. //{
320. //    Close(client);
321. //}
322. //\_clientCount = 0;
323. //\_clients.Clear();
324. }
326. #endregion
328. #region 释放
329. /// <summary>
330. /// Performs application-defined tasks associated with freeing,
331. /// releasing, or resetting unmanaged resources.
332. /// </summary>
333. **public** **void** Dispose()
334. {
335. Dispose(**true**);
336. GC.SuppressFinalize(**this**);
337. }
339. /// <summary>
340. /// Releases unmanaged and - optionally - managed resources
341. /// </summary>
342. /// <param name="disposing"><c>true</c> to release
343. /// both managed and unmanaged resources; <c>false</c>
344. /// to release only unmanaged resources.</param>
345. **protected** **virtual** **void** Dispose(**bool** disposing)
346. {
347. **if** (!**this**.disposed)
348. {
349. **if** (disposing)
350. {
351. **try**
352. {
353. Stop();
354. **if** (\_serverSock != **null**)
355. {
356. \_serverSock = **null**;
357. }
358. }
359. **catch** (SocketException)
360. {
361. //TODO
362. RaiseOtherException(**null**);
363. }
364. }
365. disposed = **true**;
366. }
367. }
368. #endregion
369. }
370. }

会话封装类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44280959) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net;
6. **using** System.Net.Sockets;
8. **namespace** NetFrame.Net.UDP.Sock.Asynchronous
9. {
10. **public** **class** AsyncSocketUDPState
11. {
12. // Client   socket.
13. **public** Socket workSocket = **null**;
14. // Size of receive buffer.
15. **public** **const** **int** BufferSize = 1024;
16. // Receive buffer.
17. **public** **byte**[] buffer = **new** **byte**[BufferSize];
18. // Received data string.
19. **public** StringBuilder sb = **new** StringBuilder();
21. **public** EndPoint remote = **new** IPEndPoint(IPAddress.Any, 0);
22. }
23. }

Socket异步UDP服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44280959) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.UDP.Sock.Asynchronous
7. {
8. /// <summary>
9. /// SOCKET 异步UDP 事件类
10. /// </summary>
11. **public** **class** AsyncSocketUDPEventArgs : EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** AsyncSocketUDPState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** AsyncSocketUDPEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** AsyncSocketUDPEventArgs(AsyncSocketUDPState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** AsyncSocketUDPEventArgs(**string** msg, AsyncSocketUDPState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

本文作者：小竹zz 本文地址http://blog.csdn.net/zhujunxxxxx/article/details/44258719 转载请注明出处

顶

0

踩

0

* 上一篇[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)
* 下一篇[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[**[置顶] C#网络编程系列文章(六)之Socket实现同步UDP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

2015-03-16 12:39 779人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44303697#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44303697#report)

 分类：

网络编程（16）  c#技术类（70）  UDP/TCP通讯（28）

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/zhujunxxxxx/article/details/44303697)

**原创性声明**

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

在.Net中，System.Net.Sockets 命名空间为需要严密控制网络访问的开发人员提供了 Windows Sockets (Winsock) 接口的托管实现。System.Net 命名空间中的所有其他网络访问类都建立在该套接字Socket实现之上，如TCPClient、TCPListener 和 UDPClient 类封装有关创建到 Internet 的 TCP 和 UDP 连接的详细信息；NetworkStream类则提供用于网络访问的基础数据流等，常见的许多Internet服务都可以见到Socket的踪影，如Telnet、Http、Email、Echo等，这些服务尽管通讯协议Protocol的定义不同，但是其基础的传输都是采用的Socket。 其实，Socket可以象流Stream一样被视为一个数据通道，这个通道架设在应用程序端（客户端）和远程服务器端之间，而后，数据的读取（接收）和写入（发送）均针对这个通道来进行。   
可见，在应用程序端或者服务器端创建了Socket对象之后，就可以使用Send/SentTo方法将数据发送到连接的Socket,或者使用Receive/ReceiveFrom方法接收来自连接Socket的数据；   
针对Socket编程，.NET 框架的 Socket 类是 Winsock32 API 提供的套接字服务的托管代码版本。其中为实现网络编程提供了大量的方法，大多数情况下，Socket 类方法只是将数据封送到它们的本机 Win32 副本中并处理任何必要的安全检查。如果你熟悉Winsock API函数，那么用Socket类编写网络程序会非常容易，当然，如果你不曾接触过，也不会太困难，跟随下面的解说，你会发觉使用Socket类开发windows 网络应用程序原来有规可寻，它们在大多数情况下遵循大致相同的步骤。

本节介绍使用Socket来实现一个同步的UDP服务器。

**Socket同步UDP服务器**

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303697) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **using** System.Threading;
9. **namespace** NetFrame.Net.UDP.Sock.Synchronous
10. {
11. /// <summary>
12. /// Socket 实现同步UDP服务器
13. /// </summary>
14. **public** **class** SocketUDPServer
15. {
16. #region Fields
17. /// <summary>
18. /// 服务器程序允许的最大客户端连接数
19. /// </summary>
20. **private** **int** \_maxClient;
22. /// <summary>
23. /// 当前的连接的客户端数
24. /// </summary>
25. **private** **int** \_clientCount;
27. /// <summary>
28. /// 服务器使用的同步socket
29. /// </summary>
30. **private** Socket \_serverSock;
32. /// <summary>
33. /// 客户端会话列表
34. /// </summary>
35. **private** List<SocketUDPState> \_clients;
37. **private** **bool** disposed = **false**;
39. /// <summary>
40. /// 数据接受缓冲区
41. /// </summary>
42. **private** **byte**[] \_recvBuffer;
44. #endregion
46. #region Properties
48. /// <summary>
49. /// 服务器是否正在运行
50. /// </summary>
51. **public** **bool** IsRunning { **get**; **private** **set**; }
52. /// <summary>
53. /// 监听的IP地址
54. /// </summary>
55. **public** IPAddress Address { **get**; **private** **set**; }
56. /// <summary>
57. /// 监听的端口
58. /// </summary>
59. **public** **int** Port { **get**; **private** **set**; }
60. /// <summary>
61. /// 通信使用的编码
62. /// </summary>
63. **public** Encoding Encoding { **get**; **set**; }
65. #endregion
67. #region 构造函数
69. /// <summary>
70. /// 异步Socket UDP服务器
71. /// </summary>
72. /// <param name="listenPort">监听的端口</param>
73. **public** SocketUDPServer(**int** listenPort)
74. : **this**(IPAddress.Any, listenPort,1024)
75. {
76. }
78. /// <summary>
79. /// 异步Socket UDP服务器
80. /// </summary>
81. /// <param name="localEP">监听的终结点</param>
82. **public** SocketUDPServer(IPEndPoint localEP)
83. : **this**(localEP.Address, localEP.Port,1024)
84. {
85. }
87. /// <summary>
88. /// 异步Socket UDP服务器
89. /// </summary>
90. /// <param name="localIPAddress">监听的IP地址</param>
91. /// <param name="listenPort">监听的端口</param>
92. /// <param name="maxClient">最大客户端数量</param>
93. **public** SocketUDPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
94. {
95. **this**.Address = localIPAddress;
96. **this**.Port = listenPort;
97. **this**.Encoding = Encoding.Default;
99. \_maxClient = maxClient;
100. \_clients = **new** List<SocketUDPState>();
101. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Dgram, ProtocolType.Udp);
103. \_recvBuffer=**new** **byte**[\_serverSock.ReceiveBufferSize];
104. }
106. #endregion
108. #region Method
109. /// <summary>
110. /// 启动服务器
111. /// </summary>
112. /// <returns>异步TCP服务器</returns>
113. **public** **void** Start()
114. {
115. **if** (!IsRunning)
116. {
117. IsRunning = **true**;
118. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
119. //启动一个线程监听数据
120. **new** Thread(ReceiveData).Start();
121. }
122. }
124. /// <summary>
125. /// 停止服务器
126. /// </summary>
127. **public** **void** Stop()
128. {
129. **if** (IsRunning)
130. {
131. IsRunning = **false**;
132. \_serverSock.Close();
133. //TODO 关闭对所有客户端的连接
134. CloseAllClient();
135. }
136. }
138. /// <summary>
139. /// 同步数据接收方法
140. /// </summary>
141. **private** **void** ReceiveData()
142. {
143. **int** len = -1;
144. EndPoint remote = **null**;
145. **while** (**true**)
146. {
147. **try**
148. {
149. len = \_serverSock.ReceiveFrom(\_recvBuffer, **ref** remote);
151. //if (!\_clients.Contains(remote))
152. //{
153. //    \_clients.Add(remote);
154. //}
155. }
156. **catch** (Exception)
157. {
158. //TODO 异常处理操作
159. RaiseOtherException(**null**);
160. }
161. }
162. }
163. /// <summary>
164. /// 同步发送数据
165. /// </summary>
166. **public** **void** Send(**string** msg, EndPoint clientip)
167. {
168. **byte**[] data = Encoding.Default.GetBytes(msg);
169. **try**
170. {
171. \_serverSock.SendTo(data, clientip);
172. //数据发送完成事件
173. RaiseCompletedSend(**null**);
174. }
175. **catch** (Exception)
176. {
177. //TODO 异常处理
178. RaiseOtherException(**null**);
179. }
180. }
182. #endregion
184. #region 事件
185. /// <summary>
186. /// 接收到数据事件
187. /// </summary>
188. **public** **event** EventHandler<SocketUDPEventArgs> DataReceived;
190. **private** **void** RaiseDataReceived(SocketUDPState state)
191. {
192. **if** (DataReceived != **null**)
193. {
194. DataReceived(**this**, **new** SocketUDPEventArgs(state));
195. }
196. }
198. /// <summary>
199. /// 数据发送完毕事件
200. /// </summary>
201. **public** **event** EventHandler<SocketUDPEventArgs> CompletedSend;
203. /// <summary>
204. /// 触发数据发送完毕的事件
205. /// </summary>
206. /// <param name="state"></param>
207. **private** **void** RaiseCompletedSend(SocketUDPState state)
208. {
209. **if** (CompletedSend != **null**)
210. {
211. CompletedSend(**this**, **new** SocketUDPEventArgs(state));
212. }
213. }
215. /// <summary>
216. /// 网络错误事件
217. /// </summary>
218. **public** **event** EventHandler<SocketUDPEventArgs> NetError;
219. /// <summary>
220. /// 触发网络错误事件
221. /// </summary>
222. /// <param name="state"></param>
223. **private** **void** RaiseNetError(SocketUDPState state)
224. {
225. **if** (NetError != **null**)
226. {
227. NetError(**this**, **new** SocketUDPEventArgs(state));
228. }
229. }
231. /// <summary>
232. /// 异常事件
233. /// </summary>
234. **public** **event** EventHandler<SocketUDPEventArgs> OtherException;
235. /// <summary>
236. /// 触发异常事件
237. /// </summary>
238. /// <param name="state"></param>
239. **private** **void** RaiseOtherException(SocketUDPState state, **string** descrip)
240. {
241. **if** (OtherException != **null**)
242. {
243. OtherException(**this**, **new** SocketUDPEventArgs(descrip, state));
244. }
245. }
246. **private** **void** RaiseOtherException(SocketUDPState state)
247. {
248. RaiseOtherException(state, "");
249. }
250. #endregion
252. #region Close
253. /// <summary>
254. /// 关闭一个与客户端之间的会话
255. /// </summary>
256. /// <param name="state">需要关闭的客户端会话对象</param>
257. **public** **void** Close(SocketUDPState state)
258. {
259. **if** (state != **null**)
260. {
261. \_clients.Remove(state);
262. \_clientCount--;
263. //TODO 触发关闭事件
264. }
265. }
266. /// <summary>
267. /// 关闭所有的客户端会话,与所有的客户端连接会断开
268. /// </summary>
269. **public** **void** CloseAllClient()
270. {
271. **foreach** (SocketUDPState client **in** \_clients)
272. {
273. Close(client);
274. }
275. \_clientCount = 0;
276. \_clients.Clear();
277. }
278. #endregion
280. #region 释放
281. /// <summary>
282. /// Performs application-defined tasks associated with freeing,
283. /// releasing, or resetting unmanaged resources.
284. /// </summary>
285. **public** **void** Dispose()
286. {
287. Dispose(**true**);
288. GC.SuppressFinalize(**this**);
289. }
291. /// <summary>
292. /// Releases unmanaged and - optionally - managed resources
293. /// </summary>
294. /// <param name="disposing"><c>true</c> to release
295. /// both managed and unmanaged resources; <c>false</c>
296. /// to release only unmanaged resources.</param>
297. **protected** **virtual** **void** Dispose(**bool** disposing)
298. {
299. **if** (!**this**.disposed)
300. {
301. **if** (disposing)
302. {
303. **try**
304. {
305. Stop();
306. **if** (\_serverSock != **null**)
307. {
308. \_serverSock = **null**;
309. }
310. }
311. **catch** (SocketException)
312. {
313. //TODO
314. RaiseOtherException(**null**);
315. }
316. }
317. disposed = **true**;
318. }
319. }
320. #endregion
321. }
322. }

客户状态封装类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303697) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
8. **namespace** NetFrame.Net.UDP.Sock.Synchronous
9. {
10. **public** **class** SocketUDPState
11. {
12. // Client   socket.
13. **public** Socket workSocket = **null**;
14. // Size of receive buffer.
15. **public** **const** **int** BufferSize = 1024;
16. // Receive buffer.
17. **public** **byte**[] buffer = **new** **byte**[BufferSize];
18. // Received data string.
19. **public** StringBuilder sb = **new** StringBuilder();
21. **public** EndPoint remote = **new** IPEndPoint(IPAddress.Any, 0);
22. }
23. }

服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303697) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.UDP.Sock.Synchronous
7. {
8. /// <summary>
9. /// Socket实现同步UDP服务器
10. /// </summary>
11. **public** **class** SocketUDPEventArgs:EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** SocketUDPState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** SocketUDPEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** SocketUDPEventArgs(SocketUDPState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** SocketUDPEventArgs(**string** msg, SocketUDPState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

顶

0

踩

0

* 上一篇[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)
* 下一篇[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[**[置顶] C#网络编程系列文章(七)之UdpClient实现异步UDP服务器**](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

2015-03-16 12:48 1160人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44303743#comments)(1) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44303743#report)

 分类：

UDP/TCP通讯（28）  c#技术类（70）  网络编程（16）

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/zhujunxxxxx/article/details/44303743)

**原创性声明**

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

**文章系列目录**

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

**本文介绍**

UdpClient 类在同步阻塞模式中为发送和接收无连接的 UDP 数据包而提供了简单的方法。因为 UDP 是一种无连接的传输协议，所以你不需要在发送和接收数据之前建立任何远程主机连接。你只需要按照下列方式来建立默认的远程主机选项：  
使用远程主机名称和端口号作为参数来创建 UdpClient 类的实例。  
创建 UdpClient 类的实例然后调用 Connect 方法。  
你可以使用任何由 UdpClient 所提供的发送方法把数据发送给远程设备。然后使用 Receive 方法来接收来自于远程主机的数据。  
提示：如果你已经指定了一个默认的远程主机，就不要使用主机名称或者 IPEndPoint 来调用 Send 方法。如果你这样做，那么 UdpClient 将会抛出一个异常。  
UdpClient 方法同样允许你发送和接收多点传送的数据包。而使用 JoinMulticastGroup 方法可以把 UdpClient 订阅到多点传送分组。也可以使用 DropMulticastGroup 方法把 UdpClient 从多点传送分组的订阅中取消。

本节介绍使用UdpClient实现一个异步的高性能UDP服务器

**UdpClient异步UDP服务器**

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303743) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
8. **namespace** NetFrame.Net.UDP.Listener.Asynchronous
9. {
10. /// <summary>
11. /// UdpClient 实现异步UDP服务器
12. /// </summary>
13. **public** **class** AsyncUDPServer
14. {
15. #region Fields
16. /// <summary>
17. /// 服务器程序允许的最大客户端连接数
18. /// </summary>
19. **private** **int** \_maxClient;
21. /// <summary>
22. /// 当前的连接的客户端数
23. /// </summary>
24. //private int \_clientCount;
26. /// <summary>
27. /// 服务器使用的异步UdpClient
28. /// </summary>
29. **private** UdpClient \_server;
31. /// <summary>
32. /// 客户端会话列表
33. /// </summary>
34. //private List<AsyncUDPState> \_clients;
36. **private** **bool** disposed = **false**;
38. /// <summary>
39. /// 数据接受缓冲区
40. /// </summary>
41. **private** **byte**[] \_recvBuffer;
43. #endregion
45. #region Properties
47. /// <summary>
48. /// 服务器是否正在运行
49. /// </summary>
50. **public** **bool** IsRunning { **get**; **private** **set**; }
51. /// <summary>
52. /// 监听的IP地址
53. /// </summary>
54. **public** IPAddress Address { **get**; **private** **set**; }
55. /// <summary>
56. /// 监听的端口
57. /// </summary>
58. **public** **int** Port { **get**; **private** **set**; }
59. /// <summary>
60. /// 通信使用的编码
61. /// </summary>
62. **public** Encoding Encoding { **get**; **set**; }
64. #endregion
66. #region 构造函数
68. /// <summary>
69. /// 异步UdpClient UDP服务器
70. /// </summary>
71. /// <param name="listenPort">监听的端口</param>
72. **public** AsyncUDPServer(**int** listenPort)
73. : **this**(IPAddress.Any, listenPort,1024)
74. {
75. }
77. /// <summary>
78. /// 异步UdpClient UDP服务器
79. /// </summary>
80. /// <param name="localEP">监听的终结点</param>
81. **public** AsyncUDPServer(IPEndPoint localEP)
82. : **this**(localEP.Address, localEP.Port,1024)
83. {
84. }
86. /// <summary>
87. /// 异步UdpClient UDP服务器
88. /// </summary>
89. /// <param name="localIPAddress">监听的IP地址</param>
90. /// <param name="listenPort">监听的端口</param>
91. /// <param name="maxClient">最大客户端数量</param>
92. **public** AsyncUDPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
93. {
94. **this**.Address = localIPAddress;
95. **this**.Port = listenPort;
96. **this**.Encoding = Encoding.Default;
98. \_maxClient = maxClient;
99. //\_clients = new List<AsyncUDPSocketState>();
100. \_server = **new** UdpClient(**new** IPEndPoint(**this**.Address, **this**.Port));
102. \_recvBuffer=**new** **byte**[\_server.Client.ReceiveBufferSize];
103. }
105. #endregion
107. #region Method
108. /// <summary>
109. /// 启动服务器
110. /// </summary>
111. /// <returns>异步TCP服务器</returns>
112. **public** **void** Start()
113. {
114. **if** (!IsRunning)
115. {
116. IsRunning = **true**;
117. \_server.EnableBroadcast = **true**;
118. \_server.BeginReceive(ReceiveDataAsync, **null**);
119. }
120. }
122. /// <summary>
123. /// 停止服务器
124. /// </summary>
125. **public** **void** Stop()
126. {
127. **if** (IsRunning)
128. {
129. IsRunning = **false**;
130. \_server.Close();
131. //TODO 关闭对所有客户端的连接
133. }
134. }
136. /// <summary>
137. /// 接收数据的方法
138. /// </summary>
139. /// <param name="ar"></param>
140. **private** **void** ReceiveDataAsync(IAsyncResult ar)
141. {
142. IPEndPoint remote=**null**;
143. **byte**[] buffer = **null**;
144. **try**
145. {
146. buffer = \_server.EndReceive(ar, **ref** remote);
148. //触发数据收到事件
149. RaiseDataReceived(**null**);
150. }
151. **catch** (Exception)
152. {
153. //TODO 处理异常
154. RaiseOtherException(**null**);
155. }
156. **finally**
157. {
158. **if** (IsRunning && \_server != **null**)
159. \_server.BeginReceive(ReceiveDataAsync, **null**);
160. }
161. }
163. /// <summary>
164. /// 发送数据
165. /// </summary>
166. /// <param name="msg"></param>
167. /// <param name="remote"></param>
168. **public** **void** Send(**string** msg, IPEndPoint remote)
169. {
170. **byte**[] data = Encoding.Default.GetBytes(msg);
171. **try**
172. {
173. RaisePrepareSend(**null**);
174. \_server.BeginSend(data, data.Length, **new** AsyncCallback(SendCallback), **null**);
175. }
176. **catch** (Exception)
177. {
178. //TODO 异常处理
179. RaiseOtherException(**null**);
180. }
181. }
183. **private** **void** SendCallback(IAsyncResult ar)
184. {
185. **if** (ar.IsCompleted)
186. {
187. **try**
188. {
189. \_server.EndSend(ar);
190. //消息发送完毕事件
191. RaiseCompletedSend(**null**);
192. }
193. **catch** (Exception)
194. {
195. //TODO 数据发送失败事件
196. RaiseOtherException(**null**);
197. }
198. }
200. }
201. #endregion
203. #region 事件
204. /// <summary>
205. /// 接收到数据事件
206. /// </summary>
207. **public** **event** EventHandler<AsyncUDPEventArgs> DataReceived;
209. **private** **void** RaiseDataReceived(AsyncUDPState state)
210. {
211. **if** (DataReceived != **null**)
212. {
213. DataReceived(**this**, **new** AsyncUDPEventArgs(state));
214. }
215. }
217. /// <summary>
218. /// 发送数据前的事件
219. /// </summary>
220. **public** **event** EventHandler<AsyncUDPEventArgs> PrepareSend;
222. /// <summary>
223. /// 触发发送数据前的事件
224. /// </summary>
225. /// <param name="state"></param>
226. **private** **void** RaisePrepareSend(AsyncUDPState state)
227. {
228. **if** (PrepareSend != **null**)
229. {
230. PrepareSend(**this**, **new** AsyncUDPEventArgs(state));
231. }
232. }
234. /// <summary>
235. /// 数据发送完毕事件
236. /// </summary>
237. **public** **event** EventHandler<AsyncUDPEventArgs> CompletedSend;
239. /// <summary>
240. /// 触发数据发送完毕的事件
241. /// </summary>
242. /// <param name="state"></param>
243. **private** **void** RaiseCompletedSend(AsyncUDPState state)
244. {
245. **if** (CompletedSend != **null**)
246. {
247. CompletedSend(**this**, **new** AsyncUDPEventArgs(state));
248. }
249. }
251. /// <summary>
252. /// 网络错误事件
253. /// </summary>
254. **public** **event** EventHandler<AsyncUDPEventArgs> NetError;
255. /// <summary>
256. /// 触发网络错误事件
257. /// </summary>
258. /// <param name="state"></param>
259. **private** **void** RaiseNetError(AsyncUDPState state)
260. {
261. **if** (NetError != **null**)
262. {
263. NetError(**this**, **new** AsyncUDPEventArgs(state));
264. }
265. }
267. /// <summary>
268. /// 异常事件
269. /// </summary>
270. **public** **event** EventHandler<AsyncUDPEventArgs> OtherException;
271. /// <summary>
272. /// 触发异常事件
273. /// </summary>
274. /// <param name="state"></param>
275. **private** **void** RaiseOtherException(AsyncUDPState state, **string** descrip)
276. {
277. **if** (OtherException != **null**)
278. {
279. OtherException(**this**, **new** AsyncUDPEventArgs(descrip, state));
280. }
281. }
282. **private** **void** RaiseOtherException(AsyncUDPState state)
283. {
284. RaiseOtherException(state, "");
285. }
286. #endregion
288. #region Close
289. /// <summary>
290. /// 关闭一个与客户端之间的会话
291. /// </summary>
292. /// <param name="state">需要关闭的客户端会话对象</param>
293. **public** **void** Close(AsyncUDPState state)
294. {
295. **if** (state != **null**)
296. {
297. //\_clients.Remove(state);
298. //\_clientCount--;
299. //TODO 触发关闭事件
300. }
301. }
302. /// <summary>
303. /// 关闭所有的客户端会话,与所有的客户端连接会断开
304. /// </summary>
305. **public** **void** CloseAllClient()
306. {
307. //foreach (AsyncUDPSocketState client in \_clients)
308. //{
309. //    Close(client);
310. //}
311. //\_clientCount = 0;
312. //\_clients.Clear();
313. }
315. #endregion
317. #region 释放
318. /// <summary>
319. /// Performs application-defined tasks associated with freeing,
320. /// releasing, or resetting unmanaged resources.
321. /// </summary>
322. **public** **void** Dispose()
323. {
324. Dispose(**true**);
325. GC.SuppressFinalize(**this**);
326. }
328. /// <summary>
329. /// Releases unmanaged and - optionally - managed resources
330. /// </summary>
331. /// <param name="disposing"><c>true</c> to release
332. /// both managed and unmanaged resources; <c>false</c>
333. /// to release only unmanaged resources.</param>
334. **protected** **virtual** **void** Dispose(**bool** disposing)
335. {
336. **if** (!**this**.disposed)
337. {
338. **if** (disposing)
339. {
340. **try**
341. {
342. Stop();
343. **if** (\_server != **null**)
344. {
345. \_server = **null**;
346. }
347. }
348. **catch** (SocketException)
349. {
350. //TODO
351. RaiseOtherException(**null**);
352. }
353. }
354. disposed = **true**;
355. }
356. }
357. #endregion
358. }
359. }

客户端信息封装了类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303743) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
8. **namespace** NetFrame.Net.UDP.Listener.Asynchronous
9. {
10. **public** **class** AsyncUDPState
11. {
12. // Client   socket.
13. **public** UdpClient udpClient = **null**;
14. // Size of receive buffer.
15. **public** **const** **int** BufferSize = 1024;
16. // Receive buffer.
17. **public** **byte**[] buffer = **new** **byte**[BufferSize];
18. // Received data string.
19. **public** StringBuilder sb = **new** StringBuilder();
21. **public** IPEndPoint remote = **new** IPEndPoint(IPAddress.Any, 0);
22. }
23. }

服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303743) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.UDP.Listener.Asynchronous
7. {
8. /// <summary>
9. /// UdpClient 异步UDP服务器事件参数类
10. /// </summary>
11. **public** **class** AsyncUDPEventArgs : EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** AsyncUDPState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** AsyncUDPEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** AsyncUDPEventArgs(AsyncUDPState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** AsyncUDPEventArgs(**string** msg, AsyncUDPState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

顶

0

踩

0

* 上一篇[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)
* 下一篇[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

# [[置顶] C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

2015-03-16 12:58 880人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44303787#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44303787#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

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/zhujunxxxxx/article/details/44303787)

## 原创性声明

本文作者：小竹zz 本文地址http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx/article/details/44258719 转载请注明出处

## 文章系列目录

[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

[C#网络编程系列文章(二)之Socket实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261375)

[C#网络编程系列文章(三)之TcpListener实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44261497)

[C#网络编程系列文章(四)之TcpListener实现同步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44278861)

[C#网络编程系列文章(五)之Socket实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44280959)

[C#网络编程系列文章(六)之Socket实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303697)

[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

[C#网络编程系列文章(八)之UdpClient实现同步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

## 本文介绍

UdpClient 类在同步阻塞模式中为发送和接收无连接的 UDP 数据包而提供了简单的方法。因为 UDP 是一种无连接的传输协议，所以你不需要在发送和接收数据之前建立任何远程主机连接。你只需要按照下列方式来建立默认的远程主机选项：  
使用远程主机名称和端口号作为参数来创建 UdpClient 类的实例。  
创建 UdpClient 类的实例然后调用 Connect 方法。  
你可以使用任何由 UdpClient 所提供的发送方法把数据发送给远程设备。然后使用 Receive 方法来接收来自于远程主机的数据。  
提示：如果你已经指定了一个默认的远程主机，就不要使用主机名称或者 IPEndPoint 来调用 Send 方法。如果你这样做，那么 UdpClient 将会抛出一个异常。

UdpClient 方法同样允许你发送和接收多点传送的数据包。而使用 JoinMulticastGroup 方法可以把 UdpClient 订阅到多点传送分组。也可以使用 DropMulticastGroup 方法把 UdpClient 从多点传送分组的订阅中取消。

本节介绍使用UdpClient实现一个同步的UDP服务器

## UdpClient同步UDP服务器

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303787) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **using** System.Threading;
9. **namespace** NetFrame.Net.UDP.Listener.Synchronous
10. {
11. /// <summary>
12. /// UdpClient 实现同步UDP服务器
13. /// </summary>
14. **public** **class** UDPServer
15. {
16. #region Fields
17. /// <summary>
18. /// 服务器程序允许的最大客户端连接数
19. /// </summary>
20. **private** **int** \_maxClient;
22. /// <summary>
23. /// 当前的连接的客户端数
24. /// </summary>
25. //private int \_clientCount;
27. /// <summary>
28. /// 服务器使用的异步UdpClient
29. /// </summary>
30. **private** UdpClient \_server;
32. /// <summary>
33. /// 客户端会话列表
34. /// </summary>
35. //private List<UDPState> \_clients;
37. **private** **bool** disposed = **false**;
39. /// <summary>
40. /// 数据接受缓冲区
41. /// </summary>
42. **private** **byte**[] \_recvBuffer;
44. #endregion
46. #region Properties
48. /// <summary>
49. /// 服务器是否正在运行
50. /// </summary>
51. **public** **bool** IsRunning { **get**; **private** **set**; }
52. /// <summary>
53. /// 监听的IP地址
54. /// </summary>
55. **public** IPAddress Address { **get**; **private** **set**; }
56. /// <summary>
57. /// 监听的端口
58. /// </summary>
59. **public** **int** Port { **get**; **private** **set**; }
60. /// <summary>
61. /// 通信使用的编码
62. /// </summary>
63. **public** Encoding Encoding { **get**; **set**; }
65. #endregion
67. #region 构造函数
69. /// <summary>
70. /// 同步UdpClient UDP服务器
71. /// </summary>
72. /// <param name="listenPort">监听的端口</param>
73. **public** UDPServer(**int** listenPort)
74. : **this**(IPAddress.Any, listenPort,1024)
75. {
76. }
78. /// <summary>
79. /// 同步UdpClient UDP服务器
80. /// </summary>
81. /// <param name="localEP">监听的终结点</param>
82. **public** UDPServer(IPEndPoint localEP)
83. : **this**(localEP.Address, localEP.Port,1024)
84. {
85. }
87. /// <summary>
88. /// 同步UdpClient UDP服务器
89. /// </summary>
90. /// <param name="localIPAddress">监听的IP地址</param>
91. /// <param name="listenPort">监听的端口</param>
92. /// <param name="maxClient">最大客户端数量</param>
93. **public** UDPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
94. {
95. **this**.Address = localIPAddress;
96. **this**.Port = listenPort;
97. **this**.Encoding = Encoding.Default;
99. \_maxClient = maxClient;
100. //\_clients = new List<AsyncUDPSocketState>();
101. \_server = **new** UdpClient(**new** IPEndPoint(**this**.Address, **this**.Port));
103. \_recvBuffer=**new** **byte**[\_server.Client.ReceiveBufferSize];
104. }
106. #endregion
108. #region Method
109. /// <summary>
110. /// 启动服务器
111. /// </summary>
112. /// <returns>异步TCP服务器</returns>
113. **public** **void** Start()
114. {
115. **if** (!IsRunning)
116. {
117. IsRunning = **true**;
118. \_server.EnableBroadcast = **true**;
119. **new** Thread(ReceiveData).Start();
120. }
121. }
123. /// <summary>
124. /// 停止服务器
125. /// </summary>
126. **public** **void** Stop()
127. {
128. **if** (IsRunning)
129. {
130. IsRunning = **false**;
131. \_server.Close();
132. //TODO 关闭对所有客户端的连接
134. }
135. }
137. /// <summary>
138. /// 同步接收数据的方法
139. /// </summary>
140. /// <param name="ar"></param>
141. **private** **void** ReceiveData()
142. {
143. IPEndPoint remote = **null**;
144. **while** (**true**)
145. {
146. **byte**[] buffer = **null**;
147. **try**
148. {
149. buffer = \_server.Receive(**ref** remote);
150. }
151. **catch** (Exception)
152. {
153. //异常处理操作
154. **return**;
155. }
156. **if** (buffer == **null** || buffer.Length == 0) **return**;
157. //TODO 数据包收到触发事件
158. RaiseDataReceived(**null**);
159. }
160. }
162. /// <summary>
163. /// 同步发送数据
164. /// </summary>
165. /// <param name="msg"></param>
166. /// <param name="remote"></param>
167. **public** **void** Send(**string** msg, IPEndPoint remote)
168. {
169. **byte**[] data = Encoding.Default.GetBytes(msg);
170. **try**
171. {
172. \_server.Send(data, data.Length, remote);
173. }
174. **catch** (Exception)
175. {
176. //TODO 异常处理
177. RaiseOtherException(**null**);
178. }
179. }
181. #endregion
183. #region 事件
184. /// <summary>
185. /// 接收到数据事件
186. /// </summary>
187. **public** **event** EventHandler<UDPEventArgs> DataReceived;
189. **private** **void** RaiseDataReceived(UDPState state)
190. {
191. **if** (DataReceived != **null**)
192. {
193. DataReceived(**this**, **new** UDPEventArgs(state));
194. }
195. }
197. /// <summary>
198. /// 数据发送完毕事件
199. /// </summary>
200. **public** **event** EventHandler<UDPEventArgs> CompletedSend;
202. /// <summary>
203. /// 触发数据发送完毕的事件
204. /// </summary>
205. /// <param name="state"></param>
206. **private** **void** RaiseCompletedSend(UDPState state)
207. {
208. **if** (CompletedSend != **null**)
209. {
210. CompletedSend(**this**, **new** UDPEventArgs(state));
211. }
212. }
214. /// <summary>
215. /// 网络错误事件
216. /// </summary>
217. **public** **event** EventHandler<UDPEventArgs> NetError;
218. /// <summary>
219. /// 触发网络错误事件
220. /// </summary>
221. /// <param name="state"></param>
222. **private** **void** RaiseNetError(UDPState state)
223. {
224. **if** (NetError != **null**)
225. {
226. NetError(**this**, **new** UDPEventArgs(state));
227. }
228. }
230. /// <summary>
231. /// 异常事件
232. /// </summary>
233. **public** **event** EventHandler<UDPEventArgs> OtherException;
234. /// <summary>
235. /// 触发异常事件
236. /// </summary>
237. /// <param name="state"></param>
238. **private** **void** RaiseOtherException(UDPState state, **string** descrip)
239. {
240. **if** (OtherException != **null**)
241. {
242. OtherException(**this**, **new** UDPEventArgs(descrip, state));
243. }
244. }
245. **private** **void** RaiseOtherException(UDPState state)
246. {
247. RaiseOtherException(state, "");
248. }
249. #endregion
251. #region Close
252. /// <summary>
253. /// 关闭一个与客户端之间的会话
254. /// </summary>
255. /// <param name="state">需要关闭的客户端会话对象</param>
256. **public** **void** Close(UDPState state)
257. {
258. **if** (state != **null**)
259. {
260. //\_clients.Remove(state);
261. //\_clientCount--;
262. //TODO 触发关闭事件
263. }
264. }
265. /// <summary>
266. /// 关闭所有的客户端会话,与所有的客户端连接会断开
267. /// </summary>
268. **public** **void** CloseAllClient()
269. {
270. //foreach (AsyncUDPSocketState client in \_clients)
271. //{
272. //    Close(client);
273. //}
274. //\_clientCount = 0;
275. //\_clients.Clear();
276. }
278. #endregion
280. #region 释放
281. /// <summary>
282. /// Performs application-defined tasks associated with freeing,
283. /// releasing, or resetting unmanaged resources.
284. /// </summary>
285. **public** **void** Dispose()
286. {
287. Dispose(**true**);
288. GC.SuppressFinalize(**this**);
289. }
291. /// <summary>
292. /// Releases unmanaged and - optionally - managed resources
293. /// </summary>
294. /// <param name="disposing"><c>true</c> to release
295. /// both managed and unmanaged resources; <c>false</c>
296. /// to release only unmanaged resources.</param>
297. **protected** **virtual** **void** Dispose(**bool** disposing)
298. {
299. **if** (!**this**.disposed)
300. {
301. **if** (disposing)
302. {
303. **try**
304. {
305. Stop();
306. **if** (\_server != **null**)
307. {
308. \_server = **null**;
309. }
310. }
311. **catch** (SocketException)
312. {
313. //TODO
314. RaiseOtherException(**null**);
315. }
316. }
317. disposed = **true**;
318. }
319. }
320. #endregion
321. }
322. }

用户状态封装类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303787) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
8. **namespace** NetFrame.Net.UDP.Listener.Synchronous
9. {
10. **public** **class** UDPState
11. {
12. // Client   socket.
13. **public** UdpClient udpClient = **null**;
14. // Size of receive buffer.
15. **public** **const** **int** BufferSize = 1024;
16. // Receive buffer.
17. **public** **byte**[] buffer = **new** **byte**[BufferSize];
18. // Received data string.
19. **public** StringBuilder sb = **new** StringBuilder();
21. **public** IPEndPoint remote = **new** IPEndPoint(IPAddress.Any, 0);
22. }
23. }

服务器事件参数类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/44303787) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/44303787)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
6. **namespace** NetFrame.Net.UDP.Listener.Synchronous
7. {
8. /// <summary>
9. /// UdpClient 同步UDP服务器事件类
10. /// </summary>
11. **public** **class** UDPEventArgs : EventArgs
12. {
13. /// <summary>
14. /// 提示信息
15. /// </summary>
16. **public** **string** \_msg;
18. /// <summary>
19. /// 客户端状态封装类
20. /// </summary>
21. **public** UDPState \_state;
23. /// <summary>
24. /// 是否已经处理过了
25. /// </summary>
26. **public** **bool** IsHandled { **get**; **set**; }
28. **public** UDPEventArgs(**string** msg)
29. {
30. **this**.\_msg = msg;
31. IsHandled = **false**;
32. }
33. **public** UDPEventArgs(UDPState state)
34. {
35. **this**.\_state = state;
36. IsHandled = **false**;
37. }
38. **public** UDPEventArgs(**string** msg, UDPState state)
39. {
40. **this**.\_msg = msg;
41. **this**.\_state = state;
42. IsHandled = **false**;
43. }
44. }
45. }

顶

0

踩

0

* 上一篇[C#网络编程系列文章(七)之UdpClient实现异步UDP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44303743)

# [基于C#的socket编程的TCP异步实现](http://blog.csdn.net/zhujunxxxxx/article/details/44239743)

2015-03-13 12:58 556人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44239743#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44239743#report)

 分类：

c#技术类（70）  UDP/TCP通讯（28）  网络编程（16）

using System;

using System.Net;

using System.Net.Sockets;

using System.Text;

using System.Threading;

// State object for reading client data asynchronously

public class StateObject

{

// Client socket.

public Socket workSocket = null;

// Size of receive buffer.

public const int BufferSize = 1024;

// Receive buffer.

public byte[] buffer = new byte[BufferSize];

// Received data string.

public StringBuilder sb = new StringBuilder();

}

public class AsynchronousSocketListener

{

// Thread signal.

public static ManualResetEvent allDone = new ManualResetEvent(false);

public AsynchronousSocketListener()

{

}

public static void StartListening()

{

// Data buffer for incoming data.

byte[] bytes = new Byte[1024];

// Establish the local endpoint for the socket.

// The DNS name of the computer

// running the listener is "host.contoso.com".

//IPHostEntry ipHostInfo = Dns.Resolve(Dns.GetHostName());

//IPAddress ipAddress = ipHostInfo.AddressList[0];

IPAddress ipAddress = IPAddress.Parse("127.0.0.1");

IPEndPoint localEndPoint = new IPEndPoint(ipAddress, 11000);

// Create a TCP/IP socket.

Socket listener = new Socket(AddressFamily.InterNetwork,SocketType.Stream, ProtocolType.Tcp);

// Bind the socket to the local

//endpoint and listen for incoming connections.

try

{

listener.Bind(localEndPoint);

listener.Listen(100);

while (true)

{

// Set the event to nonsignaled state.

allDone.Reset();

// Start an asynchronous socket to listen for connections.

Console.WriteLine("Waiting for a connection...");

listener.BeginAccept(new AsyncCallback(AcceptCallback),listener);

// Wait until a connection is made before continuing.

allDone.WaitOne();

}

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

Console.WriteLine("\nPress ENTER to continue...");

Console.Read();

}

public static void AcceptCallback(IAsyncResult ar)

{

// Signal the main thread to continue.

allDone.Set();

// Get the socket that handles the client request.

Socket listener = (Socket)ar.AsyncState;

Socket handler = listener.EndAccept(ar);

// Create the state object.

StateObject state = new StateObject();

state.workSocket = handler;

handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new AsyncCallback(ReadCallback), state);

}

public static void ReadCallback(IAsyncResult ar)

{

String content = String.Empty;

// Retrieve the state object and the handler socket

// from the asynchronous state object.

StateObject state = (StateObject)ar.AsyncState;

Socket handler = state.workSocket;

// Read data from the client socket.

int bytesRead = handler.EndReceive(ar);

if (bytesRead > 0)

{

// There might be more data, so store the data received so far.

state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, bytesRead));

// Check for end-of-file tag. If it is not there, read

// more data.

content = state.sb.ToString();

if (content.IndexOf("<EOF>") > -1)

{

// All the data has been read from the

// client. Display it on the console.

Console.WriteLine("Read {0} bytes from socket. \n Data : {1}", content.Length, content);

// Echo the data back to the client.

Send(handler, content);

}

else

{

// Not all data received. Get more.

handler.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new AsyncCallback(ReadCallback), state);

}

}

}

private static void Send(Socket handler, String data)

{

// Convert the string data to byte data using ASCII encoding.

byte[] byteData = Encoding.ASCII.GetBytes(data);

// Begin sending the data to the remote device.

handler.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback), handler);

}

private static void SendCallback(IAsyncResult ar)

{

try

{

// Retrieve the socket from the state object.

Socket handler = (Socket)ar.AsyncState;

// Complete sending the data to the remote device.

int bytesSent = handler.EndSend(ar);

Console.WriteLine("Sent {0} bytes to client.", bytesSent);

handler.Shutdown(SocketShutdown.Both);

handler.Close();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

public static int Main(String[] args)

{

StartListening();

return 0;

}

}

using System;

using System.Net;

using System.Net.Sockets;

using System.Threading;

using System.Text;

// State object for receiving data from remote device.

public class StateObject

{

// Client socket.

public Socket workSocket = null;

// Size of receive buffer.

public const int BufferSize = 256;

// Receive buffer.

public byte[] buffer = new byte[BufferSize];

// Received data string.

public StringBuilder sb = new StringBuilder();

}

public class AsynchronousClient

{

// The port number for the remote device.

private const int port = 11000;

// ManualResetEvent instances signal completion.

private static ManualResetEvent connectDone = new ManualResetEvent(false);

private static ManualResetEvent sendDone = new ManualResetEvent(false);

private static ManualResetEvent receiveDone = new ManualResetEvent(false);

// The response from the remote device.

private static String response = String.Empty;

private static void StartClient()

{

// Connect to a remote device.

try

{

// Establish the remote endpoint for the socket.

// The name of the

// remote device is "host.contoso.com".

//IPHostEntry ipHostInfo = Dns.Resolve("user");

//IPAddress ipAddress = ipHostInfo.AddressList[0];

IPAddress ipAddress = IPAddress.Parse("127.0.0.1");

IPEndPoint remoteEP = new IPEndPoint(ipAddress, port);

// Create a TCP/IP socket.

Socket client = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

// Connect to the remote endpoint.

client.BeginConnect(remoteEP, new AsyncCallback(ConnectCallback), client);

connectDone.WaitOne();

// Send test data to the remote device.

Send(client, "This is a test<EOF>");

sendDone.WaitOne();

// Receive the response from the remote device.

Receive(client);

receiveDone.WaitOne();

// Write the response to the console.

Console.WriteLine("Response received : {0}", response);

// Release the socket.

client.Shutdown(SocketShutdown.Both);

client.Close();

Console.ReadLine();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

private static void ConnectCallback(IAsyncResult ar)

{

try

{

// Retrieve the socket from the state object.

Socket client = (Socket)ar.AsyncState;

// Complete the connection.

client.EndConnect(ar);

Console.WriteLine("Socket connected to {0}", client.RemoteEndPoint.ToString());

// Signal that the connection has been made.

connectDone.Set();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

private static void Receive(Socket client)

{

try

{

// Create the state object.

StateObject state = new StateObject();

state.workSocket = client;

// Begin receiving the data from the remote device.

client.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new AsyncCallback(ReceiveCallback), state);

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

private static void ReceiveCallback(IAsyncResult ar)

{

try

{

// Retrieve the state object and the client socket

// from the asynchronous state object.

StateObject state = (StateObject)ar.AsyncState;

Socket client = state.workSocket;

// Read data from the remote device.

int bytesRead = client.EndReceive(ar);

if (bytesRead > 0)

{

// There might be more data, so store the data received so far.

state.sb.Append(Encoding.ASCII.GetString(state.buffer, 0, bytesRead));

// Get the rest of the data.

client.BeginReceive(state.buffer, 0, StateObject.BufferSize, 0, new AsyncCallback(ReceiveCallback), state);

}

else

{

// All the data has arrived; put it in response.

if (state.sb.Length > 1)

{

response = state.sb.ToString();

}

// Signal that all bytes have been received.

receiveDone.Set();

}

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

private static void Send(Socket client, String data)

{

// Convert the string data to byte data using ASCII encoding.

byte[] byteData = Encoding.ASCII.GetBytes(data);

// Begin sending the data to the remote device.

client.BeginSend(byteData, 0, byteData.Length, 0, new AsyncCallback(SendCallback), client);

}

private static void SendCallback(IAsyncResult ar)

{

try

{

// Retrieve the socket from the state object.

Socket client = (Socket)ar.AsyncState;

// Complete sending the data to the remote device.

int bytesSent = client.EndSend(ar);

Console.WriteLine("Sent {0} bytes to server.", bytesSent);

// Signal that all bytes have been sent.

sendDone.Set();

}

catch (Exception e)

{

Console.WriteLine(e.ToString());

}

}

public static int Main(String[] args)

{

StartClient();

return 0;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 213
* 214
* 215
* 216
* 217
* 218
* 219
* 220
* 221
* 222
* 223
* 224
* 225
* 226
* 227
* 228
* 229
* 230
* 231
* 232
* 233
* 234
* 235
* 236
* 237
* 238
* 239
* 240
* 241
* 242
* 243
* 244
* 245
* 246
* 247
* 248
* 249
* 250
* 251
* 252
* 253
* 254
* 255
* 256
* 257
* 258
* 259
* 260
* 261
* 262
* 263
* 264
* 265
* 266
* 267
* 268
* 269
* 270
* 271
* 272
* 273
* 274
* 275
* 276
* 277
* 278
* 279
* 280
* 281
* 282
* 283
* 284
* 285
* 286
* 287
* 288
* 289
* 290
* 291
* 292
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 213
* 214
* 215
* 216
* 217
* 218
* 219
* 220
* 221
* 222
* 223
* 224
* 225
* 226
* 227
* 228
* 229
* 230
* 231
* 232
* 233
* 234
* 235
* 236
* 237
* 238
* 239
* 240
* 241
* 242
* 243
* 244
* 245
* 246
* 247
* 248
* 249
* 250
* 251
* 252
* 253
* 254
* 255
* 256
* 257
* 258
* 259
* 260
* 261
* 262
* 263
* 264
* 265
* 266
* 267
* 268
* 269
* 270
* 271
* 272
* 273
* 274
* 275
* 276
* 277
* 278
* 279
* 280
* 281
* 282
* 283
* 284
* 285
* 286
* 287
* 288
* 289
* 290
* 291
* 292

顶

0

踩

0

* 上一篇[一个UDP异步通信例子](http://blog.csdn.net/zhujunxxxxx/article/details/44238239)
* 下一篇[C#网络编程系列文章(一)之Socket实现异步TCP服务器](http://blog.csdn.net/zhujunxxxxx/article/details/44258719)

# [一个UDP异步通信例子](http://blog.csdn.net/zhujunxxxxx/article/details/44238239)

标签： [异步](http://www.csdn.net/tag/%e5%bc%82%e6%ad%a5)[udp](http://www.csdn.net/tag/udp)

2015-03-13 10:50 609人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/44238239#comments)(0) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/44238239#report)

 分类：

UDP/TCP通讯（28）  网络编程（16）  c#技术类（70）

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace ConsoleUdpServer

{

class Program

{

static void Main(string[] args)

{

Socket socket = new Socket(AddressFamily.InterNetwork,

SocketType.Dgram,

ProtocolType.Udp);

EndPoint localEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8002);

socket.Bind(localEP);

//启动异步接收

State state = new State(socket);

socket.BeginReceiveFrom(

state.Buffer, 0, state.Buffer.Length,

SocketFlags.None,

ref state.RemoteEP,

EndReceiveFromCallback,

state);

Console.ReadLine();

socket.Close();

}

//异步接收回调函数

static void EndReceiveFromCallback(IAsyncResult iar)

{

State state = iar.AsyncState as State;

Socket socket = state.Socket;

try

{

//完成接收

int byteRead = socket.EndReceiveFrom(iar, ref state.RemoteEP);

//显示客户端地址和端口

Console.WriteLine("客户端终结点：{0}", state.RemoteEP.ToString());

//显示接收信息

Console.WriteLine("接收数据字节数：{0}", byteRead);

string message = Encoding.Default.GetString(state.Buffer, 0, byteRead);

Console.WriteLine("来着客户端信息：{0}", message);

//向客户端发送信息

SendMeaage(socket, state.RemoteEP, "成功接收！");

}

catch (Exception e)

{

Console.WriteLine("发生异常！异常信息：");

Console.WriteLine(e.Message);

}

finally

{

//非常重要：继续异步接收

socket.BeginReceiveFrom(

state.Buffer, 0, state.Buffer.Length,

SocketFlags.None,

ref state.RemoteEP,

EndReceiveFromCallback,

state);

}

}

/// <summary>

/// 向客户端发送信息

/// </summary>

/// <param name="socket">本地Socket（服务器Socket）</param>

/// <param name="remoteEndPoint">客户端终结点</param>

/// <param name="Message">信息</param>

static void SendMeaage(Socket socket, EndPoint remoteEndPoint, string Message)

{

byte[] bytes = Encoding.Default.GetBytes(Message);

socket.SendTo(bytes, remoteEndPoint);

}

}

/// <summary>

/// 用于异步接收处理的辅助类

/// </summary>

class State

{

public State(Socket socket)

{

this.Buffer = new byte[1024];

this.Socket = socket;

this.RemoteEP = new IPEndPoint(IPAddress.Any, 0);

}

/// <summary>

/// 获取本机（服务器）Socket

/// </summary>

public Socket Socket { get; private set; }

/// <summary>

/// 获取接收缓冲区

/// </summary>

public byte[] Buffer { get; private set; }

/// <summary>

/// 获取/设置客户端终结点

/// </summary>

public EndPoint RemoteEP;

}

}

UDP客户端

using System;

using System.Text;

using System.Net;

using System.Net.Sockets;

namespace ConsoleUdpClient

{

class Program

{

static void Main(string[] args)

{

Socket socket = new Socket(AddressFamily.InterNetwork,

SocketType.Dgram,

ProtocolType.Udp);

//客户端使用的终结点

EndPoint localEP = new IPEndPoint(IPAddress.Any, 0);

socket.Bind(localEP);

//启动异步接收

State state = new State(socket);

socket.BeginReceiveFrom(

state.Buffer, 0, state.Buffer.Length,

SocketFlags.None,

ref state.RemoteEP,

EndReceiveFromCallback,

state);

//向服务器发送信息

EndPoint remoteEP = new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8002);

SendMeaage(socket, remoteEP, "你好，这里是客户端，收到请回复！Over~~~~");

Console.ReadLine();

socket.Close();

}

//异步接收回调函数

static void EndReceiveFromCallback(IAsyncResult iar)

{

State state = iar.AsyncState as State;

Socket socket = state.Socket;

try

{

//完成接收

int byteRead = socket.EndReceiveFrom(iar, ref state.RemoteEP);

//显示服务器地址和端口

Console.WriteLine("服务器终结点：{0}", state.RemoteEP.ToString());

//显示接收信息

Console.WriteLine("接收数据字节数：{0}", byteRead);

string message = Encoding.Default.GetString(state.Buffer, 0, byteRead);

Console.WriteLine("来着服务器的信息：{0}", message);

}

catch (Exception e)

{

Console.WriteLine("发生异常！异常信息：");

Console.WriteLine(e.Message);

}

finally

{

//非常重要：继续异步接收

socket.BeginReceiveFrom(

state.Buffer, 0, state.Buffer.Length,

SocketFlags.None,

ref state.RemoteEP,

EndReceiveFromCallback,

state);

}

}

/// <summary>

/// 向服务器发送信息

/// </summary>

/// <param name="socket">本地Socket</param>

/// <param name="remoteEndPoint">服务器终结点</param>

/// <param name="Message">信息</param>

static void SendMeaage(Socket socket, EndPoint remoteEndPoint, string Message)

{

byte[] bytes = Encoding.Default.GetBytes(Message);

socket.SendTo(bytes, remoteEndPoint);

}

}

/// <summary>

/// 用于异步接收处理的辅助类

/// </summary>

class State

{

public State(Socket socket)

{

this.Buffer = new byte[1024];

this.Socket = socket;

this.RemoteEP = new IPEndPoint(IPAddress.Any, 0);

}

/// <summary>

/// 获取本机Socket

/// </summary>

public Socket Socket { get; private set; }

/// <summary>

/// 获取接收缓冲区

/// </summary>

public byte[] Buffer { get; private set; }

/// <summary>

/// 获取/设置客户端终结点

/// </summary>

public EndPoint RemoteEP;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212
* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134
* 135
* 136
* 137
* 138
* 139
* 140
* 141
* 142
* 143
* 144
* 145
* 146
* 147
* 148
* 149
* 150
* 151
* 152
* 153
* 154
* 155
* 156
* 157
* 158
* 159
* 160
* 161
* 162
* 163
* 164
* 165
* 166
* 167
* 168
* 169
* 170
* 171
* 172
* 173
* 174
* 175
* 176
* 177
* 178
* 179
* 180
* 181
* 182
* 183
* 184
* 185
* 186
* 187
* 188
* 189
* 190
* 191
* 192
* 193
* 194
* 195
* 196
* 197
* 198
* 199
* 200
* 201
* 202
* 203
* 204
* 205
* 206
* 207
* 208
* 209
* 210
* 211
* 212

顶

0

踩

0

* 上一篇[c#异步Socket Tcp服务器实现](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)
* 下一篇[基于C#的socket编程的TCP异步实现](http://blog.csdn.net/zhujunxxxxx/article/details/44239743)

# [c#异步Socket Tcp服务器实现](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

标签： [异步](http://www.csdn.net/tag/%e5%bc%82%e6%ad%a5)[TCP](http://www.csdn.net/tag/TCP)

2015-02-06 21:31 2235人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/43574655#comments)(2) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/43574655#report)

本文章已收录于：

[**计算机网络知识库**](http://lib.csdn.net/base/37?source=blogtop)

 分类：

UDP/TCP通讯（28）  计算机网络（2）  c#技术类（70）

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/zhujunxxxxx/article/details/43574655)

## 原创性申明

本文作者: 小竹zz  本文地址:http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/zhujunxxxxx 转载请注明出处。

## 介绍

我之前写过一篇IOCP的文章: <http://blog.csdn.net/zhujunxxxxx/article/details/43573879> 这个比异步socket性能好，因为它复用对象了。

在c#中微软已经提供了TcpListener和TcpClient来实现Tcp的通讯，这部分已经有人写了比较好的异步服务器代码 <http://www.cnblogs.com/gaochundong/archive/2013/04/14/csharp_async_tcp_server.html> 这位博主写的博客质量真是高，我经常浏览他的博客总是有很多让我惊喜的地方，他使用的是TcpListener来实现的异步服务器的。

我的socket版本其实本质上和他的没有区别，就只是改写了一点点，所以在这里贴一份代码就是了，多的不解释了

## 代码

服务器核心代码 AsyncServer.cs

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43574655) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

1. /// <summary>
2. /// 异步SOCKET 服务器
3. /// </summary>
4. **public** **class** AsyncServer : IDisposable
5. {
7. #region Fields
8. /// <summary>
9. /// 服务器程序允许的最大客户端连接数
10. /// </summary>
11. **private** **int** \_maxClient;
13. /// <summary>
14. /// 当前的连接的客户端数
15. /// </summary>
16. **private** **int** \_clientCount;
18. /// <summary>
19. /// 服务器使用的异步socket
20. /// </summary>
21. **private** Socket \_serverSock;
23. /// <summary>
24. /// 客户端会话列表
25. /// </summary>
26. **private** List<Session> \_clients;
28. **private** **bool** disposed = **false**;
30. #endregion

33. #region Properties
35. /// <summary>
36. /// 服务器是否正在运行
37. /// </summary>
38. **public** **bool** IsRunning { **get**; **private** **set**; }
39. /// <summary>
40. /// 监听的IP地址
41. /// </summary>
42. **public** IPAddress Address { **get**; **private** **set**; }
43. /// <summary>
44. /// 监听的端口
45. /// </summary>
46. **public** **int** Port { **get**; **private** **set**; }
47. /// <summary>
48. /// 通信使用的编码
49. /// </summary>
50. **public** Encoding Encoding { **get**; **set**; }

53. #endregion
55. #region Ctors
57. /// <summary>
58. /// 异步Socket TCP服务器
59. /// </summary>
60. /// <param name="listenPort">监听的端口</param>
61. **public** AsyncServer(**int** listenPort)
62. : **this**(IPAddress.Any, listenPort,1024)
63. {
64. }
66. /// <summary>
67. /// 异步Socket TCP服务器
68. /// </summary>
69. /// <param name="localEP">监听的终结点</param>
70. **public** AsyncServer(IPEndPoint localEP)
71. : **this**(localEP.Address, localEP.Port,1024)
72. {
73. }
75. /// <summary>
76. /// 异步Socket TCP服务器
77. /// </summary>
78. /// <param name="localIPAddress">监听的IP地址</param>
79. /// <param name="listenPort">监听的端口</param>
80. /// <param name="maxClient">最大客户端数量</param>
81. **public** AsyncServer(IPAddress localIPAddress, **int** listenPort,**int** maxClient)
82. {
83. **this**.Address = localIPAddress;
84. **this**.Port = listenPort;
85. **this**.Encoding = Encoding.Default;
87. \_maxClient = maxClient;
88. \_clients = **new** List<Session>();
89. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
90. }
92. #endregion

95. #region Server
97. /// <summary>
98. /// 启动服务器
99. /// </summary>
100. /// <returns>异步TCP服务器</returns>
101. **public** AsyncServer Start()
102. {
103. **if** (!IsRunning)
104. {
105. IsRunning = **true**;
106. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
107. \_serverSock.Listen(1024);
108. \_serverSock.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), \_serverSock);
109. }
110. **return** **this**;
111. }
113. /// <summary>
114. /// 启动服务器
115. /// </summary>
116. /// <param name="backlog">
117. /// 服务器所允许的挂起连接序列的最大长度
118. /// </param>
119. /// <returns>异步TCP服务器</returns>
120. **public** AsyncServer Start(**int** backlog)
121. {
122. **if** (!IsRunning)
123. {
124. IsRunning = **true**;
125. \_serverSock.Bind(**new** IPEndPoint(**this**.Address, **this**.Port));
126. \_serverSock.Listen(backlog);
127. \_serverSock.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), \_serverSock);
128. }
129. **return** **this**;
130. }
132. /// <summary>
133. /// 停止服务器
134. /// </summary>
135. /// <returns>异步TCP服务器</returns>
136. **public** AsyncServer Stop()
137. {
138. **if** (IsRunning)
139. {
140. IsRunning = **false**;
141. \_serverSock.Close();
142. //TODO 关闭对所有客户端的连接
144. }
145. **return** **this**;
146. }
148. #endregion
150. #region Receive
151. /// <summary>
152. /// 处理客户端连接
153. /// </summary>
154. /// <param name="ar"></param>
155. **private** **void** HandleAcceptConnected(IAsyncResult ar)
156. {
157. **if** (IsRunning)
158. {
159. Socket server = (Socket)ar.AsyncState;
160. Socket client = server.EndAccept(ar);
162. //检查是否达到最大的允许的客户端数目
163. **if** (\_clientCount == \_maxClient)
164. {
165. //TODO 触发事件
166. RaiseServerException(**null**);
167. }
168. **else**
169. {
170. Session session = **new** Session(client);
171. **lock** (\_clients)
172. {
173. \_clients.Add(session);
174. \_clientCount++;
175. RaiseClientConnected(session); //触发客户端连接事件
176. }
177. session.RecvDataBuffer = **new** **byte**[client.ReceiveBufferSize];
178. //开始接受来自该客户端的数据
179. client.BeginReceive(session.RecvDataBuffer, 0, session.RecvDataBuffer.Length, SocketFlags.None,
180. **new** AsyncCallback(HandleDataReceived), session);
181. }
182. //接受下一个请求
183. server.BeginAccept(**new** AsyncCallback(HandleAcceptConnected), ar.AsyncState);
184. }
185. }
186. /// <summary>
187. /// 处理客户端数据
188. /// </summary>
189. /// <param name="ar"></param>
190. **private** **void** HandleDataReceived(IAsyncResult ar)
191. {
192. **if** (IsRunning)
193. {
194. Session session = (Session)ar.AsyncState;
195. Socket client = session.ClientSocket;
196. **try**
197. {
198. //如果两次开始了异步的接收,所以当客户端退出的时候
199. //会两次执行EndReceive
200. **int** recv = client.EndReceive(ar);
201. **if** (recv == 0)
202. {
203. //TODO 触发事件 (关闭客户端)
204. CloseSession(session);
205. RaiseNetError(session);
206. **return**;
207. }
208. //TODO 处理已经读取的数据 ps:数据在session的RecvDataBuffer中
209. RaiseDataReceived(session);
210. //TODO 触发数据接收事件
211. }
212. **catch** (SocketException ex)
213. {
214. //TODO 异常处理
215. RaiseNetError(session);
216. }
217. **finally**
218. {
219. //继续接收来自来客户端的数据
220. client.BeginReceive(session.RecvDataBuffer, 0, session.RecvDataBuffer.Length, SocketFlags.None,
221. **new** AsyncCallback(HandleDataReceived), session);
222. }
223. }
224. }
225. #endregion
227. #region Send
228. /// <summary>
229. /// 发送数据
230. /// </summary>
231. /// <param name="session">接收数据的客户端会话</param>
232. /// <param name="data">数据报文</param>
233. **public** **void** Send(Session session, **byte**[] data)
234. {
235. Send(session.ClientSocket,data);
236. }
238. /// <summary>
239. /// 异步发送数据至指定的客户端
240. /// </summary>
241. /// <param name="client">客户端</param>
242. /// <param name="data">报文</param>
243. **public** **void** Send(Socket client, **byte**[] data)
244. {
245. **if** (!IsRunning)
246. **throw** **new** InvalidProgramException("This TCP Scoket server has not been started.");
248. **if** (client == **null**)
249. **throw** **new** ArgumentNullException("client");
251. **if** (data == **null**)
252. **throw** **new** ArgumentNullException("data");
253. client.BeginSend(data, 0, data.Length, SocketFlags.None,
254. **new** AsyncCallback(SendDataEnd), client);
255. }
257. /// <summary>
258. /// 发送数据完成处理函数
259. /// </summary>
260. /// <param name="ar">目标客户端Socket</param>
261. **private** **void** SendDataEnd(IAsyncResult ar)
262. {
263. ((Socket)ar.AsyncState).EndSend(ar);
264. }
265. #endregion
267. #region Events
268. /// <summary>
269. /// 接收到数据事件
270. /// </summary>
271. **public** **event** EventHandler<EventArgs> DataReceived;
273. **private** **void** RaiseDataReceived(Session session)
274. {
275. **if** (DataReceived != **null**)
276. {
277. DataReceived(**this**, **new** AsyncEventArgs(session));
278. }
279. }
281. /// <summary>
282. /// 与客户端的连接已建立事件
283. /// </summary>
284. **public** **event** EventHandler<AsyncEventArgs> ClientConnected;
285. /// <summary>
286. /// 与客户端的连接已断开事件
287. /// </summary>
288. **public** **event** EventHandler<AsyncEventArgs> ClientDisconnected;
290. /// <summary>
291. /// 触发客户端连接事件
292. /// </summary>
293. /// <param name="session"></param>
294. **private** **void** RaiseClientConnected(Session session)
295. {
296. **if** (ClientConnected != **null**)
297. {
298. ClientConnected(**this**, **new** AsyncEventArgs(session));
299. }
300. }
301. /// <summary>
302. /// 触发客户端连接断开事件
303. /// </summary>
304. /// <param name="client"></param>
305. **private** **void** RaiseClientDisconnected(Socket client)
306. {
307. **if** (ClientDisconnected != **null**)
308. {
309. ClientDisconnected(**this**, **new** AsyncEventArgs("连接断开"));
310. }
311. }
312. /// <summary>
313. /// 网络错误事件
314. /// </summary>
315. **public** **event** EventHandler<AsyncEventArgs> NetError;
316. /// <summary>
317. /// 触发网络错误事件
318. /// </summary>
319. /// <param name="client"></param>
320. **private** **void** RaiseNetError(Session session)
321. {
322. **if** (NetError != **null**)
323. {
324. NetError(**this**, **new** AsyncEventArgs(session));
325. }
326. }
328. /// <summary>
329. /// 异常事件
330. /// </summary>
331. **public** **event** EventHandler<AsyncEventArgs> ServerException;
332. /// <summary>
333. /// 触发异常事件
334. /// </summary>
335. /// <param name="client"></param>
336. **private** **void** RaiseServerException(Session session)
337. {
338. **if** (ServerException != **null**)
339. {
340. ServerException(**this**, **new** AsyncEventArgs(session));
341. }
342. }
343. #endregion

346. #region Close
347. /// <summary>
348. /// 关闭一个与客户端之间的会话
349. /// </summary>
350. /// <param name="closeClient">需要关闭的客户端会话对象</param>
351. **public** **void** CloseSession(Session session)
352. {
353. **if** (session != **null**)
354. {
355. session.Datagram = **null**;
356. session.RecvDataBuffer = **null**;
358. \_clients.Remove(session);
359. \_clientCount--;
360. //TODO 触发关闭事件
361. session.Close();
362. }
363. }
364. /// <summary>
365. /// 关闭所有的客户端会话,与所有的客户端连接会断开
366. /// </summary>
367. **public** **void** CloseAllClient()
368. {
369. **foreach** (Session client **in** \_clients)
370. {
371. CloseSession(client);
372. }
373. \_clientCount = 0;
374. \_clients.Clear();
375. }
377. /// <summary>
378. /// Performs application-defined tasks associated with freeing,
379. /// releasing, or resetting unmanaged resources.
380. /// </summary>
381. **public** **void** Dispose()
382. {
383. Dispose(**true**);
384. GC.SuppressFinalize(**this**);
385. }
387. /// <summary>
388. /// Releases unmanaged and - optionally - managed resources
389. /// </summary>
390. /// <param name="disposing"><c>true</c> to release
391. /// both managed and unmanaged resources; <c>false</c>
392. /// to release only unmanaged resources.</param>
393. **protected** **virtual** **void** Dispose(**bool** disposing)
394. {
395. **if** (!**this**.disposed)
396. {
397. **if** (disposing)
398. {
399. **try**
400. {
401. Stop();
402. **if** (\_serverSock != **null**)
403. {
404. \_serverSock = **null**;
405. }
406. }
407. **catch** (SocketException ex)
408. {
409. //TODO
410. RaiseServerException(**null**);
411. }
412. }
413. disposed = **true**;
414. }
415. }
416. #endregion
417. }

其中使用了一个Session类，来封装对客户端的连接

Session,cs

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43574655) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

1. /// <summary>
2. /// 客户端与服务器之间的会话类
3. /// </summary>
4. **public** **class** Session
5. {
6. #region 字段
7. /// <summary>
8. /// 接收数据缓冲区
9. /// </summary>
10. **private** **byte**[] \_recvBuffer;
12. /// <summary>
13. /// 客户端发送到服务器的报文
14. /// 注意:在有些情况下报文可能只是报文的片断而不完整
15. /// </summary>
16. **private** **string** \_datagram;
18. /// <summary>
19. /// 客户端的Socket
20. /// </summary>
21. **private** Socket \_clientSock;
23. #endregion
25. #region 属性
27. /// <summary>
28. /// 接收数据缓冲区
29. /// </summary>
30. **public** **byte**[] RecvDataBuffer
31. {
32. **get**
33. {
34. **return** \_recvBuffer;
35. }
36. **set**
37. {
38. \_recvBuffer = value;
39. }
40. }
42. /// <summary>
43. /// 存取会话的报文
44. /// </summary>
45. **public** **string** Datagram
46. {
47. **get**
48. {
49. **return** \_datagram;
50. }
51. **set**
52. {
53. \_datagram = value;
54. }
55. }
57. /// <summary>
58. /// 获得与客户端会话关联的Socket对象
59. /// </summary>
60. **public** Socket ClientSocket
61. {
62. **get**
63. {
64. **return** \_clientSock;
66. }
67. }

70. #endregion
72. /// <summary>
73. /// 构造函数
74. /// </summary>
75. /// <param name="cliSock">会话使用的Socket连接</param>
76. **public** Session(Socket cliSock)
77. {
79. \_clientSock = cliSock;
80. }
81. /// <summary>
82. /// 关闭会话
83. /// </summary>
84. **public** **void** Close()
85. {
87. //关闭数据的接受和发送
88. \_clientSock.Shutdown(SocketShutdown.Both);
90. //清理资源
91. \_clientSock.Close();
92. }
93. }

事件类

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43574655) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

1. **class** AsyncEventArgs : EventArgs
2. {
3. /// <summary>
4. /// 提示信息
5. /// </summary>
6. **public** **string** \_msg;
8. **public** Session \_sessions;
10. /// <summary>
11. /// 是否已经处理过了
12. /// </summary>
13. **public** **bool** IsHandled { **get**; **set**; }
15. **public** AsyncEventArgs(**string** msg)
16. {
17. **this**.\_msg = msg;
18. IsHandled = **false**;
19. }
20. **public** AsyncEventArgs(Session session)
21. {
22. **this**.\_sessions = session;
23. IsHandled = **false**;
24. }
25. **public** AsyncEventArgs(**string** msg, Session session)
26. {
27. **this**.\_msg = msg;
28. **this**.\_sessions = session;
29. IsHandled = **false**;
30. }
31. }

 Socket版本的异步客户端代码

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43574655) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

1. /// <summary>
2. /// 异步客户端
3. /// </summary>
4. **public** **class** AsyncClient
5. {
6. #region Fields
8. **private** Socket \_client;
9. **private** **bool** disposed = **false**;
10. **private** **int** retries = 0;
12. #endregion
14. #region Properties
16. /// <summary>
17. /// 是否已与服务器建立连接
18. /// </summary>
19. **public** **bool** Connected { **get** { **return** \_client.Connected; } }
20. /// <summary>
21. /// 远端服务器的IP地址列表
22. /// </summary>
23. **public** IPAddress[] Addresses { **get**; **private** **set**; }
24. /// <summary>
25. /// 远端服务器的端口
26. /// </summary>
27. **public** **int** Port { **get**; **private** **set**; }
28. /// <summary>
29. /// 连接重试次数
30. /// </summary>
31. **public** **int** Retries { **get**; **set**; }
32. /// <summary>
33. /// 连接重试间隔
34. /// </summary>
35. **public** **int** RetryInterval { **get**; **set**; }
36. /// <summary>
37. /// 远端服务器终结点
38. /// </summary>
39. **public** IPEndPoint RemoteIPEndPoint
40. {
41. **get** { **return** **new** IPEndPoint(Addresses[0], Port); }
42. }
43. /// <summary>
44. /// 本地客户端终结点
45. /// </summary>
46. **protected** IPEndPoint LocalIPEndPoint { **get**; **private** **set**; }
47. /// <summary>
48. /// 通信所使用的编码
49. /// </summary>
50. **public** Encoding Encoding { **get**; **set**; }
52. #endregion
54. #region 构造函数
56. /// <summary>
57. /// 异步TCP客户端
58. /// </summary>
59. /// <param name="remoteEP">远端服务器终结点</param>
60. **public** AsyncClient(IPEndPoint remoteEP)
61. : **this**(**new**[] { remoteEP.Address }, remoteEP.Port)
62. {
63. }
64. /// <summary>
65. /// 异步TCP客户端
66. /// </summary>
67. /// <param name="remoteEP">远端服务器终结点</param>
68. /// <param name="localEP">本地客户端终结点</param>
69. **public** AsyncClient(IPEndPoint remoteEP, IPEndPoint localEP)
70. : **this**(**new**[] { remoteEP.Address }, remoteEP.Port, localEP)
71. {
72. }
74. /// <summary>
75. /// 异步TCP客户端
76. /// </summary>
77. /// <param name="remoteIPAddress">远端服务器IP地址</param>
78. /// <param name="remotePort">远端服务器端口</param>
79. **public** AsyncClient(IPAddress remoteIPAddress, **int** remotePort)
80. : **this**(**new**[] { remoteIPAddress }, remotePort)
81. {
82. }
84. /// <summary>
85. /// 异步TCP客户端
86. /// </summary>
87. /// <param name="remoteIPAddress">远端服务器IP地址</param>
88. /// <param name="remotePort">远端服务器端口</param>
89. /// <param name="localEP">本地客户端终结点</param>
90. **public** AsyncClient(
91. IPAddress remoteIPAddress, **int** remotePort, IPEndPoint localEP)
92. : **this**(**new**[] { remoteIPAddress }, remotePort, localEP)
93. {
94. }
96. /// <summary>
97. /// 异步TCP客户端
98. /// </summary>
99. /// <param name="remoteHostName">远端服务器主机名</param>
100. /// <param name="remotePort">远端服务器端口</param>
101. **public** AsyncClient(**string** remoteHostName, **int** remotePort)
102. : **this**(Dns.GetHostAddresses(remoteHostName), remotePort)
103. {
104. }
106. /// <summary>
107. /// 异步TCP客户端
108. /// </summary>
109. /// <param name="remoteHostName">远端服务器主机名</param>
110. /// <param name="remotePort">远端服务器端口</param>
111. /// <param name="localEP">本地客户端终结点</param>
112. **public** AsyncClient(
113. **string** remoteHostName, **int** remotePort, IPEndPoint localEP)
114. : **this**(Dns.GetHostAddresses(remoteHostName), remotePort, localEP)
115. {
116. }
118. /// <summary>
119. /// 异步TCP客户端
120. /// </summary>
121. /// <param name="remoteIPAddresses">远端服务器IP地址列表</param>
122. /// <param name="remotePort">远端服务器端口</param>
123. **public** AsyncClient(IPAddress[] remoteIPAddresses, **int** remotePort)
124. : **this**(remoteIPAddresses, remotePort, **null**)
125. {
126. }
128. /// <summary>
129. /// 异步TCP客户端
130. /// </summary>
131. /// <param name="remoteIPAddresses">远端服务器IP地址列表</param>
132. /// <param name="remotePort">远端服务器端口</param>
133. /// <param name="localEP">本地客户端终结点</param>
134. **public** AsyncClient(
135. IPAddress[] remoteIPAddresses, **int** remotePort, IPEndPoint localEP)
136. {
137. **this**.Addresses = remoteIPAddresses;
138. **this**.Port = remotePort;
139. **this**.LocalIPEndPoint = localEP;
140. **this**.Encoding = Encoding.Default;
142. **if** (**this**.LocalIPEndPoint != **null**)
143. {
144. \_client = **new** Socket(LocalIPEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
145. }
147. Retries = 3;
148. RetryInterval = 5;
149. }
151. #endregion
153. #region Connect
154. /// <summary>
155. /// 连接到服务器
156. /// </summary>
157. /// <returns>异步TCP客户端</returns>
158. **public** AsyncClient Connect()
159. {
160. **if** (!Connected)
161. {
162. // start the async connect operation
163. \_client.BeginConnect(
164. Addresses, Port, HandleTcpServerConnected, \_client);
165. }
167. **return** **this**;
168. }
169. /// <summary>
170. /// 关闭与服务器的连接
171. /// </summary>
172. /// <returns>异步TCP客户端</returns>
173. **public** AsyncClient Close()
174. {
175. **if** (Connected)
176. {
177. retries = 0;
178. \_client.Close();
179. RaiseServerDisconnected(Addresses, Port);
180. }
182. **return** **this**;
183. }
185. #endregion
187. #region Receive
189. **private** **void** HandleTcpServerConnected(IAsyncResult ar)
190. {
191. **try**
192. {
193. \_client.EndConnect(ar);
194. RaiseServerConnected(Addresses, Port);
195. retries = 0;
196. }
197. **catch** (Exception ex)
198. {
200. }
201. // we are connected successfully and start asyn read operation.
202. **byte**[] buffer = **new** **byte**[\_client.ReceiveBufferSize];
203. \_client.BeginReceive(buffer, 0, buffer.Length, SocketFlags.None, HandleDatagramReceived, buffer);
204. }
206. **private** **void** HandleDatagramReceived(IAsyncResult ar)
207. {
208. **try**
209. {
210. **int** recv = \_client.EndReceive(ar);
212. **byte**[] buffer = (**byte**[])ar.AsyncState;
214. **byte**[] receivedBytes = **new** **byte**[recv];
216. Buffer.BlockCopy(buffer, 0, receivedBytes, 0, recv);
218. RaiseDatagramReceived(\_client, receivedBytes);
220. // then start reading from the network again
221. \_client.BeginReceive(buffer, 0, buffer.Length, SocketFlags.None, HandleDatagramReceived, buffer);
222. }
223. **catch** (Exception)
224. {
226. }
227. }
229. #endregion
231. #region Events
233. /// <summary>
234. /// 接收到数据报文事件
235. /// </summary>
236. **public** **event** EventHandler<EventArgs> DatagramReceived;
238. **private** **void** RaiseDatagramReceived(Socket sender, **byte**[] datagram)
239. {
240. **if** (DatagramReceived != **null**)
241. {
242. DatagramReceived(**this**, **new** EventArgs());
243. }
244. }
246. /// <summary>
247. /// 与服务器的连接已建立事件
248. /// </summary>
249. **public** **event** EventHandler<EventArgs> ServerConnected;
250. /// <summary>
251. /// 与服务器的连接已断开事件
252. /// </summary>
253. **public** **event** EventHandler<EventArgs> ServerDisconnected;
254. /// <summary>
255. /// 与服务器的连接发生异常事件
256. /// </summary>
257. **public** **event** EventHandler<EventArgs> ServerExceptionOccurred;
259. **private** **void** RaiseServerConnected(IPAddress[] ipAddresses, **int** port)
260. {
261. **if** (ServerConnected != **null**)
262. {
263. ServerConnected(**this**, **new** EventArgs());
264. }
265. }
267. **private** **void** RaiseServerDisconnected(IPAddress[] ipAddresses, **int** port)
268. {
269. **if** (ServerDisconnected != **null**)
270. {
271. ServerDisconnected(**this**, **new** EventArgs());
272. }
273. }
275. **private** **void** RaiseServerExceptionOccurred(
276. IPAddress[] ipAddresses, **int** port, Exception innerException)
277. {
278. **if** (ServerExceptionOccurred != **null**)
279. {
280. ServerExceptionOccurred(**this**, **new** EventArgs());
281. }
282. }
284. #endregion
286. #region Send
288. /// <summary>
289. /// 发送报文
290. /// </summary>
291. /// <param name="datagram">报文</param>
292. **public** **void** Send(**byte**[] datagram)
293. {
294. **if** (datagram == **null**)
295. **throw** **new** ArgumentNullException("datagram");
297. **if** (!Connected)
298. {
299. RaiseServerDisconnected(Addresses, Port);
300. **throw** **new** InvalidProgramException(
301. "This client has not connected to server.");
302. }
304. \_client.BeginSend(datagram, 0, datagram.Length, SocketFlags.None, HandleDataSend, \_client);
305. }
307. **private** **void** HandleDataSend(IAsyncResult ar)
308. {
309. ((Socket)ar.AsyncState).EndSend(ar);
310. }
312. /// <summary>
313. /// 发送报文
314. /// </summary>
315. /// <param name="datagram">报文</param>
316. **public** **void** Send(**string** datagram)
317. {
318. Send(**this**.Encoding.GetBytes(datagram));
319. }
321. #endregion
323. #region IDisposable Members
325. /// <summary>
326. /// Performs application-defined tasks associated with freeing,
327. /// releasing, or resetting unmanaged resources.
328. /// </summary>
329. **public** **void** Dispose()
330. {
331. Dispose(**true**);
332. GC.SuppressFinalize(**this**);
333. }
335. /// <summary>
336. /// Releases unmanaged and - optionally - managed resources
337. /// </summary>
338. /// <param name="disposing"><c>true</c> to release both managed
339. /// and unmanaged resources; <c>false</c>
340. /// to release only unmanaged resources.
341. /// </param>
342. **protected** **virtual** **void** Dispose(**bool** disposing)
343. {
344. **if** (!**this**.disposed)
345. {
346. **if** (disposing)
347. {
348. **try**
349. {
350. Close();
352. **if** (\_client != **null**)
353. {
354. \_client = **null**;
355. }
356. }
357. **catch** (SocketException)
358. {
359. }
360. }
362. disposed = **true**;
363. }
364. }
366. #endregion
368. }

顶

2

踩

0

* 上一篇[C#高性能Socket服务器SocketAsyncEventArgs的实现(IOCP)](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)
* 下一篇[一个UDP异步通信例子](http://blog.csdn.net/zhujunxxxxx/article/details/44238239)

# [C#高性能Socket服务器SocketAsyncEventArgs的实现(IOCP)](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

标签： [IOCP](http://www.csdn.net/tag/IOCP)[socket](http://www.csdn.net/tag/socket)[异步](http://www.csdn.net/tag/%e5%bc%82%e6%ad%a5)[高性能](http://www.csdn.net/tag/%e9%ab%98%e6%80%a7%e8%83%bd)

2015-02-06 21:05 5393人阅读 [评论](http://blog.csdn.net/zhujunxxxxx/article/details/43573879#comments)(8) [收藏](javascript:void(0);) [举报](http://blog.csdn.net/zhujunxxxxx/article/details/43573879#report)

本文章已收录于：

[**.NET知识库**](http://lib.csdn.net/base/44?source=blogtop)

 分类：

UDP/TCP通讯（28）  计算机网络（2）  c#技术类（70）

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/zhujunxxxxx/article/details/43573879)

## 原创性申明

本文作者:小竹zz  博客地址:<http://blog.csdn.net/zhujunxxxxx/article/details/43573879>转载请注明出处

## 引言

我一直在探寻一个高性能的Socket客户端代码。以前，我使用Socket类写了一些基于传统异步编程模型的代码（BeginSend、BeginReceive，等等）也看过很多博客的知识，在linux中有poll和epoll来实现，在windows下面  
微软MSDN中也提供了SocketAsyncEventArgs这个类来实现IOCP 地址:<https://msdn.microsoft.com/zh-cn/library/system.net.sockets.socketasynceventargs.aspx>  
NET Framework中的APM也称为Begin/End模式。这是因为会调用Begin方法来启动异步操作，然后返回一个IAsyncResult 对象。可以选择将一个代理作为参数提供给Begin方法，异步操作完成时会调用该方法。或者，一个线程可以等待 IAsyncResult.AsyncWaitHandle。当回调被调用或发出等待信号时，就会调用End方法来获取异步操作的结果。这种模式很灵活，使用相对简单，在 .NET Framework 中非常常见。  
但是，您必须注意，如果进行大量异步套接字操作，是要付出代价的。针对每次操作，都必须创建一个IAsyncResult对象，而且该对象不能被重复使用。由于大量使用对象分配和垃圾收集，这会影响性能。为了解决这个问题，新版本提供了另一个使用套接字上执行异步I/O的方法模式。这种新模式并不要求为每个套接字操作分配操作上下文对象。

代码下载：<http://download.csdn.net/detail/zhujunxxxxx/8431289> 这里的代码优化了的

## 目标

在上面微软提供的例子我觉得不是很完整，没有具体一个流程，只是受到客户端消息后发送相同内容给客户端，初学者不容易看懂流程，因为我花了一天的时间来实现一个功能齐全的IOCP服务器，

效果如下

## 代码

首先是ICOPServer.cs 这个类是IOCP服务器的核心类，目前这个类是网络上比较全的代码，MSDN上面的例子都没有我的全

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43573879) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Net.Sockets;
6. **using** System.Net;
7. **using** System.Threading;
9. **namespace** ServerTest
10. {
11. /// <summary>
12. /// IOCP SOCKET服务器
13. /// </summary>
14. **public** **class** IOCPServer : IDisposable
15. {
16. **const** **int** opsToPreAlloc = 2;
17. #region Fields
18. /// <summary>
19. /// 服务器程序允许的最大客户端连接数
20. /// </summary>
21. **private** **int** \_maxClient;
23. /// <summary>
24. /// 监听Socket，用于接受客户端的连接请求
25. /// </summary>
26. **private** Socket \_serverSock;
28. /// <summary>
29. /// 当前的连接的客户端数
30. /// </summary>
31. **private** **int** \_clientCount;
33. /// <summary>
34. /// 用于每个I/O Socket操作的缓冲区大小
35. /// </summary>
36. **private** **int** \_bufferSize = 1024;
38. /// <summary>
39. /// 信号量
40. /// </summary>
41. Semaphore \_maxAcceptedClients;
43. /// <summary>
44. /// 缓冲区管理
45. /// </summary>
46. BufferManager \_bufferManager;
48. /// <summary>
49. /// 对象池
50. /// </summary>
51. SocketAsyncEventArgsPool \_objectPool;
53. **private** **bool** disposed = **false**;
55. #endregion
57. #region Properties
59. /// <summary>
60. /// 服务器是否正在运行
61. /// </summary>
62. **public** **bool** IsRunning { **get**; **private** **set**; }
63. /// <summary>
64. /// 监听的IP地址
65. /// </summary>
66. **public** IPAddress Address { **get**; **private** **set**; }
67. /// <summary>
68. /// 监听的端口
69. /// </summary>
70. **public** **int** Port { **get**; **private** **set**; }
71. /// <summary>
72. /// 通信使用的编码
73. /// </summary>
74. **public** Encoding Encoding { **get**; **set**; }
76. #endregion
78. #region Ctors
80. /// <summary>
81. /// 异步IOCP SOCKET服务器
82. /// </summary>
83. /// <param name="listenPort">监听的端口</param>
84. /// <param name="maxClient">最大的客户端数量</param>
85. **public** IOCPServer(**int** listenPort,**int** maxClient)
86. : **this**(IPAddress.Any, listenPort, maxClient)
87. {
88. }
90. /// <summary>
91. /// 异步Socket TCP服务器
92. /// </summary>
93. /// <param name="localEP">监听的终结点</param>
94. /// <param name="maxClient">最大客户端数量</param>
95. **public** IOCPServer(IPEndPoint localEP, **int** maxClient)
96. : **this**(localEP.Address, localEP.Port,maxClient)
97. {
98. }
100. /// <summary>
101. /// 异步Socket TCP服务器
102. /// </summary>
103. /// <param name="localIPAddress">监听的IP地址</param>
104. /// <param name="listenPort">监听的端口</param>
105. /// <param name="maxClient">最大客户端数量</param>
106. **public** IOCPServer(IPAddress localIPAddress, **int** listenPort, **int** maxClient)
107. {
108. **this**.Address = localIPAddress;
109. **this**.Port = listenPort;
110. **this**.Encoding = Encoding.Default;
112. \_maxClient = maxClient;
114. \_serverSock = **new** Socket(localIPAddress.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
116. \_bufferManager = **new** BufferManager(\_bufferSize \* \_maxClient \* opsToPreAlloc,\_bufferSize);
118. \_objectPool = **new** SocketAsyncEventArgsPool(\_maxClient);
120. \_maxAcceptedClients = **new** Semaphore(\_maxClient, \_maxClient);
121. }
123. #endregion

126. #region 初始化
128. /// <summary>
129. /// 初始化函数
130. /// </summary>
131. **public** **void** Init()
132. {
133. // Allocates one large byte buffer which all I/O operations use a piece of.  This gaurds
134. // against memory fragmentation
135. \_bufferManager.InitBuffer();
137. // preallocate pool of SocketAsyncEventArgs objects
138. SocketAsyncEventArgs readWriteEventArg;
140. **for** (**int** i = 0; i < \_maxClient; i++)
141. {
142. //Pre-allocate a set of reusable SocketAsyncEventArgs
143. readWriteEventArg = **new** SocketAsyncEventArgs();
144. readWriteEventArg.Completed += **new** EventHandler<SocketAsyncEventArgs>(OnIOCompleted);
145. readWriteEventArg.UserToken = **null**;
147. // assign a byte buffer from the buffer pool to the SocketAsyncEventArg object
148. \_bufferManager.SetBuffer(readWriteEventArg);
150. // add SocketAsyncEventArg to the pool
151. \_objectPool.Push(readWriteEventArg);
152. }
154. }
156. #endregion
158. #region Start
159. /// <summary>
160. /// 启动
161. /// </summary>
162. **public** **void** Start()
163. {
164. **if** (!IsRunning)
165. {
166. Init();
167. IsRunning = **true**;
168. IPEndPoint localEndPoint = **new** IPEndPoint(Address, Port);
169. // 创建监听socket
170. \_serverSock = **new** Socket(localEndPoint.AddressFamily, SocketType.Stream, ProtocolType.Tcp);
171. //\_serverSock.ReceiveBufferSize = \_bufferSize;
172. //\_serverSock.SendBufferSize = \_bufferSize;
173. **if** (localEndPoint.AddressFamily == AddressFamily.InterNetworkV6)
174. {
175. // 配置监听socket为 dual-mode (IPv4 & IPv6)
176. // 27 is equivalent to IPV6\_V6ONLY socket option in the winsock snippet below,
177. \_serverSock.SetSocketOption(SocketOptionLevel.IPv6, (SocketOptionName)27, **false**);
178. \_serverSock.Bind(**new** IPEndPoint(IPAddress.IPv6Any, localEndPoint.Port));
179. }
180. **else**
181. {
182. \_serverSock.Bind(localEndPoint);
183. }
184. // 开始监听
185. \_serverSock.Listen(**this**.\_maxClient);
186. // 在监听Socket上投递一个接受请求。
187. StartAccept(**null**);
188. }
189. }
190. #endregion
192. #region Stop
194. /// <summary>
195. /// 停止服务
196. /// </summary>
197. **public** **void** Stop()
198. {
199. **if** (IsRunning)
200. {
201. IsRunning = **false**;
202. \_serverSock.Close();
203. //TODO 关闭对所有客户端的连接
205. }
206. }
208. #endregion

211. #region Accept
213. /// <summary>
214. /// 从客户端开始接受一个连接操作
215. /// </summary>
216. **private** **void** StartAccept(SocketAsyncEventArgs asyniar)
217. {
218. **if** (asyniar == **null**)
219. {
220. asyniar = **new** SocketAsyncEventArgs();
221. asyniar.Completed += **new** EventHandler<SocketAsyncEventArgs>(OnAcceptCompleted);
222. }
223. **else**
224. {
225. //socket must be cleared since the context object is being reused
226. asyniar.AcceptSocket = **null**;
227. }
228. \_maxAcceptedClients.WaitOne();
229. **if** (!\_serverSock.AcceptAsync(asyniar))
230. {
231. ProcessAccept(asyniar);
232. //如果I/O挂起等待异步则触发AcceptAsyn\_Asyn\_Completed事件
233. //此时I/O操作同步完成，不会触发Asyn\_Completed事件，所以指定BeginAccept()方法
234. }
235. }
237. /// <summary>
238. /// accept 操作完成时回调函数
239. /// </summary>
240. /// <param name="sender">Object who raised the event.</param>
241. /// <param name="e">SocketAsyncEventArg associated with the completed accept operation.</param>
242. **private** **void** OnAcceptCompleted(**object** sender, SocketAsyncEventArgs e)
243. {
244. ProcessAccept(e);
245. }
247. /// <summary>
248. /// 监听Socket接受处理
249. /// </summary>
250. /// <param name="e">SocketAsyncEventArg associated with the completed accept operation.</param>
251. **private** **void** ProcessAccept(SocketAsyncEventArgs e)
252. {
253. **if** (e.SocketError == SocketError.Success)
254. {
255. Socket s = e.AcceptSocket;//和客户端关联的socket
256. **if** (s.Connected)
257. {
258. **try**
259. {
261. Interlocked.Increment(**ref** \_clientCount);//原子操作加1
262. SocketAsyncEventArgs asyniar = \_objectPool.Pop();
263. asyniar.UserToken = s;
265. Log4Debug(String.Format("客户 {0} 连入, 共有 {1} 个连接。", s.RemoteEndPoint.ToString(), \_clientCount));
267. **if** (!s.ReceiveAsync(asyniar))//投递接收请求
268. {
269. ProcessReceive(asyniar);
270. }
271. }
272. **catch** (SocketException ex)
273. {
274. Log4Debug(String.Format("接收客户 {0} 数据出错, 异常信息： {1} 。", s.RemoteEndPoint, ex.ToString()));
275. //TODO 异常处理
276. }
277. //投递下一个接受请求
278. StartAccept(e);
279. }
280. }
281. }
283. #endregion
285. #region 发送数据
287. /// <summary>
288. /// 异步的发送数据
289. /// </summary>
290. /// <param name="e"></param>
291. /// <param name="data"></param>
292. **public** **void** Send(SocketAsyncEventArgs e, **byte**[] data)
293. {
294. **if** (e.SocketError == SocketError.Success)
295. {
296. Socket s = e.AcceptSocket;//和客户端关联的socket
297. **if** (s.Connected)
298. {
299. Array.Copy(data, 0, e.Buffer, 0, data.Length);//设置发送数据
301. //e.SetBuffer(data, 0, data.Length); //设置发送数据
302. **if** (!s.SendAsync(e))//投递发送请求，这个函数有可能同步发送出去，这时返回false，并且不会引发SocketAsyncEventArgs.Completed事件
303. {
304. // 同步发送时处理发送完成事件
305. ProcessSend(e);
306. }
307. **else**
308. {
309. CloseClientSocket(e);
310. }
311. }
312. }
313. }
315. /// <summary>
316. /// 同步的使用socket发送数据
317. /// </summary>
318. /// <param name="socket"></param>
319. /// <param name="buffer"></param>
320. /// <param name="offset"></param>
321. /// <param name="size"></param>
322. /// <param name="timeout"></param>
323. **public** **void** Send(Socket socket, **byte**[] buffer, **int** offset, **int** size, **int** timeout)
324. {
325. socket.SendTimeout = 0;
326. **int** startTickCount = Environment.TickCount;
327. **int** sent = 0; // how many bytes is already sent
328. **do**
329. {
330. **if** (Environment.TickCount > startTickCount + timeout)
331. {
332. //throw new Exception("Timeout.");
333. }
334. **try**
335. {
336. sent += socket.Send(buffer, offset + sent, size - sent, SocketFlags.None);
337. }
338. **catch** (SocketException ex)
339. {
340. **if** (ex.SocketErrorCode == SocketError.WouldBlock ||
341. ex.SocketErrorCode == SocketError.IOPending ||
342. ex.SocketErrorCode == SocketError.NoBufferSpaceAvailable)
343. {
344. // socket buffer is probably full, wait and try again
345. Thread.Sleep(30);
346. }
347. **else**
348. {
349. **throw** ex; // any serious error occurr
350. }
351. }
352. } **while** (sent < size);
353. }

356. /// <summary>
357. /// 发送完成时处理函数
358. /// </summary>
359. /// <param name="e">与发送完成操作相关联的SocketAsyncEventArg对象</param>
360. **private** **void** ProcessSend(SocketAsyncEventArgs e)
361. {
362. **if** (e.SocketError == SocketError.Success)
363. {
364. Socket s = (Socket)e.UserToken;
366. //TODO
367. }
368. **else**
369. {
370. CloseClientSocket(e);
371. }
372. }
374. #endregion
376. #region 接收数据

379. /// <summary>
380. ///接收完成时处理函数
381. /// </summary>
382. /// <param name="e">与接收完成操作相关联的SocketAsyncEventArg对象</param>
383. **private** **void** ProcessReceive(SocketAsyncEventArgs e)
384. {
385. **if** (e.SocketError == SocketError.Success)//if (e.BytesTransferred > 0 && e.SocketError == SocketError.Success)
386. {
387. // 检查远程主机是否关闭连接
388. **if** (e.BytesTransferred > 0)
389. {
390. Socket s = (Socket)e.UserToken;
391. //判断所有需接收的数据是否已经完成
392. **if** (s.Available == 0)
393. {
394. //从侦听者获取接收到的消息。
395. //String received = Encoding.ASCII.GetString(e.Buffer, e.Offset, e.BytesTransferred);
396. //echo the data received back to the client
397. //e.SetBuffer(e.Offset, e.BytesTransferred);
399. **byte**[] data = **new** **byte**[e.BytesTransferred];
400. Array.Copy(e.Buffer, e.Offset, data, 0, data.Length);//从e.Buffer块中复制数据出来，保证它可重用
402. **string** info=Encoding.Default.GetString(data);
403. Log4Debug(String.Format("收到 {0} 数据为 {1}",s.RemoteEndPoint.ToString(),info));
404. //TODO 处理数据
406. //增加服务器接收的总字节数。
407. }
409. **if** (!s.ReceiveAsync(e))//为接收下一段数据，投递接收请求，这个函数有可能同步完成，这时返回false，并且不会引发SocketAsyncEventArgs.Completed事件
410. {
411. //同步接收时处理接收完成事件
412. ProcessReceive(e);
413. }
414. }
415. }
416. **else**
417. {
418. CloseClientSocket(e);
419. }
420. }
422. #endregion
424. #region 回调函数
426. /// <summary>
427. /// 当Socket上的发送或接收请求被完成时，调用此函数
428. /// </summary>
429. /// <param name="sender">激发事件的对象</param>
430. /// <param name="e">与发送或接收完成操作相关联的SocketAsyncEventArg对象</param>
431. **private** **void** OnIOCompleted(**object** sender, SocketAsyncEventArgs e)
432. {
433. // Determine which type of operation just completed and call the associated handler.
434. **switch** (e.LastOperation)
435. {
436. **case** SocketAsyncOperation.Accept:
437. ProcessAccept(e);
438. **break**;
439. **case** SocketAsyncOperation.Receive:
440. ProcessReceive(e);
441. **break**;
442. **default**:
443. **throw** **new** ArgumentException("The last operation completed on the socket was not a receive or send");
444. }
445. }
447. #endregion
449. #region Close
450. /// <summary>
451. /// 关闭socket连接
452. /// </summary>
453. /// <param name="e">SocketAsyncEventArg associated with the completed send/receive operation.</param>
454. **private** **void** CloseClientSocket(SocketAsyncEventArgs e)
455. {
456. Log4Debug(String.Format("客户 {0} 断开连接!",((Socket)e.UserToken).RemoteEndPoint.ToString()));
457. Socket s = e.UserToken **as** Socket;
458. CloseClientSocket(s, e);
459. }
461. /// <summary>
462. /// 关闭socket连接
463. /// </summary>
464. /// <param name="s"></param>
465. /// <param name="e"></param>
466. **private** **void** CloseClientSocket(Socket s, SocketAsyncEventArgs e)
467. {
468. **try**
469. {
470. s.Shutdown(SocketShutdown.Send);
471. }
472. **catch** (Exception)
473. {
474. // Throw if client has closed, so it is not necessary to catch.
475. }
476. **finally**
477. {
478. s.Close();
479. }
480. Interlocked.Decrement(**ref** \_clientCount);
481. \_maxAcceptedClients.Release();
482. \_objectPool.Push(e);//SocketAsyncEventArg 对象被释放，压入可重用队列。
483. }
484. #endregion
486. #region Dispose
487. /// <summary>
488. /// Performs application-defined tasks associated with freeing,
489. /// releasing, or resetting unmanaged resources.
490. /// </summary>
491. **public** **void** Dispose()
492. {
493. Dispose(**true**);
494. GC.SuppressFinalize(**this**);
495. }
497. /// <summary>
498. /// Releases unmanaged and - optionally - managed resources
499. /// </summary>
500. /// <param name="disposing"><c>true</c> to release
501. /// both managed and unmanaged resources; <c>false</c>
502. /// to release only unmanaged resources.</param>
503. **protected** **virtual** **void** Dispose(**bool** disposing)
504. {
505. **if** (!**this**.disposed)
506. {
507. **if** (disposing)
508. {
509. **try**
510. {
511. Stop();
512. **if** (\_serverSock != **null**)
513. {
514. \_serverSock = **null**;
515. }
516. }
517. **catch** (SocketException ex)
518. {
519. //TODO 事件
520. }
521. }
522. disposed = **true**;
523. }
524. }
525. #endregion
527. **public** **void** Log4Debug(**string** msg)
528. {
529. Console.WriteLine("notice:"+msg);
530. }
532. }
533. }

BufferManager.cs 这个类是缓存管理类，是采用MSDN上面的例子一样的 地址: <https://msdn.microsoft.com/zh-cn/library/bb517542.aspx>

SocketAsyncEventArgsPool.cs 这个类也是来自MSDN的 地址:<https://msdn.microsoft.com/zh-cn/library/system.net.sockets.socketasynceventargs.aspx>

需要的话自己到MSDN网站上去取，我就不贴出来了

## 服务器端

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43573879) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

1. **static** **void** Main(**string**[] args)
2. {
4. IOCPServer server = **new** IOCPServer(8088, 1024);
5. server.Start();
6. Console.WriteLine("服务器已启动....");
7. System.Console.ReadLine();
8. }

## 客户端

客户端代码也是很简单

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43573879) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

1. **static** **void** Main(**string**[] args)
2. {
3. IPAddress remote=IPAddress.Parse("192.168.3.4");
4. client c = **new** client(8088,remote);
6. c.connect();
7. Console.WriteLine("服务器连接成功!");
8. **while** (**true**)
9. {
10. Console.Write("send>");
11. **string** msg=Console.ReadLine();
12. **if** (msg == "exit")
13. **break**;
14. c.send(msg);
15. }
16. c.disconnect();
17. Console.ReadLine();
18. }

client.cs

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43573879) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

1. **public** **class** client
2. {
4. **public** TcpClient \_client;
6. **public** **int** port;
8. **public** IPAddress remote;
10. **public** client(**int** port,IPAddress remote)
11. {
13. **this**.port = port;
14. **this**.remote = remote;
15. }
17. **public** **void** connect()
18. {
19. **this**.\_client=**new** TcpClient();
20. \_client.Connect(remote, port);
21. }
22. **public** **void** disconnect()
23. {
24. \_client.Close();
25. }
26. **public** **void** send(**string** msg)
27. {
28. **byte**[] data=Encoding.Default.GetBytes(msg);
29. \_client.GetStream().Write(data, 0, data.Length);
30. }
31. }

IOCPClient类，使用SocketAsyncEventArgs类建立一个Socket客户端。虽然MSDN说这个类特别设计给网络服务器应用，但也没有限制在客户端代码中使用APM。下面给出了IOCPClient类的样例代码：

**[csharp]** [view plain](http://blog.csdn.net/zhujunxxxxx/article/details/43573879) [copy](http://blog.csdn.net/zhujunxxxxx/article/details/43573879)

1. **public** **class** IOCPClient
2. {
3. /// <summary>
4. /// 连接服务器的socket
5. /// </summary>
6. **private** Socket \_clientSock;
8. /// <summary>
9. /// 用于服务器执行的互斥同步对象
10. /// </summary>
11. **private** **static** Mutex mutex = **new** Mutex();
12. /// <summary>
13. /// Socket连接标志
14. /// </summary>
15. **private** Boolean \_connected = **false**;
17. **private** **const** **int** ReceiveOperation = 1, SendOperation = 0;
19. **private** **static** AutoResetEvent[]
20. autoSendReceiveEvents = **new** AutoResetEvent[]
21. {
22. **new** AutoResetEvent(**false**),
23. **new** AutoResetEvent(**false**)
24. };
26. /// <summary>
27. /// 服务器监听端点
28. /// </summary>
29. **private** IPEndPoint \_remoteEndPoint;
31. **public** IOCPClient(IPEndPoint local,IPEndPoint remote)
32. {
33. \_clientSock = **new** Socket(local.AddressFamily,SocketType.Stream, ProtocolType.Tcp);
34. \_remoteEndPoint = remote;
35. }
37. #region 连接服务器
39. /// <summary>
40. /// 连接远程服务器
41. /// </summary>
42. **public** **void** Connect()
43. {
44. SocketAsyncEventArgs connectArgs = **new** SocketAsyncEventArgs();
46. connectArgs.UserToken = \_clientSock;
47. connectArgs.RemoteEndPoint = \_remoteEndPoint;
48. connectArgs.Completed += **new** EventHandler<SocketAsyncEventArgs>(OnConnected);
49. mutex.WaitOne();
50. **if** (!\_clientSock.ConnectAsync(connectArgs))//异步连接
51. {
52. ProcessConnected(connectArgs);
53. }
55. }
56. /// <summary>
57. /// 连接上的事件
58. /// </summary>
59. /// <param name="sender"></param>
60. /// <param name="e"></param>
61. **void** OnConnected(**object** sender, SocketAsyncEventArgs e)
62. {
63. mutex.ReleaseMutex();
64. //设置Socket已连接标志。
65. \_connected = (e.SocketError == SocketError.Success);
66. }
67. /// <summary>
68. /// 处理连接服务器
69. /// </summary>
70. /// <param name="e"></param>
71. **private** **void** ProcessConnected(SocketAsyncEventArgs e)
72. {
73. //TODO
74. }
76. #endregion
78. #region 发送消息
79. /// <summary>
80. /// 向服务器发送消息
81. /// </summary>
82. /// <param name="data"></param>
83. **public** **void** Send(**byte**[] data)
84. {
85. SocketAsyncEventArgs asyniar = **new** SocketAsyncEventArgs();
86. asyniar.Completed += **new** EventHandler<SocketAsyncEventArgs>(OnSendComplete);
87. asyniar.SetBuffer(data, 0, data.Length);
88. asyniar.UserToken = \_clientSock;
89. asyniar.RemoteEndPoint = \_remoteEndPoint;
90. autoSendReceiveEvents[SendOperation].WaitOne();
91. **if** (!\_clientSock.SendAsync(asyniar))//投递发送请求，这个函数有可能同步发送出去，这时返回false，并且不会引发SocketAsyncEventArgs.Completed事件
92. {
93. // 同步发送时处理发送完成事件
94. ProcessSend(asyniar);
95. }
96. }
98. /// <summary>
99. /// 发送操作的回调方法
100. /// </summary>
101. /// <param name="sender"></param>
102. /// <param name="e"></param>
103. **private** **void** OnSendComplete(**object** sender, SocketAsyncEventArgs e)
104. {
105. //发出发送完成信号。
106. autoSendReceiveEvents[SendOperation].Set();
107. ProcessSend(e);
108. }
110. /// <summary>
111. /// 发送完成时处理函数
112. /// </summary>
113. /// <param name="e">与发送完成操作相关联的SocketAsyncEventArg对象</param>
114. **private** **void** ProcessSend(SocketAsyncEventArgs e)
115. {
116. //TODO
117. }
118. #endregion
120. #region 接收消息
121. /// <summary>
122. /// 开始监听服务端数据
123. /// </summary>
124. /// <param name="e"></param>
125. **public** **void** StartRecive(SocketAsyncEventArgs e)
126. {
127. //准备接收。
128. Socket s = e.UserToken **as** Socket;
129. **byte**[] receiveBuffer = **new** **byte**[255];
130. e.SetBuffer(receiveBuffer, 0, receiveBuffer.Length);
131. e.Completed += **new** EventHandler<SocketAsyncEventArgs>(OnReceiveComplete);
132. autoSendReceiveEvents[ReceiveOperation].WaitOne();
133. **if** (!s.ReceiveAsync(e))
134. {
135. ProcessReceive(e);
136. }
137. }
139. /// <summary>
140. /// 接收操作的回调方法
141. /// </summary>
142. /// <param name="sender"></param>
143. /// <param name="e"></param>
144. **private** **void** OnReceiveComplete(**object** sender, SocketAsyncEventArgs e)
145. {
146. //发出接收完成信号。
147. autoSendReceiveEvents[ReceiveOperation].Set();
148. ProcessReceive(e);
149. }
151. /// <summary>
152. ///接收完成时处理函数
153. /// </summary>
154. /// <param name="e">与接收完成操作相关联的SocketAsyncEventArg对象</param>
155. **private** **void** ProcessReceive(SocketAsyncEventArgs e)
156. {
157. **if** (e.SocketError == SocketError.Success)
158. {
159. // 检查远程主机是否关闭连接
160. **if** (e.BytesTransferred > 0)
161. {
162. Socket s = (Socket)e.UserToken;
163. //判断所有需接收的数据是否已经完成
164. **if** (s.Available == 0)
165. {
166. **byte**[] data = **new** **byte**[e.BytesTransferred];
167. Array.Copy(e.Buffer, e.Offset, data, 0, data.Length);//从e.Buffer块中复制数据出来，保证它可重用
169. //TODO 处理数据
170. }
172. **if** (!s.ReceiveAsync(e))//为接收下一段数据，投递接收请求，这个函数有可能同步完成，这时返回false，并且不会引发SocketAsyncEventArgs.Completed事件
173. {
174. //同步接收时处理接收完成事件
175. ProcessReceive(e);
176. }
177. }
178. }
179. }
181. #endregion

184. **public** **void** Close()
185. {
186. \_clientSock.Disconnect(**false**);
187. }
189. /// <summary>
190. /// 失败时关闭Socket，根据SocketError抛出异常。
191. /// </summary>
192. /// <param name="e"></param>
194. **private** **void** ProcessError(SocketAsyncEventArgs e)
195. {
196. Socket s = e.UserToken **as** Socket;
197. **if** (s.Connected)
198. {
199. //关闭与客户端关联的Socket
200. **try**
201. {
202. s.Shutdown(SocketShutdown.Both);
203. }
204. **catch** (Exception)
205. {
206. //如果客户端处理已经关闭，抛出异常
207. }
208. **finally**
209. {
210. **if** (s.Connected)
211. {
212. s.Close();
213. }
214. }
215. }
216. //抛出SocketException
217. **throw** **new** SocketException((Int32)e.SocketError);
218. }

221. /// <summary>
222. /// 释放SocketClient实例
223. /// </summary>
224. **public** **void** Dispose()
225. {
226. mutex.Close();
227. autoSendReceiveEvents[SendOperation].Close();
228. autoSendReceiveEvents[ReceiveOperation].Close();
229. **if** (\_clientSock.Connected)
230. {
231. \_clientSock.Close();
232. }
233. }
235. }

这个类我没有测试，但是理论上是没问题的。

本文地址:<http://blog.csdn.net/zhujunxxxxx/article/details/43573879>

顶

1

踩

0

* 上一篇[Trie-Tree字典树](http://blog.csdn.net/zhujunxxxxx/article/details/43340325)
* 下一篇[c#异步Socket Tcp服务器实现](http://blog.csdn.net/zhujunxxxxx/article/details/43574655)

# C# TcpListener和TcpClient的应用

 (2013-01-14 10:33:06)

[转载▼](javascript:;)

|  |  |
| --- | --- |
| 标签： [杂谈](http://search.sina.com.cn/?c=blog&q=%D4%D3%CC%B8&by=tag) |  |

//服务器端程序

using System.IO;

using System.Net.Sockets;

using System.Net;

namespace PictureServer

{

    class Program

    {

        static void Main(string[] args)

        {

//在本机创建一个TcpListener，端口是8888，

            TcpListener listener = new TcpListener(IPAddress.Any, 8888);

//开始监听，

            listener.Start();

//循环，等待客户端连接

            while (true)

            {

                const int bufferSize = 256;

//接受客户端的连接，利用client保存连接的客户端

                TcpClient client = listener.AcceptTcpClient();

//获取客户端的流stream

                NetworkStream clientStream = client.GetStream();

                byte[] buffer = new byte[bufferSize];

                int readBytes = 0;

//将客户端流读入到buffer中

                readBytes = clientStream.Read(buffer, 0, bufferSize);

//将从客户端流读取的数据保存到字符串request中

                string request = Encoding.ASCII.GetString(buffer).Substring(0, readBytes);

//如果客户端的命令以LIST开头，

                if (request.StartsWith("LIST"))

                {

                    // LIST request - return list

//利用类PictureHelper的函数GetFileListBytes，获取图片文件列表

                    byte[] responseBuffer = PictureHelper.GetFileListBytes();

//将服务器获取的图片文件列表写入到clientStream中

                    clientStream.Write(responseBuffer, 0, responseBuffer.Length);

                }

//如果客户端的请求命令以FILE开头，即获取单个图片文件

                else if (request.StartsWith("FILE"))

                {

                // FILE request - return file

                    // get the filename

//获取请求的文件名字

                    string[] requestMessage = request.Split(':');

                    string filename = requestMessage[1];

//利用File.ReadAllBytes函数将文件里面的文件filename读入到字节数组data中，

    byte[] data = File.ReadAllBytes(Path.Combine(@"C:Documents and SettingsAdministratorMy DocumentsMy Pictures", filename));

                    // Send the picture to the client.

//将data中的文件内容写入到客户端clientStream中，传回客户端

                    clientStream.Write(data, 0, data.Length);

                }

//关闭客户端流

                clientStream.Close();

            }

        }

//静态类PictureHelper，

        public static class PictureHelper

        {

//提供文件夹中的文件列表

            public static string[] GetFileList()

            {

                string[] files = Directory.GetFiles(@"C:Documents and SettingsAdministratorMy DocumentsMy Pictures");

//去掉文件夹路径，只保留文件名

                // Remove the directory path from the filename.

                for (int i = 0; i < files.Length; i++)

                {

                    files[i] = Path.GetFileName(files[i]);

                }

                return files;

            }

//将文件filename的内容读到字节数组中

            public static byte[] GetPictureBytes(string filename)

            {

                FileInfo fileInfo = new FileInfo(filename);

                byte[] buffer = new byte[fileInfo.Length];

                using (FileStream stream = fileInfo.OpenRead())

                {

                    stream.Read(buffer, 0, buffer.Length);

                }

                return buffer;

            }

            public static byte[] GetFileListBytes()

            {

                // LIST request - return list

                string[] files = PictureHelper.GetFileList();

                StringBuilder responseMessage = new StringBuilder();

                foreach (string s in files)

                {

                    responseMessage.Append(s);

                    responseMessage.Append(":");

                }

                byte[] responseBuffer = Encoding.ASCII.GetBytes(

                responseMessage.ToString());

                return responseBuffer;

            }

        }

    }

}

//////客户端程序

using System.Net;

using System.Net.Sockets;

using System.IO;

        private void buttonListPictures\_Click(object sender, EventArgs e)

        {

            const int bufferSize = 4096;

            // Connect to the server.

//生成一个TcpClinet

            TcpClient client = new TcpClient();

//生成服务器的IPHostEntry,因为我的服务器和客户端在同一台计算机上，所以此处用localhost，实际应用中，此处的localhost应用用服务器的ip地址代替，

            IPHostEntry host = Dns.GetHostEntry("localhost");

// 连接到服务器的8888端口

 client.Connect(host.AddressList[0],8888);

            // Send a request to the server.

//创建一个NetworkStream,

         NetworkStream clientStream = client.GetStream();

//request是"LIST"

           string request = "LIST";

//将request放入到字节数组requestBuffer中

            byte[] requestBuffer = Encoding.ASCII.GetBytes(request);

//将请求写入到NetworkStream中，发送到服务器端

            clientStream.Write(requestBuffer, 0, requestBuffer.Length);

//获取服务器端的回应

            // Read the response from the server.

            byte[] responseBuffer = new byte[bufferSize];

//生成一个内存流memstream

            MemoryStream memStream = new MemoryStream();

            int bytesRead = 0;

            do

            {

//将网络流中的数据按256字节一组的顺序写入到memStream中

            bytesRead = clientStream.Read(responseBuffer, 0, bufferSize);

            memStream.Write(responseBuffer, 0, bytesRead);

            } while (bytesRead > 0);

            clientStream.Close();

            client.Close();

//从内存流读取数据

            byte[] buffer = memStream.GetBuffer();

            string response = Encoding.ASCII.GetString(buffer);

//将服务器返回的文件列表分解开，文件名保存到字符串数组fileName中

            string[] fileNames = response.Split(':');

//将字符串数组填到ListBox中

            this.listFiles.DataSource = fileNames;

        }

//获取单个图片文件，并显示之

        private void buttonGetPicture\_Click(object sender, EventArgs e)

        {

            const int bufferSize = 4096;

            TcpClient client = new TcpClient();

            IPHostEntry host = Dns.GetHostEntry("localhost");

            client.Connect(host.AddressList[0],8888);

            NetworkStream clientStream = client.GetStream();

            try

            {

//生成请求，以FIlE：开头，后接在listbox中选择的文件名

                string request = "FILE:" + this.listFiles.SelectedItem.ToString();

                byte[] requestBuffer = Encoding.ASCII.GetBytes(request);

//将请求写入到clientStream中，发送到服务器端

                clientStream.Write(requestBuffer, 0, requestBuffer.Length);

                byte[] responseBuffer = new byte[bufferSize];

            MemoryStream memStream = new MemoryStream();

            int bytesRead = 0;

            do

            {

//获取服务器端的回应，

                bytesRead = clientStream.Read(responseBuffer, 0, bufferSize);

//服务器的回应写入到memStream

                memStream.Write(responseBuffer, 0, bytesRead);

            } while (bytesRead > 0);

            clientStream.Close();

            client.Close();

 //从内存流中读取图片文件数据，显示在picturebox控件上

            pictureBox.Image = Image.FromStream(memStream);

        }

        catch (Exception exce)

        {

            MessageBox.Show(exce.Message);

        }

        }