ESFramework 资料合集

傲瑞科技

www.oraycn.com

武 汉 傲 瑞 科 技 有 限 公 司

Wuhan Oray Technology Co.,Ltd.

http://www.oraycn.com

2017 年 04 月

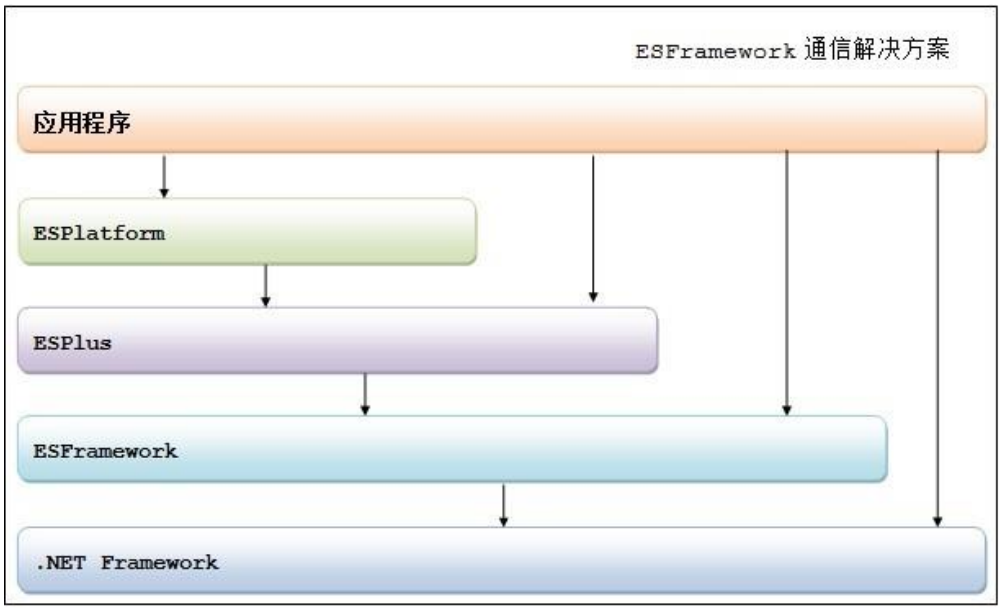
ESFramework 开发手册

（00）—— 概述

ESFramework 是一套性能卓越、稳定可靠、强大易用的跨平台通信框架，支持应用服务器集群。其内置了消息的收发与自定义处理（支持同步/异步模型）、消息广播、P2P 通道、文件传送（支持断点续传）、心跳检测、断线重连、登录验证、在线用户管理、好友与群组管理、性能诊断等功能。基于 ESFramework，您可以方便快捷地开发出各种优秀的网络通信应用。此外，我们在长期实践中所积累的丰富经验，更将成为您强大的技术保障，从开发到上线直至后续运维，全程为您保驾护航，让您高枕无忧。

一. ESFramework 体系的构成

ESFramework 体系直接构建在.NET Framework 2.0 上，它由通信核心 ESFramework、应用增强 ESPlus、以及群集平台 ESPlatform 构成。它们的层次依赖关系如下图所示：



我们的应用程序可以直接基于通信核心 ESFramework 来构建 这样做可以拥有最大限度的灵活性来组装整个应用，但是需要手动做的工作也最多。为了快速而高效地构建应用程序，我们推荐基于 ESPlus 进行开发。ESPlus 内置众多组件供我们直接使用，像消息头、解析器、消息处理器、序列化器、自定义信息、文件传送、P2P 通道、好友/组友状态改变通知、等等。

基于 ESPlus 构建的通信应用程序，当同时在线用户数量剧增的时候，只要修改仅仅几行代码和配置，就可以将其平滑地迁移到 ESPlatform 平台，以实现应用服务器的群集和负载均衡。

跨平台也是 ESFramework 体系的重要特性之一，ESFramework 将通过提供多平台的客户端引擎来实现这一点。

跨平台解决方案的第一阶段主要任务是对主流移动设备的支持。其规划图如下所示：

所有类型的客户端都使用几乎完全一致的 API 接口，且都可与同一 ESFramework 服务端进行通信，从而使得异构平台变得相对透明。

二. ESPlus 快速开发

本手册将着重介绍如何使用 ESPlus 提供的 Rapid 引擎（客户端引擎为 ESPlus.Rapid.IRapidPassiveEngine，服务端引擎为 ESPlus.Rapid.IRapidServerEngine）来进行 ESFramework 通信系统的快速开发。欲掌握 ESPlus 快速开发，需要抓住三个方面：Rapid 引擎、四大核心武器、一个可选功能。

1.Rapid 引擎

在使用 ESPlus 开发的时候，首先要初始化 Rapid 引擎 ——服务端是 IRapidServerEngine、 客户端是 IRapidPassiveEngine。在引擎对象初始化成功之后，我们就可以使用引擎对象暴露出的四大武器了。

2.四大核心武器

通过 Rapid 引擎对象暴露出的属性，可以获得 ESPlus 提供的四大武器，大多数情况下，我们正是靠使用这四大武器来进行快速应用开发的。四大武器都都位于 ESPlus.Application 对应的子空间下：

（1）CustomizeInfo 子空间：用于发送和处理自定义信息。

（2）Basic 子空间：用于完成在线用户管理、基础功能（如获取在线好友列表等）、和接收用户状态改变通知（如好友上下线等）。

（3）FileTransfering 子空间：用于完成与文件传送相关的所有功能。

（4）P2PSession 子空间：用于完成与 P2P 打洞、P2P 通信相关的所有功能。

3.可选功能

很多分布式通信应用都涉及到客户端与客户端之间需要交互，或者涉及到群组的功能需求（比如在目标组内广播消息）。为了方便更多的开发者，ESPlus 对此提供了直接的支持，并将它们统一称为“联系人”。对于开发一般的系统而言，它们并不是必需的，而且，我们完全可以基于上面的四大武器自己实现这个可选功能。 该可选武器位于 ESPlus.Application.Contacts 命名空间下。

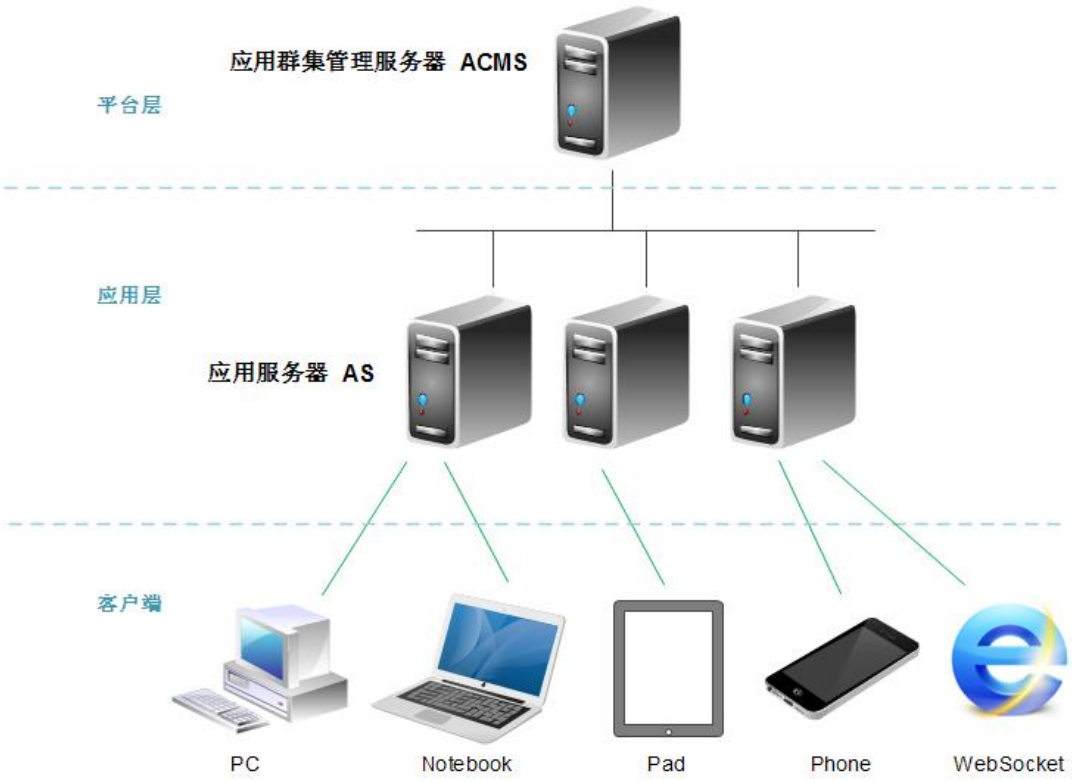
“联系人”功能的核心就是 IContactsManager 接口。注意，类似联系人的添加/移除，组成员的添加/移除等，ESPlus 是不关心的，ESPlus 关心的是“关系” —— 即某个用户有哪些相关联系人，某个组包含哪些成员。有了这些信息，在用户上/下线时，ESPlus 就知道要将上/下线事件通知给哪些相关联系人；组广播时，ESPlus 也才知道要将广播消息发送给哪些用户。

三. ESPlatform 群集平台

ESPlatform 平台用于将基于 ESFramework/ESPlus 开发的应用服务器 AS 进行群集，以实现负载均衡而达到能负载巨大量的用户同时在线，并且，登录到群集系统中的不同 AS 上的用户之间可以相互通信，就好像它们登陆在同一个 AS 上一样。ESPlatform 提供了可直接运行的平台服务器（即用于应用服务器群集管理的服务器 ACMS），并内置了三种常用的负载均衡策略。

ESFramework 体系的一个最大优势在于，从单服务器应用转变为群集应用，是如此的简单。要做的所有事情包括：启动平台服务器 ACMS、更改数行代码、修改数行配置。

ESPlatform 所支持的的群集模型如下图所示：



在后面介绍 ESPlatform 的章节，我们将详细介绍群集平台的功能特点，以及如何迁移到群集平台。

四. ESFramework 体系的特点

1.高性能

ESFramework 底层使用 IOCP 模型，使得数据收发与处理达到最高性能。当前主流配置的服务器（如至强 4 核双 CPU、4-8G 内存）可轻松处理 10000 个同时在线连接，每秒处理 50000 个以上的请求。当然，最终能达到的并发，更取决于具体应用的业务逻辑，如果业务逻辑复杂、处理单个请求都对 CPU 和内存的消耗都比较高，那么就会导致并发数下降，这时也许就要优化我们的业务逻辑代码了、或者使用更多的服务器来分担负载（比如迁移到 ESPlatform）。关于 ESFramework 的性能测试的更多内容可以参见 ESFramework 4.0 性能测试。

2.可靠、稳定

ESFramework 起始于 2004 年，从 0.1 版本不断完善和优化到今天 6.0 版本已经有 12 年时间，其内核（ESFramework.dll）已经相当成熟稳定，所有已经发现的内核 bug 都已被解决，而且自 2009 年以来，没有新的内核的 bug 被发现。另外，ESFramework 只会在需要时才使用必要的资源（如 CPU、内存），并且会及时释放持有的资源，不会有内存泄露等情况发生。如果您的业务逻辑代码没有 bug，那么基于 ESFramework 的服务端正常运行一年，都不用重启一次。

3.功能强大丰富

现有的大多数通信框架仅仅解决了通信层的问题，而且几乎都是部分地解决。而 ESFramework 不仅仅完整地解决了通信层的需求，并且也解决了绝大多数通信系统中都关注的应用层的重要问题，这使得基于 ESFramework 开发分布式通信系统更迅速、更高效。

在通信层：ESFramework 支持 TCP/UDP、文本协议/二进制协议、服务端/客户端以及它们组合的任意方式，并提供多种通信引擎供服务端开发人员和客户端开发人员使用。

在应用层：ESFramework 内置了在线用户管理、消息拦截、消息同步调用、回复异步调用、通道智能选择、客户端登录验证、心跳检测、消息优先级、断线自动重连、在线状态改变自动通知（好友与组友）、重登陆模式选择、完整的异常日志、文件传送（支持断点续传）、组广播、带 ACK 机制的信息发送、高效的二进制序列化器、等等功能。

在安全性：ESFramework 内置了常见的重要安全机制以防止恶意用户在应用层对服务端进行试探或攻击。这些机制有：消息格式完整验证、消息加密、验证未绑定的消息、绑定连接、关闭空连接等。

4.可靠的 P2P

ESFramework 提供了基于 TCP 和 UDP 的 P2P 通信（不仅是局域网，还支持广域网 P2P 通信），而且基于 UDP 的 P2P 做了增强，以保证基于 UDP 的 P2P 通信也像 TCP 一样可靠。在客户端之间需要高频通信的分布式系统中（如 IM 系统等），可靠的 P2P 通信将为您节省巨大的带宽和服务器成本。

5.高伸缩性的群集平台

ESPlatform 平台支持基于 ESFramework 的应用程序的 Cluster（群集），其内置了 3 种最常用的负载均衡策略。仅仅通过修改几行代码就可以将一个基于 ESFramework 的应用程序平滑迁入到 ESPlatform 平台中，以实现多台应用服务器的 Cluster，从而应对日益增长的巨大并发。使用 ESPlatform 群集，我们可以非常方便地在运行时动态添加/移除应用服务器（AS）实例节点。

6.跨平台

跨平台是 ESFramework 的重要目标之一，ESFramework 通过提供多平台的客户端引擎来实现这一点。基于 ESFramework 开发的不同平台的客户端之间可以相互通信，如此，异构环境将变得透明化。ESFramework 目前支持的平台包括：.NET、Android、iOS、Xamarin、Mono、Unity、WebSocket、WPF 等。

7.服务端性能诊断

ESFramework 框架在服务端内置了性能跟踪诊断功能。如果基于 ESFramework 构建的服务端在运行时，遇到性能问题或某些故障，那么可以开启诊断功能，ESFramework 将自动跟踪每种类型消息的处理情况，之后通过分析日志，就可以很快发现问题所在。

8.适用范围广

ESFramework 可以用于任何需要分布式通信的软件系统中，而且其群集功能还可以支持那些同时在线用户数巨大的系统。比如，即时通讯系统（IM），大型多人在线游戏（MMORPG）、在线网页游戏（WebGame）、文件传送系统（FTS）、数据采集系统（DGS）、分布式 OA 系统等等。

9.文档齐全、接口清晰规范

ESFramework 提供的所有接口和 API 都具有良好的编码风格，与.NET Framework 完全一致。我们提供了 MSDN 风格的帮助文档、Demo 源码、ESFramework 开发手册系列文章供您研究和学习使用 ESFramework 进行开发。

10.历史经验分享

我们在过去的 10+年里，基于 ESFramework 开发了很多网络通信系统，也为诸多客户提供技术支持和运行故障排查服务，所以，在以下方面积累了丰富的经验：如基于 ESFramework 的最佳开发实践、服务端性能瓶颈排查、服务端运行故障排查、性能优化等等。如果您是第一次基于 ESFramework 进行二次开发，那么，我们分享的这些经验将为您的项目上线运行提供更强大的后续保障。

五. ESFramework 与 WCF 等技术的区别

WCF 以及 WebService、.NET Remoting，还有更古老的 RPC、DCOM 等，这些技术一脉相承，它们要达到的最核心目标就是要像调用本地方法一样调用远程方法。它们是标准的 C/S 结构，且服务端处于被动的状态，即，通常都是客户端主动向服务器请求并获取结果，服务端一般不主动发信息给客户端。

对于那些只需要客户端主动请求服务器的应用来说，使用这些技术是非常合适的。但是，也有很多应用不仅仅需要客户端主动请求服务端，同时也需要服务端能主动通知客户端，有的甚至需要客户端之间相互传递消息。像很多网络游戏、IM 系统等等，就有类似的需求。在这种情况下，使用 WCF 等技术就不太合适，虽然，我们可以手动做很多工作来模拟达到类似的效果，但是，这并不划算，而且，这也偏离了 WCF 等技术的设计目标。

相反，ESFramework 就非常适合类似的需求，并且 ESFramework 与应用贴得更近，为应用提供了更强大的支持（像可靠 P2P、服务器群集等），就如上面介绍 ESFramework 特点时所讲述的那样。

（01）—— 发送和处理信息

本文介绍 ESFramework 开发手册（00）—— 概述一文中提到的四大武器的第一个：发送和处理自定义信息。

使用通信框架最基础的需求就是收发信息，ESFramework 底层已经为我们封装好了所有与信息收发相关的操作，我们只要调用 ESPlus.Application.CustomizeInfo 命名空间下的相关组件的 API 来发送信息，以及实现对应的处理器接口来处理收到的信息就可以了。

一.客户端发送信息

客户端可以发送信息给服务端，也可以发送信息给其他在线用户。

客户端通过 ESPlus.Application.CustomizeInfo.Passive.ICustomizeOutter 接口提供的方法来发送信息。

我们可以从 ESPlus.Rapid.IRapidPassiveEngine 暴露的 CustomizeOutter 属性来获取 ICustomizeOutter 引用。

API：

```
// 摘要:
//     该接口用于向服务器或其它在线用户发送自定义信息、同步调用请求。 zhuweisky 2010.08.17
public interface ICustomizeOutter
{
    // 摘要:
    //     向服务器提交请求信息，并返回服务器的应答信息。如果超时没有应答则将抛出Timeout异常。
    //
    // 参数:
```

```
        //   informationType:
        //     自定义请求信息的类型
        //
        //   info:
        //     请求信息
        //
        // 返回结果:
        //     服务器的应答信息
        byte[] Query(int informationType, byte[] info);
        //
        // 摘要:
        //     向在线目标用户或服务器提交请求信息，并返回应答信息。如果目标用户不在线，或超时没有应答则将抛出TimeoutException。如果对方在处理请求时出现未捕获的异常，则该调用会抛出HandingException。
        //
        // 参数:
        //   targetUserID:
        //     接收并处理请求消息的目标用户ID。如果为null，表示信息接收者为服务端。
        //
        //   informationType:
        //     自定义请求信息的类型
        //
        //   info:
        //     请求信息
        //
        // 返回结果:
        //     应答信息
        byte[] Query(string targetUserID, int informationType, byte[] info);
        //
        // 摘要:
        //     回复异步调用。向在线目标用户或服务器提交请求信息，当收到应答信息或超时时，将回调CallbackHandler函数。
        //
        // 参数:
        //   targetUserID:
        //     接收并处理请求消息的目标用户ID。如果为null，表示信息接收者为服务端。
        //
        //   informationType:
        //     自定义请求信息的类型
        //
        //   info:
        //     请求信息
        //
        //   handler:
        //     用于处理回复信息的处理器
        //
        //   tag:
        //     携带的状态数据，将被传递给回调函数handler
        void Query(string targetUserID, int informationType, byte[] info, CallbackHandler handler,
object tag);
```

```
        //
        // 摘要:
        //     向服务器发送信息。
        //
        // 参数:
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        void Send(int informationType, byte[] info);
        //
        // 摘要:
        //     向在线用户targetUserID发送信息。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        void Send(string targetUserID, int informationType, byte[] info);
        //
        // 摘要:
        //     向在线用户或服务器发送信息。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示接收者为服务器。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   post:
        //     是否采用Post模式发送消息
        //
        //   action:
        //     当通道繁忙时所采取的动作
        //
        // 返回结果:
        //     如果成功发送，将返回true；否则，（比如丢弃）返回false。
        bool Send(string targetUserID, int informationType, byte[] info, bool post,
ActionTypeOnChannelIsBusy action);
        //
        // 摘要:
        //     向在线用户或服务器发送信息。
```

```
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示接收者为服务器。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   post:
        //     是否采用Post模式发送消息
        //
        //   action:
        //     当通道繁忙时所采取的动作
        //
        //   mode:
        //     通道选择模型。如果接收者为服务器，则忽略该参数。
        //
        // 返回结果:
        //     如果成功发送，将返回true；否则，（比如丢弃）返回false。
        bool Send(string targetUserID, int informationType, byte[] info, bool post,
ActionTypeOnChannelIsBusy action, ChannelMode mode);
        //
        // 摘要:
        //     向在线用户或服务器发送大的数据块信息。直到数据发送完毕，该方法才会返回。如果担心长时
间阻塞调用线程，可考虑异步调用本方法。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示接收者为服务器。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   blobInfo:
        //     大的数据块信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        void SendBlob(string targetUserID, int informationType, byte[] blobInfo, int fragmentSize);
        //
        // 摘要:
        //     通过P2P通道向在线用户发送大的数据块信息。直到数据发送完毕，该方法才会返回。如果担心长
时间阻塞调用线程，可考虑异步调用本方法。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示接收者为服务器。
        //
        //   informationType:
```

```
        //     自定义信息类型
        //
        //   blobInfo:
        //     大的数据块信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        //
        //   mode:
        //     通道选择模型。如果接收者为服务器，则忽略该参数。
        void SendBlob(string targetUserID, int informationType, byte[] blobInfo, int fragmentSize,
ChannelMode mode);
        //
        // 摘要:
        //     通过可靠的P2P通道向在线用户targetUserID发送信息。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID，不能为null。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        //
        //   actionType:
        //     当P2P通道不存在时，采取的操作
        //
        //   post:
        //     是否采用Post模式发送消息
        //
        //   action:
        //     当通道繁忙时所采取的动作
        //
        // 返回结果:
        //     如果成功发送，将返回true；否则，（比如丢弃）返回false。
        bool SendByP2PChannel(string targetUserID, int informationType, byte[] info,
ActionTypeOnNoP2PChannel actionType, bool post, ActionTypeOnChannelIsBusy action);
        //
        // 摘要:
        //     向在线用户或服务器发送信息，并等待其ACK。当前调用线程会一直阻塞，直到收到ACK；如果超
时都没有收到ACK，则将抛出TimeoutException。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示信息接收者为服务端。
        //
        //   informationType:
```

```
        //     自定义信息类型
        //
        //   info:
        //     信息
        void SendCertainly(string targetUserID, int informationType, byte[] info);
        //
        // 摘要:
        //     向在线用户或服务器发送信息。当前调用线程会立即返回。当收到ACK或者超时都没有收到ACK,
将回调ResultHandler。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID。如果为null，表示信息接收者为服务端。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        //
        //   ackHandler:
        //     被回调的处理器
        //
        //   tag:
        //     携带的状态数据，将被传递给回调函数handler
        void SendCertainly(string targetUserID, int informationType, byte[] info, ResultHandler
ackHandler, object tag);
        //
        // 摘要:
        //     即使与目标用户之间有可靠的P2P通道存在，也要通过服务器转发信息。
        //
        // 参数:
        //   targetUserID:
        //     接收消息的目标用户ID，不能为null。
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        void TransferByServer(string targetUserID, int informationType, byte[] info);
    }
```

发送信息有几种方式：

1. 普通发送：

调用 Send 方法进行普通发送，即将信息写入网络流后就立即返回。

Send 方法的重载有个 ActionTypeOnChannelIsBusy 参数，用于指示当通道繁忙时所采取的动作：继续发送、或丢弃数据。在某些系统中，对于一些非重要非紧急信息的发送，可以为 ActionTypeOnChannelIsBusy 参数传入枚举

值 Discard（丢弃）。

2. 带 ACK 机制的发送：

调用 SendCertainly 方法发送信息时会启用 ACK 机制，即将信息发送出去后，调用并不返回，而是要等到接收方的 ACK 后，才返回。如果接收方不在线，SendCertainly 调用会在阻塞一段时间后（默认为 30 秒），抛出超时异常。

ACK 机制是由 ESFramework 底层实现的，我们直接使用，不需要做任何额外的其它工作。关于带 ACK 机制的信息发送的更多内容可以参见 ACK 机制。

3. 信息同步调用：

调用 Query 方法可以发送请求信息，并返回接收方处理请求后的应答信息。就像方法调用一样 - - 使用参数调用方法并返回结果。从 Query 方法的重载看到，信息同步调用的对象既可以是服务端、也可以是另外一个在线客户端。关于信息同步调用的更多内容可以参见消息同步调用。

4. 回复异步调用：

重载的 Query 方法（带有 CallbackHandler 参数的）在发送请求信息后，不会阻塞而继续向下执行，而框架在收到对应的回复信息时，会回调 CallbackHandler 委托指向的方法。由于调用线程与回复回调的线程不是同一个线程，所以称这种机制为回复异步调用。

5. 使用 P2P 通道发送：

调用 SendByP2PChannel 方法可以明确指定使用 P2P 通道进行发送，这是一种普通发送。如果与接收者之间的 P2P 通道不存在，则由 ActionTypeOnNoP2PChannel 参数指示如何动作：使用服务器转发、或者丢弃信息。关于 P2P 通道的更多内容可参考 ESFramework 开发手册（04） —— 可靠的 P2P。

6. 强制经过服务器转发：

对于某些类型的 P2P 信息，我们可能想在服务端监控它，如果这些信息还是经过 P2P 通道发送的话，那么服务端将捕获不到这些信息。TransferByServer 方法用于解决这一问题。如果调用 TransferByServer 方法发送信息，那么即使有可靠的 P2P 通道存在，信息仍然会经过服务器中转。

7. 发送大数据块：

调用 SendBlob 方法可以将大数据块信息（比如一张大图片）发送给服务端或任何其他的在线用户。 关于大数据块的更多内容，可以参考 ESFramework 使用技巧 —— 大数据块信息。

二.服务端发送信息

服务端可以通过 ESPlus.Application.CustomizeInfo.Server.ICustomizeController 接口向客户端发送信息和广播、以及同步调用客户端。

我们可以从 ESPlus.Rapid.IRapidServerEngine 暴露的 CustomizeController 属性来获取 ICustomizeController 引用。

API：

```
// 摘要:
    //     服务端主动向用户发送/投递自定义信息以及同步调用客户端的控制接口。 zhuweisky 2010.08.17
    public interface ICustomizeController
    {
        // 摘要:
        //     当服务端接收到来自客户端的信息时（包括转发的信息，但不包括Blob信息），触发此事件。
```

```
        event CbGeneric<Information> InformationReceived;
        //
        // 摘要:
        //     当因为目标用户不在线而导致服务端转发自定义信息失败时（不包括Blob及其片段信息），将触发该事件。参数为转发失败的信息。
        event CbGeneric<Information> TransmitFailed;
        // 摘要:
        //     询问当前AS的在线用户，并返回应答信息。如果超时没有应答则将抛出Timeout异常。如果客户端在处理请求时出现未捕获的异常，则该调用会抛出HandingException。
        //
        // 参数:
        //   userID:
        //     接收并处理服务器询问的目标用户ID
        //
        //   informationType:
        //     自定义请求信息的类型
        //
        //   info:
        //     请求信息
        //
        // 返回结果:
        //     客户端给出的应答信息
        byte[] QueryLocalClient(string userID, int informationType, byte[] info);
        //
        // 摘要:
        //     回复异步调用。当前AS的在线用户发送请求信息，当收到应答信息或超时时，将回调CallbackHandler函数。
        //
        // 参数:
        //   userID:
        //     接收并处理服务器询问的目标用户ID
        //
        //   informationType:
        //     自定义请求信息的类型
        //
        //   info:
        //     请求信息
        //
        //   handler:
        //     用于处理回复信息的处理器
        //
        //   tag:
        //     携带的状态数据，将被传递给回调函数handler
        void QueryLocalClient(string userID, int informationType, byte[] info, CallbackHandler handler, object tag);
        //
        // 摘要:
        //     向ID为userID的在线用户发送信息。如果用户不在线，则直接返回。
        //
```

```
        // 参数:
        //   userID:
        //     接收消息的用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        void Send(string userID, int informationType, byte[] info);
        //
        // 摘要:
        //     向ID为userID的在线用户发送信息。如果用户不在线，则直接返回。
        //
        // 参数:
        //   userID:
        //     接收消息的用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        //
        //   post:
        //     是否采用Post模式发送消息
        //
        //   action:
        //     当通道繁忙时所采取的动作
        //
        // 返回结果:
        //     如果成功发送，将返回true；否则，（比如丢弃）返回false。
        bool Send(string userID, int informationType, byte[] info, bool post, ActionTypeOnChannelIsBusy
action);
        //
        // 摘要:
        //     向ID为userID的在线用户发送大数据块信息。直到数据发送完毕，该方法才会返回。如果担心长
时间阻塞调用线程，可考虑异步调用本方法。
        //
        // 参数:
        //   userID:
        //     接收消息的用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   blobInfo:
        //     大数据块信息
        //
```

```
        //   fragmentSize:
        //     分片传递时，片段的大小
        void SendBlob(string userID, int informationType, byte[] blobInfo, int fragmentSize);
        //
        // 摘要:
        //     向当前AS上的在线用户发送信息，并等待其ACK。当前调用线程会一直阻塞，直到收到ACK；如果超时都没有收到ACK，则将抛出Timeout异常。
        //
        // 参数:
        //   userID:
        //     接收消息的用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        void SendCertainlyToLocalClient(string userID, int informationType, byte[] info);
        //
        // 摘要:
        //     向当前AS上的在线用户发送信息。当前调用线程会立即返回。当收到ACK或者超时都没有收到ACK，将回调ResultHandler。
        //
        // 参数:
        //   userID:
        //     接收消息的用户ID
        //
        //   informationType:
        //     自定义信息类型
        //
        //   info:
        //     信息
        //
        //   handler:
        //     被回调的处理器
        //
        //   tag:
        //     携带的状态数据，将被传递给回调函数handler
        void SendCertainlyToLocalClient(string userID, int informationType, byte[] info, ResultHandler handler, object tag);
    }
```

各个方法含义几乎与客户端是一致的。ICustomizeController 接口还暴露了 TransmitFailed 事件，我们可以通过预定该事件来监控那些转发失败的信息。

SendCertainlyToLocalClient 方法和 QueryLocalClient 方法的名称都以 LocalClient 结尾，其在 ESPlatform 群集平台中就会显示其特别的含义 -- 这两个方法只能针对连接到当前服务端的客户端进行调用。而其它的几个方法，则是可以将信息发送给任何在线用户的，即使该用户位于群集中的其它应用服务器上。

三.处理信息

客户端可以收到来自其它客户端或服务端的信息、大数据块、以及同步调用。服务端也可以收到来自客户端的信息（转发的信息除外）及同步调用。那么，我们如何处理这些接收到的信息了？

无论是服务端，还是客户端，都只要实现 ESPlus.Application.CustomizeInfo.ICustomizeHandler 接口即可。

API：

```
// 摘要:
//     自定义信息处理器接口。
public interface ICustomizeHandler
{
    // 摘要:
    //     处理接收到的自定义信息（包括大数据块信息）。
    //
    // 参数:
    //   sourceUserID:
    //     发送该信息的用户ID。如果为null，表示信息发送者为服务端。
    //
    //   informationType:
    //     自定义信息类型
    //
    //   info:
    //     信息
    void HandleInformation(string sourceUserID, int informationType, byte[] info);
    //
    // 摘要:
    //     处理接收到的请求并返回应答信息。
    //
    // 参数:
    //   sourceUserID:
    //     发送该请求信息的用户ID。如果为null，表示信息发送者为服务端。
    //
    //   informationType:
    //     自定义请求信息的类型
    //
    //   info:
    //     请求信息
    //
    // 返回结果:
    //     应答信息
    byte[] HandleQuery(string sourceUserID, int informationType, byte[] info);
}
```

（1）凡是 sourceUserID 参数为 null 的，都表示被处理的信息是来自服务端的；否则，表示被处理的信息是由其它在线客户端发出的。

（2）ICustomizeHandler 即用于客户端、也用于服务端。如果是用于服务端，则方法的第一个参数 sourceUserID 是绝对不会为 null 的。

（3）ICustomizeHandler 接口的所有方法都是在后台线程中被调用的，所以如果这些方法的实现中涉及到了操作 UI，一定要将调用转发到 UI 线程。

（4）在客户端，将 ICustomizeHandler 的实现类的实例传递给 ESPlus.Rapid.IRapidPassiveEngine 的 Initialize 方法以挂接到框架；

在服务端，则将 ICustomizeHandler 的实现类的实例传递给 ESPlus.Rapid.IRapidServerEngine 的 Initialize 方法以挂接到框架。

四.更多说明

1. 信息发送模型

信息发送可以使用同步模型或异步模型，在方法中通过 bool 型 post 参数体现出来。如果其值为 true，表示使用异步模型（即发送方法的调用立即返回，不用等到信息发送完毕）；否则使用同步模型（阻塞调用线程，直到信息发送完毕）。

2. 信息处理

客户端和服务端的 ICustomizeHandler，我们称之为自定义信息处理器，或者业务处理器，表示其用于处理我们应用系统的具体业务逻辑。

（1）业务处理器将在后台线程中被调用，所以，实现业务处理器的方法中如果涉及到了 UI 操作，则必须将调用转发到 UI 线程。

（2）业务处理器的方法必须尽可能快地返回，否则，将不能及时地处理后续的消息。如果某个业务处理方法非常耗时，可以考虑使用异步方式。

3. 大数据块

当发送大数据块时，发送方会将其拆分为许多连续的片段逐个发送，而在接收方会自动将接收到的片段重组起来构成一个完整的信息。而且无论是发送大数据块，还是普通信息，在接收方都是调用相同的方法（ICustomizeHandler 的 HandleInformation 方法）来处理的。

五.另一套消息收发机制

除了通过 Customize 空间发送消息和处理消息外，ESFramework 的最新版本增加了另一套收发消息的机制 —— 直接通过引擎接口收发消息。

与 Customize 机制的区别在于，通过引擎接口发送的消息，接收方将触发 MessageReceived 事件，而不是回调 Handler 接口。这两套机制是相互独立的，不会相互影响。下面我们把引擎接口中与消息收发相关的 API 摘出来：

API：

```
public interface IRapidPassiveEngine
{
    /// <summary>
    /// 当接收到来自服务器或其它用户的消息时，触发此事件。
    /// 事件参数：sourceUserID - informationType - message - tag 。
    /// 如果消息来自服务器，则sourceUserID为null。
    /// </summary>
    event CbGeneric<string, int, byte[], string> MessageReceived;
    /// <summary>
    /// 向服务器或其它在线用户发送消息。如果其它用户不在线，消息将被丢弃。
    /// </summary>
    /// <param name="targetUserID">接收者的UserID，如果为服务器，则传入null</param>
    /// <param name="informationType">消息类型</param>
    /// <param name="message">消息内容</param>
    /// <param name="tag">附加内容</param>
    void SendMessage(string targetUserID, int informationType, byte[] message, string tag);
```

```
        /// <summary>
        /// 向服务器或其它在线用户发送消息。如果其它用户不在线，消息将被丢弃。
        /// </summary>
        /// <param name="targetUserID">接收者的UserID，如果为服务器，则传入null</param>
        /// <param name="informationType">消息类型</param>
        /// <param name="message">消息内容</param>
        /// <param name="tag">附加内容</param>
        /// <param name="fragmentSize">消息将被分块发送，分块的大小</param>
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize);
        /// <summary>
        /// 向服务器或其它在线用户异步发送消息（当前调用线程立即返回）。如果其它用户不在线，消息将被
丢弃。
        /// </summary>
        /// <param name="targetUserID">接收者的UserID，如果为服务器，则传入null</param>
        /// <param name="informationType">消息类型</param>
        /// <param name="message">消息内容</param>
        /// <param name="tag">附加内容</param>
        /// <param name="fragmentSize">消息将被分块发送，分块的大小</param>
        /// <param name="handler">当发送任务结束时，将回调该处理器</param>
        /// <param name="handlerTag">将回传给ResultHandler的参数</param>
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize, ResultHandler handler,
                         object handlerTag);
    }
```

```
    public interface IRapidServerEngine
    {
        /// <summary>
        /// 当接收到来自客户端的消息时，触发此事件。
        /// 事件参数：sourceUserID - informationType - message - tag 。
        /// </summary>
        event CbGeneric<string, int, byte[], string> MessageReceived;
        /// <summary>
        /// 向在线用户发送消息。如果目标用户不在线，消息将被丢弃。
        /// </summary>
        /// <param name="targetUserID">接收者的UserID</param>
        /// <param name="informationType">消息类型</param>
        /// <param name="message">消息内容</param>
        /// <param name="tag">附加内容</param>
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag);
        /// <summary>
        /// 向在线用户发送消息。如果目标用户不在线，消息将被丢弃。
        /// </summary>
        /// <param name="targetUserID">接收者的UserID</param>
        /// <param name="informationType">消息类型</param>
        /// <param name="message">消息内容</param>
        /// <param name="tag">附加内容</param>
        /// <param name="fragmentSize">消息将被分块发送，分块的大小</param>
```

```
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize);
    }
```

比如 客户端通过 IRapidPassiveEngine 的 SendMessage 方法发送消息给服务端 则服务端的 IRapidServerEngine 将触发 MessageReceived 事件。预定 MessageReceived 事件，我们就可以处理接收到的消息。

在实际使用时，究竟该使用哪种机制比较好了？

我们的建议是，以 Customize 机制为主，以引擎直接收发消息机制为辅助。在 Customize 机制中，通过实现处理器接口，我们可以将所有的消息处理集中到一个 Handler 类中；而如果使用引擎直接收发消息的机制，则在很多地方都可以预定 MessageReceived 事件，使得消息处理的代码分散在不同的地方。

（02）—— 在线用户管理、基础功能及状态通知

本文介绍 ESFramework 开发手册（00） —— 概述一文中提到的四大武器的第二个：在线用户管理、基础功能及状态通知。

在解决了发送信息和处理信息之后，还有一些基础功能是很多分布式通信系统都需要用到的，比如，查询某个用户是否在线、获取在线用户列表、自己掉线时得到通知，等等。ESPlus.Application.Basic 命名空间下的组件，为我们解决了这些基础问题。

一. 客户端

客户端通过调用 ESPlus.Application.Basic.Passive.IBasicOutter 接口对应的方法以及预定其相关的事件，就可以完成基础功能或得到相关状态改变通知。

我们可以从 ESPlus.Rapid.IRapidPassiveEngine 暴露的 BasicOutter 属性来获取 IBasicOutter 引用。

API：

```
    // 摘要:
    //     用于客户端向服务器发送Basic信息，并发布与自己或好友状态相关的事件。
    public interface IBasicOutter
    {
        // 摘要:
        //     当自己被服务端踢出掉线时，触发此事件。此时，客户端引擎已被Dispose。
        event CbGeneric BeingKickedOut;
        //
        // 摘要:
        //     当自己被同名用户挤掉线时，触发此事件。此时，客户端引擎已被Dispose。
        event CbGeneric BeingPushedOut;

        // 摘要:
        //     获取当前AS上的所有在线的用户列表。【该方法仅仅用于demo和测试】
        List<string> GetAllOnlineUsers();
        //
        // 摘要:
        //     获取自己的IPE。(通常是经过NAT之后的IPE)
        IPEndPoint GetMyIPE();
        //
```

```
        // 摘要:
        //     获取某个在线用户的IPE。如果用户不在线，则返回null。【适用于ESPlatform平台】
        //
        // 返回结果:
        //     通常是经过NAT之后的IPE
        IPEndPoint GetUserIPE(string userID);
        //
        // 摘要:
        //     查询用户是否在线。【适用于ESPlatform平台】
        Dictionary<string, bool> IsUserOnline(List<string> userIDList);
        //
        // 摘要:
        //     查询用户是否在线。【适用于ESPlatform平台】
        bool IsUserOnline(string userID);
        //
        // 摘要:
        //     命令服务端将目标用户踢出。如果目标用户不在当前AS上，则直接返回。
        //
        // 参数:
        //   targetUserID:
        //     要踢出的用户ID
        void KickOut(string targetUserID);
        //
        // 摘要:
        //     ping服务器。在应用层模拟ping，比普通的ICMP的ping大一些（如8-10ms）。
        //
        // 返回结果:
        //     ping耗时，单位毫秒
        int Ping();
        //
        // 摘要:
        //     ping其他在线用户（通过P2P通道，回复仍是由服务器中转）。 如果P2P通道不存在，则抛出通道不存在的异常。如果目标用户不在线，将抛出Timeout异常。
        //
        // 参数:
        //   targetUserID:
        //     要Ping的目标用户ID
        //
        // 返回结果:
        //     ping耗时，单位毫秒
        int PingByP2PChannel(string targetUserID);
        //
        // 摘要:
        //     ping其他在线用户(通过服务器中转)。 如果目标用户不在线，将抛出Timeout异常。
        //
        // 参数:
        //   targetUserID:
        //     要Ping的目标用户ID
        //
```

```
        // 返回结果:
        //     ping耗时，单位毫秒
        int PingByServer(string targetUserID);
        //
        // 摘要:
        //     向服务器发送心跳消息。被框架ESPlus.Application.Basic.Passive.HeartBeater使用。
        void SendHeartBeatMessage();
    }
```

1. 状态改变事件通知

首先，我们看看 IBasicOutter 暴露的两个事件：

（1）BeingKickedOut 当自己被踢出时将触发该事件。

（2）BeingPushedOut 发生于当服务端将重登陆模式设置为 ReplaceOld 时，并且同名用户的成功登录，将会把老的在线用户挤掉而导致其下线。

关于重登陆模式的更多内容可以参见重登陆模式。

2. 基础 API

接下来，我们简单看看 IBasicOutter 的几个方法。

（1）GetAllOnlineUsers 用于获取所有在线用户，通常该方法仅仅用于 demo，因为在正式的系统中，在线用户数可能是非常巨大的，这将导致 GetAllOnlineUsers 的返回消息非常大，甚至可能超过框架的最大消息尺寸的限制。

（2）Ping 系列方法，用于获取当前客户端到服务端或到另一个在线客户端的消息来回的耗时，由于其是在应用层来模拟类似 ICMP 的 ping，所以这个方法返回的值通常比 ICMP 的 ping 大一些。尽管如此，在一些应用中，该 Ping 的结果还是有一些参考价值的。

（3）有时，我们需要命令服务器将一些恶意的用户从服务端踢出（断开其连接），那么可以调用 KickOut 方法，被踢出的客户端将会触发上述的 BeingKickedOut 事件。

（4）SendHeartBeatMessage 方法用于向服务器发送心跳消息。如果我们使用的是 Rapid 引擎，那么框架会自动发送心跳消息，所以，我们通常不需要手动调用该方法。关于心跳消息的更多内容可以参见心跳机制。

3. TCP 连接状态

Basic 空间提供了一部分基础功能，还有另一部分很重要的基础功能需要涉及到客户端的 Rapid 引擎，我们在这里也一并介绍一下。客户端如何知道自己与服务器的 TCP 连接的状态及其变化了？ESPlus.Rapid.IRapidPassiveEngine 的几个事件和属性来获取这些信息。

API：

```
    /// <summary>
    /// 当客户端与服务器的TCP连接断开时，将触发此事件。
    /// </summary>
    event CbGeneric ConnectionInterrupted;

    /// <summary>
    /// 自动重连开始时，触发此事件。如果重连成功则将重新登录，并触发RelogonCompleted事件。
    /// </summary>
    event CbGeneric ConnectionRebuildStart;

    /// <summary>
    /// 当断线重连成功时，会自动登录服务器验证用户账号密码，并触发此事件。如果验证失败，则与服务器的连接将会断开，且后续不会再自动重连。事件参数表明了登录验证的结果。
```

```
  /// </summary>
  event CbGeneric<LogonResponse> RelogonCompleted;

  /// <summary>
  /// 当前引擎所连接的服务器的地址。
  /// </summary>
  AgileIPE ServerAddress { get; }

  /// <summary>
  /// 当前是否处于连接状态。
  /// </summary>
  bool Connected { get; }

  /// <summary>
  /// 与服务器之间的通道是否处于繁忙状态？
  /// </summary>
  bool ChannelIsBusy { get; }
```

注释已经很好的说明了每个事件和属性的用途，这里就不赘述了。

4. 建立 TCP 连接与登录

在 ESFramework 体系中，与服务端建立 TCP 连接，并进行帐号登录是在 IRapidPassiveEngine 的 Initialize 方法中完成的：

API：

```
  //
  // 摘要:
  //   完成客户端引擎的初始化，与服务器建立TCP连接，连接成功后立即验证用户密码。如果连接失败，则抛出异常。
  //
  // 参数:
  //  userID:
  //   当前登录的用户ID，由数字和字母组成，最大长度为10
  //
  //  logonPassword:
  //   用户登陆密码。
  //
  //  serverIP:
  //   服务器的IP地址。
  //
  //  serverPort:
  //   服务器的端口。
  //
  //  customizeHandler:
  //   自定义处理器，用于处理服务器或其它用户发送过来的消息
  LogonResponse Initialize(string userID, string logonPassword, string serverIP, int serverPort, ICustomizeHandler customizeHandler);
```

值得一提的是，RelogonCompleted 事件。当网络恢复 TCP 重连成功时，将自动登录服务器并重新验证用户账号和密码，然后触发 RelogonCompleted 事件。RelogonCompleted 事件的参数为 LogonResult，表明了重新登录验证的结果。而且，如果验证失败，与服务器的连接将会再次断开，且后续不会再自动重连。所以，当开发人员在进行

二次开发时，一定要注意 RelogonCompleted 事件的参数的值来作为重连成功/失败的依据。

二.服务端

1. 基础控制

Basic 的服务端就相当简单了。首先，我们可以通过 ESPlus.Application.Basic.Server.IBasicController 的 KickOut 方法来在服务端进行踢人操作。

我们可以从 ESPlus.Rapid.IRapidServerEngine 暴露的 BasicController 属性来获取 IBasicController 引用。

2. 登录验证

刚刚我们提到客户端可以调用 IBasicOutter 的 Logon 方法进行登陆验证 那么这个验证服务端是在哪里做的了？服务端正是通过 ESPlus.Application.Basic.Server.IBasicHandler 的 VerifyUser 方法来验证用户账号密码的。

API：

```
public interface IBasicHandler
{
    // 摘要:
    //     客户端登陆验证。
    //
    // 参数:
    //   userID:
    //     登陆用户账号
    //
    //   systemToken:
    //     系统标志。用于验证客户端是否与服务端属于同一系统。
    //
    //   password:
    //     登陆密码
    //
    //   failureCause:
    //     如果登录失败，该out参数指明失败的原因
    //
    // 返回结果:
    //     如果密码和系统标志都正确则返回true；否则返回false。
    bool VerifyUser(string systemToken, string userID, string password, out string failureCause);
}
```

请注意，如果账号密码验证不通过，可以通过 failureCause 参数返回不通过的原因。failureCause 的值将被传递并赋值给 Logon 方法返回的 LogonResponse 的 FailureCause 属性。

同上一章讲到的 ICustomizeHandler 一样，我们要在系统中根据项目的具体需求来实现 IBasicHandler 接口并将其注入到框架中。

3. 在线用户管理器

服务端如何知道用户上下线、以及每个在线用户的状态了？

只要通过 ESFramework.Server.UserManagement.IUserManager 的相关事件和方法就能得到这些信息。我们可以从 ESPlus.Rapid.IRapidServerEngine 暴露的 UserManager 属性来获取 IUserManager 引用。IUserManager 接口定义如下：

API：

```
    // 摘要:
    //     在线用户管理器接口。注意，IUserManager仅仅管理当前AS上的所有在线用户，所以其暴露的所有事件、属性、方法都是针对当前AS上的用户的。
    //     如果要获取ESPlatform全局在线用户的信息，需要访问IPlatformUserManager接口。
    public interface IUserManager
    {
        // 摘要:
        //     重登陆模式。
        RelogonMode RelogonMode { get; set; }
        //
        // 摘要:
        //     当前在线用户的数量。
        int UserCount { get; }
        //
        // 摘要:
        //     用户管理器依赖该属性显示所有在线用户的状态信息。
        IUserDisplayer UserDisplayer { set; }

        // 摘要:
        //     如果RelogonMode为IgnoreNew，并且当从一个新连接上收到一个同名ID用户的消息时将触发此事件。 注意，只有在该事件处理完毕后，才会关闭新连接。可以在该事件的处理函数中，将相关情况通知给客户端。
        //     【一般情况下，都会由登录回复告知客户端已经登录，而不会进入到这里触发该事件！】
        event CbGeneric<string, IPEndPoint> NewConnectionIgnored;
        //
        // 摘要:
        //     当某个用户的P2PAddress地址被修改时，将触发此事件。参数为 UserID - P2PAddress
        event CbGeneric<string, P2PAddress> P2PAddressChanged;
        //
        // 摘要:
        //     如果RelogonMode为ReplaceOld，并且当从另外一个新连接上收到一个同名ID用户的消息时将触发此事件。 注意，只有在该事件处理完毕后，才会真正关闭旧的连接并使用新的地址取代旧的地址。可以在该事件的处理函数中，将相关情况通知给旧连接的客户端。
        event CbGeneric<UserData> SomeOneBeingPushedOut;
        //
        // 摘要:
        //     当客户端登录成功时，触发此事件。不要远程预定该事件。
        event CbGeneric<UserData> SomeOneConnected;
        //
        // 摘要:
        //     客户端连接断开下线时，触发此事件。不要远程预定该事件。
        event CbGeneric<UserData, DisconnectedType> SomeOneDisconnected;
        //
        // 摘要:
        //     用户心跳超时。 只有在该事件处理完毕后，才关闭对应的连接，并将其从用户列表中删除。可以在该事件的处理函数中，将相关情况通知给客户端。
        event CbGeneric<UserData> SomeOneTimeOuted;
        //
        // 摘要:
        //     当在线用户数发生变化时，触发此事件。
```

```
        event CbGeneric<int> UserCountChanged;
        //
        // 摘要:
        //     当某个用户的携带数据被修改时，将触发此事件。参数为 UserID - tag
        event CbGeneric<string, object> UserTagChanged;

        // 摘要:
        //     获取所有在线用户信息。
        List<UserData> GetAllUserData();
        //
        // 摘要:
        //     获取在线用户的ID列表。
        List<string> GetOnlineUserList();
        //
        // 摘要:
        //     如果用户不在线，返回null
        IPEndPoint GetUserAddress(string userID);
        //
        // 摘要:
        //     获取目标在线用户的基础信息。
        //
        // 参数:
        //   userID:
        //     目标用户的ID
        //
        // 返回结果:
        //     如果目标用户不在线，则返回null
        UserData GetUserData(string userID);
        //
        // 摘要:
        //     初始化用户管理器。
        void Initialize();
        //
        // 摘要:
        //     目标用户是否在线。
        bool IsUserOnLine(string userID);
        //
        // 摘要:
        //     该方法用于Platform，用于接受平台的回调通知。当RelogonMode为ReplaceOld时，同名用户在同一群集的其它的应用服务器上登陆。将触发SomeOneBeingPushedOut事件。
        //
        // 参数:
        //   userID:
        //     同名登陆的用户ID。
        //
        //   newServerID:
        //     新登录到哪台服务器。
        void LogonOtherServer(string userID, string newServerID);
        //
```

```
        // 摘要:
        //     从目标用户集合中挑出在线用户的ID列表。
        List<string> SelectOnlineUserFrom(IEnumerable<string> users);
        //
        // 摘要:
        //     记录客户端上报的P2P地址。如果目标用户不在线，则忽略。
        //
        // 参数:
        //   userID:
        //     客户端用户的ID
        //
        //   addr:
        //     P2P地址
        void SetP2PAddress(string userID, P2PAddress addr);
    }
```

RelogonMode 属性用于设置重登录模式，关于重登陆模式的更多内容可以参见重登陆模式。

三.UserID 的长度

ESFramework 的 Rapid 引擎使用的消息头的默认长度是 36 字节，允许的 UserID 最大长度为 11 字节。但是，如果你的系统中需要用到的 UserID 长度超过了 11 字节，该怎么办了？我们可以通过调用 GlobalUtil 静态类的 SetMaxLengthOfUserID 静态方法来设定 ESFramework 允许的 UserID 的最大长度：

API：

```
  //
  // 摘要:
  //     设置UserID（包括GroupID）的最大长度(不能超过255)。必须在Rapid引擎初始化之前设置才有效。注意，客户端与服务端要统一设置。
  public static void SetMaxLengthOfUserID(byte maxLen);
```

注意，我们必须在 Rapid 引擎的 Initialize 方法执行之前调用 SetMaxLengthOfUserID 方法。而且，客户端和服务端必须采用相同的设置，否则，就一定会导致服务端和客户端通信出现异常。如果你的客户端是使用的 Silverlight，那么使用 ESFramework.SL 时也是如此。

（1）服务端和桌面客户端请调用 ESPlus.GlobalUtil 的 SetMaxLengthOfUserID 方法进行设置。

（2）Silverlight 的客户端请调用 ESFramework.SL.GlobalUtil 的 SetMaxLengthOfUserID 方法进行设置。

（3）在 ESFramework 内部，组（Group）ID 也采用与 UserID 相同的规则。

（4）在满足项目需求的情况下，尽可能使 UserID 的最大长度短一点，这样可以使得消息头更加短小，从而避免浪费本不需要的带宽。

四.消息的最大长度

Rapid 引擎内部默认设置的消息的最大长度为 1M（1024*1024），并且这个长度还包含了上述消息头的长度。如果您的应用需要发送的单个信息的长度超过了 1M，就会被 ESFramework 认为是恶意的消息，ESFramework 会丢弃该消息并关闭对应的连接。

我们建议：在能同样满足项目的需求下，应该尽可能地使传送的消息小，这样不仅可以节省带宽，而且还有助于提升并发的性能。如果应用中确实有信息的长度超过最大限制，那么还可以通过 ICustomizeOutter 的 SendBlob 方法来将其作为大数据块进行发送。我们可以通过调用 GlobalUtil 静态类的 SetMaxLengthOfMessage 静态方法来设定 ESFramework 允许的最大消息长度。

API：

```
  //
  // 摘要:
  //      设置消息的最大长度,初始值为1M。必须在Rapid引擎初始化之前设置才有效。注意，客户端与服务端要统一设置。
  public static void SetMaxLengthOfMessage(int maxLen);
```

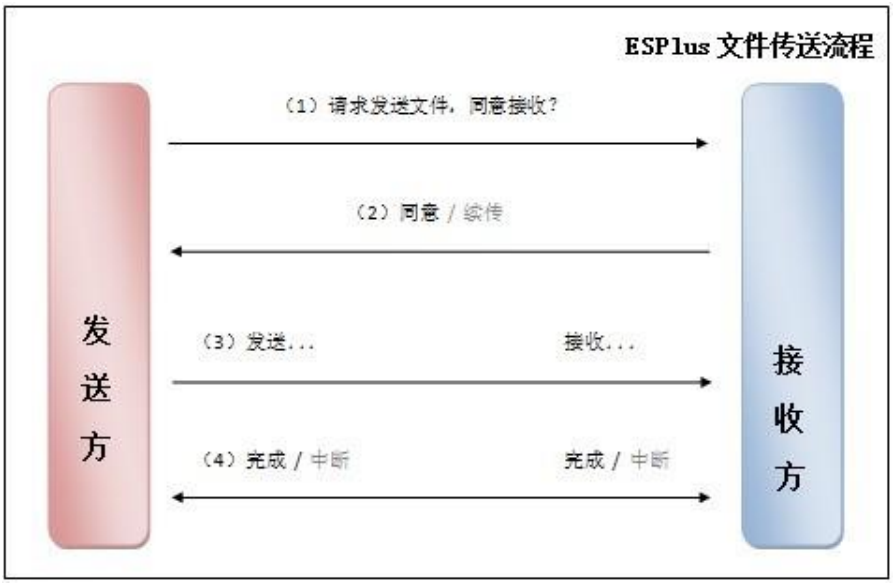
（03）—— 文件（夹）传送

本文介绍 ESFramework 开发手册（00） —— 概述一文中提到的四大武器中的第三个：文件传送。

在很多分布式系统中，都有文件传送的需求。ESPlus 内置了文件传送（支持自动断点续传）的功能，通过 ESPlus.Application.FileTransfering 命名空间提供相关服务。最新的 ESPlus 不仅支持传送单个文件，还支持传送整个文件夹，而且，传送文件夹与传送文件采用完全相同的 API 和模型。

一.ESPlus 的文件传送流程

ESPlus 定义了文件传送的标准流程，可以用下图表示：



（1） 由发送方发起传送文件的请求。

（2） 接收方回复同意或者拒绝接收文件。如果拒收，则流程结束；否则进入下一步。

（3） 发送方发送文件数据，接收方接收文件数据。

（4） 如果文件传送过程中，接收方或发送方掉线或者取消文件传送，则文件传送被中断，流程结束。如果文件传送过程一直正常，则到最后完成文件的传送。

有几点需要说明一下：

（1） 发送方可以是客户端，也可以是服务器；接收方也是如此。但无论发送方和接收方的类别如何，它们都遵守这一文件传送流程。

（2） 当接收方同意接收后，框架会自动搜索是否存在匹配的续传项目，若存在，则会启动断点续传。

（3） 进行文件传送的线程是由框架自动控制的，只要发送方收到了接收方同意接收的回复，框架就会自动在后台线程中发送文件数据包；而接收方也会自动处理接收到的文件数据包。

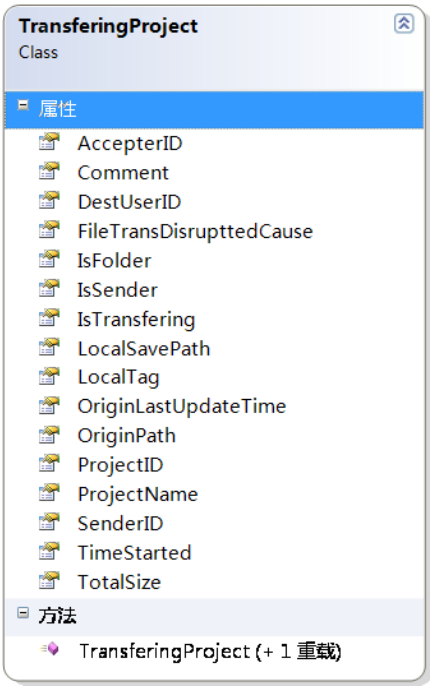
（4） 发送方或接收方都可随时取消正在传送的文件。

（5） 当文件传送被中断或完成时，发送方和接收方都会有相应的事件通知。

二.用于支持文件传送的基础设施

1. TransferingProject

无论发送方还是接收方，每个文件传送任务，都有一个对象来表示它，ESPlus.FileTransceiver.TransferingProject 便是一个文件传送项目的封装，里面包含了类似发送者 ID、接收者 ID、文件名称、是否已经开始传送，等等相关信息。



TransferingProject 的大部分属性对于发送方和接收方都是有效的，而有几个属性只对发送方有效（比如 SendingFileParas），有几个属性只对接收方有效（如 LocalSaveFilePath），这些在帮助文档中都有详细的说明。而且，有些属性（如 OriginFileLastUpdateTime）的存在是用于支持断点续传功能的。

2. FileTransDisrupttedType

ESPlus 使用 FileTransDisrupttedType 枚举定义了所有可能导致文件传送中断的原因：

API：

```
// 摘要:
    //     文件传输中断的原因。
    public enum FileTransDisrupttedType
    {
        // 摘要:
        //     接收方拒绝接收
        RejectAccepting = 0,
        //
        // 摘要:
        //     自己主动取消
        ActiveCancel = 1,
        //
        // 摘要:
        //     对方取消
        DestCancel = 2,
        //
        // 摘要:
        //     对方掉线
```

```
        DestOffline = 3,
        //
        // 摘要:
        //     与对方的可靠的P2P通道关闭
        ReliableP2PChannelClosed = 4,
        //
        // 摘要:
        //     网络中断、自己掉线
        SelfOffline = 5,
        //
        // 摘要:
        //     自己系统内部错误，如文件读取失败等
        InnerError = 6,
        //
        // 摘要:
        //     对方系统内部错误，如文件读取失败等
        DestInnerError = 7,
        //
        // 摘要:
        //     网速太慢。
        NetworkSpeedSlow = 8,
        //
        // 摘要:
        //     【发送方】读取文件卡住。
        ReadFileBlocked = 9,
        //
        // 摘要:
        //     发送线程没有启动。
        SendThreadNotStarted = 10,
        //
        // 摘要:
        //     【接收方】在一定的时间内，针对传送的目标文件，第一个文件块都尚未收到。 真正原因可能是：
(1) 网络太慢；(2) 发送方ReadFileBlocked
        //     或 SendThreadNotStarted。
        Timeout4FirstPackage = 11,
    }
```

当文件传送通过 P2P 通道进行时，如果 P2P 通道意外中断，则文件传送也会中断，其中断的原因就是枚举中的 ReliableP2PChannelClosed。

3. IFileTransferingEvents 接口

ESPlus 定义了 IFileTransferingEvents 接口，用于暴露所有与文件传送相关的状态和事件：

API：

```
// 摘要:
    //     与文件传送相关的事件。
    public interface IFileTransferingEvents
    {
        // 摘要:
        //     当某个文件续传开始时，触发该事件。（将不再触发FileTransStarted事件）
```

```
        event CbGeneric<TransferingProject> FileResumedTransStarted;
        //
        // 摘要:
        //     文件传送完成时，触发该事件。
        event CbGeneric<TransferingProject> FileTransCompleted;
        //
        // 摘要:
        //     文件传送中断时，触发该事件。
        event CbGeneric<TransferingProject, FileTransDisruptedType> FileTransDisrupted;
        //
        // 摘要:
        //     文件传送的进度。参数为projectID（文件传送项目的编号），total（文件大小），transfered
（已传送字节数）
        event CbFileSendedProgress FileTransProgress;
        //
        // 摘要:
        //     当某个文件开始传送时，触发该事件。
        event CbGeneric<TransferingProject> FileTransStarted;
    }
```

通过预定这些事件，我们可以知道每个传送的文件什么时候开始（或断点续传）、什么时候完成、传递的实时进度、传送中断的原因等等。要注意的是，这些事件都是在后台线程中触发的，如果在事件处理函数中需要更新 UI，则需要将调用转发到 UI 线程。

4. SendingFileParas

该对象仅仅包含两个属性：SendingSpanInMSecs 和 FilePackageSize。发送方可以通过 SendingFileParas 对象来指定发送文件数据包时的频率与每个数据包的大小。一般来说，为了达到最快的传送速度，SendingSpanInMSecs 可以设为 0。而 FilePackageSize 的大小则要根据发送方与接收方的网络环境的好坏进行决定，在 Internet 上，一般可以设为 2048 或 4096 左右 ;而在局网内 ,可以设为 204800 甚至更大(在局网的传送速度可以达到 30M/s 以上)。

5. IBaseFileController

通过 ESPlus.Application.FileTransfering.IBaseFileController 接口，我们可以提交发送文件的请求，并且可以主动取消正在接收或发送的文件。IBaseFileController 即可用于客户端也可用户服务端。

API：

```
  // 摘要:
    //     用于控制文件（夹）发送及文件传送状态的查看。即可用于客户端也可用户服务端。
    public interface IBaseFileController
    {
        // 摘要:
        //     该事件接口暴露了所有正在接收的文件（夹）的实时状态。
        IFileTransferingEvents FileReceivingEvents { get; }
        //
        // 摘要:
        //     该事件接口暴露了所有正在发送文件（夹）的实时状态。
        IFileTransferingEvents FileSendingEvents { get; }
        //
        // 摘要:
        //     当一个续传项超过多长时间没被使用，则将其移除，同时删除对应的临时文件（夹）。单位：秒。
```

```
默认值：300。 如果值小于等于0，表示不使用续传功能。
        int TTL4ResumedFileItem { get; set; }

        // 摘要:
        //     当文件接收方收到了来自发送方发送文件（夹）的请求时，触发此事件。该事件将在后台线程中触发，如果处理该事件时需要刷新UI，则需要转发到UI线程。 当接收方确定要接收或拒绝文件时，请调用BeginReceiveFile方法或RejectFile方法。
        event CbFileRequestReceived FileRequestReceived;
        //
        // 摘要:
        //     当文件接收方回复了同意/拒绝接收文件（夹）时，在发送方触发此事件。参数为TransmittingProject - bool（同意？）。可以通过参数TransmittingProject的AccepterID属性得知接收方的UserID。
        //     通常，客户端预定该事件，只需要告知文件发送者，而不需要再做任何额外处理。该事件将在后台线程中触发，如果处理该事件时需要刷新UI，则需要转发到UI线程。
        event CbGeneric<TransferingProject, bool> FileResponseReceived;

        // 摘要:
        //     接收方如果同意接收文件（夹），则调用该方法。
        //
        // 参数:
        //   projectID:
        //     文件传送项目的编号
        //
        //   savePath:
        //     存储文件（夹）的路径。请特别注意，如果已经存在同名的文件（夹），将覆盖之。
        void BeginReceiveFile(string projectID, string savePath);
        //
        // 摘要:
        //     接收方如果同意接收文件（夹），则调用该方法。
        //
        // 参数:
        //   projectID:
        //     文件传送项目的编号
        //
        //   savePath:
        //     存储文件（夹）的路径。请特别注意，如果已经存在同名的文件（夹），将覆盖之。
        //
        //   allowResume:
        //     如果满足续传条件，是否启用续传。
        void BeginReceiveFile(string projectID, string savePath, bool allowResume);
        //
        // 摘要:
        //     发送方准备发送文件（夹）。目标用户必须在线。如果对方同意接收，则后台会自动发送文件（夹）；如果对方拒绝接收，则会取消发送。（通过FileAnswerReceived事件，可以得知对方是否同意接收。）
        //
        // 参数:
        //   accepterID:
        //     接收文件（夹）的用户ID
```

```
        //
        //   fileOrDirPath:
        //     被发送文件（夹）的路径
        //
        //   comment:
        //     其它附加备注。如果是在类似FTP的服务中，该参数可以是保存文件（夹）的路径
        //
        //   projectID:
        //     返回文件传送项目的编号
        void BeginSendFile(string accepterID, string fileOrDirPath, string comment, out string
projectID);
        //
        // 摘要:
        //     发送方准备发送文件（夹）。目标用户必须在线。如果对方同意接收，则后台会自动发送文件；
如果对方拒绝接收，则会取消发送。（通过FileResponseReceived事件，可以得知对方是否同意接收。）
        //
        // 参数:
        //   accepterID:
        //     接收文件（夹）的用户ID
        //
        //   fileOrDirPath:
        //     被发送文件（夹）的路径
        //
        //   comment:
        //     其它附加备注。如果是在类似FTP的服务中，该参数可以是保存文件（夹）的路径
        //
        //   paras:
        //     发送参数设定。传入null，表示采用IFileSenderManager的默认设置。
        //
        //   projectID:
        //     返回文件传送项目的编号
        void BeginSendFile(string accepterID, string fileOrDirPath, string comment, SendingFileParas
paras, out string projectID);
        //
        // 摘要:
        //     取消所有正在传送项目（包括未被接收方回复的传送项目）。
        void CancelAllTransfering();
        //
        // 摘要:
        //     主动取消正在发送或接收的文件（夹）（包括未被接收方回复的传送项目），并通知对方。
        void CancelTransfering(string projectID);
        //
        // 摘要:
        //     主动取消正在发送或接收的文件（夹）（包括未被接收方回复的传送项目），并通知对方。
        void CancelTransfering(string projectID, string cause);
        //
        // 摘要:
        //     取消与目标用户相关的正在传送项目（包括未被接收方回复的传送项目）。
        //
```

```
        // 参数:
        //   destUserID:
        //     目标用户ID。如果为null，则表示取消所有正在传送项目。
        void CancelTransferingAbout(string destUserID);
        //
        // 摘要:
        //     获取与目标用户相关的所有文件传送项目的projectID的列表(包括未被接收方回复的传送项目)。
        //
        // 参数:
        //   destUserID:
        //     目标用户ID。如果为null，则表示获取所有正在传送项目的projectID。
        //
        // 返回结果:
        //     projectID的列表
        List<string> GetTransferingAbout(string destUserID);
        //
        // 摘要:
        //     获取正在发送或接收中的文件传送项目的传送进度。如果不存在目标项目或传送尚未开始，则返回null。
        FileTransferingProgress GetTransferingProgress(string projectID);
        //
        // 摘要:
        //     获取正在发送或接收中的文件传送项目（包括未被接收方回复的传送项目）。如果不存在目标项目，则返回null。
        TransferingProject GetTransferingProject(string projectID);
        //
        // 摘要:
        //     接收方如果拒绝接收文件（夹），则调用该方法。
        //
        // 参数:
        //   projectID:
        //     文件传送项目的编号
        void RejectFile(string projectID);
        //
        // 摘要:
        //     接收方如果拒绝接收文件（夹），则调用该方法。
        //
        // 参数:
        //   projectID:
        //     文件传送项目的编号
        //
        //   cause:
        //     拒绝接收的原因
        void RejectFile(string projectID, string cause);
    }
```

请求传送

1．BeginSendFile 用于向接收方提交发送文件的请求，如果对方同意，则后台会自动开始传递文件。该方法有个 out 参数 projectID，用于传出标记该文件传送项目的唯一编号，比如，你打算将同一个文件发送给两个好友，将

会调用两次 BeginSendFile 方法，而两次得到的 projectID 是不一样的。也就是说，projectID 是用于标记文件传送项目的，而不是标记文件的。该方法有两个重载，区别在于第二个 BeginSendFile 方法多了一个 SendingFileParas 参数，用于主动控制文件数据包的大小和发送频率。

在客户端使用时，BeginSendFile 方法不仅可以向其他在线用户提交发送文件的请求，也可以直接向服务器提交发送文件的请求 —— 即此时文件的接收者为服务端。我们只需要将 accepterID 参数传入 NetServer.SystemUserID，以指明由服务端而不是其他用户来接收即将发送的文件。

2．当发送方调用了 BeginSendFile 方法后，接收方会触发 FileRequestReceived 事件，事件参数包含了与此次文件传送相关的详细信息，请特别注意其 ResumedProjectItem 参数，若该参数的值不为 null，则表示发现了**续传项目**，可以启用续传，而接下来是要续传还是重新传送，取决于接收方调用 BeginReceiveFile 方法时传入的 allowResume 参数的值。

3．另外，由于 FileRequestReceived 事件是在后台线程中被框架调用的，如果处理该事件的方法中需要刷新应用程序的 UI，则注意一定要转发到 UI 线程。

同意/拒绝接收

1．如果接收方同意接收文件，则应该调用 BeginReceiveFile 方法；否则，调用 RejectFile 方法。注意，有可能在调用 BeginReceiveFile 方法或 RejectFile 方法之前，发送方已经取消了文件的发送（此时，会在接收方会触发 FileReceivingEvents 属性的 FileTransDisruptted 事件）。

2.无论接收方是调用 BeginReceiveFile 方法还是调用 RejectFile 方法，都会在发送方触发 FileResponseReceived 事件，事件的第二个 bool 参数表明了对方是同意还是拒绝接收文件；还可以通过事件的第一个参数 TransferingProject 的 AccepterID 属性得知接收方的 UserID。应用程序在处理 FileResponseReceived 事件时，最多只需要告知文件发送者，而不需要再做任何其它的额外处理，因为框架已经帮你打理好了一切。

3．当接收方同意接收文件后，与该文件传送项目相关的事件会通过 IFileTransferingEvents 接口（FileSendingEvents 和 FileReceivingEvents 属性）相继触发。

4.当接收方调用 RejectFile 方法拒绝接收文件时，发送方会触发 FileSendingEvents 的 FileTransDisruptted 事件，接收方会自己也会触发 FileReceivingEvents 的 FileTransDisruptted 事件。且触发的这两个事件的第二个参数的值都是 FileTransDisrupttedType.RejectAccepting。

获取传送状态

1．GetTransferingProject 方法可以获取任何一个正在发送或正在接收的项目信息，该方法也可获取一个还未被接收方回复的文件传送项目的信息。通过 TransferingProject 的 IsTransfering 属性，我们可以间接地知道接收方对发送请求是否给出了回复。如果 IsTransfering 为 false，则表示接收方还未给出回复，文件传送还未开始；反之亦然。

2．GetTransferingAbout 方法可以获取与目标用户相关的所有文件传送项目的 projectID 的列表，其中还包括那些还未被接收方回复的文件传送项目。

3．FileSendingEvents 属性用于暴露自己作为发送者的所有正在进行的文件传送项目的实时状态；而 FileReceivingEvents 属性用于暴露自己作为接收者的所有正在进行的文件传送项目的实时状态。

取消传送项目

1．CancelTransfering 方法用于取消正在发送或接收的某个文件传送项目，该方法还可以取消已经请求发送但还未收到接收方回复的文件传送项目。调用该方法时，框架会自动通知文件传送的另一端用户，并触发 FileReceivingEvents 或 FileSendingEvents 中的 FileTransDisruptted 事件，而另一端也会自动触发 FileTransDisruptted 事件。

2．CancelTransferingAbout 方法用于取消与某个指定用户相关的正在传送项目（包括已经请求发送但还未收到接收方回复的文件传送项目）。比如，我们正在与 aa01 用户聊天，并且与 aa01 有多个文件正在传送，此时，如

果要关闭与 aa01 的聊天窗口，那么关闭之前，通常会先调用 CancelTransferingAbout 方法来取消与 aa01 相关的所有文件传送。所以你经常会看到类似的提示：“您与 aa01 有文件正在传送中，关闭当前窗口将导致正在传送的文件中断，您确定要关闭吗？”。如果用户确认关闭，此时就正是我们要调用 CancelTransferingAbout 方法的时候了。

3．要特别注意一点，对于那些已经发送文件请求但还未收到接收方回复的文件传送项目，其与正在传送的文件项目采用的是相同的处理模式。我们可以通过 TransferingProject 的 IsTransfering 属性来区分这两种类型。

三.断点续传

1. 断点续传是由接收方来管理的。

2. 如果之前用户 A 发送给用户 B 的某个文件传送到一半时中断了（可能是因为主动取消、或网络断开等），那么，现在 A 又发送文件给 B，当 B 端程序检测到以下条件满足时，则会启动续传：

（1）A 发送的是同一个文件。即被发送文件的绝对路径、文件大小、文件的最后修改时间， 这三个要素完全一致。

（2）本次发送文件的请求距离上次中断的时间间隔不超过 5 分钟。

（3）在此期间，接收方 B 的程序没有重启过，也没有调用过 Rapid 引擎的 Initialize 方法。

（4）B 方上次接收中断对应的临时文件（扩展名.tmpe$）没有被手动或其它程序删除。

（5）B 同意接收文件，且存放接收文件的路径与上次选择的路径完全一致。

3. 当上面的前（1）（2）（3）（4）满足时，B 端触发的 FileRequestReceived 事件的 ResumedProjectItem 参数值就不为 null，表示可以开启续传。

4. 即使达到了续传的条件，B 仍然可以要求重新传送整个文件，只要 B 在同意接收文件时，调用 BeginReceiveFile 方法传入的 allowResume 参数的值为 false 即可。

四.客户端 IFileOutter

同 ESPlus 的 Basic 应用或 CustomizeInfo 应用一样，在客户端支持文件传送功能需要使用到相应的“Outter”组件 IFileOutter。

客户端通过 ESPlus.Application.FileTransfering.Passive.IFileOutter 接口提供的方法来提交发送文件请求等操作。我们可以从 ESPlus.Rapid.IRapidPassiveEngine 暴露的 FileOutter 属性来获取 IFileOutter 引用。IFileOutter 接口直接从 IBaseFileController 继承，且未增加任何新的内容：

API：

```
// 摘要:
//     用于文件传送及其相关控制消息。
public interface IFileOutter : IBaseFileController
{
}
```

ESFramework.Boost（源码开放）提供了默认的传送项目的状态查看器控件 FileTransferingViewer，如果没有特殊需求，大家在项目中可以直接使用它来显示文件传送的实时状态，它的界面截图如下所示：

你只需要把这个控件拖拽到你的 UI 上，然后将 IFileOutter 传入 FileTransferingViewer 的 Initialize 方法，它就可以正常工作了。

FileTransferingViewer 的 Initialize 方法的第一个参数 friendUserID 表示当前的 FileTransferingViewer 控件要显示与哪个好友相关的所有文件传送项目的状态。以 QQ 作类比，你同时在与多个好友传送文件，那么就会有多个聊天窗口，每个聊天窗口都会有一个 FileTransferingViewer 实例，而这个 FileTransferingViewer 实例仅仅显示与当前聊天窗口对应的好友的传送项目。如此一来，你与 aa01 用户传送文件的进度查看器就不会在你与 aa02 的聊天窗口上显示出来。

如果你的 FileTransferingViewer 需要捕捉所有正在传送的项目的实时状态，那么，调用其 Initialize 方法时，friendUserID 参数传入 null 就可以了。另外，FileTransferingViewer 实现了 IFileTransferingViewer 接口：

API：

```
    // 摘要:
    //     文件传送查看器。用于查看与某个好友的所有文件传送的进度状态信息。 注意，
FileTransferingViewer的所有事件都是在UI线程中触发的。
    public interface IFileTransferingViewer
    {
        // 摘要:
        //     当前是否有文件正在传送中。
        bool IsFileTransfering { get; }

        // 摘要:
        //     当所有文件都传送完成时，触发该事件。
        event CbSimple AllTaskFinished;
        //
        // 摘要:
        //     当某个文件开始续传时，触发该事件。参数为FileName - isSending
        event CbGeneric<string, bool> FileResumedTransStarted;
        //
        // 摘要:
        //     当某个文件传送完毕时，触发该事件。参数为FileName - isSending - comment - isFolder
        event CbGeneric<string, bool, string, bool> FileTransCompleted;
        //
        // 摘要:
        //     当某个文件传送中断时，触发该事件。参数为FileName - isSending - FileTransDisrupttedType
        event CbGeneric<string, bool, FileTransDisrupttedType> FileTransDisrupted;
        //
        // 摘要:
        //     当某个文件传送开始时，触发该事件。参数为FileName - isSending
        event CbGeneric<string, bool> FileTransStarted;

        // 摘要:
        //     在该控件释放之前（比如在其宿主窗体的FormClosing事件中），一定要调用此方法！
        void BeforeDispose();
        //
        // 摘要:
        //     获取所有正在传送的项目的ID集合。
        List<string> GetTransferingProjectIDsInCurrentViewer();
        //
```

```
        // 摘要:
        //     当收到发送文件的请求时,接收方调用此方法显示fileTransferItem。
        //
        // 参数:
        //   projectID:
        //     传送项目的ID
        //
        //   offlineFile:
        //     是否为离线文件
        //
        //   doneAccepted:
        //     如果当前是接收方，是否已经同意接收了？（不显示“接收”、“拒绝”按钮？）
        void NewFileTransferItem(string projectID, bool offlineFile, bool doneAgreed);
    }
```

你也可以通过该接口来关注 FileTransferingViewer 查看器捕捉到的（正如前所述，不一定是全部）文件传送项目的状态，而且，该接口的事件都是在 UI 线程中触发的，你可以直接在其处理函数中操控 UI 显示。

五. 服务端 IFileController

服务端也可以接收客户端发送的文件（即上传），甚至可以发送文件给客户端（即下载），它遵循同样的文件传送流程。如果需要服务端也参与到文件的发送与接收中来，可以使用服务端的 IFileController 接口，直接从 IBaseFileController 接口继承，而且未增加任何新的内容。我们可以从 IRapidServerEngine 暴露的 FileController 属性来获取 IFileController 引用。

您可以通过查看文件传送 demo 以进一步了解如何使用相关的 API。

（04）—— 可靠的 P2P

本文介绍 ESFramework 开发手册（00）——概述一文中提到的四大武器中的最后一个：P2P 通道。

ESPlus 提供了基于 TCP 和 UDP 的 P2P 通道（不仅支持局域网，还支持广域网的 P2P 通信），而无论我们是使用基于 TCP 的 P2P 通道 还是使用基于 UDP 的 P2P 通道 ,ESPlus 保证所有的 P2P 通信都是可靠的。这是因为 ESPlus 在原始 UDP 的基础上模拟 TCP 的机制进行了再次封装，以使 UDP 像 TCP 一样可靠。在客户端之间需要高频通信的分布式系统中（如 IM 系统等），可靠的 P2P 通信将为您节省巨大的带宽和服务器成本。

一.P2P 打洞

了解 P2P 的朋友都知道，P2PChannel 的建立需要通过“打洞”来完成，而运行于两个 NAT 设备后面的 PC 上的客户端实例之间的 P2P 打洞能否成功，或者说，P2P 通道能否成功建立，取决于 NAT 设备的类型。

1.UDP 打洞

就目前我们常用的路由器、防火墙等 NAT 设备来说，大多都是 Cone 型（FullCone、RestrictedCone、PortRestrictedCone 之一）的，所以 UDP 打洞的成功率还是非常大的（70%以上）。

基于 UDP 的 P2P 打洞成功率与 NAT 设备的类型的关系一览表如下所示：

NAT 设备类型	Full Cone	RestrictedCone	PortRestrictedCone	Symmetric
Full Cone	Yes	Yes	Yes	Yes
RestrictedCone	Yes	Yes	Yes	Not Sure
PortRestrictedCone	Yes	Yes	Yes	Not Sure
Symmetric	Yes	Not Sure	Not Sure	No

从列表可以看出，最麻烦的是 Symmetric 类型，如果需要 P2P 通信的双方有一方是 Symmetric，那么打洞就需要用到端口预测技术，而且其打洞成功的希望非常渺茫。

大家可以通过从网上下载 NAT 类型的检测程序（比如 STUN）来检测自己路由器等 NAT 设备的类型，以此来确定通信的双方基于 UDP 的 P2P 通道是否可以创建成功。

2.TCP 打洞

TCP 打洞的原理几乎与 UDP 是一样的，但不幸的是，目前支持 TCP 打洞的 NAT 设备非常少，以至于位于两个 NAT 后面的客户端实例之间能成功建立基于 TCP 的 P2P 连接的机会就很小了。希望在不久的将来，支持 TCP 打洞的 NAT 设备会逐渐多起来，这需要时间。关于基于 TCP 的 P2P 的更多介绍可以参见这里。

即使如此，ESFramework 支持基于 TCP 的 P2P 还是非常必要的，因为在以下两种情况下，基于 TCP 的 P2P 通道肯定是可以成功创建的。

（1）通信的双方位于同一个局网内。

（2）通信的双方中至少有一方运行于具有公网 IP 的机器上。

在这两种情况下，都不需要 TCP 打洞，也不需要 NAT 设备的额外支持，基于 TCP 的 P2P 通道就可以成功建立。

二.P2P 通道的可靠性

由于我们的 P2P 通道可能是基于 TCP 的、也可能是基于 UDP 的，所以 P2P 通道就继承了协议的特性：基于 TCP 的 P2P 通道是可靠的、而基于 UDP 的 P2P 通道是不可靠的。

值得庆幸的是，ESFramework/ESPlus 内部使用了增强的 UDP——在 UDP 的基础上模拟 TCP 机制，以保证通信的可靠性。所以，ESPlus 提供的 P2P 通道都是可靠的。

三.通道选择

在介绍 CustomizeInfo 空间时，我们提到可以使用 ICustomizeOutter 基于 P2P 通道发送消息给另外一个在线的用户，该接口的如下几个方法都可能采用 P2P 通道发送消息：

API：

```
public interface ICustomizeOutter
{
    void Send(string targetUserID, int informationType, byte[] info);

    void SendCertainly(string targetUserID, int informationType, byte[] info);

    byte[] Query(string targetUserID, int informationType, byte[] info);

    /// <summary>
    /// 通过P2P通道（即使是不可靠的）向在线用户targetUserID发送信息。
```

```
        /// </summary>
        /// <param name="targetUserID">接收消息的目标用户ID</param>
        /// <param name="informationType">自定义信息类型</param>
        /// <param name="info">信息</param>
        /// <param name="actionType">当P2P通道又不存在时，采取的操作</param>
        void SendByP2PChannel(string targetUserID, int informationType, byte[] info,
ActionTypeOnNoP2PChannel actionType);
    }
```

除 SendByP2PChannel 方法外，其它的方法都将使用可靠的 P2P 通道来发送消息，如果通信双方的 P2P 通道没有创建成功，则消息将通过服务器中转。

SendByP2PChannel 方法是当与目标用户之间的 P2P 通道存在时，一定采用 P2P 通道发送消息。如果与目标用户之间的 P2P 通道不存在，那么将采取的操作取决于该方法的第 4 个参数 ActionTypeOnNoP2PChannel 的值，可以是通过服务器中转、也可以是丢弃消息。所以，调用 SendByP2PChannel 方法发送目标消息，意味着，目标消息是可以被丢弃的。

ActionTypeOnNoP2PChannel 定义如下：

API：

```
    // 摘要:
    //     当要经过P2P发送消息而P2P通道又不存在时，采取的操作。
    public enum ActionTypeOnNoP2PChannel
    {
        // 摘要:
        //     通过服务器中转
        TransferByServer = 0,
        //
        // 摘要:
        //     丢弃消息
        Discard = 1,
    }
```

两个客户端之间有可能同时存在两个 P2P 通道：一个是 TCP 的，一个是 UDP 的。在这种情况下，ESFramework 将优先使用基于 TCP 的 P2P 通道。

还记得前面我们介绍的 ICustomizeOutter 接口的 TransferByServer 方法，它的意思是，即使有可靠的 P2P 通道存在，信息也一定要通过服务器中转。

顺便说一下，当我们采用前面介绍的 IFileOutter 来发送文件给其他在线用户时，如果存在 P2P 通道，则 ESFramework 将通过 P2P 通道来发送文件数据块。

四.P2P 通道控制器 IP2PController

ESPlus 为客户端提供了 ESPlus.Application.P2PSession.Passive.IP2PController 接口以控制和管理 P2P 通道。通过 IRapidPassiveEngine 暴露了 P2PController 属性，我们可以获取 IP2PController 的引用。IP2PController 接口的定义如下：

API：

```
    // 摘要:
    //     P2P通道控制器。控制和查看所有的P2P通道。
    public interface IP2PController
    {
        // 摘要:
```

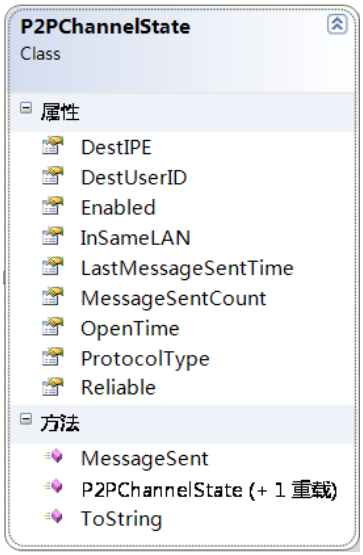
```
        //     采用的P2P通道的类型。默认为TcpAndUdp，表示TCP打洞和UDP打洞都进行尝试。
        P2PChannelMode P2PChannelMode { get; set; }

        // 摘要:
        //     当所有的P2P通道关闭时（比如因为自己断线），触发此事件。
        event CbGeneric AllP2PChannelClosed;
        //
        // 摘要:
        //     当某个P2P Channel关闭时，触发此事件。
        event CbGeneric<P2PChannelState> P2PChannelClosed;
        //
        // 摘要:
        //     当某个P2P Channel创建成功时，触发此事件。
        event CbGeneric<P2PChannelState> P2PChannelOpened;
        //
        // 摘要:
        //     当尝试建立P2P连接失败时，触发此事件。参数为对方的UserID。
        event CbGeneric<string> P2PConnectFailed;

        // 摘要:
        //     获取所有P2P通道的状态。
        Dictionary<string, P2PChannelState> GetP2PChannelState();
        //
        // 摘要:
        //     获取目标用户的P2P通道的状态。
        P2PChannelState GetP2PChannelState(string destUserID);
        IUdpSessionStateViewer GetUdpSessionStateViewer();
        //
        // 摘要:
        //     与目标用户之间是否存在P2P通道。
        bool IsP2PChannelExist(string destUserID);
        //
        // 摘要:
        //     P2P通道是否繁忙。如果返回null，表示没有P2P通道。
        bool? P2PChannelIsBusy(string destUserID);
        //
        // 摘要:
        //     尝试与目标用户建立P2P Channel。（异步方式。）
        //
        // 参数:
        //   destUserID:
        //     目标用户的UserID
        void P2PConnectAsyn(string destUserID);
    }
```

首先，我们可以通过设置 P2PChannelMode 属性，来要求 ESPlus 在尝试创建 P2P 通道时是使用 UDP 还是 TCP，或者都进行尝试。

当我们要与某个其他在线用户 P2P 会话之前，可以先调用 IP2PController 的 P2PConnectAsyn 方法，该方法将会在后台线程中尝试与目标用户建立 P2P 连接（即进行 UDP 打洞和 TCP 打洞）。当 P2P 连接建立成功时，会触发

P2PChannelOpened 事件，而接下来后续的 P2P 消息就可以通过 P2P 通道发送。如果 P2P 连接建立失败，则将触发 P2PConnectFailed 事件。

每一个 P2P 通道在内存中都对应着一个 P2PChannelState 实例，该实例记录着 P2P 通道的相关信息和实时状态，比如：P2P 会话对方的 UserID 和地址信息，P2P 通道的协议类型、通道的创建时间、通过该通道发送的消息个数、以及发送的最后一个消息的时间、当前通道是否可靠等。P2PChannelState 的类图如下：



当已经建立的 P2P 通道关闭时（可能是因为对方下线、或者 P2P 连接中断、或者 UDP 的 P2P 心跳超时），IP2PController 将触发 P2PChannelClosed 的事件。

任何时候，我们都可以通过 IP2PController 的 IsP2PChannelExist 方法查询与目标用户之间是否存在 P2P 通道。我们还可以通过其 GetP2PChannelState 方法来获取所有的或某个特定的 P2P 通道的实时状态。

（05）—— 联系人

联系人（ESPlus.Application.Contacts 命名空间）是 ESFramework 6.0 新增的功能，用于取代之前的 好友与组。

大部分分布式通信系统中，除了客户端与服务器进行通信外，都还会涉及到客户端之间相互通信、以及需要将客户端进行分组的功能，或者是类似这方面的需求。ESFramework 对这一常见的任务内置了强大的支持，包括从客户端到服务端、一直到 ESPlatform 群集。在设计时，我们就考虑到了如何对常见的好友通信与组广播通信进行最大的支持，以期让 ESFramework 的使用者非常容易的就能够使用这些功能。

在 ESFramework 中，好友与组成员统称为“联系人”，“联系人”并不仅仅是指用户（某人），而是可以指任何一个运行的客户端实例。只要两个客户端实例之间需要频繁相互通信，那么它们就可以建立好友关系（friend）。如果需要在某些特定的客户端实例间进行广播通信，那么这些实例就可以被划分到同一个组，成为组友（groupmate）。如果是使用 ESFramework 开发类似 IM 的系统，那么这两种关系就更明显，它们就类似 QQ 的好友与群。

一. 非强制性依赖

ESPlus 定义了联系人管理器接口（IContactsManager），如果你的应用需要联系人方面的功能，那么只要实现对应的接口，并通过 IRapidServerEngine 对应的属性注入到框架中即可。如果不需要联系人相关的功能，那么可以完全忽略它的存在。ESFramework 并不会强制性地要求你的应用必须要实现一个与自己的项目需求没有任何关系的接口（比如 IContactsManager）。我们将选择的权利交到了你的手中，你可以根据项目的具体需求，决定要实现哪

些接口，并注入到 ESFramework 框架中。

二. 联系人功能服务端详解

1.IContactsManager

ESPlus.Application.Contacts 命名空间用于实现联系人功能，在服务端，定义了 IContactsManager 接口：

API：

```
// 摘要:
//     联系人管理器接口。
public interface IContactsManager
{
    // 摘要:
    //     获取用户的相关联系人。
    List<string> GetContacts(string userID);
    //
    // 摘要:
    //     获取目标组的所有成员。
    List<string> GetGroupMemberList(string groupID);
}
```

（1）GetContacts 方法用于获取某个用户的所有相关联系人的 UserID 列表。同很多常见的返回集合的方法设计规则一样，该方法不允许返回 null，如果目标用户没有任何联系人，那么请返回元素个数为 0 的 List。

GetContacts 方法在框架内的主要用途：当某个用户上下线时，框架可以自动通知该用户的所有相关联系人其状态变化事件。

（2）GetGroupMemberList 用于获取一个组的所有成员列表。比如，在实现该接口时，我们可以从 DB 中加载目标组及组成员，然后返回成员列表。

GetGroupMemberList 方法在框架内的主要用途：在发送组内广播消息功能时，框架只有通过 GetGroupMembers 才知道要将消息转发给哪些用户。

如果要使用联系人功能，请实现 IContactsManager 接口，并在 IRapidServerEngnie 初始化之前，注入到其 ContactsManager 属性。

（3）ESPlus 内置了 IContactsManager 接口的实现：DefaultContactsManager。DefaultContactsManager 假设所有的在线用户都是相关联系人 —— 即其 GetContacts 方法始终返回所有在线用户列表。DefaultFriendsManager 一般用于 demo 或者测试程序中，使得 demo 或测试程序的实现更简单。

2. IContactsController

当服务端引擎初始化成功后，还可以通过其暴露的 ContactsController 属性来对联系人功能进行某些设置和向某个组广播消息：

API：

```
// 摘要:
//     联系人控制器。
public interface IContactsController
{
    // 摘要:
    //     是否监听来自客户端的Blob广播，默认值为false。如果为true，即使客户端广播的是blob消息，也将触发BroadcastReceived事件（将需要更多的内存）。
    bool BroadcastBlobListened { get; set; }
    //
```

```
        // 摘要:
        //     用户上线时，是否通知相关联系人。默认值为true。
        bool ContactsConnectedNotifyEnabled { get; set; }
        //
        // 摘要:
        //     用户下线时，是否通知相关联系人。默认值为true。
        bool ContactsDisconnectedNotifyEnabled { get; set; }
        //
        // 摘要:
        //     联系人上下线通知是否使用单独的线程。默认值为false。
        bool UseContactsNotifyThread { get; set; }

        // 摘要:
        //     当因为某个组成员不在线而导致对其广播失败时（不包括Blob及其片段信息），将触发该事件。参数：UserID（组成员ID） - BroadcastInformation。
        event CbGeneric<string, BroadcastInformation> BroadcastFailed;
        //
        // 摘要:
        //     当服务端接收到要转发的广播消息时（BroadcastBlobListened属性决定了是否包括blob广播），触发此事件。参数：broadcasterID
        //     - groupID - broadcastType - broadcastContent - tag
        event CbGeneric<string, string, int, byte[], string> BroadcastReceived;

        // 摘要:
        //     在组内广播信息。
        //
        // 参数:
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   broadcastContent:
        //     广播的内容
        //
        //   tag:
        //     附加信息
        //
        //   action:
        //     当通道繁忙时采取的操作。
        void Broadcast(string groupID, int broadcastType, byte[] broadcastContent, string tag, ActionTypeOnChannelIsBusy action);
        //
        // 摘要:
        //     在组内广播大数据块信息。直到数据发送完毕，该方法才会返回。如果担心长时间阻塞调用线程，可考虑异步调用本方法。
        //
        // 参数:
```

```
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   blobContent:
        //     大数据块的内容
        //
        //   tag:
        //     附加信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        void BroadcastBlob(string groupID, int broadcastType, byte[] blobContent, string tag, int
fragmentSize);
        //
        // 摘要:
        //     在组内广播大数据块信息。直到数据发送完毕，该方法才会返回。如果担心长时间阻塞调用线程，
可考虑异步调用本方法。
        //
        // 参数:
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   blobContent:
        //     大数据块的内容
        //
        //   tag:
        //     附加信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        //
        //   handler:
        //     当发送任务结束时，将回调该处理器
        //
        //   handlerTag:
        //     将回传给ResultHandler的参数
        void BroadcastBlob(string groupID, int broadcastType, byte[] blobContent, string tag, int
fragmentSize, ResultHandler handler, object handlerTag);
    }
```

（１）ContactsConnectedNotifyEnabled 用于控制是否开启联系人上线通知；ContactsDisconnectedNotifyEnabled 用于控制是否开启联系人下线通知。

（２）UseContactsNotifyThread 用于控制是否开启了联系人通知线程。如果为 true，表示所有的上下线通知消

息都集中在一个单独的线程中发送。我们建议，在一个需要承载大量用户同时在线的系统中，如果开启了联系人上下线通知功能，请务必将 UseContactsNotifyThread 也设置为 true。

（3）BroadcastReceived 事件可用于在服务端监控所有转发的广播信息。

（4）Broadcast 和 BroadcastBlob 方法使得我们可以在服务端直接向某个组广播信息。

三. 联系人功能客户端详解

1.IContactsOutter

在客户端 定义了 ESPlus.Application.Contacts.Passive.IContactsOutter 接口 我们可以通过 IRapidPassiveEngine 暴露出的 ContactsOutter 属性，来使用联系人功能。

API：

```
    // 摘要:
    //     用于客户端发送与联系人操作相关的信息和广播。
    public interface IContactsOutter
    {
        // 摘要:
        //     当接收到某个组内的广播消息（包括大数据块信息）时，触发此事件。参数：broadcasterID - groupID - broadcastType
        //     - broadcastContent - tag。 如果broadcasterID为null，表示是服务端发送的广播。
        event CbGeneric<string, string, int, byte[], string> BroadcastReceived;
        //
        // 摘要:
        //     当组成员上线时，触发该事件。参数：UserID
        event CbGeneric<string> ContactsConnected;
        //
        // 摘要:
        //     当组成员下线时，触发该事件。参数：UserID
        event CbGeneric<string> ContactsOffline;

        // 摘要:
        //     在组内广播信息。
        //
        // 参数:
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   broadcastContent:
        //     信息的内容
        //
        //   tag:
        //     附加信息
        //
        //   action:
        //     当通道繁忙时采取的操作。
        void Broadcast(string groupID, int broadcastType, byte[] broadcastContent, string tag,
```

```
ActionTypeOnChannelIsBusy action);
        //
        // 摘要:
        //     在组内广播大数据块信息。直到数据发送完毕，该方法才会返回。若不想阻塞调用线程，可考虑
使用异步广播重载方法。
        //
        // 参数:
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   blobContent:
        //     大数据块的内容
        //
        //   tag:
        //     附加信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        void BroadcastBlob(string groupID, int broadcastType, byte[] blobContent, string tag, int
fragmentSize);
        //
        // 摘要:
        //     在组内异步广播大数据块信息（当前调用线程立即返回）。
        //
        // 参数:
        //   groupID:
        //     接收广播信息的组ID
        //
        //   broadcastType:
        //     广播信息的类型
        //
        //   blobContent:
        //     大数据块的内容
        //
        //   tag:
        //     附加信息
        //
        //   fragmentSize:
        //     分片传递时，片段的大小
        //
        //   handler:
        //     当发送任务结束时，将回调该处理器
        //
        //   handlerTag:
        //     将回传给ResultHandler的参数
        void BroadcastBlob(string groupID, int broadcastType, byte[] blobContent, string tag, int
```

```
fragmentSize, ResultHandler handler, object handlerTag);
        //
        // 摘要:
        //     获取所有在线的联系人。
        List<string> GetAllOnlineContacts();
        //
        // 摘要:
        //     获取联系人列表。
        List<string> GetContacts();
        //
        // 摘要:
        //     获取组的成员。
        Groupmates GetGroupMembers(string groupID);
    }
```

（1）GetGroupMembers 会返回某个组的所有成员，并将在线成员与不在线成员区分开来。

（2）当用户上线或下线时，框架会回调 IContactsOutter 接口的 ContactsConnected 或 ContactsOffline 事件以通知其所有相关联系人。

（3）可以通过 Broadcast 和 BroadcastBlob 方法向任何一个组发送广播，目标组的每个在线成员都将会通过 IContactsOutter 的 BroadcastReceived 事件来获得广播内容。

2.关注联系人的实时状态

在类似 IM 的系统中，每个运行的客户端实例，在其运行的整个生命周期中，都需要清楚地知道与其相关每个联系人的实时状态，这个需求可以这样来实现：

（1）当某个客户端登陆成功后，就调用 GetContacts 方法和 GetAllOnlineContacts 方法以获取联系人列表和所有的在线联系人列表。这样，就知道了所有联系人的初始状态。

（2）预定 IContactsOutter 的 ContactsConnected 和 ContactsOffline 事件，然后在运行的过程中，当 ContactsConnected 和 ContactsOffline 事件触发时，就修改对应联系人的状态。

这样就保证我们的客户端可以实时地知道每个相关的联系人是否在线。

四. 请注意性能

如果具体的项目中需要频繁地用到联系人特性，那么在实现 IContactsManager 接口时，要特别注意性能问题。

1. 实现时使用缓存

因为 IContactsManager 接口的方法会被框架频繁调用，所以，必须想办法提高 IContactsManager 接口的实现的性能。

如果 IContactsManager 接口的方法被调用时，每次都需要从外部介质（比如 DB、文件等）重新加载相关联系人，那么毫无疑问将严重地降低应用程序的性能。通常的解决方案是，将联系人关系缓存在内存中，以避免重复从外部介质加载。

至于究竟采用何种策略来提升 IContactsManager 的性能，需要根据你的项目具体情况而作妥当设计。特别是在高性能的分布式通信系统中，这一点是万万不可忽视的。

2. 精简联系人列表

IContactsManager 返回的联系人列表，请尽可能的精简，不要包含垃圾数据。

特别是当服务端引擎 ContactsController 的联系人上下线通知设置为 true 时，用户的状态变化会自动通知其所有相关联系人。如果用户状态改变频繁，而其相关联系人的数量又很巨大时，这种开销是非常大的。必要时，可以

考虑将 ContactsConnectedNotifyEnabled 或 ContactsDisconnectedNotifyEnabled 设置为 false 以关闭状态改变自动通知。

（06）—— Rapid 通信引擎

ESPlus.Rapid 命名空间提供了我们可以直接使用的客户端 Rapid 引擎和服务端 Rapid 引擎。Rapid 引擎将 ESFramework 和 ESPlus 提供的各种组件装配成一个整体，将结构的复杂性隐藏在引擎的内部，而提供给我们一个简单易用的接口。在引擎对象初始化成功之后，我们就可以使用引擎对象暴露出的四大武器和两个可选功能了。Rapid 引擎底层使用的核心是 ESFramework 的 StreamTcpEngine，其采用 TCP 协议、并使用了紧凑的二进制消息格式。

一. 客户端 Rapid 引擎

ESPlus.Rapid. IRapidPassiveEngine 是客户端 Rapid 引擎的接口。观察 IRapidPassiveEngine 接口的定义，我们就可以了解客户端 Rapid 引擎的使用方法了。

API：

```
    // 摘要:
    //     迅捷的客户端引擎。基于TCP、使用二进制协议。
    public interface IRapidPassiveEngine : IRapidEngine
    {
        // 摘要:
        //     客户端引擎高级控制选项。如果要设置AdvancedOptions的某些属性，必须在调用Initialize方法之前设置才有效。
        AdvancedOptions Advanced { get; }
        //
        // 摘要:
        //     掉线后，是否自动重连。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        bool AutoReconnect { get; set; }
        //
        // 摘要:
        //     该接口用于向服务器发送基本的请求，如获取自己的IP、获取所有在线用户列表等等。（只有Initialize方法调用成功之后，该属性才可正常使用）
        IBasicOutter BasicOutter { get; }
        //
        // 摘要:
        //     与服务器之间的通道是否处于繁忙状态？
        bool ChannelIsBusy { get; }
        //
        // 摘要:
        //     当前是否处于连接状态。如果为true，则表示不仅TCP连接正常，而且是登录成功的状态。
        bool Connected { get; }
        //
        // 摘要:
        //     该接口用于客户端发送与联系人操作相关的信息和广播。（只有Initialize方法调用成功之后，该属性才可正常使用）
        IContactsOutter ContactsOutter { get; }
```

```
        //
        // 摘要:
        //     当前登录的用户UserID。（只有Initialize方法调用成功之后，该属性才可正常使用）
        string CurrentUserID { get; }
        //
        // 摘要:
        //     该接口用于向服务器或其它在线用户发送自定义信息。（只有Initialize方法调用成功之后，该
属性才可正常使用）
        ICustomizeOutter CustomizeOutter { get; }
        //
        // 摘要:
        //     该接口用于向服务器或其它在线用户发送文件。（只有Initialize方法调用成功之后，该属性才
可正常使用）
        IFileOutter FileOutter { get; }
        //
        // 摘要:
        //     该接口用于获取好友列表及接收好友状态改变通知。（只有Initialize方法调用成功之后，该属
性才可正常使用）
        IFriendsOutter FriendsOutter { get; }
        //
        // 摘要:
        //     该接口用于客户端发送与组操作相关的信息和广播信息。（只有Initialize方法调用成功之后，
该属性才可正常使用）
        IGroupOutter GroupOutter { get; }
        //
        // 摘要:
        //     每隔多长时间（秒）发送一次心跳消息。如果小于等于0，表示不发送定时心跳。默认值为10秒。
（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        int HeartBeatSpanInSecs { get; set; }
        //
        // 摘要:
        //     该接口用于控制和管理所有的P2P通道。（只有Initialize方法调用成功之后，该属性才可正常使
用）
        IP2PController P2PController { get; }
        //
        // 摘要:
        //     P2P服务器的地址。默认值为null。（如果要set该属性，则必须在调用Initialize方法之前设置
才有效。） 如果服务端引擎开启了UseAsP2PServer，则可以不设置该属性。如果同时又设置该属性为非null值，
则以设置值为准。
        AgileIPE P2PServerAddress { get; set; }
        //
        // 摘要:
        //     当前引擎所连接的服务器的地址。
        AgileIPE ServerAddress { get; }
        //
        // 摘要:
        //     Sock5代理服务器信息。如果不需要代理，则设置为null。（如果要set该属性，则必须在调用
Initialize方法之前设置才有效。）
        Sock5ProxyInfo Sock5ProxyInfo { get; set; }
```

```
        //
        // 摘要:
        //     是否用于性能测试（默认值：false）。如果是，则设为true，以加快客户端引擎的启动速度，方便测试。
        bool StressTesting { get; set; }
        //
        // 摘要:
        //     系统标志。引擎在初始化时会提交给服务器验证客户端是否是正确的系统。（也可以被借用于登陆增强验证）。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        string SystemToken { get; set; }
        //
        // 摘要:
        //     当通过ICustomizeOutter进行同步调用时，等待回复的最长时间。如果小于等于0，表示一直阻塞调用线程直到等到回复为止。默认值为30秒。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        int WaitResponseTimeoutInSecs { get; set; }

        // 摘要:
        //     当客户端与服务器的TCP连接断开时，将触发此事件。
        event CbGeneric ConnectionInterrupted;
        //
        // 摘要:
        //     自动重连开始时，触发此事件。如果重连成功则将重新登录，并触发RelogonCompleted事件。
        event CbGeneric ConnectionRebuildStart;
        //
        // 摘要:
        //     当接收到来自服务器或其它用户的消息时，触发此事件。 事件参数：sourceUserID - informationType - message -
        //     tag 。 如果消息来自服务器，则sourceUserID为null。
        event CbGeneric<string, int, byte[], string> MessageReceived;
        //
        // 摘要:
        //     当断线重连成功时，会自动登录服务器验证用户账号密码，并触发此事件。如果验证失败，则与服务器的连接将会断开，且后续不会再自动重连。事件参数表明了登录验证的结果。
        event CbGeneric<LogonResponse> RelogonCompleted;

        // 摘要:
        //     关闭并释放客户端通信引擎。
        void Close();
        //
        // 摘要:
        //     主动关闭与ESFramework服务器的连接,将引发自动重连。
        //
        // 参数:
        //   reconnectNow:
        //     是否立即重连？
        void CloseConnection(bool reconnectNow);
        //
        // 摘要:
        //     完成客户端引擎的初始化，与服务器建立TCP连接，连接成功后立即验证用户密码。如果连接失败，
```

```
则抛出异常。
        //
        // 参数:
        //   userID:
        //     当前登录的用户ID，由数字和字母组成，最大长度为10
        //
        //   logonPassword:
        //     用户登陆密码。
        //
        //   serverIP:
        //     服务器的IP地址。
        //
        //   serverPort:
        //     服务器的端口。
        //
        //   customizeHandler:
        //     自定义处理器，用于处理服务器或其它用户发送过来的消息
        LogonResponse Initialize(string userID, string logonPassword, string serverIP, int serverPort,
ICustomizeHandler customizeHandler);
        //
        // 摘要:
        //     向服务器或其它在线用户发送消息。如果其它用户不在线，消息将被丢弃。
        //
        // 参数:
        //   targetUserID:
        //     接收者的UserID，如果为服务器，则传入null
        //
        //   informationType:
        //     消息类型
        //
        //   message:
        //     消息内容
        //
        //   tag:
        //     附加内容
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag);
        //
        // 摘要:
        //     向服务器或其它在线用户发送消息。如果其它用户不在线，消息将被丢弃。
        //
        // 参数:
        //   targetUserID:
        //     接收者的UserID，如果为服务器，则传入null
        //
        //   informationType:
        //     消息类型
        //
        //   message:
        //     消息内容
```

```
        //
        //   tag:
        //     附加内容
        //
        //   fragmentSize:
        //     消息将被分块发送，分块的大小
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize);
        //
        // 摘要:
        //     向服务器或其它在线用户异步发送消息（当前调用线程立即返回）。如果其它用户不在线，消息
将被丢弃。
        //
        // 参数:
        //   targetUserID:
        //     接收者的UserID，如果为服务器，则传入null
        //
        //   informationType:
        //     消息类型
        //
        //   message:
        //     消息内容
        //
        //   tag:
        //     附加内容
        //
        //   fragmentSize:
        //     消息将被分块发送，分块的大小
        //
        //   handler:
        //     当发送任务结束时，将回调该处理器
        //
        //   handlerTag:
        //     将回传给ResultHandler的参数
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize, ResultHandler handler, object handlerTag);
    }
```

1. 属性

请注意，如果需要通过引擎的 setter 属性设置某些参数（如 HeartBeatSpanInSecs 等），必须在调用 Initialize 方法之前设置才能生效。如果要通过引擎的 getter 属性使用某些组件（如 BasicOutter），必须在引擎初始化成功后，才能正常使用。

当客户端是通过代理服务器上 Internet 时，可以通过 Sock5ProxyInfo 属性来设置 Sock5 代理的相关信息。

SystemToken 属性的值用于作为参数传递到服务端 IBasicBusinessHandler 接口的 VerifyUser 方法，以进行登陆验证。如果有的系统需要验证除了密码之外更多的信息，那么也可以通过 systemToken 传递这些额外信息。

WaitResponseTimeoutInSecs 属性用于设置同步调用（以及 ACK 机制的信息发送）时等待回复的超时时间。

当独立部署的 P2P 服务器时，我们可以通过 P2PServerAddress 属性设定 P2P 服务器的地址信息。关于部署 P2P 服务器的更多内容，可以参见 ESFramework 使用技巧 —— 部署 P2P 服务器。

ChannelIsBusy：指当前客户端是否有消息正在发往服务器。另外几个只读属性：ServerAddress、Connected、CurrentUserID，注释已经很清楚了。

BasicOutter、CustomizeOutter、FileOutter、P2PController、FriendsOutter、GroupOutter 等属性，我们已经在前面章节进行了详细的描述，不再赘述。

2. 方法

参数设置好后，就可以调用 Initialize 方法进行初始化。Initialize 方法需要传入自定义信息处理器 ICustomizeHandler。

正如在 ESFramework 开发手册（02） —— 基础功能与状态通知讲到的 ，当 TCP 重连成功时，Rapid 客户端引擎会自动登录服务器重新验证用户账号和密码，并触发 RelogonCompleted 事件。开发人员应该依据 IRapidPassiveEngine 的 RelogonCompleted 事件来作为重连成功/失败的依据。

当不再需要使用引擎实例时，请调用 Close 方法以释放资源。

二. 服务端 Rapid 引擎

ESPlus.Rapid. IRapidServerEngine 是服务端 Rapid 引擎的接口。我们来看看 IRapidServerEngine 接口的定义。

API：

```
    // 摘要:
    //     迅捷的服务端引擎。基于TCP、使用二进制协议。
    public interface IRapidServerEngine : IRapidEngine
    {
        // 摘要:
        //     服务端引擎高级控制器。（如果要设置AdvancedController的某些属性，则必须在调用Initialize方法之前设置才有效。）
        IAdvancedController Advanced { get; }
        //
        // 摘要:
        //     是否自动回复客户端发过来的心跳消息（将心跳消息原封不动地发回客户端）。默认值为false。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        bool AutoRespondHeartBeat { get; set; }
        //
        // 摘要:
        //     通过此接口，服务端可以将目标用户从服务器中踢出，并关闭其对应的tcp连接。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IBasicController BasicController { get; set; }
        //
        // 摘要:
        //     当前在线连接的数量。
        int ConnectionCount { get; }
        //
        // 摘要:
        //     通过此接口，服务端可以控制联系人通知和主动向某个组广播信息。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IContactsController ContactsController { get; }
        //
        // 摘要:
        //     【可选】如果需要支持联系人，则必须设置该属性。（如果要set该属性，则必须在调用Initialize
```

```
方法之前设置才有效。）
        IContactsManager ContactsManager { set; }
        //
        // 摘要:
        //     通过此接口，服务端可以主动向在线用户发送/投递自定义信息。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        ICustomizeController CustomizeController { get; }
        //
        // 摘要:
        //     通过此接口，服务端可以主动向在线用户发送文件。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IFileController FileController { get; }
        //
        // 摘要:
        //     通过此接口，服务端可以控制好友通知。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IFriendsController FriendsController { get; }
        //
        // 摘要:
        //     【可选】如果需要好友关系支持，则必须设置该属性。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        IFriendsManager FriendsManager { set; }
        //
        // 摘要:
        //     通过此接口，服务端可以控制组友通知和主动向某个组广播信息。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IGroupController GroupController { get; }
        //
        // 摘要:
        //     【可选】如果需要组关系支持，则必须设置该属性。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        IGroupManager GroupManager { set; }
        //
        // 摘要:
        //     心跳超时间隔（秒）。即服务端多久没有收到客户端的心跳消息，就视客户端为超时掉线。默认值为30秒。如果设置小于等于0，则表示不做心跳检查。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        int HeartbeatTimeoutInSecs { get; set; }
        //
        // 摘要:
        //     通过哪个IP地址提供服务。如果设为null，则表示绑定本地所有IP。默认值为null。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        string IPAddressBinding { get; set; }
        //
        // 摘要:
        //     服务器允许最大的同时连接数。
        int MaxConnectionCount { get; }
        //
        // 摘要:
```

```
        //     平台用户管理器（用于ESPlatform，通常为ASM的远程引用）。可以获取群集中所有在线用户信息。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IPlatformUserManager PlatformUserManager { get; }
        //
        // 摘要:
        //     当前监听的端口。
        int Port { get; }
        //
        // 摘要:
        //     服务端实例的唯一编号。该属性用于ESPlatform。
        string ServerID { get; }
        IServiceTypeNameMatcher ServiceTypeNameMatcher { get; }
        //
        // 摘要:
        //     是否同时作为P2P服务器运行。默认值为true（P2P服务器将使用UDP端口，端口号为当前引擎监听的端口号加1）。（如果要set该属性，则必须要在调用Initialize方法之前设置才有效。）
        bool UseAsP2PServer { get; set; }
        //
        // 摘要:
        //     通过此接口，可以获取用户的相关信息以及用户上/下线的事件通知。（只有在RapidServerEngine初始化成功后，才能正常使用。）
        IUserManager UserManager { get; }
        //
        // 摘要:
        //     当通过ICustomizeController.QueryClient方法进行同步调用时，等待回复的最长时间。如果小于等于0，表示一直阻塞调用线程直到等到回复为止。默认值为30秒。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        int WaitResponseTimeoutInSecs { get; set; }
        //
        // 摘要:
        //     发送消息的超时，单位：秒。默认值：5秒。如果设置为小于等于0，则表示无限。（如果要set该属性，则必须在调用Initialize方法之前设置才有效。）
        int WriteTimeoutInSecs { get; set; }

        // 摘要:
        //     当接收到来自客户端的消息时，触发此事件。 事件参数：sourceUserID - informationType - message - tag
        //     。
        event CbGeneric<string, int, byte[], string> MessageReceived;

        // 摘要:
        //     关闭服务端引擎。
        void Close();
        //
        // 摘要:
        //     完成服务端引擎的初始化，并启动服务端引擎。
        //
        // 参数:
        //   port:
```

```
        //     用于提供tcp通信服务的端口
        //
        //   customizeHandler:
        //     服务器通过此接口来处理客户端提交给服务端自定义信息。
        void Initialize(int port, ICustomizeHandler customizeHandler);
        //
        // 摘要:
        //     完成服务端引擎的初始化，并启动服务端引擎。
        //
        // 参数:
        //   port:
        //     用于提供tcp通信服务的端口
        //
        //   customizeHandler:
        //     服务器通过此接口来处理客户端提交给服务端自定义信息。
        //
        //   basicHandler:
        //     用于验证客户端登陆密码。如果不需要验证，则直接传入null。
        void Initialize(int port, ICustomizeHandler customizeHandler, IBasicHandler basicHandler);
        //
        // 摘要:
        //     向在线用户发送消息。如果目标用户不在线，消息将被丢弃。
        //
        // 参数:
        //   targetUserID:
        //     接收者的UserID
        //
        //   informationType:
        //     消息类型
        //
        //   message:
        //     消息内容
        //
        //   tag:
        //     附加内容
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag);
        //
        // 摘要:
        //     向在线用户发送消息。如果目标用户不在线，消息将被丢弃。
        //
        // 参数:
        //   targetUserID:
        //     接收者的UserID
        //
        //   informationType:
        //     消息类型
        //
        //   message:
        //     消息内容
```

```
        //
        //   tag:
        //     附加内容
        //
        //   fragmentSize:
        //     消息将被分块发送，分块的大小
        void SendMessage(string targetUserID, int informationType, byte[] message, string tag, int
fragmentSize);
    }
```

1. 属性

请注意，同客户端一样，如果需要通过引擎的 setter 属性设置某些参数（如 FriendsManager、GroupManager 等），必须在调用 Initialize 方法之前设置才能生效。如果要通过引擎的 getter 属性使用某些组件（如 UserManager、FriendsController 等），必须在 Initialize 方法执行成功后，才能正常使用。

IPAddressBinding 用于服务器有多个 IP 的情况，通过设定 IPAddressBinding 的值可以指定服务端要监听哪个 IP，如果设为 null，表示监听服务器绑定的所有 IP。

UseAsP2PServer 用于指示是否在服务端中集成部署 P2P 服务器。关于部署 P2P 服务器的更多内容，可以参见 ESFramework 使用技巧 —— 部署 P2P 服务器。

PlatformUserManager 用于 ESPlatform，通过它可以获取群集中所有在线用户信息。当没有群集存在时，PlatformUserManager 等同于 UserManager。

FriendManager 和 GroupManager 属性分别用于管理好友和组，这两个是可选功能，如果不需要好友和组的支持，可以忽略它们。其详细介绍请参见联系人。

BasicController、CustomizeController、FileController、UserManager、FriendsController、GroupController 的作用我们前面已经详细介绍了，这里不再赘述。

2. 方法

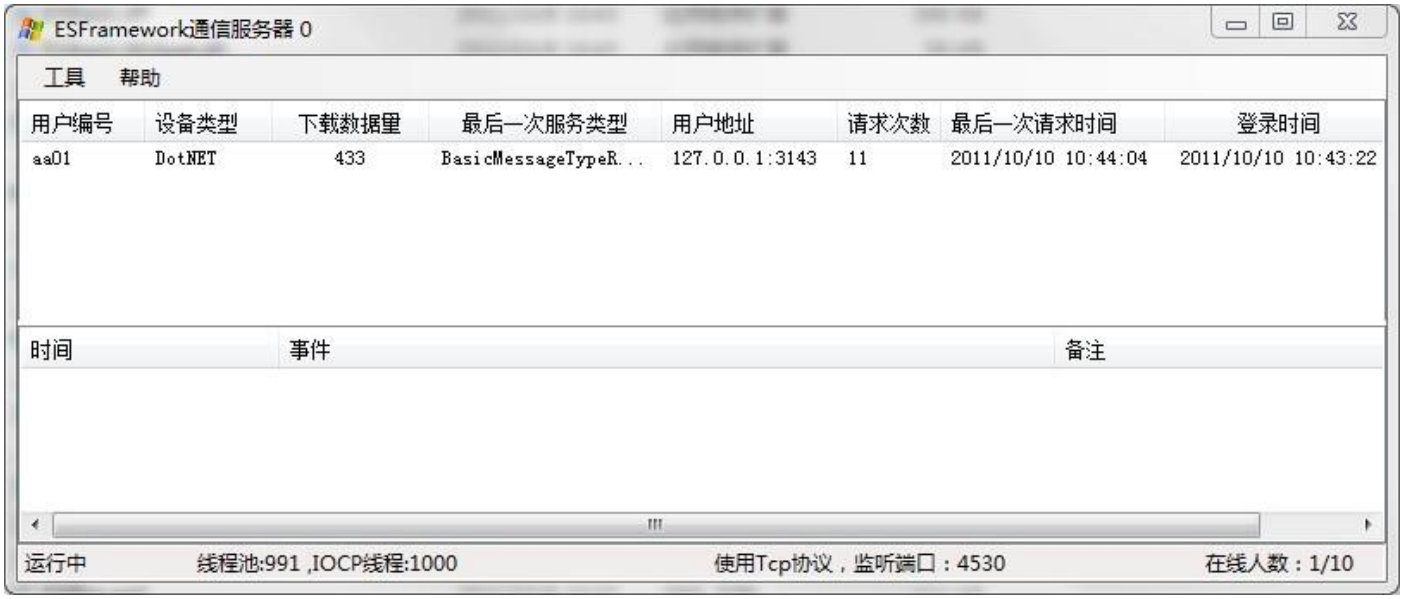
属性及参数设置好后，就可以调用 Initialize 方法进行初始化。Initialize 方法需要的 Handler 参数已经在前面详细介绍过了。

当不再需要使用引擎实例时，调用 Close 方法以释放资源。

3. 默认主窗体

我们的所有 Demo 使用的都是 ESFramework.Boost（源码开放）提供了默认的传主窗体 ESFramework.Boost.Controls.MainServerForm。一般，只有在调试和测试阶段，我们才会使用该窗体来实时显示每个在线用户的状态。在正式发布时，则建议不要使用该窗体了。特别是在高性能的应用中，由于该窗体会实时刷新每个用户的状态数据，其消耗 CPU 是不可忽视的。

当服务端 Rapid 引擎初始化成功之后，就可以构造 MainServerForm 并将其显示出来了，其截图如下：



简单提一下，图中显示的设备类型可以是.NET 客户端、Silverlight 客户端、ios 客户端等等。不同平台的客户端引擎可以连接上同一个服务器，并相互通信。

如果具体的应用需要在主窗体中做更多的事情，则需要自己来实现。可以下载 ESFramework.Boost 的源码，大家可以在其基础上添加所需的功能。

当然，我们也可以在 windows 服务中使用服务端 Rapid 引擎，只要在 windows 服务启动的时候初始化 Rapid 引擎就可以了，这样就不需要窗体了，而且还节省了 UI 刷新所消耗的性能。

三. RapidEngineFactory

我们不能通过 new 来创建服务端或客户端 Rapid 引擎的实例，必须通过 RapidEngineFactory 的静态方法来创建它们。

API：

```
// 摘要:
//     Rapid引擎工厂。
public static class RapidEngineFactory
{
    // 摘要:
    //     创建一个P2P服务器实例。
    public static IP2PServer CreateP2PServer();
    //
    // 摘要:
    //     创建一个Rapid客户端引擎实例。
    public static IRapidPassiveEngine CreatePassiveEngine();
    //
    // 摘要:
    //     创建一个Rapid客户端引擎实例。
    public static IRapidServerEngine CreateServerEngine();
}
```

我们甚至可以通过 RapidEngineFactory 的 CreateP2PServer 来创建 P2P 服务器实例，P2P 服务器用于协助客户端之间创建 P2P 通道（P2P“打洞”）。

到此为止，我们已经将基于 ESFramework 进行二次开发所需要掌握的所有基础设施都已介绍完毕，大家基本可

以上手 ESFramework 开发了。

关于二次开发的步骤，可以参考这篇总结：ESFramework 开发手册（13） —— ESFramework 二次开发说明

如果大家结合阅读我们 demo 的源码 应该就更清楚明白了。后续我们将会写一些高阶一点的以及 ESFramework 使用技巧方面的文章，以帮助大家更好地理解和使用 ESFramework。

（07）—— 掉线与心跳机制

虽然我们前面已经介绍完了 ESFramework 开发所需掌握的各种基础设施，但是还不够。想要更好地利用 ESFramework 这一利器，有些背景知识是我们必须要理解的。就像本文介绍的心跳机制，在严峻的 Internet 条件下，是通信系统中不可或缺的机制之一。

在 Internet 上采用 TCP 进行通信的系统，都会遇到一个令人头疼的问题，就是“掉线”。而“TCP 掉线”这个问题远比我们通常所能想象的要复杂的多 —— 网络拓扑纷繁复杂、而从始节点 A 到终节点 B 之间可能要经过 N 多的交换机、路由器、防火墙等等硬件设备，每个硬件设备的相关设定也不统一，再加上网络中可能出现的拥塞、延迟等，使得我们在编程时，处理掉线也非常棘手。

一. 从程序的角度看待 TCP 掉线

TCP 掉线的原因多种多样、不一而足，比如，客人的电脑突然断电、OS 崩溃、路由器重启、网线接触不良、因为 P2P 下载软件而导致网络资源短缺、Internet 网络的不稳定等等，但是从程序的角度来说，我们可以总结为两种情况：程序能立即感知的掉线和程序不能立即感知的掉线。

1. 程序能立即感知的掉线

也就是说客户端一掉线，服务器端的某个读写对应的 TCP 连接的线程就会抛出异常，这种情况相对容易处理。而 ESFramework 针对这种情况，会触发 IUserManager 的 SomeOneDisconnected 事件，来通知我们的应用程序。

API：

```
///<summary>
/// 客户端连接被关闭时，将触发此事件。不要远程预定该事件。
///</summary>
event CbGeneric<UserData ,DisconnectedType> SomeOneDisconnected;
```

2. 程序不能立即感知的掉线

我们都知道，TCP 连接的建立，需要经过三次握手；而 TCP 连接的断开，需要经过四次挥手。掉线通常没什么大不了的，掉就掉了呗，只要四次挥手顺利完成后，服务器和客户端分别做一些善后处理就可以。

麻烦的事情在于，连接在没有机会完成 4 次挥手时已经断开了（比如当客人的电脑系统死机，或客人电脑与服务器之间的某处物理网线断开），而服务端以为客户端还正常在线，而客户端也自以为还正常在线。这种程序对现实状态的错误判断有可能引发诸多悲剧。比如，在此情况下，客户端发一个指令给服务器，服务器因为没有收到而一直处于等待指令的状态；而客户端了，以为服务器已经收到了，也就一直处于等待服务端回复的状态。如果程序的其它部分需要依据当前的状态来做后续的操作，那就可能会出问题，因为程序对当前连接状态的判断是错误的。

毫无疑问，这种对连接状态错误的判断所持续的时间越久，带来可能的危害就越大。当然，如果我们不做任何额外的处理措施，服务器到最后也能感受到客户端的掉线，但是，这个时间可能已经过去了几分钟甚至几十分钟。对于大多数应用来说，这是不可忍受的。 所以，针对这种不能立即感知掉线的情况，我们要做的补救措施，就是帮助程序尽快地获知 tcp 连接已断开的信息。

首先，我们可以在 Socket 上通过 Socket.IOControl 方法设置 KeepAliveValues，来控制底层 TCP 的保活机制，比如，设定 2 秒钟检测一次，超过 10 秒检测失败时抛出异常。

API：

```
byte[] inOptionValues = FillKeepAliveStruct(1, 10000, 2000);
ocket.IOControl(IOControlCode.KeepAliveValues, inOptionValues, null);
```

ESFramework 底层已经进行了如此处理。据我们的经验，这种设定可以解决一部分问题，但是仍然会有一些连接在断开后，远远超过 10 秒才被感知掉。所以，这个补救措施还是远远不够的。我们还需要在应用层加入我们自己的 TCP 连接状态检测机制，这种机制就是通常所说的“心跳”。

二. "心跳"机制

1.原理

心跳机制的原理很简单：客户端每隔 N 秒向服务端发送一个心跳消息，服务端收到心跳消息后，回复同样的心跳消息给客户端。如果服务端或客户端在 M 秒（M>N）内都没有收到包括心跳消息在内的任何消息，即心跳超时，我们就认为目标 TCP 连接已经断开了。

由于不同的应用程序对感知 TCP 掉线的灵敏度不一样，所以，N 和 M 的值就可以设定的不一样。灵敏度要求越高，N 和 M 就要越小；灵敏度要求越低，N 和 M 就可以越大。而要求灵敏度越高，也是有代价的，那就是需要更频繁地发送心跳消息，如果有几千个连接同时频繁地发送心跳消息，那么其所消耗的资源也是不能忽略的。

当然，网络环境（如延迟的大小）的好坏，也对会对 N 和 M 的值的设定产生影响，比如，网络延迟较大，那么 N 与 M 之间的差值也应该越大（比如，M 是 N 的 3 倍）。否则，可能会产生误判 —— 即 TCP 连接没有断开，只是因为网络延迟大才及时没收到心跳消息，我们却认为连接已经断开了。

ESFramework 内置了心跳机制：

在服务端，可通过 IRapidServerEngine 的 HeartbeatTimeoutInSecs 属性来设置上面描述的 M 值。

在客户端，则通过 IRapidPassiveEngine 的 HeartBeatSpanInSecs 属性设置 N 值。

当心跳超时时，服务端会触发 IUserManager 的 SomeOneTimeOuted 事件，来通知我们的应用程序。

2.必须关闭掉线的 TCP 连接

无论是普通掉线（立即感知）还是心跳超时掉线（非立即感知），都需要关闭对应的 TCP 连接以释放系统资源。ESFramework 将会自动帮我们关闭掉线的 TCP 连接。

另外要提醒一点，当 TCP 连接超时掉线时，服务端会引擎首先会触发 IUserManager 的 SomeOneTimeOuted 事件，接着再触发 IUserManager 的 SomeOneDisconnected 事件。

3.UDP 与"心跳"

前面介绍的都是关于 TCP 的掉线的问题，下面我们看看 UDP。

由于 UDP 是无连接的协议，所以，当我们在使用 ESFramework 的 UDP 引擎的时候，几乎肯定是需要配备心跳机制的，使用心跳消息确认客户端还在线，以保证服务端不会过早释放对应的 Session 或长期保留已失效的 Session。

在 ESFramework 开发手册（04） —— 可靠的 P2P 一文中介绍的 P2P 通道如果是基于 UDP 的，则 ESPlus 内部也启动了心跳机制，以保证在基于 UDP 的 P2P 通道断开时，ESPlus 能尽快感知，并关闭对应的 P2P 通道。

4.关闭心跳机制

比如，在 LAN 中进行通信的分布式系统，由于网络延迟和意外掉线的几率微乎其微，所以，可以考虑关闭心跳机制。再比如，当我们断点调试客户端程序时，由于断点时间太久，服务端会判断为客户端已经心跳超时掉线了，在这种情况下，也可以关闭心跳机制。那么如何关闭心跳机制了？可以这样做：

将 IRapidPassiveEngine 的 HeartBeatSpanInSecs 属性设置为 0。这样客户端就不会发送定时的心跳消息了。

将 IRapidServerEngine 的 HeartbeatTimeoutInSecs 属性设置为小于等于 0。这表示服务端将不再做心跳超时检查。

三. 客户端如何快速感知自己掉线？

在某些客户端电脑上，比如拔掉网线，或断开 wifi，程序可能需要几秒到几分钟才能感知到自己掉线，不同的电脑这个感受的时间不一样。那么如何才能让客户端尽可能快地得到掉线通知了？

可以利用 socket 写超时的机制，像下面这样做：

（1）将 Socket 发送缓冲区的大小设置为 0。对应 IRapidPassiveEngine 的 Advanced 属性的 SocketSendBuffSize 属性。

（2）设置写超时为一个较小的值，如 30 秒。对应 IRapidPassiveEngine 的 Advanced 属性的 WriteTimeoutInSecs 属性。

这样，结合上面的心跳发送机制（如每隔 5 秒发送一个心跳），则当网络断开后，在发送心跳消息，最多再过 30 秒，程序就会得到掉线通知了。

（08）—— 重登陆模式与掉线重连

ESPlus 提供的 Rapid 引擎采用了这样一条规则：当客户端与服务器成功建立 TCP 连接以后，发送的第一个消息为登录消息，当登录消息中的帐号密码经过服务端的验证后，服务端就会从消息中取出 UserID 的值，并将其与对应的 TCP 连接绑定起来。这样，服务端就知道每一个 TCP 连接所对应的用户 UserID，而当我们要求服务端向某个客户端发送消息时，服务端就知道通过哪个 TCP 连接进行发送了。对同一个服务端而言，TCP 连接与 UserID 是一一对应的，一个 TCP 连接只能对应一个 UserID；同样的，一个 UserID 最多存在一个 TCP 连接。

一. 两种重登陆模式

在现实中，经常出现这样的情况：比如我们用的 QQ，当我们用一个账号在 A 地登录了，还未下线，而我又用此账号在 B 地登录，会发生什么情况了？QQ 采用的策略是用新连接取代旧连接，即通知 A 地的客户端其已经被挤掉线了（如提示“同名的用户已在其它地方登陆”），而对于后续的通信，服务器都将与 B 地的客户端进行。

QQ 采用的这种模式在 ESFramework 中称为 ReplaceOld 模式。但是，有的应用可能需要保留 A 地的连接而忽略新来的 B 地的连接，对于这种情况，我们可以采用另外一种模式：IgnoreNew。ESFramework 通过 RelogonMode 枚举来定义这两种模式：

API：

```
    // 摘要:
    //     重登陆模式。当从另外一个新连接上收到一个同名ID用户的消息时，地址管理器对旧的连接的处理模式。
    public enum RelogonMode
    {
        // 摘要:
        //     忽略新的连接。
        IgnoreNew = 0,
        //
        // 摘要:
        //     使用新的连接取代旧的连接。
        ReplaceOld = 1,
    }
```

我们可以设置用户管理器 IUserManager 的 RelogonMode 属性来控制 ESFramework 采用哪种重登陆模式。比如：

API：

```
rapidServerEngine.UserManager.RelogonMode = RelogonMode.ReplaceOld;
```

RapidServerEngine 默认的重登陆模式是 ReplaceOld。

二. ESFramework 对重登陆模式的反应

1. IgnoreNew 模式

如果我们采用的是 IgnoreNew 模式， 当服务端从另外一个新的连接上收到同名用户发来的消息时，ESFramework 会触发 IUserManager 的 NewConnectionIgnored 事件来通知服务端应用程序：

API：

```
/// 如果RelogonMode为IgnoreNew，并且当从一个新连接上收到一个同名ID用户的消息时将触发此事件。
/// 注意，只有在该事件处理完毕后，才会关闭新连接。可以在该事件的处理函数中，将相关情况通知给客户端。
///</summary>
event CbGeneric<string, IPEndPoint> NewConnectionIgnored;
```

事件的第一个参数 string 是同名用户的 ID，第二个参数是新连接的客户端地址。ESFramework/ESPlus 在触发此事件前后所做的动作有：

通知新连接对应的客户端，已经有同名的用户在线了，新的连接将被关闭。新连接对应的客户端 Rapid 引擎的 Intialize 方法将返回 LogonResult.HadLoggedOn。

关闭新的连接。

2. ReplaceOld 模式

如果我们采用的是 ReplaceOld 模式， 当服务端从另外一个新的连接上收到同名用户发来的消息时，ESFramework 会触发 IUserManager 的 SomeOneBeingPushedOut 事件来通知服务端应用程序：

API：

```
///<summary>
/// 如果RelogonMode为ReplaceOld，并且当从另外一个新连接上收到一个同名ID用户的消息时将触发此事件。
/// 注意，只有在该事件处理完毕后，才会真正关闭旧的连接并使用新的地址取代旧的地址。可以在该事件的处理函数中，将相关情况通知给旧连接的客户端。
///</summary>
event CbGeneric<UserData> SomeOneBeingPushedOut;
```

即同名的老连接对应的客户端被挤掉了，事件的参数包含了旧连接对应的相关信息。

同样的，ESFramework/ESPlus 在触发此事件前后所做的动作有：

通知旧连接对应的客户端，有同名的用户连接上来，旧的连接将被关闭。客户端将触发 IBasicOutter 的 BeingPushedOut 事件。

关闭旧的连接。

三. 掉线重连

IRapidPassiveEngine 在与服务端的 TCP 连接断开时，会尝试自动重连服务端，直到重连成功为止。

但是有两种情况的掉线，是不会自动重连的：

客户端被挤掉线。这种情况下，如果也自动重连，将会导致新登陆的同名客户端被挤掉线，新的客户端又会自动重连……，从而陷入“挤掉线-重连-挤掉线-……”的死循环。

客户端被踢出而导致的掉线。我们可以通过 IBasicOutter 的 KickOut 方法或者 IBasicController 的 KickOut 方法将某个在线用户踢掉，这种情况下，被踢掉的客户端不会再自动重连。

如果在这两种情况下掉线，应用仍然要再次连接服务端，那么我们可以手动进行——即重新调用 IRapidPassiveEngine 的初始化方法。

还有一种情况要特别提醒一下。由于客户端引擎在初次连接服务器并登录成功后，会记录当前登录用户的帐号和密码，等之后掉线重连时，会再次自动用记录的帐号密码进行登录。所以，如果在中间过程中，用户的密码被修改了，重连登录就会失败。IRapidPassiveEngine 的重连完成事件 RelogonCompleted 的参数 LogonResponse 的值将会是 LogonResult.Failed。我们可以预定 RelogonCompleted 事件来获取重连失败的通知。

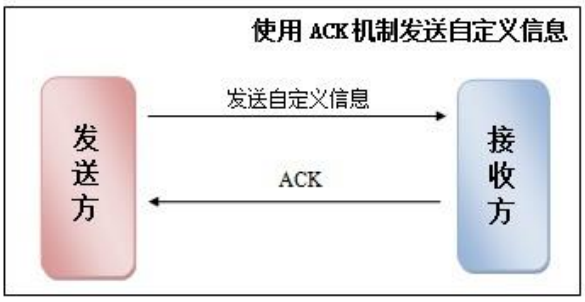
请特别注意：对于服务端而言，是无所谓“重连”的，客户端重连上服务端与客户端初次连接上服务端采用一样的处理模式。客户端只要一掉线，服务端就会清除该客户端对应的 Session 以及相关的状态数据。如果有的应用需要服务端支持重连模型，也很容易做到。我们的服务端可以在接收到用户掉线的通知时，不清除该用户相关的业务状态数据，而只是将其改为不可用，等客户端重连上来后，再激活该用户对应的业务数据就可以了。

（09）—— ACK 机制、同步调用、回复异步调用

正如 ESFramework 开发手册（01） —— 发送和处理信息一文中所介绍的，我们在客户端使用 ICustomizeOutter 接口的 Send 方法，可以给服务端或其它在线客户端发送自定义信息，那么，如何得知接收方是否已经收到了我们发出的信息了呢？特别是针对一些非常重要的信息，确认对方已经收到是非常重要的。ICustomizeOutter 接口的 SendCertainly 方法就使用了带 ACK 机制的发送。

一. 启用 ACK 机制

ACK，即确认的意思。当我们发送一个自定义信息给对方时，对方收到信息后，回复一个 ACK 给我们，我们接收到了 ACK，就知道对方一定收到信息了。



无论是客户端发送信息给服务端、还是客户端发送信息给其它客户端（ICustomizeOutter 接口的 SendCertainly 方法），或者是服务端发送信息给客户端（ICustomizeController 接口的 SendCertainlyToLocalClient 方法），它们采用的 ACK 机制的原理是一样的。

当框架接收到需要 ACK 的信息时，会自动回复 ACK 给发送方。这是由 ESPlus 底层自动完成的，应用程序不需要关心。

ESPlus 会先回复 ACK，然后才会调用处理信息的方法。所以，当发送方接收到 ACK 回复时，只是意味着接收方已经接收到了信息，并不表示接收方已经处理完了信息。

如果发送方在规定的时间内（默认为 30 秒）都没有收到 ACK，则会抛出超时的异常。当然，在规定的时间内没有收到 ACK，并不一定就是接收方没有收到信息，而是有几种可能性：（1）由于网络慢，导致 ACK 延迟抵达发送方。

（2）接收方已经掉线。

（3）发送方已经掉线。

SendCertainly 方法和 SendCertainlyToLocalClient 方法采用的是阻塞模型，即只有收到 ACK 后才会返回，否则一直阻塞当前调用线程。如果既需要 ACK 机制的发送，又不希望阻塞当前线程，那么，可以异步调用 SendCertainly 和 SendCertainlyToLocalClient 方法，并通过回调获知是否有超时异常。

二. 信息同步调用

我们首先介绍一下什么是“信息同步调用”？所谓同步调用，就是在调用线程中返回调用结果。

以客户端与服务端进行交互时最常见的一种情况为例：客户端发送一个请求给服务端，服务端处理后，返回回复消息。比如像这样，服务端提供一个加法运算的服务，客户端请求加法服务的消息类型为 1001，消息体中包含加法运算所需的两个参数；服务端计算完成后给出的回复消息的类型为 1002，回复消息的消息体中包含加法运算的结果。

站在客户端的角度，来看请求消息与应答消息：客户端在调用线程中向服务器发送请求消息，而在接收线程中收到服务器的回复消息，通常，调用线程与接收线程肯定不是同一个线程，所以，从最原始来说，请求消息与回复消息位于不同的线程中。这种模式更像是方法的异步调用。异步调用的好处是当前调用线程不会阻塞，而同步调用的好处是编程模型非常直观简单。毫无疑问，在通信框架中，原始的模型就是异步调用模型，而 ESFramework 也增加了同步调用的机制，使得编程模型更加丰富。使用者可以根据需要使用合适的模型。

使用自定义信息，我们有几个同步调用的方法，比如 ICustomizeOutter 接口的 Query 方法以及 ICustomizeController 接口的 QueryLocalClient 方法，这些同步调用的方法都是有返回值的，如果超时没有收到返回的信息，将抛出超时异常。

同步调用与带 ACK 机制的发送采用的是完全相同的模型，我们完全可以使用同步调用来模拟 ACK 机制，比如，需要确认的信息就使用同步调用发送，接收方在处理同步调用的时候直接返回 null，就可以达到同样的效果。 但是，同步调用与带 ACK 机制的发送还是有两点小区别：

在同步调用中，接收方回复的是对应请求的答案；在带 ACK 机制的发送中，接收方回复的是 ACK。

在同步调用中，接收方是处理完信息后才回复；在带 ACK 机制的发送中，接收方则是先回复 ACK，再处理收到信息。

由于同步调用和带 ACK 机制的发送都有可能超时抛出异常，所以，我们在程序中应当将其 try...catch 起来，以防止应用程序抛出未被截获的异常。

三. 回复异步调用

最后，我们来看看回复异步调用以及其使用场景。

ICustomizeOutter 还有一个重载的 Query 方法，用于回复异步调用：

API：

```
void Query(string targetUserID, int informationType, byte[] info, CallbackHandler handler, object tag);
```

如果调用该 Query 方法时，则当前线程并不阻塞以等待回复，而是继续向下执行，当接收线程接收到 Query 的回复信息时，会直接（在接收线程中）回调 CallbackHandler 委托指向的方法。这就是所谓的“回复异步调用”。

那么，何时使用回复异步调用了？

举个很常见的例子。比如，我们的 C/S 系统的客户端提供一个查询报表的服务，比较直观的方式当然是使用同步调用：客户端点击界面上的“查询”按钮，发出同步调用，等结果返回时刷新界面显示。很简单，是吧？但是这样做有个问题。假设，服务端生成报表比较复杂，需要较长的时间，又或者，网络状况不好，延时比较大，那么当用户点击按钮后，整个界面就“卡死”在哪里了，无法进行其它的操作，直到结果返回之前，界面一直不能操作。这 I 就严重地影响了用户体验。

这种情况下，使用回复异步调用就是更好的方式了。当客户端点击界面上的“查询”按钮，采用回复异步调用的模式发出请求，UI 线程不会阻塞，而是继续向下执行，所以界面不会出现“卡”的现象。当回复到来时，目标委托方法被回调，在委托方法中通过 Control.Invoke 将回复的结果数据转发到 UI 线程进行显示。

同步调用以及回复异步调用，作为两种互补的方式而存在，具体使用那种模型，需根据您的具体需求来定夺。

（10）—— 安全机制

在分布式通信系统中，安全无疑是非常重要的。

ESFramework/ESPlus 作为应用层的开发框架，在这里我们只讨论应用层的安全问题，因为如果黑客是在网络层或链路层进行攻击，位于应用层的系统几乎是无能为力的。从应用层来说，安全的重要性主要体现在以下几个方面：

（1） 防止恶意用户使用格式不正确的消息来试探服务端。

（2） 防止通信的消息被恶意用户截获，或者，即使被恶意用户截获，也无法破解其内容。

（3） 防止恶意用户在未成功登录前，就向服务器发送格式正确的伪装消息。

（4） 防止恶意用户使用巨大数量的空连接来消耗服务器的资源。

ESFramework 内置了一些安全机制，以对上述的安全性提供一些保障，下面我们一一说明。

一. 消息格式验证

ESFramework 定义了通信消息的总体格式，ESPlus 则定义了消息的详细格式。

当网络引擎（无论是服务端的还是客户端的）从网络上接收到一批二进制数据时，会尝试去解析它。如果解析时发现，这批二进制数据的格式不是我们定义好的消息的格式时，将会认为其是非法消息。此时，网络引擎将会丢弃非法数据，并关闭对应的连接（如果引擎是基于 TCP 协议的），然后再触发 INetEngine 接口的 InvalidMsgReceived 事件。

API：

```
///<summary>
/// 当接收到不完整或无法解析的数据时触发该事件
///</summary>
event CbGeneric<UserAddress, MessageInvalidType> InvalidMsgReceived;

public enum MessageInvalidType
{
    ///<summary>
    /// 正常消息。
    ///</summary>
    Valid = 0,

    ///<summary>
    /// 消息尺寸溢出。
    ///</summary>
    MessageSizeOverflow,

    ///<summary>
    /// 无效的消息头
    ///</summary>
    InvalidHeader,

    ///<summary>
    /// 无效的标识符
    ///</summary>
    InvalidToken,
```

```
        ///<summary>
        /// 数据包长度不够
        ///</summary>
        DataLacked,

        ///<summary>
        /// 无效的客户端类型
        ///</summary>
        InvalidClientType
    }
```

事件的参数 UserAddress 说明了非法消息来源于哪个用户地址；而 MessageInvalidType 参数则说明了非法消息的类型，从该枚举可以看出，网络引擎收到的数据无法解析的原因有几种：消息尺寸超过规定的大小，消息头无效、消息的标识符无效等。

二. 消息加密

对于一些关键的信息，是绝对不允许以明文的形式在网络上进行传送的。所以，消息在发送之前，必须进行加密。

在使用 ESPlus 提供的 Rapid 引擎中，内部组装骨架流程时，是没有使用加密组件的，但是，我们仍然可以在发送自定义信息时，保证信息的安全。还记得我们是使用 ICustomizeOutter 发送自定义信息的，以 Send 方法为例：

API：

```
    ///<summary>
    /// 向服务器发送信息。
    ///</summary>
    ///<param name="informationType">自定义信息类型</param>
    ///<param name="info">信息</param>
    void Send(int informationType, byte[] info);
```

在调用 Send 方法之前，我们可以先将要发送的内容 info 进行加密，然后再发送加密后的结果。

而在接收方，会调用 ICustomizeHandler 的 HandleInformation 方法来处理接收到的信息：

API：

```
    ///<summary>
    /// 处理来自客户端的自定义信息。
    ///</summary>
    ///<param name="sourceUserID">发送该信息的用户ID</param>
    ///<param name="informationType">自定义信息类型</param>
    ///<param name="info">信息</param>
    void HandleInformation(string sourceUserID, int informationType, byte[] info);
```

在实现 HandleInformation 方法时，我们可以先解密 info，然后再进行正常的业务处理。

在发送或接收自定义信息时，手动加解密信息，都要注意一点，那就是加解密都是要消耗 CPU 和内存资源的，对于那些高频通信的消息来说，这个开销是绝不可忽视的。所以，我们应该尽可能的只加密那些极其重要的消息/信息（根据 nfomationType 来进行区分），而不是将所有的消息/信息一视同仁。

三. 验证未登录的消息

有的恶意用户，在破解了消息的格式之后，会尝试在不登录的情况下，向服务器发送其他类型的请求消息。ESFramework 支持在服务端对每个连接上收到的消息进行验证，如果验证通不过，则将关闭对应的连接。

ESPlus 提供的 Rapid 引擎，其内部已经保证，恶意用户在在未登录的情况下无法进行其它类型的业务请求。

四. 绑定连接

当 Logon 消息中的帐号密码通过服务端的验证之后，服务端会将该帐号与对应的 TCP 连接绑定起来，构成一个完整的 Session。如果该连接上接收到的后续消息中，只要发现消息头中的 UserID 与该 TCP 连接绑定的帐号不一致，则认为该消息为非法消息，此时，服务端网络引擎将会关闭对应的 TCP 连接。如此，可以防止用一个帐号登录成功后，再用另一个帐号来请求服务。

五. 空连接

到这里，我们已经解决了本章开始提出的前三个问题，这就保证了恶意用户无法向服务器发送恶意的消息了。但是，恶意用户在应用层还可以做一件事情，就是消耗服务端的 TCP 连接。对于每个已成功建立的 TCP 连接，服务端都要为其分配一定的资源并对其进行管理。如果恶意的用户和服务器之间建立很多空闲的连接，对服务器资源的消耗也是不可忽视的。

ESFramework 支持一个特性：某个 TCP 连接连上后，如果在指定的时间内，服务端网络引擎都接收不到来自该连接的任何数据，则将关闭该连接。

我们可以指定这个时间为一个较短的时间（如 3s），来减轻空连接的影响。之所以说是“减轻”，而不是“消除”，是因为在应用层系统中，无法完全规避这个问题，就按照 3 秒钟的超时来说，你服务端关闭连接的速度一定赶不上恶意用户建立连接的速度。

这种情况在应用层来处理就非常的吃力。解决这个问题的更好办法，应该是在防火墙上做相关的策略设定，比如屏蔽掉恶意用户的 IP 地址，过滤由该地址发出 TCP 握手请求的 Syn 包，等等。

（11）—— 服务端信息处理模型

基于 ESFramework 进行二次开发时，主要是通过自定义信息来实现业务逻辑功能，而自定义消息的处理则是通过 ICustomizeHandler 接口的 HandleInformation（或 HandleQuery）方法来进行的。但是，ICustomizeHandler 接口的这两个方法是如何被框架调用的了？在介绍之前，我们先要了解一下.NET 的线程池（ThreadPool）。

一. 线程池

.NET 线程池中有两种类型的线程：工作者线程、完成端口 IOCP 线程。通过 ThreadPool 的静态方法 SetMinThreads、SetMaxThreads 可以对线程池做一些简单的设置。

一般在程序启动时，我们可以调用 System.Threading.ThreadPool.SetMaxThreads(100, 100); 将最大的工作者线程和 IOCP 线程数量设置为 100。

然后，在服务端运行的过程中，可定时调用（比如每秒一次）ThreadPool 的 GetAvailableThreads 方法，来查看线程中可用的（即，空闲的）工作者线程和 IOCP 线程的数量。这样就可以实时监控线程池中线程的状态。

ESFramework 对 ICustomizeHandler 接口的调用都是在线程池的 IOCP 线程或工作者线程中进行的，至于究竟使用的是哪种类型的线程，取决于信息处理模型的设定。

另外说一下，ESFramework 框架本身使用了数个工作者线程，以维持网络引擎的正常运转，但是，ESFramework 框架自身的运转并不消耗任何 IOCP 线程。

二. CustomizeInfoHandleMode

ESFramework 服务端为信息处理提供了两种方案，这两种方案通过 CustomizeInfoHandleMode 枚举进行定义，并对应该枚举有两个取值：IocpDirectly、TaskQueue。

我们知道，ESFramework 内核是基于 IOCP（Windows 系统中最高效的模型）的，即服务端接收到的信息都是在 IOCP 线程中提交的， 对于提交的信息：

（1）如果信息处理模型被指定为 IocpDirectly，则框架将直接在该 IOCP 线程中调用 ICustomizeHandler 接口来处理该信息。

（2）如果信息处理模型被指定为 TaskQueue，则框架首先会将信息放入到一个任务队列中。然后由工作者线程从队列中依次取出信息，并调用 ICustomizeHandler 接口进行处理。

这里可以看出，这两种方案有几个明显的区别：

（1）IocpDirectly 方案，ICustomizeHandler 的调用是在 IOCP 线程中进行的；而 TaskQueue 方案，ICustomizeHandler 的调用是在工作者线程中进行的。

（2）对于 TaskQueue 方案，IOCP 线程的工作仅仅是将接收的信息放入到任务队列中。

我们可以通过 IRapidServerEngine 的 Advanced 控制器的 CustomizeInfoHandleMode 属性来指定要使用的方案（默认是 IocpDirectly）， 像这样：

API：

```
rapidServerEngine.Advanced.CustomizeInfoHandleMode = CustomizeInfoHandleMode.TaskQueue;
```

对于二次开发人员而言，从一种方案切换到另一种方案，体现在代码上只是上面属性设置的修改，不需要在做任何其它的事情，框架内部会自动完成相应的工作。下面我们将深入这两种方案。

三. IocpDirectly 方案

无论是使用 IocpDirectly 方案还是 TaskQueue 方案，一个客户端连接最多用到一个 IOCP 线程（提交接收到的消息的时候才用到）。但是对于 IocpDirectly 方案，由于提交消息是直接进入 ICustomizeHandler 的 HandleInformation（或 HandleQuery）方法调用，所以，IocpDirectly 方案有以下特点：

（1）针对一个具体的 TCP 连接而言，信息处理是串行的。只有当前一个信息处理完成（即 HandleInformation / HandleQuery 方法返回）后，才处理该连接上的下一个请求信息。

（2）针对所有的 TCP 连接而言，信息处理是并行的。即同时有多个 HandleInformation / HandleQuery 方法 正在执行。

（3）通过观察 IOCP 线程的使用个数，就可以知道有多少个信息正在被处理。

（4）如果某个信息处理的耗时超过了心跳超时的设定，则对应的客户端会被当作心跳超时掉线，其 TCP 连接将被服务端主动断开。

四. TaskQueue 方案

在 TaskQueue 方案中，IOCP 线程不再重要，我们要关注的是任务队列和工作者线程。前面我们说道，工作者线程从队列中依次取出消息然后调用 ICustomizeHandler 接口进行处理。用于从事这一工作的工作者线程的数量是可以设定的，通过 IRapidServerEngine 的 Advanced 控制器的 QueueWorkerThreadCount 属性（默认是 20）， 像这样：

API：

```
rapidServerEngine.Advanced.QueueWorkerThreadCount = 50;
```

当然，这个设定的数量必须不能超过 ThreadPool 的 SetMaxThreads 方法设定的最大工作者线程数。

另外，通过 IRapidServerEngine 的 Advanced 控制器提供的 GetTaskQueueInfo 方法，我们可以获取任务队列的当前状态：

API：

```
/// <summary>
/// 当CustomizeInfoHandleMode设置为TaskQueue时，获取队列中待处理的任务个数，以及历史中最大的待处理
```

```
任务个数。
  /// </summary>
  /// <param name="taskCount">待处理的任务个数</param>
  /// <param name="maxTaskCount">历史中最大的待处理任务个数</param>
  void GetTaskQueueInfo(out int taskCount, out int maxTaskCount);
```

那么，相比于 IocpDirectly 方案，TaskQueue 方案有以下特点：

（1）针对一个具体的 TCP 连接而言，信息的处理也不是串行的，而且，后接收到的信息有可能先处理完。

（2）通过与空闲时（比如，程序启动时）线程池中可用的工作者线程数的对比，就可以知道有多少个信息正在被处理。

（3）如果某些信息处理缓慢，那么，通过 GetTaskQueueInfo 方法可看到队列中待处理的任务越来越多，同时可以观察到当前进程的内存消耗越来越大。从客户端表现来看，就是响应越来越慢。

五. 生产 - 消费模型

我们可以使用生产-消费模型来进行分析，从网络接收到消息，相当于生产消息，HandleInformation / HandleQuery 方法的执行相当于是消费消息。决定服务端吞吐量的因素有很多，但最首要也是最核心的因素就是服务端消费消息的能力。每秒消费掉的消息数量越大，吞吐量也就越大。

当生产消息的速度小于或等于消息消费的速度时，服务端的业务处理是从容不迫的。

但是，当生产消息的速度大于或远远大于消息消费的速度时，并且这一状况一直持续下去时，可以想想会发生什么状况？我们简单推导一下。

1. TaskQueue 方案

（1）为任务队列服务的工作者线程都将处于忙碌状态。

（2）队列中待处理的任务越来越多。

（3）内存消耗越来越大，可能导致进程引发 OutOfMemory（内存溢出）异常。

（4）从客户端来看，新的请求无法被服务端响应。

2. IocpDirectly 方案

（1）线程池中可用的 IOCP 线程会越来越少。

（2）每个 TCP 连接的 Socket 缓冲区中排队等待接收的消息会越来越多，当这个缓冲区满时，客户端再向服务端发送消息的方法调用将被阻塞。

（3）如果某些消息处理的异常缓慢，则可能导致某些客户端心跳超时掉线。

当生产消息的速度大于或远远大于消息消费的速度时，提升服务端信息处理能力（这涉及到的诸多方面的优化）是唯一的解决方案。但是，在未提升服务器的处理能力之前，对于这种情况，IocpDirectly 方案和 TaskQueue 方案哪种更好一点了？

虽然，无论哪种方案，客户端的体验都将比较差，但是相比而言，我们推荐是 IocpDirectly 方案，原因在于：基于上面描述 IocpDirectly 方案的第（2）点，客户端发送消息的方法调用将被阻塞，这意味着会倒逼客户端降低生产消息的速度。采用 IocpDirectly 方案是 ESFramework 框架的默认设置。

（12）—— 服务端性能诊断

基于 ESFramework 构建的系统的服务端，如果在运行时遇到性能问题或某些故障时，可以开启 ESFramework 服务端引擎的诊断功能。 诊断功能开启后，ESFramework 将自动跟踪每种类型消息的处理情况，您可以将这些信息定时记录到日志，之后通过分析日志，就可以很快发现问题所在。

针对 ESFramework 开发手册（11） —— 服务端信息处理模型 中提到的两种信息处理模型，诊断功能都是可以使用的。

一. 开启诊断功能

在 IRapidServerEngine 初始化（Initialize 方法）之前，将其 Advanced 属性的 DiagnosticsEnabled 设置为 true，即可开启诊断。

rapidServerEngine.Advanced.DiagnosticsEnabled = true;

开启诊断功能后的服务端在运行时，会额外消耗一点 CPU 和内存，以跟踪和记录诊断信息。

二. 查看诊断信息

通过 IRapidServerEngine 的 Advanced 属性的 DiagnosticsViewer，就可以得到诊断查看器（IDiagnosticsViewer）的引用，其定义如下：

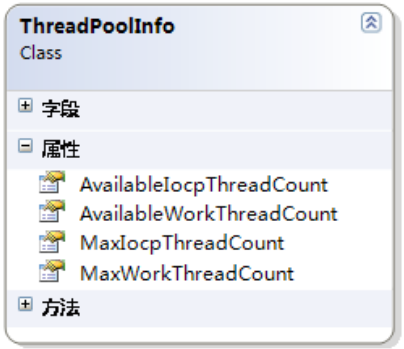
API：

```
public interface IDiagnosticsViewer
{
    /// <summary>
    /// 获取自定义信息处理的统计数据。
    /// </summary>
    List<InfoHandleRecordStatistics> GetCustomizeInfoStatistics();

    /// <summary>
    /// 获取处理尚未完成的自定义信息。
    /// </summary>
    List<InfoHandleRecord> GetUncommittedCustomizeInfos();

    /// <summary>
    /// 获取线程池信息。
    /// </summary>
    ThreadPoolInfo GetThreadPoolInfo();
}
```

IDiagnosticsViewer 提供了三个方法，用于获取 3 种信息：

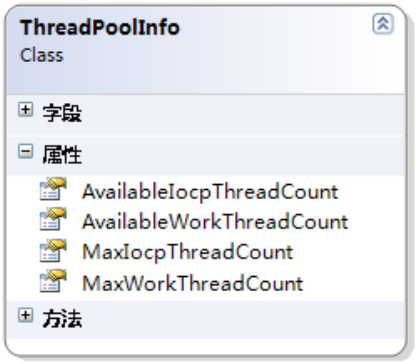
1.GetThreadPoolInfo

GetThreadPoolInfo 用于获取线程池的信息，如设定的最大的工作者线程数/IOCP 线程数，当前可用的工作者线程数/IOCP 线程数。



2.GetCustomizeInfoStatistics

GetCustomizeInfoStatistics 用于获取已处理的信息的相关情况。在其返回的 List 中，针对每种消息类型（informationType）都有一个统计。即，List 中的每个元素都对应于一种消息类型的统计。List 中的元素类型为 InfoHandleRecordStatistics，其类图如下：



InfoHandleRecordStatistics 有 5 个属性，分别解释如下：

（1）InformationType：被统计的信息类型，比如我们用户自定义的消息类型，如用 101 表示 Chat 聊天消息类型。

（2）InformationStyle：信息的风格，比如信息是来自客户端 ICustomizeOutter 的 Send 调用，还是 Query 调用。

（3）CallCount：自服务端启动以来，收到这种类型的信息的总个数。

（4）ExceptionCount：处理这种类型的信息时抛出异常的个数（如果类似在实现 HandleInformation 方法时，没有捕获异常，则其抛出的异常将被框架捕获，并记录在此）。

（5）LastRecords：记录了这种类型的最近 10 条消息的详细处理情况。这个属性至关重要。其集合中的对象是 InfoHandleRecord 类型，每一个元素对应一条信息处理记录。

通过其类图可以看到，InfoHandleRecord 其有 5 个属性。其中的 InformationType 与 InformationStyle 与上述的 InfoHandleRecordStatistics 的同名属性是一致的。

ID 是框架为收到的每个信息分配的唯一标志。重点注意其 StartTime 和 TimeSpent 属性。

StartTime 是服务端处理信息的开始时间。TimeSpent 是处理信息的耗时（毫秒）。

通过观察 TimeSpent，就可以知道哪些信息的处理花费了过多的时间。然后，通过 InformationType 的值，在程序中找到对应的代码进行分析。

3.GetUncommittedCustomizeInfos

GetUncommittedCustomizeInfos 方法用于获取此刻正在处理中的那些信息，其返回的集合中的对象类型仍然是 InfoHandleRecord。

对于尚未处理完成的信息而言，InfoHandleRecord 的 TimeSpent 属性尚未被赋值（默认为 0），所以需要特别关注的是其 StartTime 属性的值与当前时刻的差值。

通过查看 GetUncommittedCustomizeInfos 方法的返回结果，可以很容易发现那些在处理过程中一直卡住或卡死的信息。

三 .诊断日志记录器

当开启服务端的诊断功能后 通常 我们需要定时记录诊断日志。比如 每隔 10 秒就调用一次 IDiagnosticsViewer 的方法，并将得到的诊断信息记录到日志文件中。

ESFramework.Boost 类库（源码开放）提供了 DignosticLogger 类，用于完成这一目的，二次开发者不用再重复实现了，可以将其直接拿去用。DignosticLogger 的构造函数有是三个参数：

API：

```
public DignosticLogger(IDiagnosticsViewer viewer ,string logFilePath, int logSpanInSecs)
```

第一个参数就是我们前面讲到的诊断查看器 IDiagnosticsViewer，第二个参数 logFilePath 为日志文件的路径，第三个参数 logSpanInSecs 是每隔多少秒记录一次。

以下是使用 DignosticLogger 记录的诊断日志的一个片段：

```
2015/4/10 10:48:47:线程情况：WorkThreadCount=85/100, IocpThreadCount=10/10
2015/4/10 10:48:47:历史记录:
2015/4/10 10:48:47:InformationType=-10000, InformationStyle=Login, CallCount=1, ExceptionCount=0
2015/4/10 10:48:47:    ID=1, StartTime=2015/4/10 10:46:37, TimeSpent=2
2015/4/10 10:48:47:InformationType=-10003, InformationStyle=GetGroupmates, CallCount=1, ExceptionCount=0
2015/4/10 10:48:47:    ID=2, StartTime=2015/4/10 10:46:37, TimeSpent=1
2015/4/10 10:48:47:InformationType=1021, InformationStyle=Query, CallCount=1, ExceptionCount=0
2015/4/10 10:48:47:    ID=3, StartTime=2015/4/10 10:46:37, TimeSpent=17
2015/4/10 10:48:47:InformationType=1011, InformationStyle=Query, CallCount=1, ExceptionCount=0
2015/4/10 10:48:47:    ID=4, StartTime=2015/4/10 10:46:37, TimeSpent=0
2015/4/10 10:48:47:InformationType=1052, InformationStyle=Query, CallCount=1, ExceptionCount=0
2015/4/10 10:48:47:    ID=5, StartTime=2015/4/10 10:46:38, TimeSpent=9
2015/4/10 10:48:47:InformationType=1000, InformationStyle=DirectByRapidEngine, CallCount=6, ExceptionCount=0
2015/4/10 10:48:47:    ID=9, StartTime=2015/4/10 10:47:29, TimeSpent=8
2015/4/10 10:48:47:    ID=10, StartTime=2015/4/10 10:47:30, TimeSpent=0
2015/4/10 10:48:47:    ID=11, StartTime=2015/4/10 10:47:31, TimeSpent=0
2015/4/10 10:48:47:    ID=12, StartTime=2015/4/10 10:47:32, TimeSpent=0
2015/4/10 10:48:47:    ID=13, StartTime=2015/4/10 10:47:33, TimeSpent=0
2015/4/10 10:48:47:    ID=14, StartTime=2015/4/10 10:47:34, TimeSpent=0
2015/4/10 10:48:47:
2015/4/10 10:48:47:------------------ ------------------ ------------------ ------------------
```

（13）—— ESFramework 二次开发说明

一. ESFramework 主要组件

根据前面系列文章介绍的内容，我们总结一下，ESFramework 主要包含了以下组件：

功能	Client	Server
Engine	IRapidPassiveEngine	IRapidServerEngine
CustomizeInfo	ICustomizeOutter	ICustomizeController
Basic	IBasicOutter	IBasicController
FileTransfering	IFileOutter	IFileController
P2PSession	IP2PController	——
Friends	IFriendsOutter	IFriendsManager
Group	IGroupOutter	IGroupManager
信息处理	ICustomizeHandler	ICustomizeHandler
登录验证	——	IBasicHandler
用户管理	——	IUserManager

（1）用黑色字体和蓝色字体表示的组件，表示是由 ESFramework 内部实现的，提供给二次开发者可以直接使用的。

（2）用红色字体表示的接口，表示是由二次开发者提供其实现，并挂接到引擎实例上的，供框架回调（通过 Rapid 引擎的 Initialize 方法的参数传入，或其属性注入）。

二. ESFramework 二次开发要点

当基于 ESFramework 开发分布式通信应用时，可以遵循下面的步骤：

（１） 系统中的客户端用户之间是否存在好友关系、是否需要相互通信？如果是，则要实现 IFriendManager 接口。

（２） 系统中是否需要将客户端用户进行分组/群、是否需要在组内进行广播消息？如果是，则要实现 IGroupManager 接口。

（３） 分析项目需求，定义所有的自定义信息类型和自定义信息协议。

（４） 在客户端和服务端分别实现各自的 ICustomizeHandler 接口，用于处理自定义信息、完成系统业务功能。

（５） 在服务端实现 IBasicHandler 接口，用于验证用户账密。如果不需要验证，则直接使用 ESPlus 提供的 EmptyBasicHandler。

（６）通过 RapidEngineFactory 创建客户端和服务端 Rapid 引擎，设置相关属性的值，并调用其 Initialize 方法。

（７） 在客户端，预定并处理引擎的 IRapidPassiveEngine 暴露的连接状态变化事件。如有需要，还可预定并处理 BasicOutter 以及 FriendsOutter、GroupOutter 等相关状态通知事件。

（８） 在服务端，预定并处理 Rapid 引擎的 UserManager 属性的用户状态变化事件。

（９） 使用引擎实例通过属性暴露的四大武器，进一步实现系统业务功能。

三. 跨平台：支持其它平台的客户端

具体请参见：ESFramework 使用技巧 —— 跨平台开发

（14） - - Xamarin 移动端开发（Android、iOS）

ESFramework 的早期版本就已经支持了原生的 Android 和 iOS，而最新版本的 ESFramework 客户端引擎则推出了 Xamarin 版本，可用于来发 Android/iOS/WP 等移动端应用 App 和游戏。如此一来，仅仅懂.NET 的开发者也可以开发手机应用了。

相比较 ESFramework 提供的 Android 和 iOS 原生引擎而言，ESFramework 的 Xamarin 版本内置了与 ESFramework 的 PC 版完全一样的自动序列化器，这使得在打通 PC 端与移动端时，通信协议的封装与解析就不需要做任何额外的工作了。

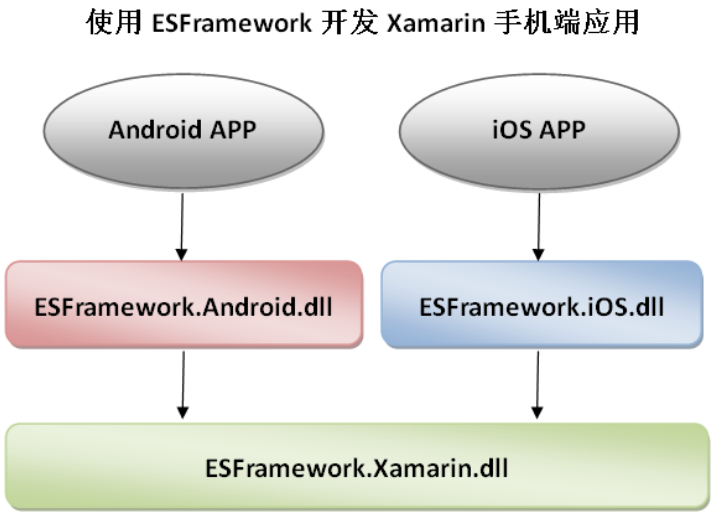
一. ESFramework Xamarin 客户端引擎结构

ESFramework 的 Xamarin 客户端引擎分为两个层次：

（１）可移植的通用部分，即 ESFramework.Xamarin.dll。

（２）特定平台的非通用部分，即 Android 平台上的 ESFramework.Android.dll，和 iOS 平台上的 ESFramework.iOS.dll。

当使用 ESFramework 的 Xamarin 引擎开发手机应用时，应用与组件之间的依赖关系如下图所示：



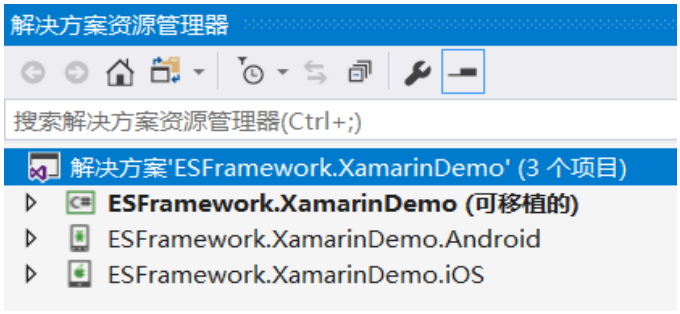
ESFramework.Xamarin.dll 中包含了 ESFramework 所提供的用于二次开发的所有 API 的定义，与 PC 版的接口完全一致。

ESFramework.Android.dll 和 ESFramework.iOS.dll 则是分别在 Android 平台和 iOS 平台上实现了 ESFramework.Xamarin.dll 中定义的接口。

二. 如何使用 ESFramework 的 Xamarin 引擎

1. 引用正确的 dll

通常我们是使用 Xamarin Forms 开发可移植的手机应用，那么，在 Xamarin Forms 项目中只需要引用 ESFramework.Xamarin.dll 就能使用 ESFramework 提供的所有功能。我们以 ESFramework 的 Xamarin 版本的 Demo 为例，如下图所示。



ESFramework.XamarinDemo 项目是可移植的，该项目内部使用 Xamarin Forms 实现了 Demo 几乎所有的功能，它只需要引用 ESFramework.Xamarin.dll。

ESFramework.XamarinDemo.Android 项目是最终发布到安卓平台的项目，它依赖于 ESFramework.XamarinDemo 项目，并实现了 Demo 中那些无法在 ESFramework.XamarinDemo 项目中实现的基于安卓平台的那些功能。所以，ESFramework.XamarinDemo.Android 项目引用了 ESFramework.Xamarin.dll 和 ESFramework.Android.dll。

ESFramework.XamarinDemo.iOS 项目是最终发布到 iOS 平台的项目，它依赖于 ESFramework.XamarinDemo 项目，并实现了 Demo 中那些无法在 ESFramework.XamarinDemo 项目中实现的基于 iOS 平台的那些功能。所以，ESFramework.XamarinDemo.iOS 项目引用了 ESFramework.Xamarin.dll 和 ESFramework.iOS.dll。

2. 使用 ESFramework 的标准接口

ESFramework.Xamarin.dll 提供了 GlobalContext 静态类，其包含了 ESFramework 的核心接口 IRapidPassiveEngine 的引用，以及包含了紧凑的序列化器 ICompactPropertySerializer 的引用。

```
    // 摘要：跨平台公用接口上下文。其实现在具体的平台中。
```

```
public static class GlobalContext
{
    public static ICompactPropertySerializer CompactPropertySerializer;
    public static IRapidPassiveEngine RapidPassiveEngine;    }
```

3. 将 ESFramework 引擎的特定平台实现注入

那么，在什么地方给 GlobalContext 的两个静态成员赋值了？

我们需要在特定平台的项目的程序启动的时候，为 GlobalContext 的两个静态成员赋值，以注入当前平台的 ESFramework 实现。

如上例中的 ESFramework.XamarinDemo.Android 项目 ,我们在 MainActivity.cs 中的 OnCreate 方法的开始处添加如下代码：

```
GlobalContext.RapidPassiveEngine =
ESPlus.Rapid.RapidEngineFactory.CreatePassiveEngine();
GlobalContext.CompactPropertySerializer = CompactPropertySerializer.Default;
```

同理，在上例中的 ESFramework.XamarinDemo.iOS 项目，我们可以在 AppDelegate.cs 中的 FinishedLaunching 方法的开始处添加同样的代码。

ESPlatform 开发手册

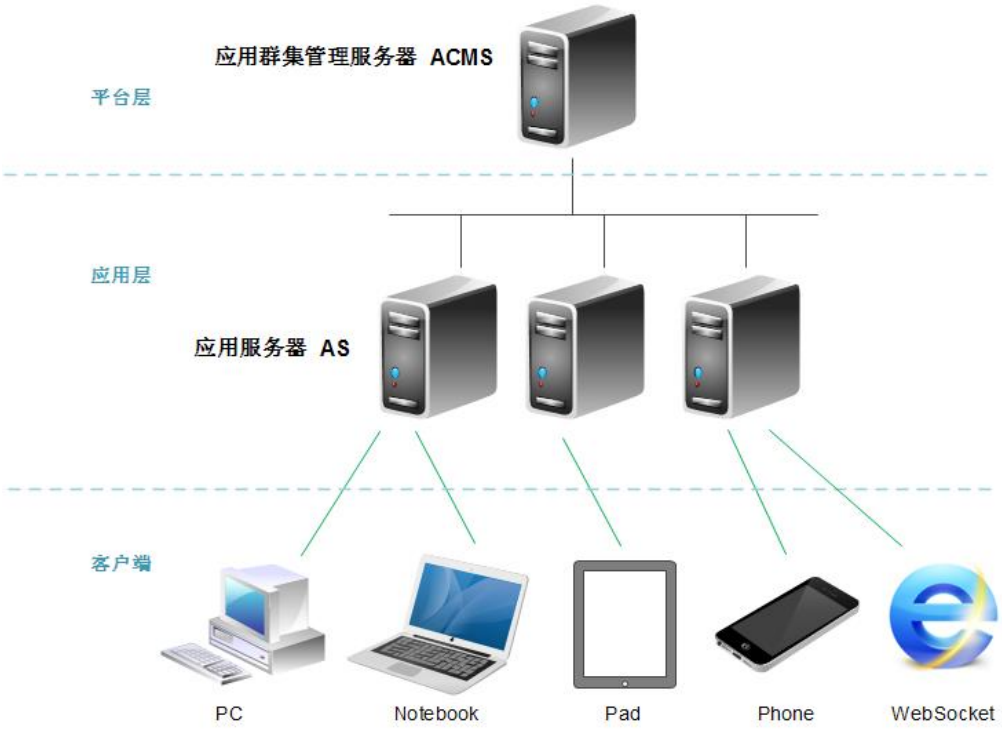
（00）—— 概念与模型

当我们将基于 ESPlus/ESFramework 开发的应用程序的服务端部署在一台服务器上时，就可以称这台服务器为应用服务器（AS）。当在线用户数量不断增加时，我们可能需要部署数台或更多的 AS 来分担负载。但是，如果没有 ESPlatform 统一管理，这些 AS 中的任何一个都是独立的孤岛，不能与其它的 AS 进行协作。对于某些特殊的应用来说，也许是可以的。但是，对于大多数需要群集的应用来说，必须要将众多的 AS 管理起来，并能协调它们的工作。

特别是那些任意两个客户端（这两个客户端可能连接上了不同的 AS）之间需要相互通信的应用，使用 ESFramework 的 P2P 技术可以解决一些问题，但是并不是所有的 P2P 通道都可以创建成功。 使用 ESPlatform 后，所有的在线用户就像连接在同一个 AS 上一样，服务端实例对于客户端而言，是透明的。

一. 群集基础模型

在单个 AS 应用中，我们的关注焦点在于服务端和客户端。而在多 AS 的群集应用中，我们需要增加一个核心关注点，那就是平台层。ESPlatform 群集模型可以划分为三层：平台层、应用层（即 AS 位于的层）、客户端。如下图所示的是 ESPlatform 所支持的最简单的群集模型：



平台层的核心是 ACMS（应用群集管理服务器），用于管理 ESPlatform 群集中的所有应用服务器。ACMS 的主要职责可概括为：

（1）管理所有的在线 AS。

（2）管理所有的在线用户及其 P2P 地址。

（3）在 AS 之间转发消息。

（4）提供服务接口给群集外的其它系统调用。

ESPlatform 提供了可直接部署运行的 ACMS 服务器，我们将在后面的文章中详细介绍它。

现在，我们以上图为例，两个客户端 Client01 与 Client02 分别连上不同的应用服务器 AS01 和 AS02，我们假设由于路由器的原因（比如两个路由器的 NAT 类型都是 Symmetric），Client01 与 Client02 之间的 P2P 通道没有建立成功。此时，如果 Client01 与 Client02 之间要相互沟通信息，那么信息就会经过 ACMS 中转。比如 Client01 要发信息给 Client02，信息经过的路线将会是：Client01 => AS01 => ACMS => AS02 => Client02。

在 ESPlatform 群集模型中，从 Client01 到 Client02 信息有三种可能的路径：

（1）当 Client01 到 Client02 之间存在 P2P 通道时，直接经由 P2P 通道到达。否则，进入（2）。

（2）当 Client01 到 Client02 之间连接的是同一个 AS 时，直接由该 AS 转发。否则，进入（3）。

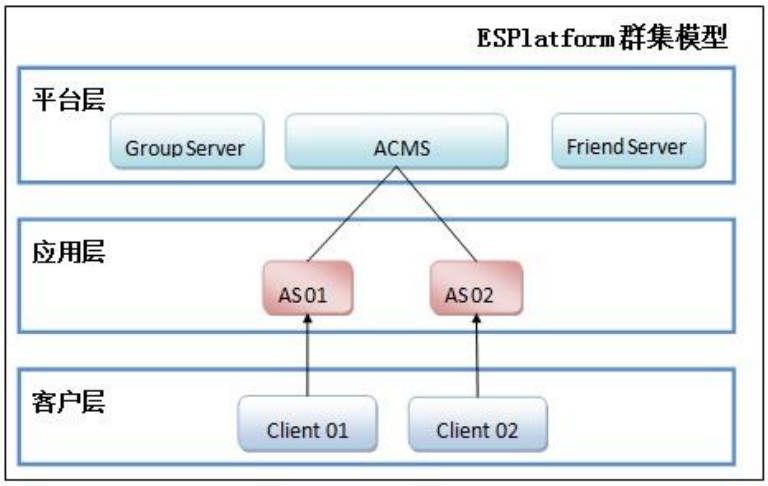
（3）经由 ACMS 转发。

对于 Client01 而言，信息到底是经 3 条路径中的哪一条到达 Client02 的，它并不需要关心， ESPlatform 保证了这一点，而且对客户端是透明的。

二. 全局的好友管理和组管理

对于那些需要好友关系或组关系的应用来说，在单 AS 应用中，通常直接在服务端进程中实例化 IFriendsManager 的实现类和 IGroupManager 的实现类就可以了。但是，在 ESPlatform 群集中，由于每个 AS 都会用到 IFriendsManager 和 IGroupManager，所以，最好有一个全局的好友管理和组管理。

我们可以这样做，将 IFriendsManager 的实现类单独部署在一台服务器上，称之为好友服务器 FriendServer；将 IGroupManager 的实现类单独部署在一台服务器上，称之为组服务器 GroupServer。FriendServer 和 GroupServer 都通过 Remoting 的方式暴露出对应的服务。所以，各个 AS 就是通过 Remoting 来访问 IFriendsManager 和 IGroupManager。很明显，全局的 FriendServer 和 GroupServer 应该位于平台层。



在部署好 FriendServer 和 GroupServer 后，只需要在初始化 AS 的服务端引擎时，将 IFriendsManager 和 IGroupManager 的 Remoting 引用注入到引擎对应的属性，接下来 AS 就会自动访问 FriendServer 和 GroupServer 暴露的服务了。

当然，这只是最简单的情况，实际上，在真正应用时，仍然要考虑到一个非常重要的方面：性能。AS 会频繁访问 FriendServer 和 GroupServer 暴露的服务，所以，非常有必要认真考虑：

（1）是否需要在 FriendServer 和 GroupServer 的内存中缓存好友/组关系信息以避免每次都从外部加载。

（2）是否需要在 AS 的内存中也缓存好友/组关系信息以避免每次都 Remoting 访问 FriendServer 和 GroupServer。

也许还有其它的方式，但是这点必需根据您的业务需求慎重考虑。

三. 策略与原则

一般而言，群集系统都是相当复杂的系统，为了处理复杂性，原则是必需要遵守的，除非你有充分的理由，并能更好地解决问题。在使用 ESPlatform 群集平台时，我们要充分重视以下几点。

1.保证 AS 与 ACMS 之间网络畅通

AS 在任何时候都有可能访问 ACMS，比如用户上下线的时候，AS 需要向 ACMS 报告；有些客户端之间的信息必需经过 ACMS 进行中转，等等。如果 AS 与 ACMS 之间的网络中断，哪怕只是片刻，可能都会导致 AS 与 ACMS 之间的状态数据不一致。

我们通常将 AS 与 ACMS 部署在同一个机房的同一个局域网内，以保证通信的质量和速度。

2.尽可能使需要相互沟通的客户端连上同一个 AS

一个新启动的客户端应该连接到哪个 AS 服务器，是由我们的 AS 分配策略来决定的。所以，为了尽可能地减少经 ACMS 转发的消息数量，也就是尽可能地降低 ACMS 的负载，在设计 AS 的分配策略时，应特别注意尽可能地使那些需要相互沟通的客户端连接到同一个 AS 服务个器上。这样，即使两个客户端之间的 P2P 通道没有打通，那么，他们之间的交互信息只要通过 AS 中转就可以了，不需要麻烦到 ACMS。

3.热机备份 ACMS

在群集模型中，很容易看到 ACMS 是整个群集系统的“单点”，如果 ACMS 挂掉，虽然每个 AS 还可以独立工作，但跨 AS 的消息就无法完成转发了。所以，在部署 ACMS 时，最好使用双热机备份或多热机备份，这样在主 ACMS 意外当机后，另一台备份机就立马可以接手工作。

四. 小结

ESPlatform 提供了最基础的群集模型（本文的第一个模型图），而后续的进化模型都是在这个基础上根据可能的需求所进行的常规变化。

但无论怎么变化，其基础仍然是我们第一个模型图所示的基础模型，它是 ESPlatform 的核心模型。对于某些业务简单的系统，最简单的模型可能就已经足够了。对于那些复杂的系统，在这个核心模型上要如何变化、如何深化，没有固定的方向可言，其最终取决于我们系统的需求。正如有句话是这样说的：凭空想象的模型是没有用的，必须要与实际的需求结合起来，并在实战中不断淬炼。

（01）—— 迁移到群集平台

在 ESFramework 开发手册（00） —— 概述中，我们提到过 ESFramework 的一个优势：仅仅通过修改几行代码或配置就可以将一个基于 ESFramework 的应用程序平滑迁入到 ESPlatform 平台中。现在，是到了兑现这一承诺的时候了。将单 AS 的 ESFramework 应用迁移到 ESPlatform 群集平台，在通常情况下，只需要两个步骤：

（1）部署并启动应用群集管理服务器 ACMS。

（2）服务端使用 ESPlatform.Rapid.IRapidServerEngine 替换 ESPlus.Rapid.IRapidServerEngine。客户端几乎不用做任何修改。

一. ESPlatform.Rapid.IRapidServerEngine

在单 AS 的 ESFramework 应用中，我们的服务端引擎使用的是 ESPlus.Rapid.IRapidServerEngine，当迁移时，我们需要使用 ESPlatform.Rapid.IRapidServerEngine。ESPlatform.Rapid.IRapidServerEngine 实现了该接口。

API：

```
public interface IPlatformRapidServerEngine : IRapidServerEngine
  {
      /// <summary>
      /// 当与平台服务器ACMS断开连接时，触发此事件。
      /// </summary>
      event CbGeneric PlatformConnectionInterrupted;

      /// <summary>
      /// 当与平台服务器ACMS连接重新建立时，触发此事件。
      /// </summary>
      event CbGeneric PlatformConnectionRebuild;

      /// <summary>
      /// 与平台服务器ACMS是否连接？
      /// </summary>
      bool PlatformConnected { get; }

      /// <summary>
      /// ACMS回调处理器。用于处理ACMS对当前AS的回调。默认值为null。
      /// </summary>
      ICallbackHandler CallbackHandler { set; }
  }
```

可见，IPlatformRapidServerEngine 接口从 IRapidServerEngine 继承，而且增加了两个特性：一是暴露了当前 AS 与 ACMS 之间的连接状态；二是允许注入一个 CallbackHandler，该 CallbackHandler 用于处理

IPlatformCustomizeService 发送的自定义信息（下一篇文章将会详细讲述）。

相比于 ESPlus.Rapid.IRapidServerEngine，使用 ESPlatform.Rapid.IRapidServerEngine 要注意一下几点：

1.构造与初始化

替换时，对开发者的使用来说，创建 ESPlatform.Rapid.IRapidServerEngine 对象时需要使用 ESPlatform.Rapid.RapidEngineFactory 的静态方法：

API：

```
    /// <summary>
    /// 创建基于ESPlatform的服务端引擎。
    /// </summary>
    /// <param name="currentServerID">当前应用服务器的ID</param>
    /// <param name="_acmsIP">应用群集管理服务器IP地址</param>
    /// <param name="_acmsServicePort">应用群集管理服务器提供Remoting服务的端口</param>
    /// <param name="_acmsTransferPort">应用群集管理服务器用于转发消息的端口</param>
    public static IPlatformRapidServerEngine CreateServerEngine(string currentServerID, string acmsIP,
int acmsServicePort, int acmsTransferPort)
```

方法的第一个参数为当前启动服务器的 ID，后三个参数指定了 ACMS 服务器的地址信息。

（1）在 ESPlatform 群集中，每个运行的服务端实例都有一个唯一的 ID，称之为 ServerID。在单 AS 系统中，ServerID 可以被忽略；但在群集系统中，ServerID 成了服务端实例的身份标志。

（2）acmsIP 参数是 ACMS 服务器的 IP 地址，后面两个参数对应着 ACMS 发布的 Remoting 服务的接口、以及用于转发的接口。后面会进一步介绍它们。

（3）如果目标 ACMS 服务器没有启动，那么该构造函数的执行是不会报错的，但是接下来 RapidServerEngine 的 Initialize 方法的调用会抛出异常。

（4）当 ESPlatform.Rapid.RapidServerEngine 的 Initialize 方法执行时，AS 会向 ACMS 注册，初始化完成之后，AS 还会定时向 ACMS 报告自己的状态。

2.平台用户管理器

还记得 ESPlus.Rapid.IRapidServerEngine 的 UserManager 属性吗？我们在服务端编程时，可以通过该属性访问当前 AS 服务器上的所有在线用户的信息。 而 ESPlus.Rapid.IRapidServerEngine 还有个 PlatformUserManager 属性，在单 AS 应用中，这个属性与 UserManager 属性相当于是同一个东西。但是，即使是在单 AS 应用中，PlatformUserManager 属性也有它存在的价值。它的作用在于：当我们开发单 AS 的应用的时候，依据对以后可能迁移到 ESPlatform 群集平台的预测，在需要访问任何一个在线用户的地方使用 PlatformUserManager 来代替使用 UserManager，这将为以后快速地迁移到群集平台铺平道路。

ESPlatform.Rapid.IRapidServerEngine 暴露的 PlatformUserManager 属性，就是真正的平台用户管理器了，通过它，AS 可以访问群集系统中任何一个在线的用户的信息。

3.注册回调通道

由于群集平台会在需要的时候回调服务端，所以，服务端在启动的时候需要注册供平台回调的 tcp 通道。注册回调通道很简单：

（1）在配置文件 App.config 中，配置端口号为 0 的基于 TCP 的 Remoting 通道。

API：

```
<system.runtime.remoting>
      <customErrors mode="Off"/>
      <application>
      <channels>
```

```
        <channel ref="tcp" port="0" >
          <serverProviders>
            <provider ref="wsdl" />
            <formatter ref="soap" typeFilterLevel="Full" />
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
```

（2）在构造 ESPlatform.Rapid.RapidServerEngine 之前，添加下面代码注册通道：

API：

```
RemotingConfiguration.Configure("******.exe.config");
```

4.替换之后

在将 ESPlus.Rapid.IRapidServerEngine 替换成 ESPlatform.Rapid.IRapidServerEngine 之后，ESFramework/ESPlus 提供的四大武器和两个可选功能都将按我们所期望的正常工作。比如，某个客户端通过 ICustomizeOutter 发送消息给另外一个 AS 上的客户端，那么，目标客户端是可以收到这个消息的。再比如，我们通过 IGroupOutter 发送广播消息时，即使同一个组的成员登录到了群集系统中的不同的 AS 上，那么每个成员也都是还能收到这个广播消息的。

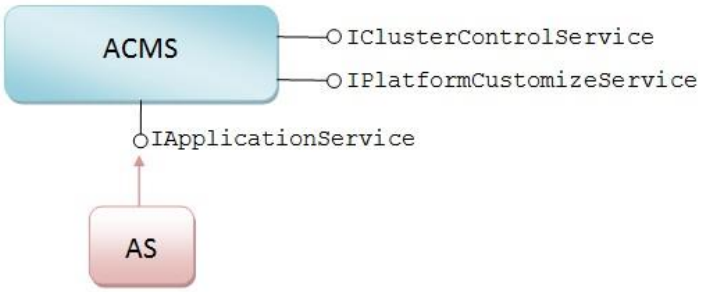
像同步调用、回复异步调用、文件传送、P2P 通道的创建，等等功能，都正常工作，而不受影响。

二. 应用群集管理服务器 ACMS

ESPlatform 提供了可直接部署运行的应用群集管理服务器 ESPlatform.ACMServer.exe，在部署 ESPlatform 群集系统时，只要修改一下配置，然后启动它就可以了。

正如 ESPlatform 群集平台（00）—— 概念与模型一文所说，ACMS 在群集系统中所扮演的重要角色，可以说，它是整个 ESPlatform 群集平台的核心。ACMS 的核心职责可概括为：管理所有的在线 AS、管理所有的在线用户、在 AS 之间转发消息、提供服务接口给群集外的其它系统调用。

ACMS 通过 Remoting 的方式对外暴露这些功能和服务。ACMS 提供了三个 Remoting 服务接口：IApplicationService、IClusterControlService、IPlatformCustomizeService。



IApplicationService 是给 ESPlatform 群集系统内部的 AS 使用的，AS 通过该接口向 ACMS 实时报告自己的状态。通过 IApplicationService，ACMS 就可以：管理在线服务器、管理在线用户。请注意，ESPlatform.Rapid.RapidServerEngine 已经在内部自动配合 ACMS 完成了这些功能，所以，我们在基于 ESFramework/ESPlus/ESPlatform 进行二次开发时，可以忽略 IApplicationService 的存在。

IClusterControlService 和 IPlatformCustomizeService 用于提供给群集外的系统（如 BL）来访问群集中的信息

或控制群集中的服务器。我们将在下篇文章中详细讨论它们。

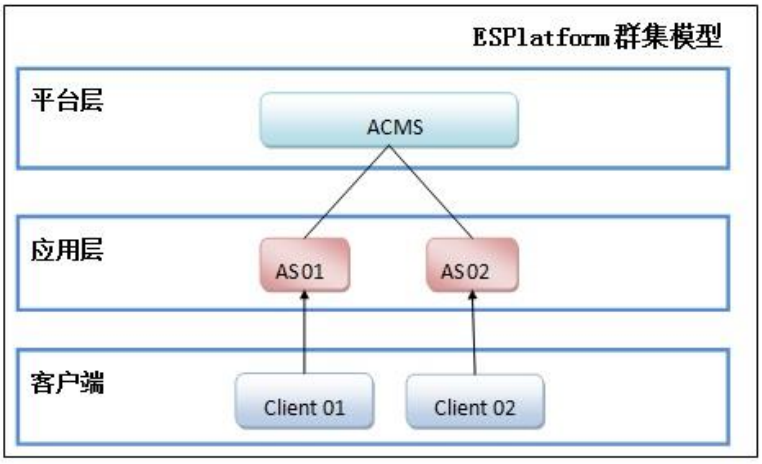
现在我们来看看 ACMS 的配置文件的内容：

API：

```
<configuration>
  <appSettings>
    <!——应用群集中的服务器分配策略——>
    <add key="ServerAssignedPolicy" value="MinUserCount"/>
    <!——用于在AS之间转发消息的Port——>
    <add key="TransferPort" value="12000"/>
  </appSettings>

  <system.runtime.remoting>
    <application>
      <channels>
        <!——提供IPlatformCustomizeService和IClusterControlService Remoting服务的Port——>
        <channel ref="tcp" port="11000" >
          <serverProviders>
            <provider ref="wsdl" />
            <formatter ref="soap" typeFilterLevel="Full" />
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>

    </application>
  </system.runtime.remoting>
</configuration>
```

上述的 3 个 Remoting 服务所使用的 Remoting 端口即是配置文件中配置的 11000 ,而为了提升消息转发的性能，ACMS 并没有使用 Remoting 来转发消息 ,而是直接基于 TCP socket ,其单独使用了一个端口即配置中的 TransferPort 项（12000）。

我们以上文中的第一个模型图的简化版为例来说明跨服务器的消息转发：

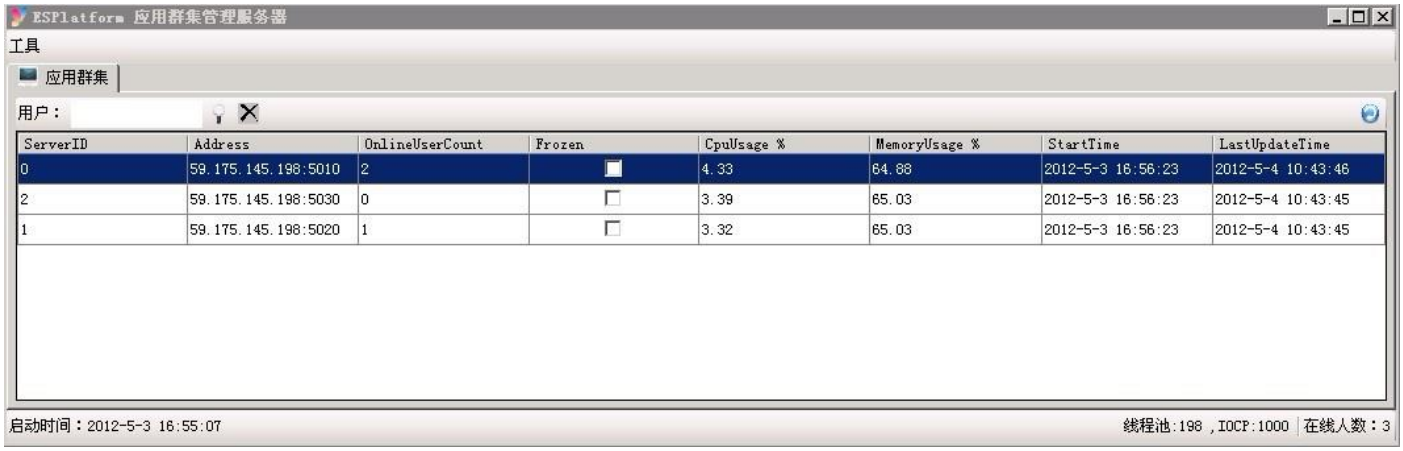


当 Client01 要发消息给 Client02 时 ,假设 Client01 与 Client02 之间不存在 P2P 通道 ,所以消息将提交给 AS01。

AS01 发现 Client02 不在当前的服务器上，所以就将消息提交给 ACMS。然后，ACMS 会查询平台用户管理器，发现 Client02 位于 AS02 上，于是 ACMS 就将消息转发给 AS02，交由 AS02 去发送，这样，Client02 就收到了来自 Client01 的消息。

三. 启动 ACMS

下图是启动 ESPlatform.ACMServer.exe 后运行的主界面：



列表中显示了每个在线的 AS 的基本信息，包括：服务器的 ID、地址、该 AS 上的在线人数、是否处于冻结状态、Cpu 使用率、内存使用率、启动时间、最后一次更新时间等。

ACMS 的主界面还直接提供了查找在线用户和踢人的功能。

通过右键快捷菜单，我们可以将某个 AS 冻结、或者从群集系统中移除。

（1）冻结 AS：当我们不想有新的用户连接到群集系统中的某个 AS 上时，我们可以冻结它。如果 AS 处于冻结状态，表示该服务器不再接收新的用户，但是，现已登陆的用户的一切活动仍然是正常的。

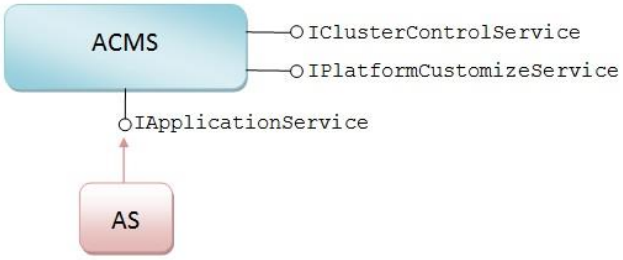
（2）移除 AS：当某个 AS 意外 down 掉（比如，硬件出现问题），我们需要手动将其从平台中移除。在移除 AS 的同时，所有位于该 AS 上的用户也会从平台中清理掉。

当然，ACMS 主界面提供的群集控制功能是非常有限的，但是，ACMS 还暴露了上面提到的两个 Remoting 接口：IClusterControlService 和 IPlatformCustomizeService，在群集系统的外部，我们可以通过这两个接口，更全面的访问和控制群集系统。这将是下一篇文章要详细阐述的内容。

（02）—— 从外部访问群集

本文我们来解决一个常见的问题，那就是在 ESPlatform 群集之外如何访问和控制 ESPlatform 群集了？

在 ESPlatform 群集平台（01）—— 迁移到群集平台一文中，我们提到 ACMS 提供了三个 Remoting 服务接口：IApplicationService、IClusterControlService、IPlatformCustomizeService。IApplicationService 我们已经介绍过了，而 IClusterControlService 和 IPlatformCustomizeService 正是 ESPlatform 群集系统提供给外部系统（如 BL）来访问或控制群集的。



通常我们会这样做，将 ACMS 暴露的 IClusterControlService 和 IPlatformCustomizeService 作为群集系统对外暴

露的唯一通道，外部系统只能通过 ACMS 提供的这两个接口与整个群集系统通信。外部系统不要直接与某个 AS 通信，除非我们有充分的理由，否则，不要轻易破坏这一规则。这一规则的目的在于，尽可能使整个系统（可能是很庞大的、复杂的）保持简洁的、清晰的结构。

一. IClusterControlService 接口

IClusterControlService 接口用于访问群集系统中的 AS 服务器信息或控制群集中的服务器。IClusterControlService 的定义如下：

API：

```
public interface IClusterControlService
{
    /// <summary>
    /// 群集中有新的服务器注册进来。
    /// </summary>
    event CbGeneric<ClusterServerInfo> ServerRegistered;

    /// <summary>
    /// 群集中的某服务已被注销。
    /// </summary>
    event CbGeneric<int> ServerUnregistered;

    /// <summary>
    /// 根据ServerAssignedPolicy，选择群集中合适的服务器。
    /// </summary>
    ServerConfiguration GetServerToLogon();

    /// <summary>
    /// 目标服务器是否正在线？
    /// </summary>
    /// <param name="serverID">目标服务器的ServerID</param>
    /// <returns>在线？</returns>
    bool IsServerOnline(int serverID);

    /// <summary>
    /// 冻结群集中的某服务器。如果群集中的某服务器处于冻结，表示该服务器不再接收新的用户（现已登陆的用户仍然正常）。
    /// </summary>
    /// <param name="serverID">目标服务器的ID</param>
    void FreezeServer(int serverID);

    /// <summary>
    /// 解冻群集中的某服务器。
    /// </summary>
    /// <param name="serverID">目标服务器的ID</param>
    void DefreezeServer(int serverID);

    /// <summary>
    /// 回调测试。如果回调成功，则正常返回，否则将抛出异常。
```

```
        /// </summary>
        /// <param name="serverID">目标服务器的ID</param>
        void TestCallbackServer(int serverID);

        /// <summary>
        /// 从群集列表中移除（异常停止的）服务器。
        /// </summary>
        /// <param name="serverID">将被移除的服务器的ID</param>
        void RemoveServer(int serverID);

        /// <summary>
        /// 获取目标服务器的信息。
        /// </summary>
        ClusterServerInfo GetServer(int serverID);

        /// <summary>
        /// 获取群集中在线的所有服务器。
        /// </summary>
        List<ClusterServerInfo> GetAllServers();
    }
```

1.AS 基础信息与事件

当我们动态地向群集中添加一台 AS 时，ACMS 将会触发 IClusterControlService 接口的 ServerRegistered 事件；同样的，当动态地从群集中移除一台 AS 时，将触发 IClusterControlService 接口的 ServerUnregistered 事件。请注意，如果某台 AS 意外挂掉，则 ACMS 是不知情的，我们必需手动调用 IClusterControlService 接口的 RemoveServer 方法，此方法的执行仍然会触发 IClusterControlService 接口的 ServerUnregistered 事件。

GetServer 用于获取群集中某台 AS 的详细信息，其返回的 ClusterServerInfo 类图如下：

通过返回的 ClusterServerInfo，我们可以知道目标 AS 基本信息：启动时间、IP 端口地址、在线用户数量、当前的 CPU 使用率、内存使用率、是否处于冻结状态。

2.群集分配策略

所谓群集分配策略，就是当一个客户端要连接到 AS 时，我们应该分配群集中的哪台 AS 给他？选择 AS 所采取的策略就是群集分配策略。

AS 内置了三种常见的群集分配策略：轮询、人数最少、CPU 利用率最小。该策略由 ServerAssignedPolicy 枚举定义：

API：

```
public enum ServerAssignedPolicy
{
    /// <summary>
    /// 轮询。
    /// </summary>
    Poll = 0,
    /// <summary>
    /// 选择群集中在线人数最小的那台Server。
    /// </summary>
    MinUserCount ,
    /// <summary>
    /// 选择群集中Cpu利用率最小的那台Server。
    /// </summary>
    MinCpuUsage
}
```

在 ACMS 的配置文件 ESPlatform.ACMServer.exe.config 中，有 key 为 ServerAssignedPolicy 的配置项，我们可以在此指定所采用的 AS 分配策略。

IClusterControlService 接口的 GetServerToLogon 方法将依据 ServerAssignedPolicy 的设置返回恰当的 AS。

当我们想让某台 AS 不再接受新的客户端连接时，可以调用 FreezeServer 方法来冻结它。当 AS 被冻结后，GetServerToLogon 方法将永远不会返回这台 AS。

当然，我们也可以完全自己定义 AS 的分配策略，而忽略 ACMS 提供的内置的分配策略的存在。

假设我们已经采用了 ACMS 内置的分配策略或实现了自定义的分配策略，那么，客户端如何知道自己要登录到哪个 AS 了？需要有个第三方来提供这一查询服务。我们可以简单地做到这一点，比如，发布一个 WebService，客户端先通过访问该 WebService 获取要登录的 AS 的地址，然后再去连接目标 AS，等等。

二. IPlatformCustomizeService 接口

如果，希望从外部发送一个指令给群集中的某台 AS 或在线的某个客户端，该如何做了？ 我们可以使用 ACMS 暴露的 IPlatformCustomizeService 接口。

API：

```
public interface IPlatformCustomizeService
{
    /// <summary>
    /// 向群集中的每台AS发送广播。将被AS的ICallbackHandler的HandleBroadcast处理。
    /// </summary>
    /// <param name="informationType">广播信息类型</param>
    /// <param name="info">广播信息</param>
    void BroadcastInCluster(int informationType, byte[] info);

    /// <summary>
    /// 向应用群集中的某个AS发送信息。如果目标AS不在线，返回false。将被AS的ICallbackHandler的HandleInformation处理。
    /// </summary>
    /// <param name="asID">接收信息的AS的ID。</param>
    /// <param name="informationType">自定义信息类型</param>
    /// <param name="info">信息</param>
```

```
        bool SendToServer(int asID, int informationType, byte[] info);

        /// <summary>
        /// 向应用群集中的某个AS查询信息。如果目标AS不在线，将抛出异常。将被AS的ICallbackHandler的
HandleQuery处理。
        /// </summary>
        /// <param name="asID">被查询的AS的ID。</param>
        /// <param name="informationType">自定义信息类型</param>
        /// <param name="info">信息</param>
        byte[] QueryServer(int asID, int informationType, byte[] info);

        /// <summary>
        /// 向平台上的用户发送自定义信息。如果目标用户不在线，返回false。
        /// </summary>
        /// <param name="clientUserID">接收信息的目标用户ID。</param>
        /// <param name="informationType">自定义信息类型</param>
        /// <param name="info">信息</param>
        bool SendToClient(string clientUserID, int informationType, byte[] info);

        /// <summary>
        /// 向平台上的用户查询信息。如果目标用户不在线，或者超时没有回复，将抛出相应的异常。
        /// </summary>
        /// <param name="clientUserID">接收信息的目标用户ID。</param>
        /// <param name="informationType">自定义信息类型</param>
        /// <param name="info">信息</param>
        byte[] QueryClient(string clientUserID, int informationType, byte[] info);

        /// <summary>
        /// 目标用户是否在线。
        /// </summary>
        bool IsUserOnLine(string userID);

        /// <summary>
        /// 获取目标在线用户的基础信息。
        /// </summary>
        /// <param name="userID">目标用户的ID</param>
        /// <returns>如果目标用户不在线，则返回null</returns>
        UserData GetUserData(string userID);

        /// <summary>
        /// 获取所有在线用户的人数。
        /// </summary>
        int GetOnlineUserCount();

        /// <summary>
        /// 将用户从某个AS上踢出。
        /// </summary>
        void KickOut(string userID);
    }
```

就像 ESPlus 提供的自定义信息功能一样，我们可以从群集外部通过 IPlatformCustomizeService 接口发送自定义信息甚至同步调用给某个 AS 或某个在线客户端，当然，这些信息都是经过 ACMS 进行中转的。接口中每个方法的注释已经描述得很清楚了，这里就不再赘述了。

1.向 AS 发送自定义信息

IPlatformCustomizeService 接口的 BroadcastInCluster、SendToServer、QueryServer 方法用于发送自定义信息或同步调用给 AS，那么 AS 如何来处理这些信息了？

AS 需要实现 ESPlatform.Server.Application.ICallbackHandler 接口来处理来自群集外部的信息。

API：

```
public interface ICallbackHandler
    {
        /// <summary>
        /// 处理ASM发送过来的广播。
        /// </summary>
        /// <param name="informationType">广播信息类型</param>
        /// <param name="info">广播信息</param>
        void HandleBroadcast(int informationType, byte[] info);

        /// <summary>
        /// 处理ASM发送过来的信息。
        /// </summary>
        /// <param name="informationType">信息类型</param>
        /// <param name="info">信息</param>
        void HandleInformation(int informationType, byte[] info);

        /// <summary>
        /// 处理ASM发送过来的查询。
        /// </summary>
        /// <param name="informationType">信息类型</param>
        /// <param name="info">信息</param>
        byte[] HandleQuery(int informationType, byte[] info);
    }
```

ICallbackHandler 接口的三个方法刚好对应 IPlatformCustomizeService 接口的三个方法。我们在实现了 ICallbackHandler 接口后，可以将其实例注入到 ESPlatform.Rapid.RapidServerEngine 的 CallbackHandler 属性上。如此，当 AS 接收到自定义信息时，便会回调 ICallbackHandler 对应的方法来处理了。

2.向客户端发送自定义信息

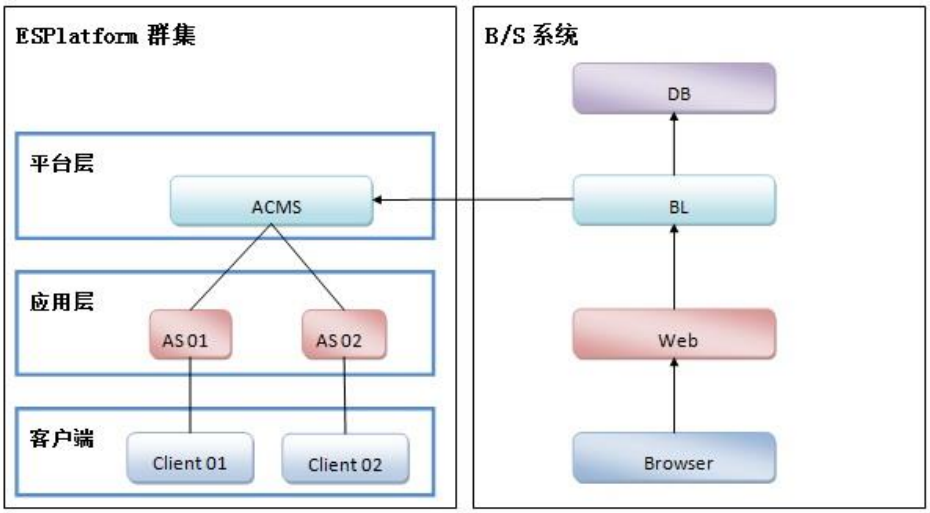
同样的问题，IPlatformCustomizeService 接口的 SendToClient、QueryClient 方法用于发送自定义信息或同步调用给某个在线的客户端，那么客户端如何来处理这些信息了？实际上，客户端不需要实现新的接口，而是通过 ICustomizeHander 接口来统一处理。无论是来自 AS 的信息，还是来自群集外部的信息，都将有 ICustomizeHander 接口来处理。关于 ICustomizeHander 接口的详细介绍，可以参见 ESFramework 开发手册（01） —— 发送和处理信息。

要注意的是，在分配信息类型 informationType 时，来自群集外的自定义信息不要与普通的自定义信息的类型重复就可以了。

三. 例子

下面我们举个简单的例子，来说明外部系统与 ESPlatform 群集的交互。

我们假设有个简单的在线的网络游戏系统，由 B/S 和 C/S 两个子系统组成。B/S 子系统用于实现像用户注册、资料修改、后台管理等业务功能；C/S 子系统则用于实现所有的游戏逻辑，由于在线用户数量巨大，所以我们使用了 ESPlatform 群集平台。整个系统结构简化后如下图所示：



下面我们举两个常见的需求。

1.修改游戏配置参数

为了方便游戏管理员 GM 的操作 ,我们的 B/S 子系统提供了后台网站给管理员进行重要的游戏参数设置。比如，像游戏中的金融平衡系数等，这些参数的值必需提交给每个 AS 生效才可以。对于类似的需求，可用类似如下的流程实现：

（1）GM 通过后台网站将修改参数的请求提交给 Web。

（2）Web 再向 BL 提交。

（3）BL 修改 DB（如果需要的话）成功后，发送自定义信息给 ACMS。（通过 IPlatformCustomizeService 接口的 BroadcastInCluster 方法）

（4）ACMS 在群集中广播自定义信息。

（5）每个 AS 都将回调上述 ICallbackHandler 接口的 HandleBroadcast 方法来使参数设定生效。

2.充值

充值功能一般在 Web 中完成，而在线的客户端却要实时显示最新的余额信息。类似这样的流程也很容易实现：

（1）玩家通过前台网站将充值请求提交给 Web。

（2）支付成功后，Web 向 BL 提交。

（3）BL 修改 DB 成功后，发送自定义信息给 ACMS。（通过 IPlatformCustomizeService 接口的 SendToClient 方法）

（4）ACMS 根据玩家的 ID 找到其所在的 AS，然后将信息转发给该 AS。

（5）AS 收到自定义信息后，在将其转发给目标玩家的客户端。

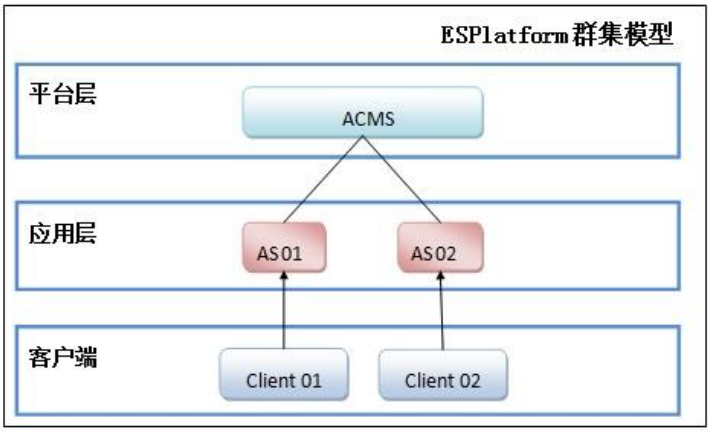
（6）客户端收到自定义信息后，更新内存中的最新余额信息，并更新 UI 显示。

为了方便讨论，我们这里对举例的场景做了很多简化，真正的系统通常要比这里描述的复杂很多。比如，ACMS 可能需要访问数据库，而 AS 可能需要访问 BL 或者缓存服务器，等等。这些扩展设计该如何做，取决于我们实际的项目需求。

（03）—— 部署群集端口开放说明

当我们要把 ESPlatform 群集系统在服务器上部署时，群集管理服务器 ACMS 和应用服务器 AS 分别需要开放哪些端口？以及这些端口开放给谁使用？

我们先回到 ESPlatform 群集模型的结构图简化版，一般在部署时，ACMS 和群集中的多个 AS 会部署在同一个 IDC 机房中的同个网段，使用路由器连接起来。



结合 ESPlatform 群集平台（01）—— 迁移到群集平台 中所描述的，我们可以总结出在部署基于 ESPlatform 的群集系统时，ACMS 和 AS 分别要开放的端口。

1. 群集管理服务器 ACMS 开放的端口

ACMS 使用了两个 TCP 端口：TransferPort 和 RemotingPort。对应着 ACMS 配置文件中设定的两个端口。

（1）TransferPort：

这是一个 TCP Socket 端口，只给 AS 使用，用于在 AS 之间转发消息。

（2）RemotingPort：

这是一个 Remoting 端口，既给 AS 使用，也给群集外的系统访问。

AS 通过该 RemotingPort 向 ACMS 注册、注销、定时报告自己的状态等。

群集外系统通过该 RemotingPort 获得 ACMS 的 IClusterControlService 和 IPlatformCustomizeService 的远程引用，以此来与整个群集进行通信。

2. 应用服务器 AS 开放的端口

AS 使用了两个 TCP 端口，一个 UDP 端口。

（1）TCP 端口 1：

这是一个 TCP Socket 端口，只给客户端使用，客户端与服务器的通信都通过该端口进行。

（2）TCP 端口 2：

这是一个 Remoting 端口，只给 ACMS 回调使用，ACMS 通过该端口向 AS 发送群集控制命令。

该端口在 AS 的配置文件中，对应着<system.runtime.remoting>节点，一般将该端口指定为 0，表示由系统自动分配。如果需要在防火墙上开放该端口，那就不能自动分配了，应手动指定一个值。

（3）UDP 端口：

这是 AS 内部集成的 P2P 服务器监听的 UDP Socket 端口，只给客户端使用，用于协助客户端之间进行 P2P 打洞。UDP 端口的值为上述的 TCP 端口 1 的值加 1。

P2P 服务器是可以独立部署的。具体可参见 ESFramework 使用技巧 —— 部署 P2P 服务器。

ESFramework 使用技巧

（00）—— 使用紧凑的序列化器，数倍提升性能

在分布式通信系统中，网络传递的是二进制流，而内存中是我们基于对象模型构建的各种各样的对象，当我们需要将一个对象通过网络传递给另一个节点时，首先需要将其序列化为字节流，然后通过网络发送给目标节点，目标节点接收后，再反序列化为对象实例。在 ESFramework 体系中，也是遵循同样的规则。

ESFramework 称这些需要经过网络传递的对象称之为协议类（Contract），协议类通常只是一个简单的数据结构封装，用于保存状态的一个哑类（不包含任何业务方法，从 object 继承的除外），有点类似于与数据库中表进行映射的贫血 Entity。基于 ESFramework 的分布式系统使用这些协议类实例进行数据交换。当我们使用 Rapid 引擎时，我们会经常使用 ESPlus.Application.CustomizeInfo.Passive.ICustomizeOutter 接口的 Send 等方法发送二进制自定义信息，这个二进制信息通常是系统中的某个业务对象的序列化结果，而这个序列化过程我们必须自己来实现。

一.序列化方式的两种选择

为了将业务对象转换为二进制流，大家通常有两种方案可以选择：使用.NET 自带的二进制序列化器，或，先将业务对象转换为字符串（比如 xml），再将结果用类似 UTF8 进行编码得到字节流。 这两种方案都有缺陷。

1..NET 自带的二进制序列化器

使用.NET 自带的二进制序列化器对我们来说是最方便的，因为只要直接调用 API 就好了，不需要我们自己动手做任何动作。但是，缺陷也很明显：

（1）强侵入性。

首先，能够序列化的类必须加上 Serializable 标签，且.NET 自带的二进制序列化器需要绑定被序列化的类的命名空间和程序集。比方说，将 windows 里序列化一个协议对象得到的结果传到 Silverlight 中，是无法完成反序列化的，即使 Silverlight 中也有完全相同的协议类定义。这也说明，如果通信的客户端是用 Silverlight 开发的，肯定是不能使用.NET 自带的二进制序列化器的。

（2）序列化结果臃肿，其 size 巨大。

在巨大并发高性能的分布式通信系统中，这将降低通信消息，并浪费大量的带宽，是不可忍受的。

（3）效率低下。

.NET 自带的二进制序列化器基于反射（Relection）的机制工作，比如读取类型的信息、读取/设置字段的值等都是通过反射来完成的。如果每秒仅仅序列化百千个对象，可能还可以应付；但是如果每秒需要序列化几万、甚至几十万个对象，就不堪重负了。

（4）加密困难。

由于.NET 自带的二进制序列化器在序列化协议对象时，会反射读取对象的内部成员（private），而如果定义协议的 dll 经过加密后，private 成员的名称通常会被混淆成随机的名称，这样就要求通信的各方都使用同一个加密的 dll，否则，协议对象的反序列化可能会失败。

2.通过 string 进行中转

为了解决类似 Silverlight 客户端与服务端通信的序列化统一的问题，我们可以使用 string 作为中转，而无论是否为 Silverlight 环境，对同一字符串进行相同格式（如 UTF8）的编码，得到的字节流肯定是一样的。反过来，同样的字节流在不同的环境中使用相同的解码器进行解码得到的字符串也一样。这是一种可行的方案，但是缺点也是有的：

（1）我们需要自己打造协议对象与字符串之间的相互转换 —— 这可能是一个费时费神的又容易出错的工作。

（2）序列化之后的结果的 Size 取决于我们协议对象与字符串之间相互转换时所采用的规则，同样也取决于我们的耐心程度 —— 为了使结果的 Size 更小，我们要动更多的脑筋。

当然，现在也有现成的将对象与字符串想换转换的工具，比如 JSON 和 XML。就一般而言，使用 JSON 转换协议对象得到的结果比使用 xml 要小很多。

（3）同.NET 自带的二进制序列化器，对象属性值的读取/设置、对象的创建等通常也是基于反射的，所以，效率同样存在问题。

二.第三种方案

ESPlus 提供了紧凑的二进制序列化器：ESPlus.Serialization.CompactPropertySerializer。

（1）CompactPropertySerializer 基于 Emit 技术和缓存技术构建，且避免了反射带来的开销，所以效率大大提升。

（2）CompactPropertySerializer 内部使用了简练、紧凑的序列化格式和规则，使得序列化的结果 Size 更短小。

（3）将被序列化的类不需要增加任何标签，且不依赖于命名空间和程序集，只要类的定义完全一致，CompactPropertySerializer 就可以正常工作。真正的弱侵入性。

（4）ESFramework.SL 也提供了完全一样的 CompactPropertySerializer ，所以基于第 3 点，服务端与 Silverlight 客户端就可以协同工作了。

（5）只要采用 CompactPropertySerializer 的序列化格式，.NET 服务端可以与任何其它语言（如 C++、JAVA 等）构建的客户端协同工作。

CompactPropertySerializer 解决了前面提出的几个问题，当然，它也不是全能的，使用它也有一些限制，下面我们即将讲到。

三.如何使用 CompactPropertySerializer

ESPlus.Serialization.CompactPropertySerializer 和 ESFramework.SL.Serialization.CompactPropertySerializer 是 ESFramework 提供的分别用于桌面应用和 Sivlerlight 客户端的二进制序列化器。它们的实现几乎一模一样，所以，使用时要注意的方面也是相同的：

（1）CompactPropertySerializer 支持类和结构的序列化，但是被序列化的类或结构必须有默认的构造函数（Ctor）。

（2）CompactPropertySerializer 只序列化那些可读写的属性，如果一个属性仅仅是只读或只写的，那么该属性不会被序列化。这也是 CompactPropertySerializer 名称中 Property 的含义。

（3）CompactPropertySerializer 支持的类型：基础数据类型(如 int、long、bool 等)、string、byte[]，以及由这些类型构成的 class 或 struct。

（4）支持多层嵌套 —— 即被序列化的 class 中可以包含别的类型的对象，只要每一个被嵌入的对象最后都是由基础数据类型构成的。

（5）除 byte[]/List<>/Dictionary<,>泛型外，不支持其它的集合类型。

（6）不支持循环引用。如果存在循环引用，序列化时将导致死循环。

正如本文开始提到的，在通信系统中用到的协议类都是一些最简单的仅仅包含数据的“哑类”，所以，上面的限制对我们在设计协议类时是没有什么约束的。尽可能地使用简单的数据类型，然后将需要序列化的字段通过可读写的属性暴露就 OK 了。

CompactPropertySerializer 包括了序列化和反序列化的两个基本方法：

API：

```
byte[] Serialize<T>(T obj);
T Deserialize<T>(byte[] buff, int startIndex) where T : new();
```

方法含义很明显，不解释了。另外，CompactPropertySerializer 采用了 Singleton，我们可以在程序中直接使用这个 Singleton ，通过 CompactPropertySerializer.Default 属性获得该 Singleton 实例的引用。

可以借由 ContractFormatGenerator 来查看 CompactPropertySerializer 对某个对象执行序列化后得到的流的具体格式，这在进行跨平台开发时，非常有用。

四.试试 CompactPropertySerializer 的性能和效率

我们以文件传送中要使用到的协议类 BeginSendFileContract 为例，BeginSendFileContract 定义如下：

View Code

之所以加上[Serializable]标签，是因为下面测试.NET 自带的二进制序列化器需要用到，正式的 BeginSendFileContract 定义是没有这个标签的。

正如大多数的协议类一样，这个类仅仅包含几个简单类型的属性，现在我们来对比一下.NET 自带的二进制序列化器与 CompactPropertySerializer 的表现。

比较序列化结果的大小：

API：

```
BeginSendFileContract contract = new BeginSendFileContract();
byte[] result1 = CompactPropertySerializer.Default.Serialize<BeginSendFileContract>(contract);
byte[] result2 = ESBasic.Helpers.SerializeHelper.SerializeObject(contract);
```

ESBasic.Helpers.SerializeHelper 就是对.NET 自带的二进制序列化器的简化封装。执行的结果如下：

result1 的长度为：32

result2 的长度为：242

.Net 自带序列化器的结果是 CompactPropertySerializer 结果的 7-8 倍，如果为 contract 的一些 string 类型的字段赋有意义的值，这个倍数会稍微降一点；如果这个 contract 的定义包含了更多的要序列化的属性，那么这个倍数还会继续提高。不管怎么样，这个比例都是很吓人的，所以在高频的通信系统中，相比于使用.Net 自带序列化器，采用 CompactPropertySerializer 可以节省大量的带宽。

比较性能：

我们分别运行两个序列化 100 万次，看所要的时间：

API：

```
BeginSendFileContract contract = new BeginSendFileContract();
Stopwatch stopwatch = new Stopwatch();
```

```
    stopwatch.Start();
    for (int i = 0; i < 1000000; i++)
    {
        byte[] result1 =
CompactPropertySerializer.Default.Serialize<BeginSendFileContract>(contract);
    }
    stopwatch.Stop();
    double span1 = stopwatch.ElapsedMilliseconds;

    stopwatch.Reset();
    stopwatch.Start();
    for (int i = 0; i < 1000000; i++)
    {
        byte[] result2 = ESBasic.Helpers.SerializeHelper.SerializeObject(contract);
    }
    stopwatch.Stop();
    double span2 = stopwatch.ElapsedMilliseconds;
```

运行结果如下：

span1：5324ms

span2：17249ms

CompactPropertySerializer 比.Net 自带的序列化器快 3 倍以上，优势不言而喻。

大家可以参考上面的 demo，写更多的测试程序，来测试更多的内容，包括它们在反序列化方面的表现的比较。

对于一般的通信应用，使用.Net 自带的二进制序列化器也许就够用了，不会有太大的影响，但是如果在类似 MMORPG、视频/音频会议等等需要高频、高性能的通信系统中，.Net 自带的二进制序列化器就不是最好的选择了。如果使用 ESFramework 来构建你的分布式通信应用，那就可以从 CompactPropertySerializer 得到更多的帮助。

（01）—— 实现离线消息

在 ESFramework 开发手册（01）——发送和处理信息一文中，我们介绍了如何使用 ESPlus.Application.CustomizeInfo 命名空间的组件来发送和处理自定义消息。而在实际的项目中，需要实现离线消息的功能是一个常见的需求，也有很多客户来咨询如何做才能实现离线消息，所以，在这里，我们简单介绍一下使用 ESFramework/ESPlus 实现离线消息的原理与步骤。

一.如何截获离线消息

用户间发送二进制信息采用的是 ESPlus.Application.CustomizeInfo.Passive.ICustomizeOutter 接口的 Send 方法，如：

API：

```
    ///<summary>
    ///向在线用户targetUserID发送二进制信息。如果目标用户不在线，则服务端会调用
    ICustomizeInfoBusinessHandler.OnTransmitFailed方法来通知应用程序。
    ///</summary>
    ///<paramname="targetUserID">接收消息的目标用户ID</param>
    ///<paramname="informationType">自定义信息类型</param>
    ///<paramname="info">二进制信息</param>
    voidSend(stringtargetUserID,intinformationType,byte[]info);
```

发送消息既可以通过 P2P 通道发送，也可以通过服务器中转。当目标用户不在线时，P2P 通道肯定是不存在的，所以消息一定是提交到服务器。服务器接收到要转发的 P2P 消息时，判断目标用户是否在线，如果在线，则直接转发；否则，框架会触发 ESPlus.Application.CustomizeInfo.Server.ICustomizeController 接口的 TransmitFailed 事件：

API：

```
///<summary>
///当因为目标用户不在线而导致服务端转发自定义信息失败时，将触发该事件。参数为转发失败的信息。
///</summary>
eventCbGeneric<Information>TransmitFailed;
```

我们只要预定 ICustomizeController 接口的这个事件就可以监控到所有的离线消息了。

二.离线消息的管理

截获到离线消息后，我们可能需要将其存到数据库（或其它地方），然后，等到目标用户上线的时候，再从数据库中提取属于该用户的离线消息发送给他即可。

首先，我们需要对离线消息做一个封装——OfflineMessage：

API：

```
   [Serializable]
    public class OfflineMessage
    {
        #region Ctor
        public OfflineMessage() { }
        public OfflineMessage(string _sourceUserID, string _destUserID, int _informationType, byte[]
info)
        {
            this.sourceUserID = _sourceUserID;
            this.destUserID = _destUserID;
            this.informationType = _informationType;
            this.information = info;
        }
        #endregion

        #region SourceUserID
        private string sourceUserID = "";
        /// <summary>
        /// 发送离线消息的用户ID。
        /// </summary>
        public string SourceUserID
        {
            get { return sourceUserID; }
            set { sourceUserID = value; }
        }
        #endregion

        #region DestUserID
        private string destUserID = "";
        /// <summary>
        /// 接收离线消息的用户ID。
        /// </summary>
```

```
        public string DestUserID
        {
            get { return destUserID; }
            set { destUserID = value; }
        }
        #endregion

        #region InformationType
        private int informationType = 0;
        /// <summary>
        /// 信息的类型。
        /// </summary>
        public int InformationType
        {
            get { return informationType; }
            set { informationType = value; }
        }
        #endregion

        #region Information
        private byte[] information;
        /// <summary>
        /// 信息内容
        /// </summary>
        public byte[] Information
        {
            get { return information; }
            set { information = value; }
        }
        #endregion

        #region Time
        private DateTime time = DateTime.Now;
        /// <summary>
        /// 服务器接收到要转发离线消息的时间。
        /// </summary>
        public DateTime Time
        {
            get { return time; }
            set { time = value; }
        }
        #endregion
    }
```

接下来，我们定义 IOfflineMessageManager 接口，用于管理离线消息：

API：

```
    /// <summary>
    /// 离线消息管理器。
    /// </summary>
```

```
    public interface IOfflineMessageManager
    {
        /// <summary>
        /// 存储离线消息。
        /// </summary>
        /// <param name="msg">要存储的离线消息</param>
        void Store(OfflineMessage msg);

        /// <summary>
        /// 提取目标用户的所有离线消息。
        /// </summary>
        /// <param name="destUserID">接收离线消息用户的ID</param>
        /// <returns>属于目标用户的离线消息列表，按时间升序排列</returns>
        List<OfflineMessage> Pickup(string destUserID);
    }
```

实现这个接口，我们便可以将离线消息存储到数据库或文本或网络等等，然后等到需要时再次从中提取。

三.存储离线消息

有了 IOfflineMessageManager 接口，我们便可以处理 ICustomizeController 接口的 TransmitFailed 事件了：

privateIOfflineMessageManagerofflineMessageManager=.....;

API：

```
        private IOfflineMessageManager offlineMessageManager = ......;
        public void OnTransmitFailed(Information information)
        {
            OfflineMessage msg = new OfflineMessage(information.SourceID, information.DestID,
information.InformationType, information.Content);
            this.offlineMessageManager.Store(msg);
        }
```

我们也许并不需要将所有的离线消息都存储起来，有些不重要的离线消息可以丢弃，而只保存那些我们关心的消息。这只需要在存储消息之前加一个条件判断进行过滤即可。

四.提取并发送离线消息

我们已经知道，可以通过 IUserManager 的 SomeOneConnected 事件来得知某个用户上线了，于是，我们可以在该事件处理函数中，提取属于该用户的离线消息并一一发送给他。我们通过类似下面的代码来做到这一点。

API：

```
    public class OfflineMessageBridge
    {
        #region UserManager
        private IUserManager userManager;
        public IUserManager UserManager
        {
            set { userManager = value; }
        }
        #endregion

        #region OfflineMessageManager
```

```
        private IOfflineMessageManager offlineMessageManager;
        public IOfflineMessageManager OfflineMessageManager
        {
            set { offlineMessageManager = value; }
        }
        #endregion

        #region CustomizeController
        private ICustomizeController customizeController;
        public ICustomizeController CustomizeController
        {
            set { customizeController = value; }
        }
        #endregion

        public void Initialize()
        {
            this.userManager.SomeOneConnected += new
CbGeneric<ESFramework.Server.UserManagement.UserData>(userManager_SomeOneConnected);
        }

        void userManager_SomeOneConnected(ESFramework.Server.UserManagement.UserData userData)
        {
            List<OfflineMessage> list = this.offlineMessageManager.Pickup(userData.UserID);
            if (list != null && list.Count > 0)
            {
                foreach (OfflineMessage msg in list)
                {
                    byte[] bMsg = CompactPropertySerializer.Default.Serialize<OfflineMessage>(msg);
                    this.customizeController.Send(msg.DestUserID, InfoTypes.OfflineMessage, bMsg);
                }
            }
        }
    }
```

当用户上线时，会将属于他的离线消息按照时间的顺序一一发送给他（ demo 中，离线消息的类型为 InfoTypes.OfflineMessage，您可以自定义它）。当然，你也可以将属于他的所有离线消息打成一个包，一次性发送也可以。如果是这样，你就需要再增加一条自定义的信息类型和相关的协议类了。

要注意的是，在服务端触发 SomeOneConnected 事件时，虽然对应的客户端已经完成了登录，但是，其可能还未完成全部的初始化动作，此时，客户端可能就无法正确地处理收到的离线消息。如何解决这种问题了？可以换种稳妥的方案：服务端并不预定 SomeOneConnected 事件，而客户端了，在完成初始化之后，向服务端发送一个请求消息（ InfoTypes.GetOfflineMessage ），服务端在接收到这个请求消息后，才将离线消息发送给它。

五.在客户端处理离线消息

根据上面第三点的描述，我们知道，当客户端上线时，客户端会接收到服务端发过来的属于当前客户端的离线消息。所以，客户端在实现的 ICustomizeHandler 接口的 HandleInformation 方法需要增加对离线消息的处理：

API：

```
    public void HandleInformation(string sourceUserID, int informationType, byte[] info)
```

```
    {
        if (informationType == InfoTypes.OfflineMessage)
        {
            OfflineMessage msg = CompactPropertySerializer.Default.Deserialize<OfflineMessage>(info,
0);
            this.HandleInformation(msg.SourceUserID, msg.InformationType, msg.Information);
            return;
        }
            ..........................
    }
```

Demo 处理中，采用了一种偷懒的方式，将离线消息解析后，直接再次递归调用了 HandleInformation 方法——就好像是现在刚收到了来自其它客户端的消息一样，忽略了原始的发送时间（OfflineMessage 的 Time 属性）。如果不能忽略原始的发送时间，您可以采取更合适的处理方式。

六.小结

从上面可以看出，基于 ESFramework/ESPlus 实现离线消息策略是相当简单的，最主要的焦点有两个：第一是可以通过处理 ICustomizeController 接口的 TransmitFailed 事件来截获到所有的离线消息；第二是通过 IUserManager 的 SomeOneConnected 事件就能知道用户上线的时刻。

有的朋友可能会问离线文件又该怎么实现了？实际上也是同样的原理，只不过要多用到 ESPlus.Application.FileTransfering 命名空间下的一些类来完成文件的收发功能，这个以后我们再介绍。

本文只是实现离线消息的一个简单示例，在实际的应用中，可能需要做更多的工作来满足项目的具体需要，这里就不再一一赘述了。

（02）—— 实现离线文件

所谓“离线文件”，就是当接收者不在线时，发送者先把文件传送给服务端，在服务器上暂时保存，等接收者上线时，服务端再把文件发送给他。当然，要想实现离线文件的功能，其最基本的前提是要先实现传送文件的功能，我们以 ESFramework 提供的传送文件的功能为基础，在其之上一步步完成一个基本的离线文件功能。

下面我们就用户在使用离线文件时，按各个动作发生的先后顺序，介绍程序方面与之对应的设计与实现。

1.客户端发送离线文件

当用户选择好一个文件，并点击“发送离线文件”按钮时，其目的是要将这个文件传送给服务端，这可以直接使用 IFileOutter 的 BeginSendFile 方法：

API：

```
    /// <summary>
    /// 发送方准备发送文件（夹）。
     /// </summary>
    /// <param name="accepterID">接收文件（夹）的用户ID</param>
    /// <param name="fileOrDirPath">被发送文件（夹）的路径</param>
    /// <param name="comment">其它附加备注。如果是在类似FTP的服务中，该参数可以是保存文件（夹）的路
径</param>
    /// <param name="projectID">返回文件传送项目的编号</param>
    void BeginSendFile(string accepterID, string fileOrDirPath, string comment, out string projectID);
```

如果将参数 accepterID 传入 null，表示文件的接收者就是服务端。那么我们要如何区分，这不是一个最终由服务端接收的文件，而是要传给另一个用户的离线文件了？这里，我们可以巧用 comment 参数，比如，comment 参数如果为 null，就表示普通的上传文件；comment 不为 null，就表示一个离线文件，并且其值就是文件最终接收者的 ID。（当然，如果在你的项目中，comment 参数已经有了其它用途，我们可以进一步扩展它，加上一些标签，使其能够标志出离线文件）。

下面这个调用示例，就是将 Test.txt 文件离线发送给 aa01。

stringfilePath=...;//要发送文件的路径

API：

```
string filePath = ...; //要发送文件的路径
string projectID = null;
fileOutter.BeginSendFile(null, filePath, "aa01", out projectID);
```

2.服务端接收离线文件

客户端调用 BeginSendFile 方法请求发送文件后，服务端会触发 IFileController 的 FileRequestReceived 事件。同理，我们判断该事件的 comment 参数，当其不为 null 时，表示是个离线文件。在答复客户端同意接收文件之前，我们需要先将离线文件的相关信息保存起来，这里我们使用 OfflineFileItem 类来封装这些信息。

API：

```
/// <summary>
/// 离线文件条目
/// </summary>
public class OfflineFileItem
{
    /// <summary>
    /// 条目的唯一编号，数据库自增序列，主键。
    /// </summary>
    public string AutoID { get; set; }

    /// <summary>
    /// 离线文件的名称。
    /// </summary>
    public string FileName { get; set; }

    /// <summary>
    /// 文件的大小。
    /// </summary>
    public ulong FileLength { get; set; }

    /// <summary>
    /// 发送者ID。
    /// </summary>
    public string SenderID { get; set; }

    /// <summary>
    /// 接收者ID。
    /// </summary>
    public string AccepterID { get; set; }
```

```
        /// <summary>
        /// 在服务器上存储离线文件的临时路径。
        /// </summary>
        public string RelayFilePath { get; set; }
    }
```

有了 OfflineFileItem 的定义之后，我们就可以处理 IFileController 的 FileRequestReceived 事件了。

API：

```
   rapidServerEngine.FileController.FileRequestReceived += new
CbFileRequestReceived(fileController_FileRequestReceived);
   ObjectManager<string, OfflineFileItem> offlineFileItemManager = new ObjectManager<string,
OfflineFileItem>(); //可以把ObjectManager类看作一个线程安全的Dictionary。
    void fileController_FileRequestReceived(string projectID, string senderID, string fileName, ulong
totalSize, ResumedProjectItem resumedFileItem, string comment)
    {
       string saveFilePath = "......" ;//根据某种策略得到存放文件的路径
       if (comment != null) //根据约定，comment不为null，表示为离线文件,其值为最终接收者的ID。
        {
            string accepterID = comment;
            OfflineFileItem item = new OfflineFileItem();
            item.AccepterID = accepterID;
            item.FileLength = totalSize;
            item.FileName = fileName;
            item.SenderID = senderID ;
            item.RelayFilePath = saveFilePath;
            offlineFileItemManager.Add(projectID, item);
        }

        //给客户端回复同意，并开始准备接收文件。
        rapidServerEngine.FileController.BeginReceiveFile(projectID ,saveFilePath);
    }
```

上面的代码做了三件事情：

（1）根据某种策略得到存放文件的路径。

（2）创建一个离线文件信息条目，保存在内存中。

（3）回复客户端，并准备接收文件。

需要重点说明的是第一点，对于一般的小型项目，在服务端我们可以将所有的离线文件存放在当前服务器的某个目录下；但是对于大型项目，一般需要使用 DFS（分布式文件系统）来存储这些临时的离线文件。

客户端收到服务器的回复后，会正式开始传送文件，如果传送过程中，因为某种原因导致传送中断，则服务端会触发 IFileController.FileReceivingEvents 的 FileTransDisruptted 事件。在该事件处理函数中，我们从内存中移除对应的离线文件信息条目：

API：

```
rapidServerEngine.FileController.FileReceivingEvents.FileTransDisruptted += new
CbGeneric<TransferingProject, FileTransDisrupttedType>(fileReceivingEvents_FileTransDisruptted);

    void fileReceivingEvents_FileTransDisruptted(TransferingProject project, FileTransDisrupttedType
type)
    {
```

```
        offlineFileItemManager.Remove(project.ProjectID);
    }
```

如果文件正常传送完毕，则服务端会触发 IFileController.FileReceivingEvents 的 FileTransCompleted 事件。此时，我们将对应的离线文件信息条目从内存转移存储到数据库中，以防止服务器重启时导致信息丢失：

API：

```
  rapidServerEngine.FileController.FileReceivingEvents.FileTransCompleted += new
CbGeneric<TransferingProject>(fileReceivingEvents_FileTransCompleted);

    IOfflineFilePersister offlineFilePersister = ......;
    void fileReceivingEvents_FileTransCompleted(TransferingProject project)
    {
        OfflineFileItem item = offlineFileItemManager.Get(project.ProjectID);
        offlineFilePersister.Add(item);
        offlineFileItemManager.Remove(project.ProjectID);
    }
```

我们设计 IOfflineFilePersister 接口，用于与数据库中的 OfflineFileItem 表交互。

API：

```
    public interface IOfflineFilePersister
    {
        /// <summary>
        /// 将一个离线文件条目保存到数据库中。
        /// </summary>
        void Add(OfflineFileItem item);

        /// <summary>
        ///  从数据库中删除主键值为ID的条目。
        /// </summary>
        void Remove(string id);

        /// <summary>
        /// 从数据库中提取接收者为指定用户的所有离线文件条目。
        /// </summary>
        List<OfflineFileItem> GetByAccepter(string accepterID);
    }
```

我们可以使用 ADO.NET 或者 EntityFramework 实现上述接口。

3.服务端发送离线文件给最终接收者

当真正的接收者上线时，服务端要把相关的离线文件发送给他。通过预定 UserManager 的 SomeOneConnected 事件，我们知道用户上线的时刻。

API：

```
rapidServerEngine.UserManager.SomeOneConnected += new
CbGeneric<UserData>(userManager_SomeOneConnected);

    void userManager_SomeOneConnected(UserData data)
    {
        List<OfflineFileItem> list = offlineFilePersister.GetByAccepter(data.UserID);
        if (list != null)
```

```
        {
            foreach (OfflineFileItem item in list)
            {
                string projectID = null ;
                rapidServerEngine.FileController.BeginSendFile(item.AccepterID, item.RelayFilePath,
item.SenderID, out projectID);
                offlineFilePersister.Remove(item.AutoID);
                File.Delete(item.RelayFilePath);
            }
        }
    }
```

上面的代码做了三件事情：

（1）从数据库中查找所有接收者为登录用户的离线文件信息条目。

（2）将离线文件逐个发送给这个用户

（3）从数据库中删除相应的条目，从磁盘上删除对应的离线文件。

实际上，第（3）点我们可以延迟到文件发送完成时，才执行删除操作。这样，就可以在发送万一意外中断时，使得重新发送成为可能。

客户端接收到服务端的发送文件请求时，会触发 IFileOutter 的 FileRequestReceived 事件，此时也可以根据 comment 参数的内容，来判断其是否为离线文件。后续的步骤的实现就相当容易了，这里就不再赘述了。

本文简洁地描述了实现离线文件功能的主要思路和基本模型，在实际的项目开发时，可以根据具体的需求在这个模型的基础上，进一步完善，包括很多细节和异常处理都需要加入进来。

（03）—— 信息处理，分而治之

ESFramework 开发手册系列文章已经详细介绍了如何使用 ESPlus 提供的 ESPlus.Application.CustomizeInfo 空间来发送和处理自定义信息，而且，在我们在前面介绍的 demo 中，也展示了如何定义信息类型、信息协议，以及如何实现 ICustomizeHandler 来处理接收到的信息。在一般业务简单的系统中，我们完全可以像 demo 一样，在一个 CustomizeHandler 类中处理所有的信息，将所有的业务逻辑集中在这一个地方。但是，当业务逐渐变得复杂时，你会发现，CustomizeHandler 类会变得越来越大，而且有很多关联不大的业务逻辑也纠缠在了一起。根据“低耦合、高内聚”的设计原则，我们需要对这个变得复杂的 CustomizeHandler 进行拆分，将一个 CustomizeHandler 拆分为多个高内聚低耦合的类，对收到的信息进行分类，分而治之。

一.分而治之的设计阶段

就像刚才提到的，分而治之的所依据的最根本原则是面向对象的基本设计理念 —— 高内聚、低耦合。

在实际的项目中，高内聚、低耦合所针对的分析目标就是我们的业务逻辑， 所以，对 CustomizeHandler 进行拆分，实际上是对业务逻辑进行拆分。再进一步，那些将被处理的自定义信息，实际上是业务逻辑类型的一个侧面的展示，所以，归根到底，在编码时，最后就是对自定义信息的类型进行拆分。

假设某个项目的主要业务逻辑可以拆分为 A、B、C 三类，那么，自定义信息也可以分为 A、B、C 三类，我们的经验是这样的，将不同类别的信息类型的值（整数）划归到不同的整数段。比如，A 类型的自定义信息的类型值为 0-100，B 类型为 101-200，C 类型为 201-300，当我们要在某类业务逻辑中增加一个信息类型时，就要在对应的数值范围内增加一个数值。这样处理之后，当我们接收到一个自定义信息，根据其类型就可以判断出它是属于哪类业务的了。

在做系统设计时，我们的设计师通常会将所有的信息类型整理成一个“协议类型”文档并将其定义放到一个 dll 中，服务端和客户端开发人员都使用这个 dll 的定义，并遵循文档中的信息类型的规范描述。比如，针对上面的示例可以设计类似如下的“协议类型”文档：

业务逻辑分类	信息类型	名称	说明
A 类 （0 －100）	1	订购	…
	2	查询订单	…
	3	取消订单	…
	…	…	…
B 类 （101 －200）	101	充值	…
	102	取款	…
	103	转账	…
	…	…	…
C 类 （201 －300）	201	查询消费记录	…
	202	查询收入记录	…
	…	…	…

二.分而治之的实现阶段

在将自定义信息分类并完成了信息的格式约定后，就可以实现信息处理器了。针对 A、B、C 三类业务，理所当然地，我们会实现三个信息处理器分别与之对应，假设命名为 ACustomizeHandler、BCustomizeHandler、CCustomizeHandler。现在的问题是，实现了这几个处理器之后，如何将它们挂接到 ESFramework/ESPlus 框架上了？幸运的是，ESFramework/ESPlus 为分而治之这种策略提供了完美的支持，我们不需要再手动去映射信息类型与对应的处理器。

ESPlus.Application.CustomizeInfo 命名空间提供了 IIntegratedCustomizeHandler 接口 —— 可被集成的处理器接口，其定义如下所示：

API：

```
///<summary>
/// 能够被ComplexCustomizeHandler集成的ICustomizeHandler。
///</summary>
public interface IIntegratedCustomizeHandler : ICustomizeHandler
{
    ///<summary>
    /// 当前的处理器能否处理目标类型的自定义信息。
    ///</summary>
    ///<param name="informationType">自定义信息的类型</param>
    ///<returns>能处理？</returns>
    bool CanHandle(int informationType);
}
```

IIntegratedCustomizeHandler 从 ICustomizeHandler 继承，说明它可以做与 ICustomizeHandler 完全一样的事情，只不过，它处理的是整个业务逻辑的一个子集。其增加的 CanHandle 方法用于说明当前处理器能处理哪些自定义信息。ACustomizeHandler、BCustomizeHandler、CCustomizeHandler 只要实现 IIntegratedCustomizeHandler 接口就可以了。处理器实现新加的 CanHandle 方法很简单，比如 BCustomizeHandler 实现 CanHandle 的代码如下所示：

API：

```
public bool CanHandle(int informationType)
{
    return informationType >= 101 && informationType <= 200;
}
```

在实现完了各个业务处理器之后，接下来就需要将它们合成起来，并挂接到 ESFramework/ESPlus 框架上。

三.分而治之的合成阶段

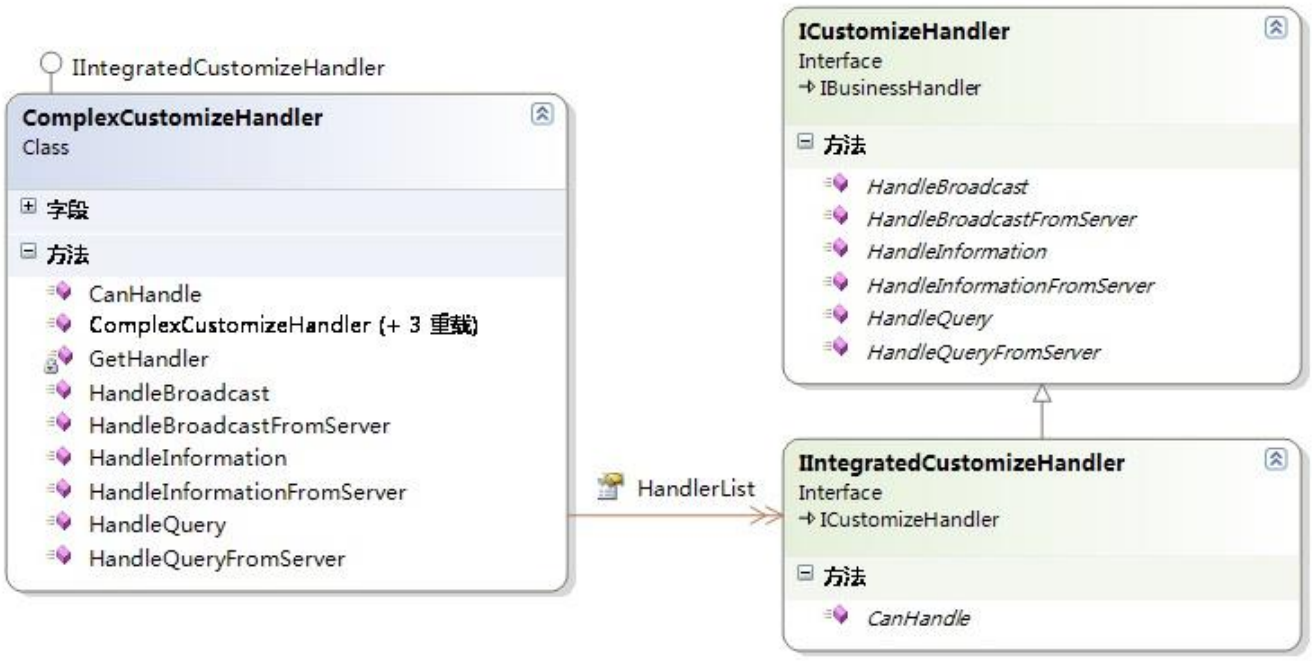
先分后合，分而治之的最后阶段是“合”，只有将 ACustomizeHandler、BCustomizeHandler、CCustomizeHandler 统合起来，才能形成一个完整的业务处理器以处理接收到的所有自定义信息。

ESPlus.Application.CustomizeInfo 命名空间提供了 ComplexCustomizeHandler 类，它是一个综合处理器，相当于一个包装，可以把 ACustomizeHandler、BCustomizeHandler、CCustomizeHandler 综合在一起，并且 ComplexCustomizeHandler 又实现了 IIntegratedCustomizeHandler 接口，这表明了两点：

ComplexCustomizeHandler 实现了 IIntegratedCustomizeHandler 接口，而 IIntegratedCustomizeHandler 又是继承自 ICustomizeHandler 接口，所以可以将其直接挂接到 ESFramework/ESPlus 框架。

ComplexCustomizeHandler 实现了 IIntegratedCustomizeHandler 接口，表明其可以再度被其它的 ComplexCustomizeHandler 集成。就像在一个巨型的系统中，业务逻辑可以被逐级向下拆分，最后可以通过 ComplexCustomizeHandler 逐级向上合成。

ComplexCustomizeHandler 的实现原理很简单，它只是将接收到的自定义信息分派给正确的处理器去处理，而自己并不参与任何实际的业务过程。其类图如下所示：



针对上面的示例，我们将 ACustomizeHandler、BCustomizeHandler、CCustomizeHandler 的实例放到 ComplexCustomizeHandler 的 HandlerList 中，并且将 ComplexCustomizeHandler 对象注入到 RapidPassiveEngine 和 RapidServerEngine 的 Initialize 方法中，即可挂接到 ESFramework/ESPlus 框架。

四.更多说明

虽然，ESFramework/ESPlus 为分而治之这种策略提供了很好的支持，但这并不是实现分而治之策略的唯一的模式。您完全可以抛开 IIntegratedCustomizeHandler 和 ComplexCustomizeHandler，按照自己的习惯和方式，来拆分业务逻辑并进行合成，最后也会殊途同归——只要我们遵循了“低耦合、高内聚”这一最根本的设计原则。

（04）—— 署 P2P 服务器

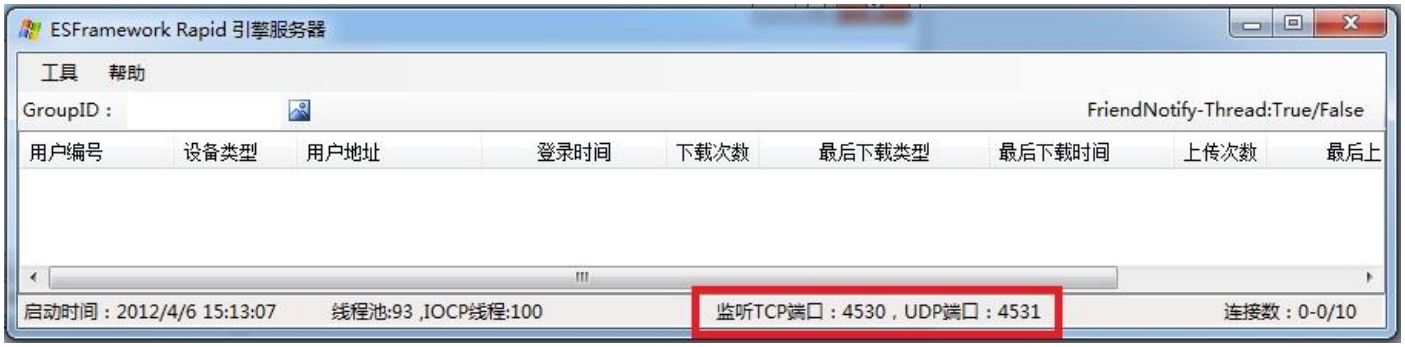
可靠的 P2P 通信功能是 ESFramework/ESPlus 提供的一个基础而又强大的功能，ESFramework 开发手册（04）——可靠的 P2P 详细描述了在客户端如何使用框架提供的这一武器。本文我们则将介绍的重点放到 P2P 服务端。

P2P 服务器用于协助客户端之间创建 P2P 通道。在 ESPlus3.0 以前的版本中，P2P 服务器是集成在服务端 Rapid 引擎中的。在 2012.04.23 最新发布的 3.0 版本中，这种模式依然被支持，而且，ESPlus 又提供了另外一种部署模型：独立部署 P2P 服务器。

一.集成部署 P2P 服务器

1.服务端 IRapidServerEngine 有一个 UseAsP2PServer 属性，用于指示服务端是否同时作为 P2P 服务器运行。如果将该属性设置为 true，然后启动服务端，P2P 服务器就会被集成在服务端中同时运行起来。

2.集成启动的 P2P 服务器监听的 UDP 端口号为当前 IRapidServerEngine 监听的 TCP 端口号加 1。比如：IRapidServerEngine 监听的 TCP 端口号为 4530（通过其 Initialize 方法初始化时设定），则 P2P 服务器监听的 UDP 端口就是 4531。如果是服务端使用框架内置的 MainServerForm 作为主界面显示，将会看到：



3.如果服务端集成启动了 P2P 服务器，那么客户端不再需要设置 IRapidPassiveEngine 的 P2PServerAddress 属性了。客户端将在登录服务端时，发现如果服务端已经集成了 P2P 服务器，则会自动与集成的 P2P 服务器建立联系。

4.如果服务端集成启动了 P2P 服务器，客户端仍然可以设置 IRapidPassiveEngine 的 P2PServerAddress 属性。比如，直接将 P2PServerAddress 设置为集成 P2P 服务器的地址：

API：

```
rapidPassiveEngine.P2PServerAddress=newAgileIPE("192.168.0.98",4531);
```

这种情况下，客户端还是会使用服务端集成的 P2P 服务器。但是，如果将 P2PServerAddress 设置为其它地址

API：

```
rapidPassiveEngine.P2PServerAddress = new AgileIPE("192.168.0.100", 4500);
```

那么，客户端就不会再使用服务端集成的 P2P 服务器，而是转向使用监听在 192.168.0.100 的 4500 端口的 P2P 服务器了，这种情况，就像是使用独立部署的 P2P 服务器一样了。

二.独立部署 P2P 服务器

我们提供了可独立启动的 P2P 服务器 exe，运行后，显示的界面如下所示：

1.配置文件

可以通过配置文件修改要监听的端口号等信息。xml 配置文件内容如下所示：

API：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Port" value="9901"/>
    <add key="MaxLengthOfUserID" value="11"/>
    <add key="SecurityLogEnabled" value="false"/>
  </appSettings>
</configuration>
```

Port 设定要监听的 UDP 端口。

MaxLengthOfUserID 用于设定 UserID 的最大长度。由于 P2P 服务器内部使用的仍然是 ESFramework 极其消息协议格式，所以 MaxLengthOfUserID 的值必需与客户端的设定完全一致。（可以参考 ESFramework 开发手册（02）——基础功能与状态通知）

SecurityLogEnabled 用于设定是否开启日志记录。

2.界面显示

P2P 服务器的界面除了显示监听的 UDP 端口号之外，还显示了 UDPSession 会话的个数，以及线程池中可用的空闲线程的个数。

客户端登录时，会与 P2P 服务器建立 UDPSession，完成了必要的通信之后，客户端会主动关闭这个 Session。会话个数表示的是当前还未关闭的 Session 的数量，而不是当前已启动的客户端的数量。

会话个数的值与 RapidServerEngine 服务端的在线人数的值几乎是没有必然的联系的。

3.如何部署

首先，我们将服务端的 IRapidServerEngine 的 UseAsP2PServer 属性设为 false，表示将要使用独立部署的 P2P 服务器。

其次，启动 P2P 服务器。我们可以把 P2P 服务器部署到不同于 RapidServerEngine 服务端所在的服务器上。

最后，将客户端 IRapidPassiveEngine 的 P2PServerAddress 属性设置为 P2P 服务器的地址就可以了。

由于 P2P 服务器消耗的资源非常的小，对服务器配置的要求不是很高。所以，多个 RapidServerEngine 服务端可以共享一个 P2P 服务器。特别是在 ESPlatform 群集中，我们可以部署许多个应用服务器，但是只要部署一个 P2P 服务器就可以了。

三.禁用 P2P

我们只要将服务端的 IRapidServerEngine 的 UseAsP2PServer 属性设为 false，并将客户端 IRapidPassiveEngine 的 P2PServerAddress 属性设置为 null，就表示禁用 P2P。这样，客户端 IRapidPassiveEngine 初始化完成后，其

P2PController 属性的值将为 null。

四.下载

ESFrameworkP2P 服务器。

（04）—— 大数据块信息

在一般的通信系统中，我们都是通过网络来传递尺寸较小的单个信息（数十个字节，最大也就几 K 或几十 K），但是有时候，需要传递的信息的个头很大，比如在内存中产生的一个数据报表，可能有数 M 之大。我们将这样的大尺寸信息称为“大数据块信息”（Blob）。

一.发送 Blob 的几种方案

在 ESFramework/ESPlus 中，发送的自定义信息的最大尺寸是有限制的，这个限制源于框架对单个消息的最大尺寸有限制（默认为 100K，可以通过 GlobalUtil 的 SetMaxLengthOfMessage 静态方法进行修改）。 在之前的版本中，如果要发送巨大尺寸的消息，有以下几个方案：

（1）修改框架允许的单个消息的最大尺寸的默认值，比如从 100K 改为 10M。

（2）通过文件中转。即将要发送的大尺寸信息先存储到一个文件中， 然后使用 ESPlus 提供的传送文件的功能将其传递给对方。

（3）在发送方可以手动拆分信息，逐个发送信息片段，然后在接收方将接收到的片段拼接为一个完整的信息。

我们应当极力避免方案（1）的做法。我们建议，最好不要通过单个消息来发送一个巨大的 Blob。虽然，TCP 连接允许我们一次性将一个大尺寸的数据交给它去发送，但是对某些系统而言，这样做是很危险的。为什么了？因为当这个大尺寸的数据还没有发送完毕时，那么，在该 TCP 连接上继续发送任何其它数据，都会被阻塞。这意味着，如果大尺寸的数据发送需要 30s，那么在这 30 秒内，整个通道都被 Blob 独占，无法发送任何其他数据，包括可能必须的紧急命令信息。

而且，方案（1）需要框架预留更多的内存作为数据缓冲区，在某些时候，这可能是一种很大的浪费。

方案（2）涉及到硬盘 IO，而且以文件作为中转，不够直观。

方案（3）是最佳的，Blob 的片段可以和其他的信息交叉发送。但是，手动实现该方案还是比较繁琐的。幸运的是在最新的 ESPlus 3.0 中，已经内置了该方案的实现。

二.ESPlus 3.0 提供的发送 Blob 方法

ESPlus 3.0 在自定义信息空间提供了发送 Blob 的几个 API：

1.ICustomizeOutter 的 SendBlob 方法：用于在客户端将 Blob 信息发送给其他在线用户或服务器。

API：

```
/// <summary>
/// 向在线用户或服务器发送大的数据块信息。直到数据发送完毕，该方法才会返回。如果担心长时间阻塞调用线程，可考虑异步调用本方法。
/// </summary>
/// <param name="targetUserID">接收消息的目标用户ID。如果为null，表示接收者为服务器。</param>
/// <param name="informationType">自定义信息类型</param>
/// <param name="blobInfo">大的数据块信息</param>
/// <param name="fragmentSize">分片传递时，片段的大小</param>
```

```
void SendBlob(string targetUserID, int informationType, byte[] blobInfo  ,int fragmentSize);
```

2.ICustomizeController 的 SendBlob 方法：用于在服务端将 Blob 信息发送给某个在线用户。

API：

```
/// <summary>
/// 向ID为userID的在线用户发送大数据块信息。直到数据发送完毕，该方法才会返回。如果担心长时间阻塞调用线程，可考虑异步调用本方法。
/// </summary>
/// <param name="userID">接收消息的用户ID</param>
/// <param name="informationType">自定义信息类型</param>
/// <param name="blobInfo">大数据块信息</param>
/// <param name="fragmentSize">分片传递时，片段的大小</param>
void SendBlob(string userID, int informationType, byte[] blobInfo, int fragmentSize);
```

对于发送 Blob 信息，要注意以下几点：

（1）选择合适的片段大小（fragmentSize）。fragmentSize 的参数的值取决于网络的畅通状态和我们期望的单个片段发送所需的时间。

（2）无论是发送方、还是接收方都需要将 Blob 信息当作一个整体。在传送 Blob 片段的过程中，如果接收方或发送方任何一方掉线，则整个 Blob 信息的传送都会被取消。接收方决不会提交不完整的 blob 信息给自定义信息处理器（ICustomizeHandler）去处理。

（3）SendBlob 方法是在当前调用线程中，逐个发送 Blob 片段的。如果 Blob 信息很大，而又不想阻塞调用线程，请使用异步调用。

我们以常见的发送图片为例，由于大一点的图片可能有几 M，所以通常采用 Blob 分片段发送。我们使用异步调用的方式：

API：

```
private RapidPassiveEngine rapidPassiveEngine = ...;
public static int ImageInformationType = 101;
public void SendImage(Image img, string destUserID)
{
        MemoryStream memoryStream = new MemoryStream() ;
      img.Save(memoryStream, img.RawFormat);
        byte[] blob = memoryStream.ToArray();
      memoryStream.Close();
        CbGeneric<byte[], string> cb = new CbGeneric<byte[], string>(this.SendBlobThread);
      cb.BeginInvoke(blob, destUserID, null, null);
}
private void SendBlobThread(byte[] blob ,string destUserID)
{
    this.rapidPassiveEngine.CustomizeOutter.SendBlob(destUserID, ImageInformationType, blob, 2048);
}
```

（4）在接收方，无论接收的是普通自定义信息，还是 Blob 信息，都会提交给同一个方法（ICustomizeHandler.HandleInformation 方法）处理，没有分别。

我们可以在接收方像下面这样处理刚才发送的图片：

API：

```
public void HandleInformation(string sourceUserID, int informationType, byte[] info)
{
    if (informationType == ImageInformationType)
```

```
        {
            MemoryStream memoryStream = new MemoryStream(info) ;
            Image img = Image.FromStream(memoryStream);
            memoryStream.Close();
             //...... 后续的业务逻辑
        }
    }
```

三.Blob 信息与 P2P

如果收发 Blob 信息的双方是两个在线的用户，并且这两个用户之间已经建立了 P2P 通道，那么 Blob 信息将直接经过 P2P 通道传送，而不经过服务器中转。这和普通的自定义信息是没有分别的。（关于如何在两个客户端之间创建 P2P 通道，请参考 ESFramework 开发手册（04） —— 可靠的 P2P ）

（05）—— 兼容 IPv6

随着互联网越来越普及，以及物联网的兴起，IPv4 地址已远远不够用，IPv6 的普及将是不可避免的趋势。最新版本的 ESFramework（ESFramework v4.0.10.0、ESPlus v3.2.0.0）已经增加了对 IPv6 的支持。

首先，要了解两个最基础的事实：

（1）通信的双方，无论是服务端与客户端之间，或是客户端与客户端之间的 P2P 通信，必须使用相同的协议版本 —— 要么都是 IPv4，要么都是 IPv6。

（2）在没有特别安装附件的情况下，有的 OS 可能只支持 IPv4，有的可能只支持 IPv6，有的即支持 IPv4 也支持 IPv6。可以通过 Socket 类的 OSSupportsIPv6 和 OSSupportsIPv4 属性来作判断。

1.服务端

基于 ESFramework 的服务端程序若需同时支持 IPv4 和 IPv6，需要保证两点：

（1）服务器系统的网卡必需绑定至少一个 IPv4 地址和至少一个 IPv6 地址。

（2）IRapidServerEngine 的 IPAddressBinding 属性设置为 null。

当然，如果想让服务端仅仅对 IPv6 的客户端提供服务，可以将 IPAddressBinding 设置为网卡的 IPv6 地址；同理，如果想让服务端仅仅对 IPv4 的客户端提供服务，可以将 IPAddressBinding 设置为网卡的 IPv4 地址。

2.客户端

基于 ESFramework 的客户端程序在升级兼容 IPv6 时，要稍作改动。

我们现在假设服务端程序已经兼容了 IPv6，并且其提供服务的 IPv4 地址为 192.168.0.104，IPv6 地址为 fe80::14d8:a209:89e6:c162%14。

那么基于 ESFramework 的客户端在与服务端建立连接之前，要看本地 OS 对 IPv4 和 IPv6 的支持情况：

（1）如果本地 OS 仅支持 IPv4，或者同时支持 IPv4 和 IPv6，那么简单地，就让其连接到服务器的 IPv4 地址。

（2）如果本地 OS 仅支持 IPv6，那么，就让其连接到服务器的 IPv6 地址。

示例代码如下所示：

API：

```
    RapidPassiveEngine rapidPassiveEngine = new RapidPassiveEngine();
    if (Socket.OSSupportsIPv4)
    {
```

```
        LogonResponse logonResponse = rapidPassiveEngine.Initialize("userID", "password",
"192.168.0.104", 9900, null);
    }
    else if (Socket.OSSupportsIPv6)
    {
        LogonResponse logonResponse = rapidPassiveEngine.Initialize("userID", "password",
"fe80::14d8:a209:89e6:c162%14", 9900, null);
    }
    else
    {
        throw new Exception("当前OS既不支持IPv4，也不支持IPv6。");
    }
```

（07）—— 跨平台开发

随着智能手机和移动平台的迅速崛起，现在的应用除了支持 PC 的桌面平台和 Web 外，还需要对移动平台进行支持。跨平台是 ESFramework 体系的重要特性之一，ESFramework 通过提供多个平台的客户端引擎来实现这一点。

一. 支持的客户端平台

目前 ESFramework 支持的客户端平台有：

（1）.NET

（2）WinForm

（3）WPF

（4）SilverLight

（5）Windows phone

（6）Windows CE

（7）iOS

（8）Android

（9）Xamarin

所有不同平台类型的客户端引擎都使用几乎完全一致的 API 接口 ,所以 ,就开发 ESFramework 客户端程序而言，从一个平台转向另一个平台，不需付出任何额外的代价。

通过公用同一个服务端实例，基于 ESFramework 开发的不同平台的客户端之间可以相互通信，如此，异构环境将变得透明化。而且，将服务端迁移到 ESPlatform 群集平台时，不同平台上的客户端程序不需要做任何改变。

二. 让应用支持其它平台的客户端

如果希望为已经开发好的基于 ESFramework 的系统增加其它类型的客户端(如 iOS 和 android 的原生程序等)，那么需要做到两点：

（1）根据客户端的平台类型，选择 ESFramework 的对应版本。

比如，新增的客户端类型是 android，那么，在开发 android 客户端时，就要基于 ESFramework 的 andorid 版本来进行。

（2）新的客户端要遵循应用层的消息的协议格式。

一般情况下，基于 ESFramework 开发的应用（如 OrayTalk、GG 等）的内部消息也是使用 ESFramework 提供的紧凑的序列化器来进行序列化和反序列化的。这时，情况就容易一些了，我们有提供了一个小工具，可以根据协议类的定义，自动生成对应的协议格式。具体可参见：ESFramework 使用技巧 —— 协议格式自动生成器（跨平台开发小工具）。

（08）—— 协议格式自动生成器（跨平台开发小工具）

一.跨平台开发需要通信消息的协议格式统一

有些客户希望将已有的基于 ESFramework 开发的系统，如 OrayTalk 或 GG（可在广域网部署的 QQ 高仿版）扩展到 android 和 iOS 平台 ,除了在这些客户端平台使用 ESFramework 对应版本的开发包之外 ,最重要的一点就是：通信消息的格式必须达成一致。所以，必须先搞清楚 GG 现有的协议格式，然后让 iOS 或 android 开发遵循现有的协议格式就 OK 了。

二.紧凑的序列化器的协议格式

OrayTalk 和 GG 内部的消息几乎都是使用 ESPlus 提供的紧凑的二进制序列化器（CompactPropertySerializer）来进行序列化和反序列化的，CompactPropertySerializer 是一个紧凑的二进制序列化器（是基于字节流的，而非基于文本的）。CompactPropertySerializer 在序列化和反序列化时遵循的是以下策略：

字符串一律使用 UTF-8 编码。

如果是基础数据类型，则直接记录其字节 。

如果是 bool，则用一个字节表示，0 表示 false，1 表示 true。

如果是 string，先记录其长度(int，-1 表示为 null，0 表示 string.Empty)，再记录 UTF8 编码的字节。

如果是 byte[]，先记录其长度(int，-1 表示为 null)，再记录其内容。

如果是 Image ，先记录图片数据（byte[]）的长度(int，-1 表示为 null)，再记录是否为 Gif（一个 byte），再记录序列化内容。

如果是 Color ，长度固定为 3 个字节，依次记录 R、G、B 的值。

如果是 List<>，先记录元素数(int，-1 表示为 null)，再依次记录每个元素的内容。

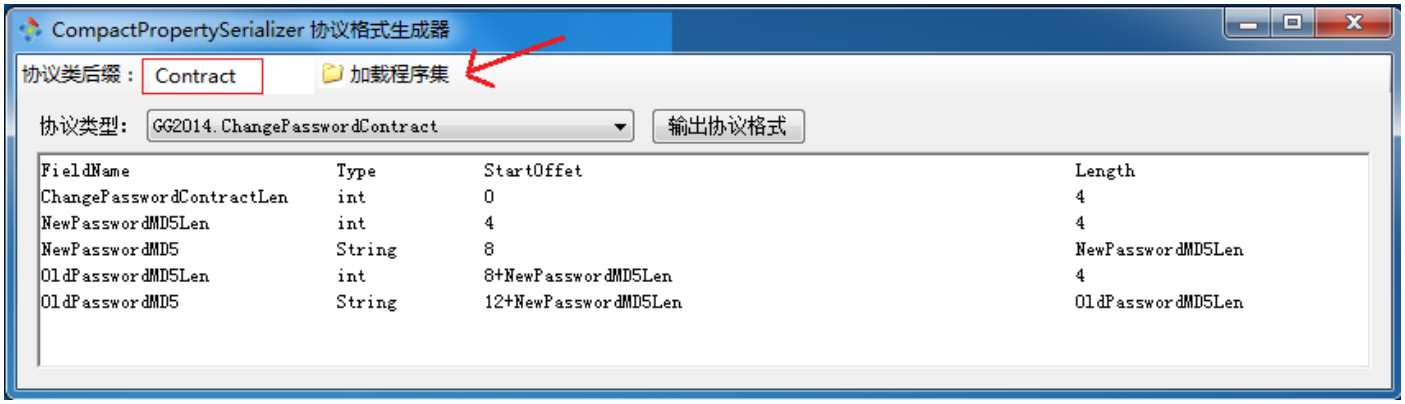
如果是 Dictionary<,>，先记录元素对的个数(int，-1 表示为 null)，再依次记录每个元素的 Key,Value。

如果是自定义的 class 和 struct，则先记录其序列化的长度(int，-1 表示为 null)，再记录其序列化后的内容。

如果是 Font，则等价于 ESPlus.Serialization.SimpleFont。

我们提供了一个工具（ContractFormatGenerator），可以查看一个协议类（contract 类，用于通信的实体类）转换成字节流后的格式。其它的平台只要按照这些格式来构造消息，就可以与 OrayTalk 或 GG 的 PC 版本进行通信了。

ContractFormatGenerator 工具运行起来后，如下所示：



点击“加载程序集”的按钮，可以选择协议类所在的 dll（需要把该 dll 以及其所依赖的相关 dll 都拷贝到 ContractFormatGenerator.exe 所在的目录，否则“加载程序集”会报错：无法加载请求类型。），这样，下拉列表中就会列出该程序集中所有的协议类（默认只显示以 Contract 结尾的类，可以在文本框中修改这个后缀），选中某个协议类，点击“输出协议格式”按钮，就会生成该协议类对象序列化后的格式说明。

上面示例中选中的是“修改密码”功能用到的 ChangePassword 协议类，这个类的定义如下：

API：

```
public class ChangePasswordContract
{
    public ChangePasswordContract() { }
    public ChangePasswordContract(string oldPasswordMD5, string newPasswordMD5)
    {
        this.OldPasswordMD5 = oldPasswordMD5;
        this.NewPasswordMD5 = newPasswordMD5;
    }

    public string OldPasswordMD5 { get; set; }

    public string NewPasswordMD5 { get; set; }
}
```

这个类只有两个字段（属性），而生成的协议格式实际上就是对这两个字段的描述。下面我们简单讲讲协议格式的各个列的含义：

（1）FieldName：字段的名称。字段名称一般与协议类的属性名是对应的，如果某个属性的类型的长度是可变的（比如 string），那么就要先加一个 Field，来描述这个属性值转换给字节后的长度。

（2）Type：Field 的类型。

（3）StartOffset：当前 Field 在 byte[]中的起始偏移。

（4）Length：当前 Field 的值的长度。

要注意，协议格式中，第一个 int 是一个长度（ChangePasswordContractLen），用来记录当前协议类序列化后

的总长度（这个 int 也包含在内）。

所以，我们在 android 或 iOS 按照上面的协议格式来构造“修改密码”的消息（得到一个 byte[]），然后，将其发送到 GG 的服务器，GG 的服务器就可以正常的解析并完成处理了。

三.协议格式生成器下载

下面给出可执行 ContractFormatGenerator 及其源码下载，如果有些特殊需要修改的，可以在源码上修改。

（1）可执行的 ContractFormatGenerator

（2）ContractFormatGenerator 源码

（09）—— 支持同帐号多设备同时登录（PC 端和移动端同时在线）

在 ESFramework/ESPlatform 体系中，是使用 UserID 作为唯一标志来标记每一个用户的，也就是说，对于一个指定的 UserID，只能有一个客户端在线。所以，ESFramework 虽然支持多种类型的客户端设备，但是，ESFramework 并不支持不同类型的客户端设备同时登录同一个 UserID。

如果我们开发的系统要支持同帐号多设备同时登录的场景，即需要像 QQ 一样，在 PC 端登录的同时，也可以使用同一个帐号登录移动端（iOS 或 Android），那么，如何才能做到了？

解决方案的原理是比较简单的：既然 ESFramework 要求 UserID 作为用户标记必须是唯一的，那么我们就引入一个称为“LoginID”的概念，对于同一个用户，在不同类型的设备上就使用不同的 LoginID，但是这些 LoginID 都指向同一个真正的 UserID。

一. LoginID 与 真正的 UserID

1. 不需要支持同帐号多设备同时登录的简单场景

在之前不支持同帐号多设备同时登录的场景中（简称“单设备登录”场景），登录用的帐号就是真正的 UserID，也就是说 ESFramework 框架中各个 API（各个方法以及事件）的参数涉及到的用户帐号都是真正的 UserID。比如，一个帐号 abc001，该帐号是存在于数据库的用户表中的；使用 abc001 登录到 ESFramework 服务器，ESFramework 在整个的运作过程中，也是使用 abc001 来标记对应的客户端实例。在该场景中，不会存在多个运行的客户端实例都对应帐号 abc001 的情况。如果有个客户端已经使用 abc001 登录，然后再用该帐号在其它地方登录，默认的机制是会把之前登录的那个客户端挤掉线。

2. 需要支持同帐号多设备同时登录的复杂场景

如果现在我们要支持同帐号多设备同时登录的场景（简称“多设备登录”场景），那么，ESFramework 在整个的运作过程中，就不能使用 abc001 来标记对应的客户端实例了，因为存在多个客户端实例都对应同一个 abc001 帐号的情况。于是，我们使用 LoginID 来区分这种情况下不同的客户端实例。

常用的方法是，在真正的 UserID 前加上两个字符的前缀以构成 LoginID。比如，对于 abc001 这个帐号，在使用 iOS 设备登录时，我们选择使用前缀“1#”，这样 iOS 设备使用的 LoginID 就是 1#abc001；同理，Android 设备就使用 2#abc001。

该两个字符的前缀的含义是这样的：

（1）第二个字符“#”，是一个标志（token），表示该 ID 是一个 LoginID。

（2）第一个字符，表示设备的类型。比如“0”表示.NET 设备（PC），“1”表示 iOS 设备，“2”表示 Android 设备，等等。

当使用 LoginID 后，ESFramework 在整个的运作过程中就不再是使用真正的 UserID 来标记客户端实例了，而是

使用 LoginID —— 也就是说，ESFramework 框架中各个 API（各个方法以及事件）的参数涉及到的用户帐号都是 LoginID 了。

二. MultiDeviceHelper 类

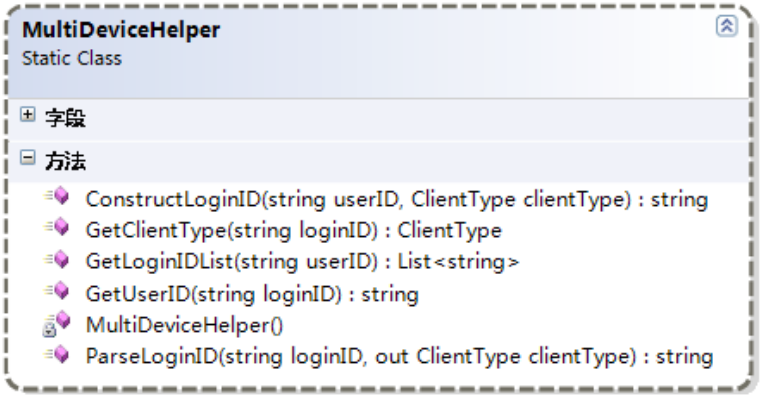
ESFramework.Boost 类库中提供了 MultiDeviceHelper 类，用于为多设备同时登录提供支持。特别是，提供了与 LoginID 的构造和解析相关的 API。

在 MultiDeviceHelper 的静态构造函数中，规定了每种设备的前缀，如下所示：

API：

```
static MultiDeviceHelper()
{
    #region 如果在当前的应用中，不存在某种类型的设备，则注释掉下面对应的语句即可。
    MultiDeviceHelper.LoginIDPrefixMapping.Add(ClientType.IOS, "1#");
    MultiDeviceHelper.LoginIDPrefixMapping.Add(ClientType.Android, "2#");
    MultiDeviceHelper.LoginIDPrefixMapping.Add(ClientType.WebSocket, "3#");
    MultiDeviceHelper.LoginIDPrefixMapping.Add(ClientType.WindowsPhone, "4#");
    #endregion
}
```

然后， MultiDeviceHelper 提供了多个静态方法以完成真正 UserID、设备类型与 LoginID 之间的转换：



三. 登录和登录验证

客户端在登录时会调用 IRapidPassiveEngine 的 Initialize 方法:

LogonResponse Initialize(string userID, string logonPassword, string serverIP, int serverPort, ICustomizeHandler customizeHandler);

该方法的第一个参数就需要传入 LoginID，比如 1#abc001。

在服务端会回调 IBasicHandler 接口的 VerifyUser 方法来进行帐号密码验证：

bool VerifyUser(string systemToken, string userID, string password, out string failureCause);

此时要注意的是，VerifyUser 方法传入的 userID 参数实际上是 LoginID，即 1#abc001。我们需要通过调用 MultiDeviceHelper 的 ParseLoginID 方法来获取真正的 UserID，该方法会返回 abc001，并且 out 参数指明设备类型为 iOS。

四. 处理消息及其它

在 ESFramework 架构中 ,服务端是通过回调 ICustomizeHandler 接口的 HandleInformation 方法来处理接收到的消息的：

void HandleInformation(string sourceUserID, int informationType, byte[] info);

同上面一样，此处的 sourceUserID 参数实际上也是 LoginID，所以，也需要调用 MultiDeviceHelper 的 ParseLoginID 方法来将其转换成真正的 UserID。

同理，在多设备登录场景中，ESFramework 框架中各个 API（各个方法以及事件）的 userID 参数实际上都是 LoginID，在处理时都需要做类似的处理，这里就不一一列举了。

五. 多设备聊天消息同步

在解决了多设备同时登录的问题后，还有一个常见的需求：类似 QQ 的 PC 和手机端同时在线时，别人给我发一条消息，手机端和 PC 端都能接收到。这样的功能是怎么实现的了？

在单设备登录场景中，我们通常是在客户端调用 ICustomizeOutter 接口的下列 Send 方法来发送聊天消息的：

```
void Send(string targetUserID, int informationType, byte[] info);
```

该方法的第一个参数是接收者的 UserID，表示直接将聊天消息发送给对方（可能是经过服务器中转，或者是经 P2P 通道直接传送）。

但是，在多设备登录场景中，不能再直接发送了，而是必须要经过服务器中转，通过调用下面的 Send 方法：

```
void Send(int informationType, byte[] info);
```

该 Send 方法将消息直接发送给服务端，在 info 参数中包含要消息接收者的 UserID。服务端在处理该消息时，需要从 info 中将接收者 UserID 解析出来，然后，调用 MultiDeviceHelper 的 GetLoginIDList 方法来获取各个设备类型对应的 LoginID，然后，服务端在把该消息发送给每一个 LoginID。如此，手机端和 PC 端就都能收到这条聊天消息了。

ESFramework 经验分享

（00）—— 服务端性能瓶颈

如果基于 ESFramework 开发的服务端程序在实际运行时，没有达到预期的性能（如在线人数到达一个值后就很难再增加了，或者吞吐量上不去等等 ），那么该如何定位问题了？首先，我们要对 ESFramework 的核心机制有个大概的了解，这对服务端的整个消息处理流程与模型能有个整体的把控，可以参见这篇文章：ESFramework 开发手册（11） —— 服务端信息处理模型 。

一. 决定吞吐量的核心因素

虽然，决定吞吐量的因素有很多，但最首要也是最核心的因素就是消息处理的快慢（ HandleInformation / HandleQuery 方法执行的耗时） 。

HandleInformation / HandleQuery 返回得越快，每秒处理的消息数量就越大，即吞吐量也越大。

二. 生产 - 消费 失配

我们可以使用生产-消费模型来分析消息，从网络接收到消息，相当于生产消息， HandleInformation / HandleQuery 方法的执行相当于是消费消息。

当生产消息的速度小于或等于消息消费的速度时，服务端的业务处理是从容不迫的。

但是，当生产消息的速度大于或远远大于消息消费的速度时，可以想想会发生什么状况？我们简单推导一下：

（１）线程池中可用的线程数会越来越少（ IocpDirectly 模型）或任务队列中积累的待处理消息越来越多、内存消耗越来越大（ TaskQueue 模型）。

（２）而在客户端看来，请求响应的时间越来越长。

（３）如果某些 HandleInformation / HandleQuery 调用在执行到某个地方时，卡死了，那么该调用所占用的线程将一直无法被归还给线程池。

所以，观察线程池中的可用 IOCP 线程数或任务队列中的等待处理的消息数，就可以发现生产-消费失配的情形。

三. 发现性能瓶颈

压力测试是发现性能问题的最直接方法，如果结合 ESFramework 提供的服务端性能诊断功能，就更容易发现问题了。

具体的压力测试要怎么做了？总体上而言，我们可以将基于 ESFramework 开发的服务端程序分为两层：通信层和信息处理层（包括业务逻辑处理、数据库操作等）。 任何一层出问题都将成为性能的短板：

（１）如果通信层每秒最多只能接收数百个消息，而信息处理层每秒可以处理上万个消息，那问题出在通信层。

（２）反过来，如果通信层每秒能接收上万个消息，而信息处理层每秒最多可以处理数百个消息，那问题就出在信息处理层。

（３）还有一种比较少见的情况，就是通信层和信息处理层的单独压力测试的效果都很好，但是组合到一起后，效果却不理想。

我们可以分别对这三个方面进行针对性的压力测试：单纯的通信层压力测试、单纯的信息处理层压力测试、两层组合后的压力测试（即对服务端整体进行压力测试），按照这个顺序测下来，肯定可以找到性能问题的根源。

1.单纯的通信层压力测试

通信层由 ESFramework 负责，而正确、高效、稳定地收发消息是 ESFramework 的优点，做得更好则是我们持续努力的目标，单纯通信层的压力测试可以参见 ESFramework 4.0 性能测试。

2.单纯的信息处理层压力测试

信息处理层是基于 ESFramework 进行二次开发的业务逻辑部分，如何撇开通信层来对其进行单纯的压力测试了？

（１）二次开发时，我们会实现 ICustomizeHandler 接口，服务端收到的所有消息都会提交给该接口的 HandleInformation 和 HandleQuery 方法来处理。

所以，可以将这两个方法看作是信息处理层的总入口，我们只要调用这两个方法，就可以驱动整个信息处理层。

（２）由于撇开了通信层，所以，我们需要模拟消息来调用总入口的两个方法。注意：模拟的消息应该尽可能地能与系统实际运行时接收的消息状况一致。

（3）使用多个线程（如 20 个、50 个 、100 个、500 个）模拟多个客户端同时发消息。在这多个线程中，分别循环创建模拟的消息，并调用 HandleInformation 或 HandleQuery 方法。

（4）统计每秒处理的消息个数。即用模拟的消息调用 HandleInformation 或 HandleQuery 方法成功返回一次，就将完成数加 1，并每隔一秒就小计一次，如此就得出了信息处理层的处理能力。

3.服务端整体压力测试

只有当前面两层的压力测试都达到了我们的预期效果时，做整体压力测试才更有意义。服务端整体压力测试，最接近于系统实际运行的情况。即运行完整的服务端程序，然后大量的客户端连接上来，进行测试。

四.整体压力测试经验分享

就我们以往基于 ESFramework 开发 C/S 系统的诸多经验而言，对于整体压力测试，感觉最好用的是使用日志记录的方式，将一些重要的数据记录下来，之后进行详细分析以发现前两个压力测试没有暴露的更为隐蔽的问题。我们建议：

1.开启服务端性能诊断功能

参考 ESFramework 开发手册（12） —— 服务端性能诊断这篇文章所描述的，将诊断功能开启，并将诊断信息记录到日志文件中。

2. 记录用户上下线日志

（1）预定 IUserManager 的 SomeOneConnected 事件，记录用户的 ID 及上线时间。

（2）预定 IUserManager 的 SomeOneDisconnected 事件，记录用户的 ID、下线时间、以及连接断开的原因（对应事件的参数 DisconnectedType）。

3. 如何记录日志？

（1） 方法执行的耗时（精确到毫秒）记录，可以使用 Stopwatch 类，其比直接使用 DateTime 更准确。

（2） 日志记录可以直接在服务端的运行目录下生成一个 txt 日志文件。

（3） 为日志记录的启用/禁用加上一个开关控制，可以通过配置来开启或关闭日志记录功能。

（4） 如果没有可复用的写 txt 文件的类，可以考虑 ESBasic.dll 提供的 ESBasic.Loggers.FileAgileLogger 类，使用其 LogWithTime 方法即可。

————————— 详细分析服务端记录的日志信息，90%以上的性能问题应该都可以被发现，然后，就可以进行针对性的优化。

五. 技术顾问服务

如果通过以上步骤的排查，还是找不到问题所在，那我们可以为您提供技术顾问性质的有偿服务，该服务将针对您项目的实际情况（我们会深入了解项目的业务需求和现有源码实现等细节信息），提供专业的更具针对性的排查指导和性能优化建议。

（01）—— 故障排查：服务器端口 telnet 失败

telnet 命令的主要作用是与目标端口进行 TCP 连接（即完成 TCP 三次握手）。

当服务端启动后，但是 telnet 其监听的端口，却失败了。或者，当服务端运行了一段时间后，突然其监听的端口 telnet 不通了。当类似这样的 telnet 失败的情况出现时，都可以按照如下方面进行排查：

1.观察一下服务端进程的 CPU 和内存是否有异常。

比如，当 CPU 持续在 100%时，就有可能导致来自客户端的 TCP 连接请求被丢弃或无暇处理。

2.端口监听器是否运行正常？

可以通过 IRapidServerEngine 的 Advanced 属性的 GetPortListenerState 方法来获取端口监听器的状态 ,该方法返回一个 PortListenerState 对象，其包含 5 个属性：

（1）IsAuthorized：服务端实例是否被授权。

（2）IsMaxConnection：是否达到了最大连接数的限制。比如，某些授权的服务端实例只允许最多 100 个客户端同时在线。

（3）IsListening：是否正在监听端口。如果未授权，或达到了最大连接数限制，则将会停止监听端口。

（4）LastDetectTime：最后一次检测 TCP 连接队列（已完成 OS 底层的三次握手，但尚未被 ESFramework 提取的 TCP 连接存放于该队列中）的时间。

（5）PendingInQueue ：队列中是否有待提取的 TCP 连接。（即 TcpListener 的 Pending()方法的返回值）

果上述两点都正常，则接下来，需要专业的运维人员或网管人当员参与进来协助排查。

3.在当前服务器上执行 telnet 命令，看能否连接成功？

如果能连接成功，至少表明本机的 TCP 握手请求是能正常地被接收和处理的。

4.在服务器上执行 netstat 命令

netstat 是一个非常有用的查看端口状态的命令，执行 netstat 命令后，请注意查看以下信息：

（1）目标端口是否处于监听状态？

（2）目标端口上是否存在已成功建立的 TCP 连接（ESTABLISHED）？其数量是多少？

（3）是否存在半开连接（SYN_RECV）？其数量是多少？

（4）是否存在等待关闭的连接（TIME_WAIT）？其数量是多少？

这里，最有可能的原因是半开连接数达到最大限制，导致 windows 系统丢弃后续的 TCP 连接请求。 要查看或修改半开连接数限制，请查看本文最后的附录说明。

5.TCP 三次握手是否正常？

对于一些奇怪现象的跟踪与分析，数据抓包工具是不可缺少的。

在服务器上将抓包工具（如 Sniffer）运行起来，然后在其他的电脑上 telnet 该服务器的目标端口，通过抓包工具观察目标端口上 TCP 三次握手的过程是否正常：

（1）目标端口是否收到了来自客户端的 SYN 请求？

（2）目标端口有回复 SYN_ACK 给客户端？

（3）目标端口有收到来自客户端的第三次握手？

只有当 TCP 三次握手顺利完成后，windows 底层才会将建立好的 TCP 连接放入队列中，提交给上层的应用程序。

6. 技术顾问服务

如果通过以上步骤的排查，还是找不到问题所在，那我们可以为您提供技术顾问性质的有偿服务，该服务将针对您项目的实际情况（我们会深入了解服务器部署的网络拓扑结构、防火墙、路由器的规则设定、网路安全监控软硬件等相关的详细信息），提供专业的更具针对性的排查指导和解决方案。

附：如何查看与修改半开连接数限制值

执行 regedit 命令，打开注册表，按照顺序依次打开一下选项： HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Services \ Tcpip \ Parameters

在窗口右侧选项栏内找到这个 EnableConnectionRateLimiting 键值。

（1）如果没有此键：表示系统没有半开连接数的限制。

（2）键值为 0：也表示系统没有半开连接数的限制。

（3）键值为一个正数：即半开连接数的限制值。可以修改它。

（02）—— 故障排查：批量心跳超时掉线

心跳超时指的是：针对某个在线的客户端（TCP 连接），ESFramework 服务端在指定的时间内（默认为 30 秒）没有收到来自该客户端的任何消息，则认为该客户端已经掉线。

为什么需要心跳机制了？因为针对某些客户端掉线（可能是因为网络断开、或客户端程序退出），服务端不能立即感受到（有的可能需要过很长的时间才能感受到），所以，需要引入心跳机制，让服务端尽可能早地发现客户端已经不在线了。关于心跳机制，更详细的介绍请参见：ESFramework 开发手册（07） —— 掉线与心跳机制。

如果发生了很多客户端批量心跳超时掉线的情况，就说明服务端在过去的某段时间内，从未收到来自这些客户端的任何心跳消息。通常有 3 种可能性导致该情况发生：

1.CPU 或内存使用率过高

在该情况发生时，观察一下服务端进程的 CPU 和内存是否有异常。

比如，当 CPU 持续在 100%时，就有可能导致接收数据的操作被停止。

2.处理某些信息所花费的时间过长

如果服务端的信息处理模型设定的是 IocpDirectly，那么依据 IocpDirectly 的原理，当处理某个信息所花费的时间超过了服务端设定的心跳超时的时间，服务端就会将对应的客户端误判为心跳超时掉线。

假设是该原因导致的心跳超时，则对应的解决方案有：

（1）找出那些处理非常耗时的信息，进行优化理，加快处理速度。

（2）将超时时间间隔设定位一个更大的值或关闭心跳检测。

（3）将信息处理修改为异步模式。

（4）将服务端信息处理模型修改为 TaskQueue 模式，这样就完全避免了由于信息处理时间过长导致误判的情况。

很显然，方案（1）是最好的也是根本性的解决方案。

3.服务器网络拓扑结构、防火墙、路由器、网络安全监控等相关软硬件

如果排除了前面的可能性（比如，即使改成了 TaskQueue 模式，批量掉线仍然发生），那么，几乎就只剩下一个可能：服务端在心跳超时时间间隔内未收到来自这些客户端的任何消息。很可能来自客户端的消息被防火墙、路由器、或某些网络完全监控的相关软硬件给挡住了。

此时，需要专业的运维人员或网管人员参与进来，协助排查问题，比如：

（1）在服务器上执行 netstat 命令，查看目标端口的相关状态信息。

（2）在服务器上执行抓包工具（如 Sniffer），监测目标端口上是否有数据从客户端过来。

（3）分析服务器的网络拓扑结构，并以服务器为中心，依次向外检查防火墙、路由器、网络安全监控等相关

软硬件等的设定，并进行针对性的排查测试。

4. 技术顾问服务

如果通过以上步骤的排查，还是找不到问题所在，那我们可以为您提供技术顾问性质的有偿服务，该服务将针对您项目的实际情况（我们会深入了解服务器部署的网络拓扑结构、防火墙、路由器的规则设定、网路安全监控软硬件等相关的详细信息），提供专业的更具针对性的排查指导和解决方案。

ESFramework Demo 详解

入门 Demo，简单的即时通讯系统(包含.NET/Android/iOS/WebSocket 源码)

（本 Demo 提供了.NET/Android/iOS/WebSocket 客户端版本，本文以.NET 版本作为示例讲解）

前面的文章已经介绍完了基于 ESFramework/ESPlus 进行二次开发的所有要点，现在，我们可以开始小试牛刀了。

本文将介绍使用 ESFramework 开发的一个入门 Demo，该 Demo 演示了以下功能：

（1）客户端用户上下线时，通知其他在线用户。

（2）当客户端与服务端网络断开时，进行自动重连，当网络恢复后，重连成功。

（3）所有在线用户之间可以进行文字聊天。

（4）文件传送。

（5）P2P 通道。

（6）消息同步调用。

（7）重登陆模式。当同名的用户登陆时，会把前面的用户挤掉。

一.基础步骤

按照 ESFramework 开发手册（06）——Rapid 通信引擎一文中提到的二次开发步骤，我们按照顺序来一步步来实现这个 demo。（请从文末下载 demo 源码，然后对照着看，将更容易理解）

1.确定是否需要 Contacts 功能

（1）确定是否需要联系人关系。在本 demo 中，我们假定所有的在线用户相互之间都是联系人。

（2）确定是否需要分组。在本 demo 中，我们不需要广播消息，不需要进行分组。

结合（1）和（2），这样就可以直接使用 ESPlus 提供的 DefaultContactsManager。

2.定义信息类型

在本 demo 中，我们定义 3 个信息类型，一个类型表示文字聊天消息，一个类型表示振动提醒，第三个表示客户端同步调用服务端。其定义如下：

API:

```
public static class InformationTypes
 {
     /// <summary>
     /// 聊天信息
     /// </summary>
     public const int Chat = 0;

     /// <summary>
     /// 振动提醒
     /// </summary>
     public const int Viberation = 1;

     /// <summary>
     /// 客户端同步调用服务端
     /// </summary>
     public const int ClientCallServer = 100;
 }
```

由于 InformationTypes 的定义以及接下来定义的信息协议类在客户端和服务端都需要用到，所以，我们将其放在一个单独的项目 ESFramework.EntranceDemo.Core 中，客户端和服务端都可以引用它。

3.定义协议类

信息类型定义好后，我们接下来定义信息协议。

对于聊天消息（InformationTypes.Chat），由于消息中除了聊天内容外，还包含了该消息的产生时间，所以我们专门定义了一个协议类：TextChatContract。

对于振动提醒（InformationTypes.Viberation），由于不需要传送额外的内容，所以不需要对应的协议类。

对于同步调用（InformationTypes.ClientCallServer），我们的 demo 假设请求信息是一个字符串，回复信息也是字符串，直接使用 UTF-8 编码就可以了，不需要专门的协议类。或者说，string 就是我们用到的协议类。

4.实现自定义信息处理器

客户端的 MainForm 实现了 ICustomizeHandler 接口，其主要实现 HandleInformation 方法，来处理收到的聊天信息和振动提醒。

API:

```
void HandleInformation(string sourceUserID, int informationType, byte[] info);
```

服务端的 CustomizeHandler 实现了服务端的 ICustomizeHandler 接口，其主要实现 HandleQuery 方法来处理来自客户端的同步调用（InformationTypes.ClientCallServer）。

API:

```
byte[] HandleQuery(string sourceUserID, int informationType, byte[] info);
```

5.服务端验证用户登录的帐号

服务端的 BasicHandler 类实现 IBasicHandler 接口，以验证登录用户的账号密码。

本 demo 中，假设所有的验证都通过，所以验证方法直接返回 true。

6.客户端基础逻辑

（1）处理状态变化事件

客户端在 MainForm 的 Initialize 方法中，预定了 Rapid 客户端引擎暴露的连接状态变化事件，如连接断开、重连开始、重连成功并重新登录完成。

另外，客户端还预定了 IBasicOutter 和 IContactsOutter 的相关状态改变事件，以处理联系人的上下线通知、自己被挤掉线通知、自己被踢出通知等。

（2）创建 P2P 通道

每次双击好友头像，弹出聊天窗口时，就尝试与对方建立 P2P 通道：

API:

```
//尝试与目标用户建立P2P通道，尝试的结果将由P2PController的P2PChannelOpened或P2PConnectFailed事件来通知。
this.rapidPassiveEngine.P2PController.P2PConnectAsyn(info.Item.Text);
```

（3）传送文件

在聊天窗口上，点击传送文件的按钮，便可选择文件进行发送。

7.初始化 Rapid 引擎

在服务端和客户端各自的 Program 类的 Main 方法中，初始化 Rapid 引擎。

上面我们介绍了这个 demo 需要的主要元件，我们接下来看看客户端和服务端所需的额外基础设施。

二.服务端说明

如果获取了 ESFramework 的正式授权，那么应该在初始化服务端引擎之前，设定正确的授权用户帐号和密码。demo 中使用的是免费帐户“FreeUser”。

API:

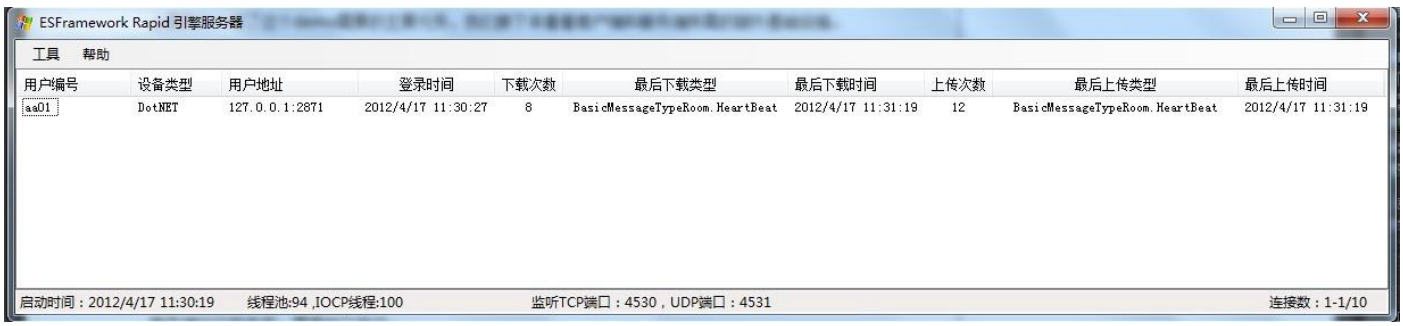
```
ESPlus.GlobalUtil.SetAuthorizedUser(AuthorizationVerifier.FreeUser, "");
```

服务端直接使用了 ESFramework.Boost 提供的默认的主窗体 MainServerForm，来显示在线用户状态信息。

API:

```
//如果不需要默认的UI显示，可以替换下面这句为自己的Form
MainServerForm mainForm = new MainServerForm(RapidServerEngine);
```

服务端运行起来后，界面如下所示：



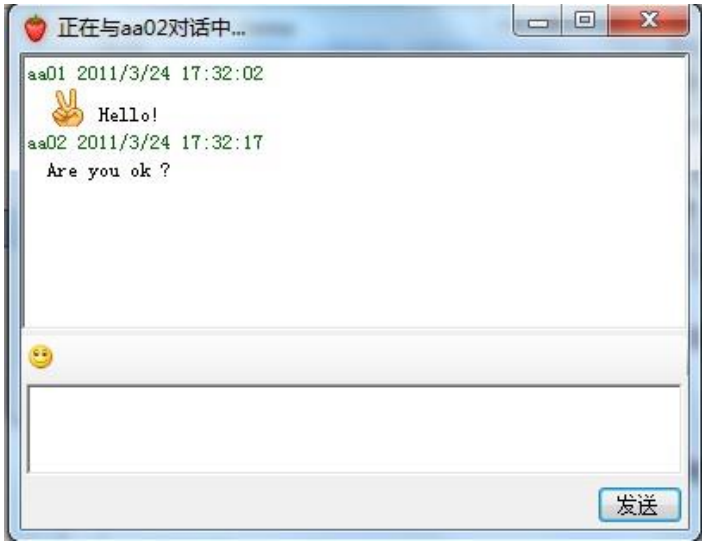
（1）最下面的状态栏，显示了线程池中剩余的线程数，初始设定的后台线程池和 IOCP 线程都是 100 个。最右边的“在线人数”还显示了已成功建立的连接数-用户数。

（2）当有用户登陆时，在线用户列表中会实时显示每个用户的状态。

三.客户端说明

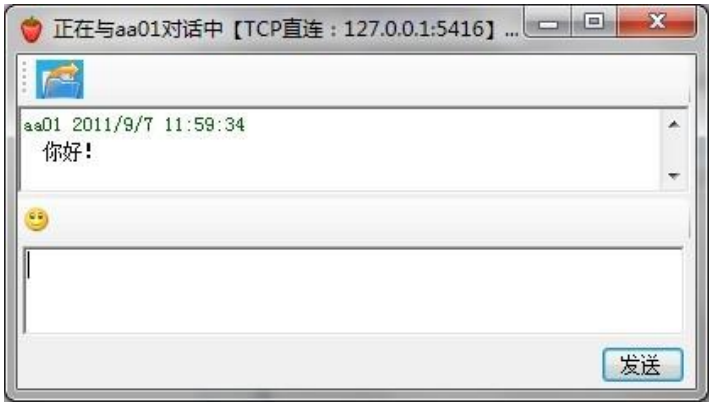
客户端启动登陆后，显示的主界面 MainForm 如下：

当有其他用户登陆时，会在“在线用户列表”中显示出来，双击头像，会弹出聊天窗口，就可以对话了。

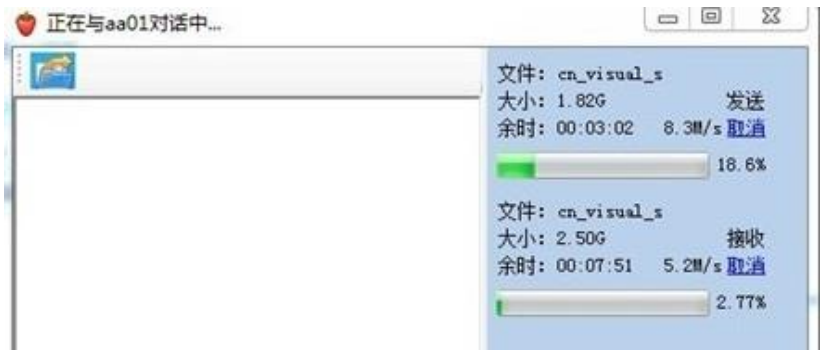


当该窗口会灰掉不可用时，表示好友已经掉线。

如果与对方的 P2P 通道创建成功，则会在窗体的 Title 显示类似“TCP 直连：*****”的信息：



下图是演示传送文件功能的截图：



如果我们再登陆一个 aa01（演示重登陆），则原来 aa01 的主界面状态栏会显示如下：

如果 Client 与服务器不在一台机器上，你还可以通过断开网络在恢复网络以观察自动重连的效果，主界面的状态栏也有相应的文字提示。或者更简单地，直接退出服务端，这会断开所有的客户端连接，也类似于断线的效果；再重启服务端，客户端就会重连上来了。

最后，点击“测试请求 123”后面的按钮，可以测试消息同步调用，消息同步调用是通过客户端的 ICustomizeOutter 接口的 Query 方法来发出的，同步调用请求将被前述的服务端的 HandleQuery 方法处理的。

要提醒的是，本示例仅仅是一个 demo，是为了展示如何使用 ESFramework 提供的功能而存在，为了简洁，省略了很多工作，比如，将 Rapid 引擎实例作为 Program 的静态成员，又如上面的 MainForm 实现了多个接口，使得 MainForm 类变得比较巨大。大家在正式的项目中，应该根据项目实际情况做正确的结构设计，不要效仿本例。

四.源码下载

（1）ESFramework.EntranceDemo 源码（内含.NET/Android/iOS/WebSocket 客户端版本）

（2）ESFramework.Boosts 源码：ESFramework 增强库，里面有本文用到的 MainServerForm 窗体的源码

动态组及群聊 Demo（附源码）

所谓“动态组”，就是在服务器内存中动态创建的组，不需要序列化存储到比如数据库或磁盘中，更像是一个临时的东西，需要的时候就创建一个，然后加入多个成员进行组内沟通，当不再使用的时候，就直接从内存中销毁了。在阅读本文之前，请务必先掌握 ESFramework 开发手册（05）—— 联系人一文中介绍的关于组关系的基础知识以及相关 API 的用法。

本 Demo 主要演示以下功能：

（1）创建动态组

（2）加入动态组

（3）在组内广播消息。（群聊）

（4）退出动态组

（5）销毁动态组

一.公共定义

对于一个 C/S 系统来说，客户端和服务端必需在某些定义上达成一致，比如信息类型的定义、协议类的定义等。DynamicGroupDemo.Core 这个程序集就是我们这个 demo 用于公共定义的地方，它将被客户端和服务端共同使用。

首先，我们要根据需求确定所需的自定义信息类型，我们用 GroupInformationTypes 静态类来表示：

API：

```
public static class GroupInformationTypes
 {
     //Client -> Server
     public const int CreateGroup = 0;
     public const int JoinGroup = 1;
     public const int DestroyGroup = 2;
     public const int QuitGroup = 3;

     //Server -> Client
     public const int SomeoneJoinGroup = 4;
     public const int SomeoneQuitGroup = 5;

     //广播消息
     public const int GroupChat = 100; //群聊
 }
```

然后，根据不同的信息类型，定义对应的信息协议类，像 GroupContract、TextChatContract。

二.服务端

1.实现组管理器接口 IGroupManager

既然是要实现动态组的功能，那么服务端就必须实现 IGroupManager 接口，Demo 里面定义了 DynamicGroupManager 类从 IGroupManager 接口继承。

DynamicGroupManager 类除了实现 IGroupManager 接口的方法以外，另外还实现了创建组、加入组、退出组和删除组的方法。

为了框架能够使用我们自定义的组管理器 DynamicGroupManager，我们必须在服务端引擎初始化之前，进行相关设置：

API：

```
IRapidServerEngine rapidServerEngine = ESPlus.Rapid.RapidEngineFactory.CreateServerEngine();
DynamicGroupManager groupManager = new DynamicGroupManager();
CustomizeHandler customizeHandler = new CustomizeHandler();
rapidServerEngine.ContactsManager = groupManager;
rapidServerEngine.Initialize(4530, customizeHandler);
```

客户端想要进行动态组的操作，必须通过自定义消息把指令发送到服务端，所以服务端还必须实现 ICustomizeHandler 接口，Demo 里面定义了 CustomizeHandler 类。

2.实现自定义处理器接口 ICustomizeHandler

服务端自定义处理器处理了创建组、加入组、退出组、销毁组等请求，并且当有人加入/退出组时，通知其他组成员。由于 CustomizeHandler 在处理这些请求时用到了组管理器，所以上面的代码，将组管理器的引用作为构造参数传递给自定义信息处理器。

另外，群聊消息不需要被自定义处理器处理，它将直接由框架提供的广播机制进行自动转发。

三.客户端

1.封装工具类 GroupTools

首先，我们将与组操作相关的动(如创建组、加入组等)作封装成一个类 GroupTools，后面直接使用 GroupTools 就好了。封装成 GroupTools 的好处是，在以后我们正式的项目中，可以直接将其拷贝过去做适当的修改就可以用于

我们自己的项目了。

创建组（CreateGroup）的时候，调用 RapidPassiveEngine 的 CustomizeOutter 的 Send 方法发送 GroupInformationTypes.CreateGroup 类型的自定义信息给服务端即可。

加入组（JoinGroup）的时候，调用 RapidPassiveEngine 的 CustomizeOutter 的 Query 方法消息给服务端，等待服务端返回加入是否成功的结果。

API：

```
  public bool JoinGroup(string groupID)
  {
      GroupContract contract = new GroupContract(groupID);
      byte[] resultBytes = this.rapidEngine.CustomizeOutter.Query(GroupInformationTypes.JoinGroup,
CompactPropertySerializer.Default.Serialize<GroupContract>(contract));
      if (resultBytes != null)
      {
          return BitConverter.ToBoolean(resultBytes, 0);
      }
      return false;
  }
```

除了 CreateGroup、JoinGroup 外，GroupTools 还实现了 QuitGroup 和 DestroyGroup 方法。

2.UI 与 Demo 逻辑

用户登录成功以后，进入组选择的界面。可以自己创建一个组（默认组名和用户名一致），也可以加入已经存在的组（如果组存在，则加入成功；如果组不存在，则加入失败）。

加入组或创建组成功后，就会进入到群聊的界面，可以开始群聊：

群聊消息直接通过框架提供的 IGroupOutter 的 Broadcast 方法进行广播发送，客户端通过预定 IGroupOutter 的 BroadcastReceived 事件来处理接收到的群聊消息。

客户端通过 IGroupOutter 的 GroupmateConnected 和 GroupmateOffline 事件，来得到组友的上下线通知。

要注意的是，我们在 demo 中还处理了以下几种情况：

（1）成员加入组或组成员上线的时候，客户端会将其显示在组成员列表中。

（2）如果组成员掉线或退出组，服务端自动将其从组中移除，客户端则将其从组成员列表中移除。

（3）如果自己掉线重连成功后，会重新加入之前的组。

四.源码下载

（1）ESFramework.Demos.DynamicGroup 源码

（2）ESFramework.Boosts 源码：ESFramework 增强库，里面有动态组（DynamicGroup）的标准可复用实现。

简单的网络硬盘 Demo（附源码）

FTP 服务器最核心的功能就是提供文件的上传、下载服务。在 ESFrameworkDemo——文件传送 Demo(附源码)一文中，我们演示了如何在客户端与客户端之间相互传送文件，现在我们就实现一个简单的 FTP 服务器，以演示如何在客户端与服务器之间传送文件。在阅读本文之前，请务必先掌握 ESFramework 开发手册（03）——文件（夹）传送一文中介绍的文件传送的流程及相关的 API 的用法。

本 Demo 主要演示以下功能：

（1）客户端浏览服务器默认目录下的所有文件。

（2）客户端上传文件到服务器的默认目录下。

（3）客户端可以下载服务器默认目录下任何一个文件。

一.定义信息类型

根据上面提到的功能需求，我们需要定义相应的信息类型：

API：

```
public static class FtpInformationTypes
{
    /// <summary>
    /// 获取所有文件名。C->S
    /// </summary>
    public const int GetAllFileNames = 0;

    /// <summary>
    /// 请求下载文件。C->S
    /// </summary>
    public const int DownloadFile = 1;
}
```

上传文件就不用定义额外的信息类型了，可以直接使用 IFileOutter 的请求发送文件方法就可以了。

二.服务端

服务端将文件目录设定在运行目录下的"FileFold"文件夹，所有上传的文件都将被保存到这个目录，所有要下载的文件也来自这个目录。

服务端的 CustomizeHandler 类实现了自定义信息处理器接口 ICustomizeHandler，当收到来自客户端的 FtpInformationTypes.GetAllFileNames 同步调用时，就将 FileFold 目录下的所有文件列表返回给客户端。当收到请求下载文件的信息时，就调用 IFileController.BeginSendFile 方法将指定的文件发给客户端。

当客户端要上传文件时，会直接调用 IFileOutter 的 BeginSendFile，此时，服务端将触发 IFileController 的

FileRequestReceived 事件。所以，服务端需要预定并处理这个事件：

API：

```
    void fileController_FileRequestReceived(string fileID, string senderID, string fileName, ulong
fileLength, ESPlus.FileTransceiver.ResumedProjectItem resumedProjectItem, string comment)
    {
            int index = fileName.LastIndexOf('\\');
            string filePath = string.Format(@"{0}FileFold\{1}",
AppDomain.CurrentDomain.BaseDirectory, fileName.Substring(index +1));
            this.fileController.BeginReceiveFile(fileID, filePath);
    }
```

服务端将保存文件的路径设定在 FileFold 目录下，然后调用 IFileController.BeginReceiveFile 方法开始接收文件。当然，这里的处理做了很多简化，比如没有判断磁盘空间是否足够、是否有同名文件等等。

三.客户端

客户端登录成功后，进入主界面。主界面初始化时，将向服务器发送 FtpInformationTypes.GetAllFileNames 同步调用，然后将返回的文件列表显示在 ListView 中。

双击 ListView 中的某个文件时，就向服务器发送 FtpInformationTypes.DownloadFile 信息。就像上面描述的一样，服务端就会调用 IFileController.BeginSendFile 方法发送指定的文件，然后，客户端也会触发 IFileOutter.FileRequestReceived 事件，处理这个事件时，我们让用户选择要存储的路径。

API：

```
    void fileOutter_FileRequestReceived(string projectID, string senderID, string fileName, ulong
totalSize, ResumedProjectItem resumedFileItem, string comment)
    {
        if (this.InvokeRequired)
        {
            this.Invoke(new CbReadyToAcceptFileAsyn(this.fileOutter_FileRequestReceived), projectID,
senderID, fileName, totalSize, resumedFileItem, comment);
        }
        else
        {
            string savePath = ESBasic.Helpers.FileHelper.GetPathToSave("保存", fileName, null);
            if (string.IsNullOrEmpty(savePath))
            {
                this.fileOutter.RejectFile(projectID);
            }
            else
            {
                this.fileOutter.BeginReceiveFile(projectID, savePath);
            }
        }
    }
```

如果用户取消了保存路径的选择，表示放弃下载文件，这样就调用 IFileOutter.RejectFile 来进行取消操作。

当客户端点击上传按钮时，就直接调用 IFileOutter.BeginSendFile 来准备上传文件。

API：

```
    private void toolStripButton_upLoad_Click(object sender, EventArgs e)
    {
```

```
        string filePath = ESBasic.Helpers.FileHelper.GetFileToOpen("打开");
        if (filePath == null)
        {
            return;
        }
        string fileID;
        SendingFileParas sendingFileParas = new SendingFileParas(2048, 0);//文件数据包大小，可以根据网络状况设定，局网内可以设为204800，传输速度可以达到30M/s以上；公网建议设定为2048或4096或8192
        this.fileOutter.BeginSendFile(NetServer.SystemUserID, filePath, null, sendingFileParas, out fileID);
    }
```

这将引发服务端 IFileController 的 FileRequestReceived 事件触发，然后，服务端会调用 IFileController.BeginReceiveFile 方法，从而启动文件的正式传递。

下图是客户端正在进行上传下载文件时的截图：

本文是一个最简单的演示文件上传下载功能的 demo，非常的粗糙，仅仅用于示范如何使用 ESPlus 提供的文件传送功能在服务端和客户端之间传递文件。若要正式开发一个文件服务器系统，本文只能算是一个简陋的起点，还有很多复杂的事情要做，那已经超出了本文的内容，但你若有任何想法，欢迎与我们讨论。

四.源码下载

（1）ESFramework.Demos.Ftp 源码

（2）ESFramework.Boosts 源码：ESFramework 增强库，里面有网盘（NetworkDisk）完整版的标准可复用实现。

OVCS 视频会议系统 Demo

OVCS 是我们基于 ESFramework 和 OMCS 实现的一个视频会议系统的 Demo。OVCS 主要功能有：

1.多人 视频/语音/文字 会话。

（1）视频编码质量根据网络状况动态调节，且当网络拥塞时，主动弃帧。

（2）优先保证语音质量。

（3）支持回音消除（AEC）、噪音抑制（DENOISE）、自动增益（AGC）、静音检测（VAD） 等语音技术。

（4）最多支持 16 路混音。

2.多人协作 电子白板。

（1）支持常用的视图元素、可插入图片、截屏，可将整个白板保存为位图。

（2）提供观看模式和控制模式两种选择。

（3）断线自动重连，始终保持白板内容为最新。

3.文件共享。

（1）房间内任何一个成员都可以共享自己的文件，其他的成员可以下载这个文件。

（2）可随时取消共享的文件。

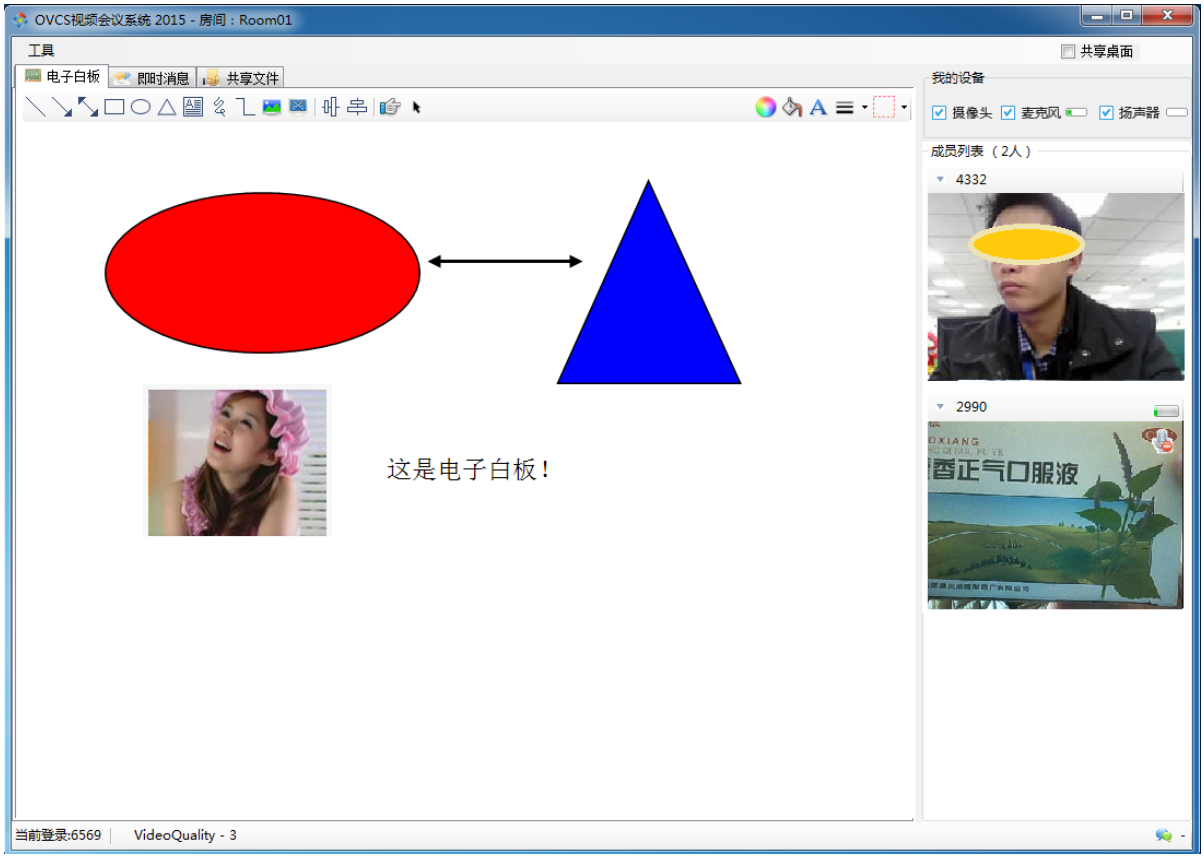
4.桌面共享

（1）桌面编码质量根据网络状况动态调节，且当网络拥塞时，主动弃帧。

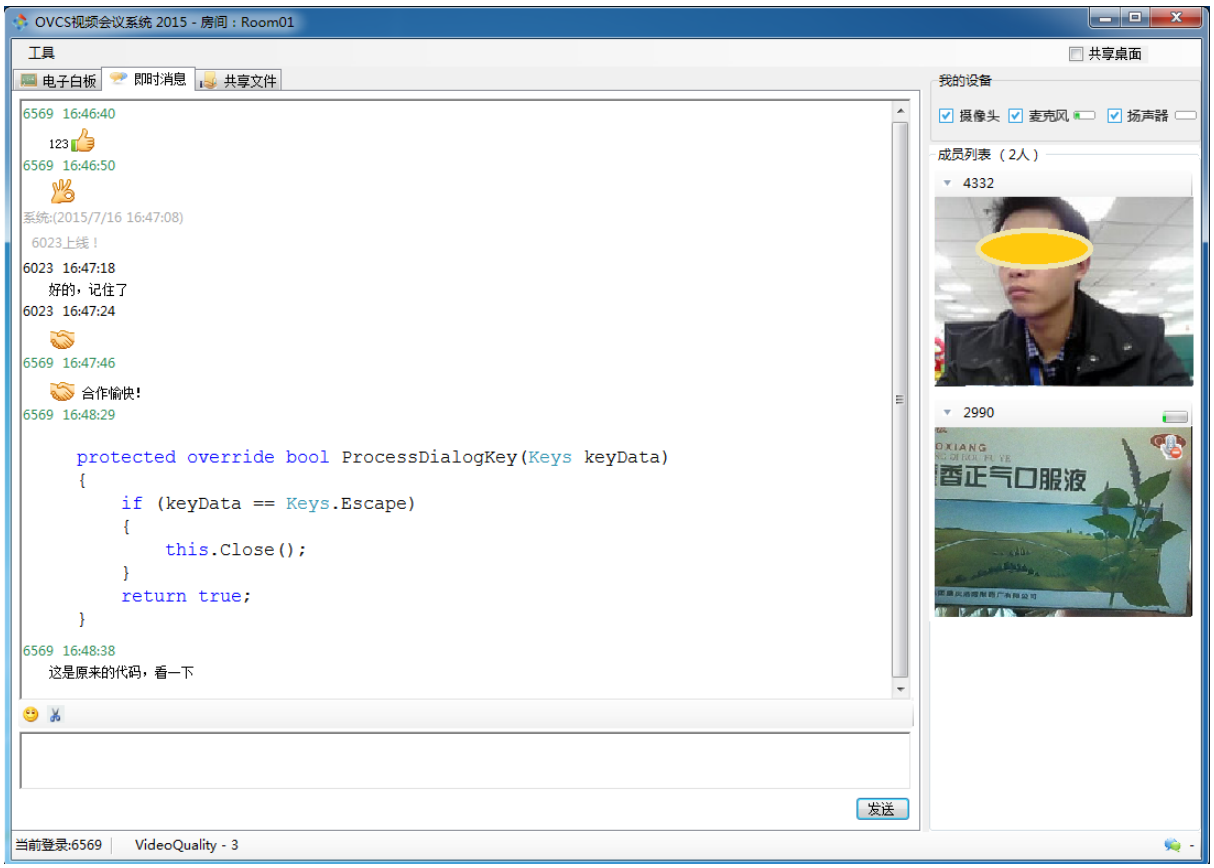
（2）房间内任何一个成员都可以共享自己的桌面，其他成员都可以观看该桌面。

（3）共享者可以授权给其他成员来操作自己的桌面。

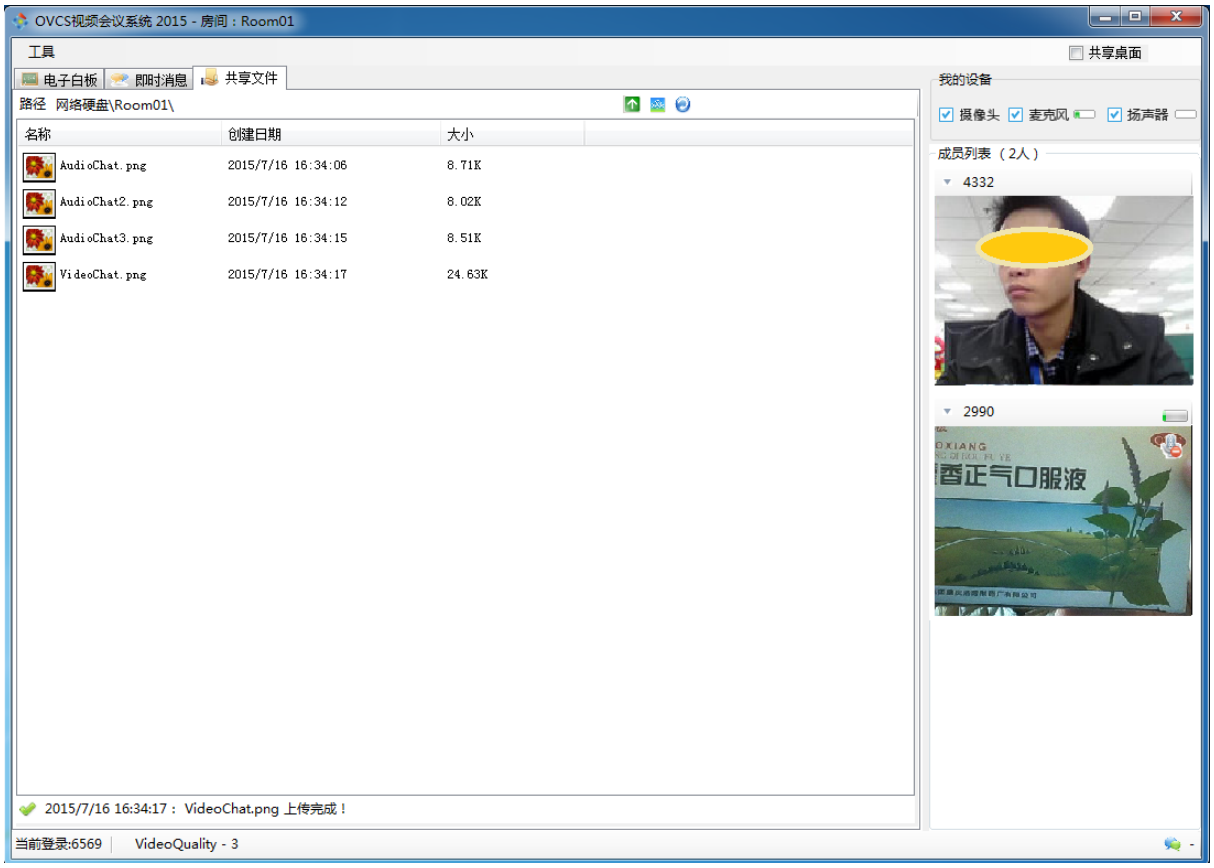
主界面，电子白板 截图：



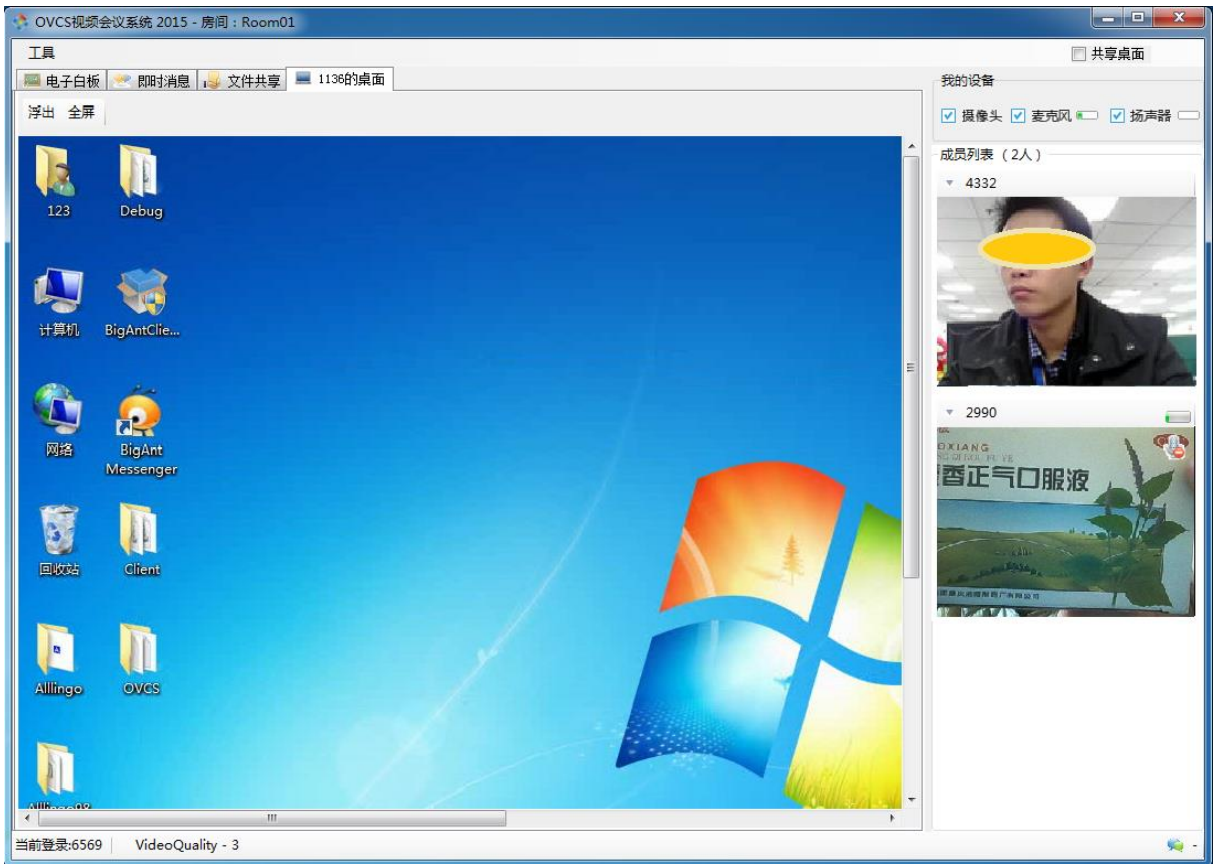
即时消息 截图：



共享文件 截图：



桌面共享 截图：



源码下载：OVCS.rar

部署说明：

（1）将 OVCS.Server 部署到服务器上，并运行起来。

（2）修改 Client 配置文件 OVCS.exe.config 中的 ServerIP 的值。

（3）运行第一个 Client 实例，以随机帐号进入测试房间。

（4）在别的机器上继续运行 Client，以随机帐号进入测试房间，大家即可在测试房间中进行视频会议。

注意：

1. 语音视频数据都是实时采集、实时播放的数据，所以测试时，服务器的带宽要求最好是独享带宽，共享带宽一般无法满足实时语音视频的要求。

对带宽的具体要求可参见 OMCS 带宽占用及网络品质测试。

2. 关于视频会议系统中 OMCS 相关视频参数的设置可参见 OMCS 开发手册（08）—— 多人语音/视频 的最后一部分讲解。

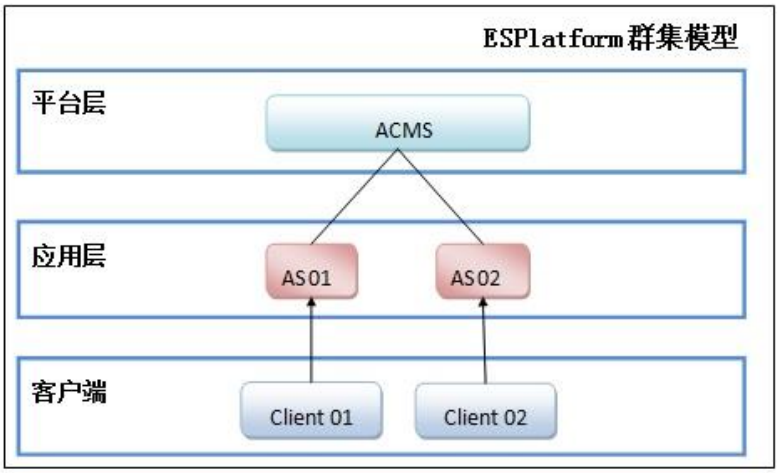
3. 如果视频或声音有卡顿现象，需要检查各客户端到服务器的网络状况（包括上行与下行）。另外，也可以调整摄像头采集分辨率和帧频以降低对带宽的要求。比如，将 IMultimediaManager 的 CameraVideoSize 设置为（160x120），MaxCameraFrameRate 设置为 6 。

ESPlatformDemo —— 入门 Demo，应用服务器群集（附源码）

ESPlatform 用于将基于 ESFramework 构建的服务器群集起来，关于 ESPlatform 的详细介绍可参见开发手册中的文章（概念与模型、迁移到群集平台、从外部访问群集）。本文我们将实现一个 Demo，来展现 ESPlatform 提供

的服务器群集以及跨服务器通信等功能。

还记得我们前面提供的 ESFramework 的入门 Demo，它演示了客户端与服务器、以及客户端与客户端之间的基本通信功能。只不过，在 ESFramework 的那个 Demo 中，相互通信的客户端连上的是同一个服务端。本文的 Demo 就是在那个老的 ESFrameworkDemo 的基础上来进行升级，使得位于不同服务器上的两个客户端之间也可以相互通信。本 Demo 运行起来的结构简化后是这样的：



一.项目结构

本 Demo 总共包含 4 个项目。

1.ESPlatform.ACMServer：这个是由官方提供的应用群集服务器 ACMS 的源码，本 demo 直接将其拷贝过来，未做任何修改。

2.ESPlatform.SimpleDemo.Core：用于定义公共的信息类型、通信协议。

3.ESPlatform.SimpleDemo.Server：Demo 的服务端。

4.ESPlatform.SimpleDemo.Client：Demo 的客户端。

二.应用群集管理服务器 ACMS

ACMS 由官方提供，本 Demo 没有任何特殊需求，所以，不需要对其进行任何修改。我们只需要关注配置文件中，TransferPort 和 Remoting 端口的值。

API：

```
<configuration>
  <appSettings>
    <!——应用群集中的服务器分配策略——>
    <add key="ServerAssignedPolicy" value="MinUserCount"/>
    <!——用于在AS之间转发消息的Port——>
    <add key="TransferPort" value="12000"/>
  </appSettings>

  <system.runtime.remoting>
    <application>
      <channels>
        <!——提供IPlatformCustomizeService和IClusterControlService Remoting服务的Port——>
        <channel ref="tcp" port="11000" >
          <serverProviders>
            <provider ref="wsdl" />
```

```
            <formatter ref="soap" typeFilterLevel="Full" />
            <formatter ref="binary" typeFilterLevel="Full" />
          </serverProviders>
          <clientProviders>
            <formatter ref="binary" />
          </clientProviders>
        </channel>
      </channels>
    </application>
  </system.runtime.remoting>
</configuration>
```

三.Demo 服务端

在升级老的 Demo 时，首先需要使用企业版的 ESFramework.dll 替换专业版，然后，使用其中的 ESPlatform.Rapid.RapidServerEngine 替代 ESPlus.Rapid.RapidServerEngine，并在构造函数中指定：当前服务端实例的 ID、ACMS 的 IP 地址及其 TransferPort 和 Remoting 端口。

API：

```
    //使用简单的好友管理器，假设所有在线用户都是好友。（仅仅用于demo）
    ESPlatform.Server.DefaultFriendsManager friendManager = new
ESPlatform.Server.DefaultFriendsManager();
    this.engine = new ESPlatform.Rapid.RapidServerEngine(this.textBox_serverID.Text,
this.textBox_acmsIP.Text,
int.Parse(this.textBox_acmsPort.Text) ,int.Parse(this.textBox_transferPort.Text));
    this.engine.FriendsManager = friendManager;
    this.engine.Initialize(int.Parse(this.textBox_serverPort.Text), new CustomizeHandler(), new
BasicHandler());
    friendManager.PlatformUserManager = this.engine.PlatformUserManager;
```

然后，本 demo 更换了主界面，直接在主界面可以指定相关配置信息，而其它的部分与老 Demo 完全一致。

四.Demo 客户端

相对于老的 Demo 而言，客户端的修改非常小，只是将配置文件中的服务器的 IP 和端口移到了登录界面上，这样方便指定要连接的服务端的地址。除此之外，没有其它变化，甚至，客户端的项目都不需要引用 ESPlatform.dll。

五.运行 Demo

1.启动应用群集管理服务器 ACMS。

2.启动第一个服务端，ServerID 指定为 0，监听 6000 端口。

3.启动第二个服务端，ServerID 指定为 1，监听 6001 端口。

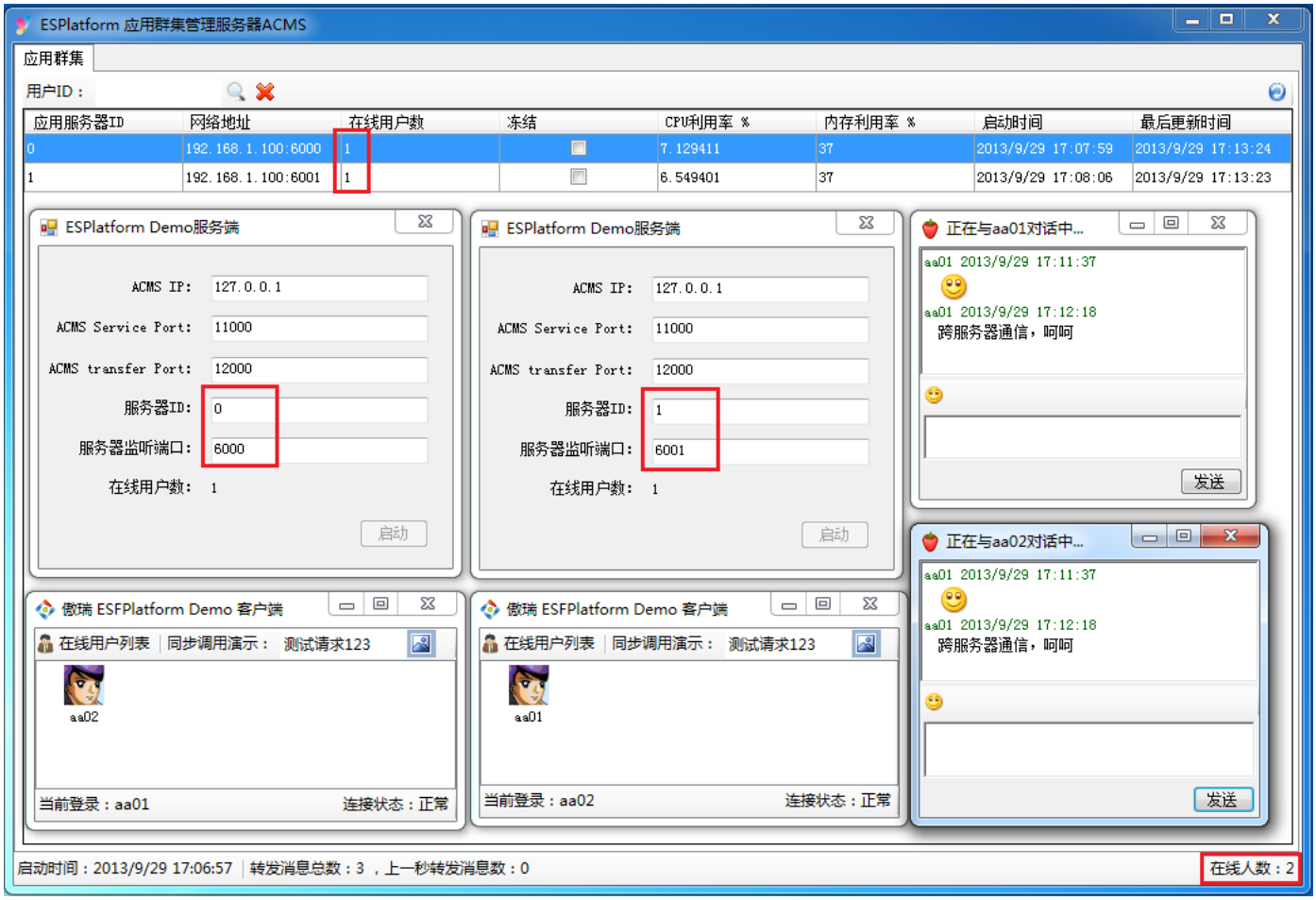
4.启动第一个客户端，连接 ServerID 为 0 的服务端（6000 端口）。

5.启动第二个客户端，连接 ServerID 为 1 的服务端（6001 端口）。

6.两个客户端之间可以相互对话了。

（在正式的应用场景中，ACMS、两个服务端、两个客户端可以部署在不同的机器器上）

下图是 Demo 运行起来的效果：



六.Demo 下载

[ESPlatform.SimpleDemo 源码]

ESFramework 相关

ESFramework 版本变更记录

第 11 次 版本更新

更新时间：2017.01.02

最新版本：ESFramework 6.2.0.0

更新列表：

（01）文件传输内核全面优化。

（02）Contacts 空间的 Broadcast 方法增加 Tag 参数（string 类型）。

（03）IContactsController 增加 BroadcastFailed 事件。当因为某个组成员不在线而导致对其广播失败时，将触发该事件。

（04）修复 bug：IContactsOutter 的 GetContacts() 方法 调用会超时 Timeout。

（05）修复 bug：WebSocket 内核解析消息有时会失败。

（06）其它小细节优化和已发现 bug 的修复。

第 10 次 版本更新

更新时间：2016.02.29

最新版本：ESFramework 6.0.0.0

更新列表：

（01）ESFramework 增加 WebSocket 客户端引擎。

（02）ESFramework 增加 Xamarin 版本（Android 和 iOS）。

（03）增加“联系人”功能，用于取代之前的“好友与组”。

（04）多文件同时发送性能优化。

（05）优化可靠 UDP 的算法，提高基于 UDP 的 P2P 传送效率。

（06）紧凑的序列化器 CompactPropertySerializer 增加了对泛型类型的支持。

（07）IRapidPassiveEngine 和 IRapidServerEngine 增加了 Advanced 属性，提供诸多高级的引擎控制选项。

（08）修复 bug：传送文件夹时，接收方子文件夹下的文件存储路径储错误。

（09）其它小细节优化和已发现 bug 的修复。

第 09 次 版本更新

更新时间：2015.03.23

最新版本：ESFramework 5.0.0.0

更新列表：

（01）IRapidPassiveEngine 增加性能跟踪诊断功能。

（02）IRapidPassiveEngine 和 IRapidServerEngine 增加 SendMessage 方法和 MessageReceived 事件，以更便捷的方式收发消息。

（03）CompactPropertySerializer 增加对 Image、Font、Color 的支持。

（04）IGroupController 增加 BroadcastBlobListened 属性，如果设置为 true，则即使客户端广播的是 blob 消息，也将触发 IGroupController 的 BroadcastReceived 事件。

（05）IBaseFileController 增加 GetTransferingProgress 方法，可获取传送项目的传送进度。

（06）文件传送 0 速度检测：如果在过去 20 秒内发送或接收的字节数为 0，则中断文件传送，原因为 NetworkSpeedSlow。

（07）P2PChannelMode 枚举增加 None 枚举值。

（08）ESPlatform 群集平台新增功能：ACMS 定时 telnet 群集中每一 AS 的端口，若发现端口不通，则将其状

态设为 TelnetFailure，并不再将其分配给后来的客户端。

（09）可靠 UDP 效率再度优化提升。

（10）修改部分，完全向下兼容。

（11）所有已发现 bug 的修复。

第 08 次 版本更新

更新时间：2013.12.10

最新版本：ESFramework 4.5.0.0

更新列表：

（01）优化 ESPlatform 平台的整体结构。

（02）提升群集平台中服务器之间的通信效率。

（03）文件传送功能增强：允许传送正在被写的文件。

（04）所有已发现 bug 的修复。

（05）ESFramework 4.5 版本相对于 4.2 而言，程序集形式和接口有稍许改变，升级时要注意以下几点：

（06）V4.5 版本中，ESPlus.dll 已经合并到 ESFramework.dll 中。

（07）RapidServerEngine 类 不再存在，改为使用 IRapidServerEngine 接口。

（08）RapidPassiveEngine 类 不再存在，改为使用 IRapidPassiveEngine 接口。

（09）由 RapidEngineFactory 的静态方法 Create****Engine()来创建引擎实例。

（10）其它部分，完全向下兼容。

第 07 次 版本更新

更新时间：2012.11.20

最新版本：ESFramework 4.2.0.0，ESPlus 3.4.0.0

更新列表：

（01）完整支持 IPv6。

（02）框架的默认最大消息尺寸由 100K 改为 1M。

（03）支持使用域名作为服务端地址。

（04）当日志文件增加到 IM 时，将自动创建一个新的日志文件。

（05）IBaseFileController 的 CancelTransfering 方法增加重载，可传入取消的原因。

（06）IBaseFileController 的 RejectFile 方法增加重载，可传入拒绝的原因。

（07）IBaseFileController 的 BeginReceiveFile 方法增加重载，可以指定是否启用续传。

（08）ICustomizeOutter 发送消息的方法增加通道选择模型参数 ChannelMode。

（09）ICustomizeOutter 的 SendBlob 方法执行中途，如果连接中断，则将抛出异常。

（10）ICustomizeController 的 QueryLocalClient 方法增加重载，以支持回复异步调用。

（11）ICustomizeController 增加 InformationReceived 事件。

（12）IP2PController 增加 AllP2PChannelClosed 事件。

（13）当客户端 Rapid 引擎关闭时(Close 方法)，释放所有使用的线程。

（14）修复在某些情况下，客户端 Rapid 引擎持续进行断线重连的 bug。

第 06 次 版本更新

更新时间：2012.04.23

最新版本：ESFramework 4.0.9.0，ESPlus 3.0.0.0

更新列表：详细介绍

（1）基于 UDP 的 P2P 优化：增强 UDP 引擎性能、降低重发率、增大发送速度。

（2）TCP 客户端引擎优化：减少初始化所需的启动时间。

（3）将好友与组二项，由 Rapid 引擎的必需两翼，转变成可选功能。

（4）客户端增加异步投递消息（Post）、回复异步调用、繁忙时丢弃消息等功能。

（5）增加发送大数据块(Blob)的功能。

（6）可独立部署的 P2P 服务器。

（7）修复基于 TCP 的 P2P 通道偶尔会自动关闭的 bug。

第 05 次 版本更新 【归入 武汉傲瑞科技有限公司】

更新时间：2011.10.08

最新版本：ESFramework 4.0.7.0，ESPlus 2.0.0.0

更新列表：

（1）可靠的 UDP。在原始 UDP 的基础上再次封装，模拟 TCP 机制，以保证基于 UDP 的通信像 TCP 一样可靠。

（2）强大的 P2P。P2P 打洞的成功率在 90%以上，而且，对于开发者，P2P 是透明的，如果底层的 P2P 通道可用，则客户端之间的通信直接使用 P2P 通道传送。

（3）文件夹传送。前面的版本，仅仅支持单个文件的传送，而在新版本中，可以直接传送整个文件夹（采用与传送文件完全相同的模型和 API）。

（4）部分重构和重命名。客户端的供框架回调的 IBasicBusinessHandler 接口、IFileBusinessHandler 接口都被删除了，其中的回调方法都转换成了对应*Outter 的事件。这将使开发者用起来更方便。

第 04 次 版本更新

更新时间：2011.04.25

最新版本：ESFramework v4.0.5.0 , ESPlus v1.3.2.0

更新列表：

（1）ESFramework 在其 ESFramework.Server.UserManagement 空间下增加了 IPlatformUserManager 接口，用于支持对 ESPlatform 群集中所有在线用户的管理。

（2）IBasicOutter 增加了 Logon 方法，用于客户端登录验证；ESPlus.Application.Basic.Server 命名空间增加了 IBasicBusinessHandler 接口，实现该接口可以验证用户的登录密码。

（3）基于（2），客户端 Rapid 引擎增加了客户端初始化时验证用户登录密码的功能。

（4）客户端 Rapid 引擎增加了 CurrentUserID 属性，其值为当前登录成功的用户的 UserID。

（5）服务端 Rapid 引擎增加了 PlatformUserManager 属性（get），通过该属性可以获取群集中所有在线用户信息；在非群集状态下，PlatformUserManager 等同于 UserManager。

（6）服务端 Rapid 引擎增加了对文件传送的支持，即服务端可以参与文件的收发，从而可以实现类似 FTP 的文件上传下载功能。

第 03 次 版本更新

更新时间：2011.04.18

最新版本：ESPlus v1.3.0.0

更新列表：

（1）ESPlus.Application.Basic.Passive.IBasicOutter 增加了获取在线的好友列表（GetAllOnlineFriends 方法）、获取所有好友列表（GetFriends 方法）、获取在线组友（GetAllOnlineGroupmates 方法）等功能。

（2）ESPlus.Application.Basic.Passive.IBasicBusinessHandler 增加了组友上下线通知（OnGroupmateConnected 方法、OnGroupmateOffline 方法）等功能。

（3）IGroupManager 增加了 GetGroupmateList 方法以获取组友列表，以支持上述的获取在线组友、组友上下线通知的功能。

（4）IGroupManager 增加了 GetOwnerGroupIDList 方法以获取目标用户加入的所有组的 ID 集合，该方法将被用于 ESPlatform。

（5）ESPlus.Application.CustomizeInfo.Passive.ICustomizeInfoOutter 增加了使用 ACK 机制发送自定义信息给服务端或其它在线用户的功能（SendCertainly 方法）。

（6）ESPlus.Application.CustomizeInfo.Server.ICustomizeInfoController 增加了使用 ACK 机制发送自定义信息给客户端的功能（SendCertainly 方法）。

（7）ESPlus.Rapid.IRapidServerEngine 增加了 FriendNotifyEnabled 属性，以控制当用户上线/掉线时，是否通知其好友。

（8）ESPlus.Rapid.IRapidServerEngine 增加了 GroupNotifyEnabled 属性，以控制当用户上线/掉线时，是否通知其组友（groupmate）。

第 02 次 版本更新

更新时间：2011.04.12

最新版本：ESPlus v1.2.1.0

更新列表：

（1）ESPlus.Application.Basic.Passive.IBasicOutter 增加了查询用户是否在线的功能（IsUserOnline）。

（2）ESPlus.Application.CustomizeInfo.Passive.ICustomizeInfoOutter 增加了客户端可以同步调用其它在线客户端的功能（CommitP2PRequest 方法）。

（3）ESPlus.Application.CustomizeInfo.Server.ICustomizeInfoController 增加了服务端可以同步调用在线客户端的功能（QueryClient 方法）。

（4）ESPlus.Application.CustomizeInfo.Server.ICustomizeInfoController 增加了 InformationReceived 事件，以监控所有自定义信。

（5）为了支持上述功能，ESPlus.Application.CustomizeInfo.Passive.ICustomizeInfoBusinessHandler 回调接口增加了相应的方法，如果不需要用到新增的功能，可以直接实现为空，或者返回 null。

第 01 次 正式发布

发布时间：2010.08.12

发布版本：ESFramework v4.0.0.0，ESPlus v1.0.0.0 。

ESFramework FAQ —— ESFramework 常见问题解答

关于在使用通信框架 ESFramework 过程中遇到的常见问题，我们做了一个整理总结，并给出了解决问题的步骤

或方案，大家可以按照指定的步骤发现和解决问题。

一. 常用网络检测命令

1. Ping —— 网络是否连通？延迟有多大？网络抖动情况如何？

如： ping 192.168.0.123 -t

2. telnet —— tcp 端口能连上？

如： telnet192.168.0.123 4530

3. tracert —— 当到服务器的网络不顺畅时，可发现路由上是哪些节点“卡”？

如： tracert -d 192.168.0.123

4.如何查看局域网中的电脑对外的公网 IP 地址？

在浏览器中打开网址 www.whereismyip.com，页面将显示当前电脑的对外 IP。

5.如何让 Windows 服务器允许 ping？

如果服务器禁止了 ping，可按如下操作：控制面板->Windows 防火墙->高级->ICMP->设置->将"允许传入回显请求"项勾上。

二. 常见异常信息解决

1. IRapidPassiveEngine 的 Initialize 方法抛出异常：Timeout waiting for reply !

（1）客户端和服务端的 UserID 的最大长度是否一致。可查找 GlobalUtil 的 SetMaxLengthOfUserID 静态方法。

（2）ping 一下服务器，看到服务器的网络情况如何。

（3）如果使用的是 ESFramework 的试用版本，则检查 SDK 是否已经过期。

2. ICustomizeOutter 的 Query 方法抛出异常：There is an exception occured when {0} handing the query of {1}.

（1）该异常说明 Query 请求的处理方（可能是服务器或另一个在线客户端）在处理该请求时出现了异常，可以查看处理方的 ESF 日志来得到异常的详细信息。

（2）异常信息中的第一个占位符{0}的值，是处理方的 UserID。如果是"_0"，则表示是服务端。

（3）异常信息中的第二个占位符{1}的值，是自定义请求信息的类型，对应 Query 方法中的 informationType 参数。

3. ICustomizeOutter 的 Query 方法抛出异常：Timeout waiting for reply !

（1）该异常含义是等待 Query 的处理回复超时。

（2）超时的常见原因有如下两个：Query 请求的处理方处理该请求使用了太多的时间；或者，网络缓慢，而且回复的数据量太大，需要很长的传送时间。

（3）解决方案：

a. 优化请求处理的流程与算法，加快请求处理速度。

b. 将等待回复的最大超时设置得大一些。（对应 IRapidPassiveEngine 的 WaitResponseTimeoutInSecs 属性，默认值为 30 秒）。

c. 使请求的数据和回复的数据都尽可能地小。

d. 如果 c 无法做到，则建议不要使用同步调用机制 Query，改用 SendBlob 方法，请求信息和/或回复信息使用 SendBlob 进行发送（内部使用分块发送机制）。

4. ICustomizeOutter 的 Query 方法抛出异常：Network connection is disconnected !

（1）该异常含义是等待 Query 回复的过程中，与服务器的 TCP 连接断开了。

（2）TCP 连接断开虽说大部分是网络原因，但是，要注意，在网络不是很顺畅时，一次性发送比较大块的数据，可能会导致 TCP 连接断开。

（3）为了防止（2）的情况出现，对于请求信息或回复信息的数据较大（比如超过 4k）的情况，那么 Query 就不再适合，建议改用 SendBlob 方法（内部使用分块发送机制）。

三. 疑问解答

1. IRapidPassiveEngine 与服务端的连接断开后，一直没有自动重连成功的原因有哪些？

（1）与服务器之间的网络是否已经恢复？

（2）IRapidPassiveEngine 的 AutoReconnect 属性是否为 false？

（3）当前客户端是否是被挤掉线（即在其它地方登录了同一个帐号，会触发 IBasicOutter 的 BeingPushedOut 事件）？如果是被挤掉线的，则不会自动重连。

（4） 当前客户端是否是被踢掉线的（会触发 IBasicOutter 的 BeingKickedOut 事件）？如果是被踢掉线的，则不会自动重连。

（5） 是否在重连时，服务端验证其帐号密码失败了？（IRapidPassiveEngine 的 RelogonCompleted 事件的 LogonResponse 参数会反应出来）

2. 向对方发送自定义信息，对方没有进入 ICustomizeHandler 的 HandleInformation 方法？

（1）发送自定义信息时，传入的对方的 UserID 参数的值是否正确？必须要与对方的登录帐号完全一致（严格区分大小写的）。

（2）自定义信息处理器（ICustomizeHandler）的实例是否注入到了 Rapid 引擎？

（3）如果处理器是实现的 IIntegratedCustomizeHandler 接口，那么，查看其 CanHandle 方法的实现，看发送的信息的类型是否在其处理范围内？

3. 客户端引擎 IRapidPassiveEngine 初始化时，serverIP 这个参数能使用域名吗？

方案 先将域名转换成IP 然后再传给Initialize方法。System.Net 命名空间下有个Dns类 ,它的GetHostAddresses方法就可以将域名转换成 IP。

四. 故障排查

1. 服务端性能瓶颈，该如何排查？
2. 服务器端口 telnet 失败，该如何定位问题？
3. 客户端批量心跳超时掉线，是什么原因？

五. 综合

1. asp.net 网站如何与 ESFramework 服务端集成？

比如，当通过网站注册了一个新会员时，需要告诉 ESFramework 的服务端这一信息。那么，通常的解决方案是：

（1）ESFramework 服务端发布一个 Remoting 服务，该服务暴露一个接口方法用于接收会员注册的通知。

（2）当网站这边注册一个会员成功的时候，网站就主动调用 ESFramework 服务端暴露的 Remoting 方法通知它。

我们还有一个复杂一点的 B/S 网站与 ESPlatform 群集平台集成的例子，可以参见《ESPlatform 群集平台（02）—— 从外部访问群集 》的文末。

2.如何为基于 ESFramework 开发的程序增加新的客户端类型（如 iOS、android 等）？

如果希望为已经开发好的基于 ESFramework 的系统增加其它类型的客户端，那么需要做到两点：

（1）根据客户端的平台类型，选择 ESFramework 的对应版本。

比如，新增的客户端类型是 android，那么，在开发 android 客户端时，就要基于 ESFramework 的 andorid 版本来进行。

（2）新的客户端要遵循应用层的消息的协议格式。

一般情况下，基于 ESFramework 开发的应用（如 OrayTalk、GG 等）的内部消息也是使用 ESFramework 提供的紧凑的序列化器来进行序列化和反序列化的。这时，情况就容易一些了，我们有提供了一个小工具，可以根据协议类的定义，自动生成对应的协议格式。具体可参见：ESFramework 使用技巧 —— 协议格式自动生成器（跨平台开发小工具。

3. 如何让 ESFramework 同时支持带 SSL 的 WebSocket ？

（1）现已部署的基于 ESFramework 的服务端程序（A 服务）不需要做任何修改。

（2）添加一个支持 SSL 的 Nginx Server 作为代理（B 服务），配置该服务器，将请求转发到上述的真正的 A 服务的端口上。（如何配置 Nginx 服务器，可参见这里。）

（3）如果一个客户端是普通的 WebSocket，则还是直接连 A 服务；如果一个客户端是带 SSL 的 WebSocket，则连接 B 服务。

ESFramework4.0 性能测试 —— 内核测试

ESFramework 底层使用最高效的 IOCP（IO 完成端口）模型，使得数据收发与处理达到最高性能。另外，ESFramework 只会在需要时才使用必要的资源（如 CPU、内存），并且会及时释放持有的资源，可以超长时间（比如数年）稳定运行，绝不会有内存泄露、死线程堆积等情况发生。由于 ESFramework 和 StriveEngine 使用的是同样的底层内核，所以本测试的结果对 StriveEngine 也是适用的。

本实验用于测试 ESFramework/StriveEngine 服务端引擎内核的性能和稳定性，测试程序使用最新发布的 ESFramework4.0 版本。

一．准备工作

测试的机器总共有 3 台，都是普通的 PC，一台作为服务器，两台作为客户端。（测试时间：2010 年 9 月 11 日）

作为服务器是 PC 配置如下：

操作系统：WindowsServer2003EnterpriseEditionSP2

CPU：PentiumDual-CoreCPUE5400@2.70GHz

内存：2G

二．测试策略

本实验所采用的策略是这样的：

（1）每个客户端实例首先与服务器建立 N 个 TCP 连接，然后依次在每个 TCP 连接上发送一个 36 字节的消息。遍历一次完毕后，等待（Sleep）M 毫秒，再进行下一轮遍历发送。

（2）服务端接收到消息后，解析消息，然后累加消息的个数。

（3）客户端统计已发消息的总数，并计算上一秒发送的请求数。

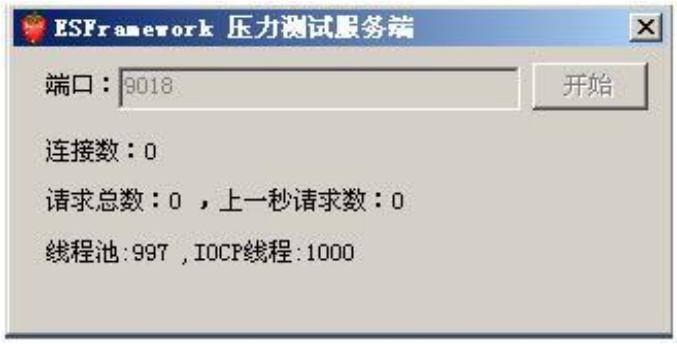
（4）服务端统计已接收消息的总数，并计算上一秒接收的请求数。

三．测试过程

测试方案一：

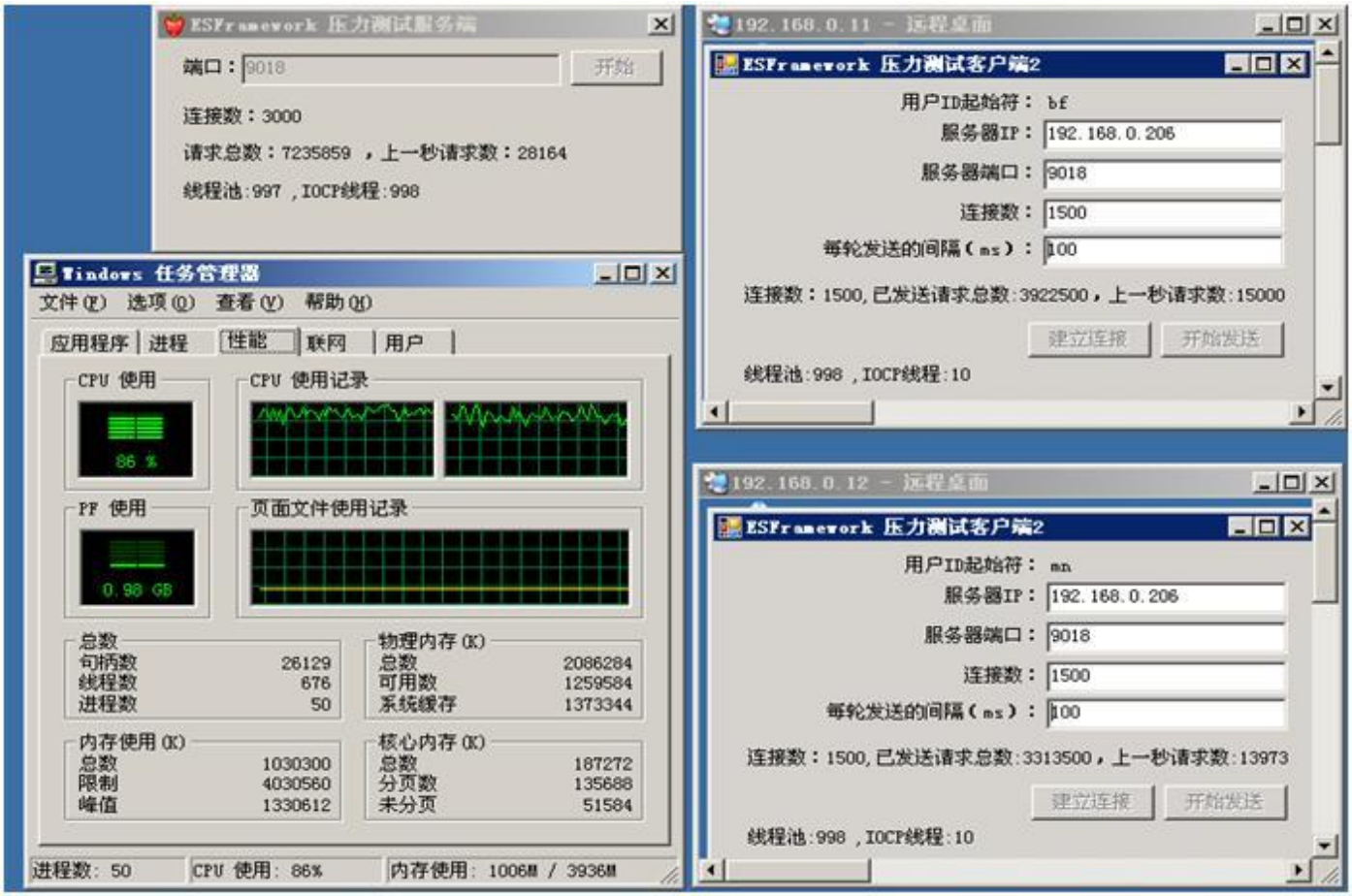
连接总数 3000，每轮发送间隔 100ms

（1）在作为服务器的 PC 上启动服务端。

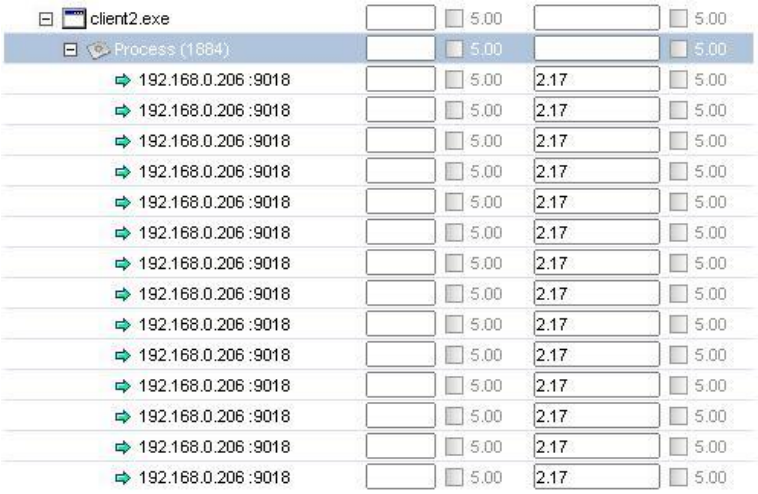


（2）在作为客户端的两台 PC 上分别运行一个客户端实例。每个客户端实例设定连接数 1500，每轮发送的间隔为 100。

（3）如此，服务端的总的连接数为 3000，以下是运行一段时间后的截图：



在该测试过程中，服务端的每秒处理的消息数量在 26000~30000 之间波动，而 CPU 持续在 80%以上。另外，在客户端的 PC 上通过 NetLimitter 可以看到每个连接上发出的数据流量：

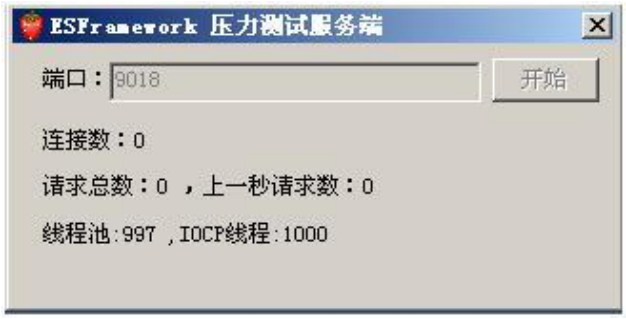


测试方案二：

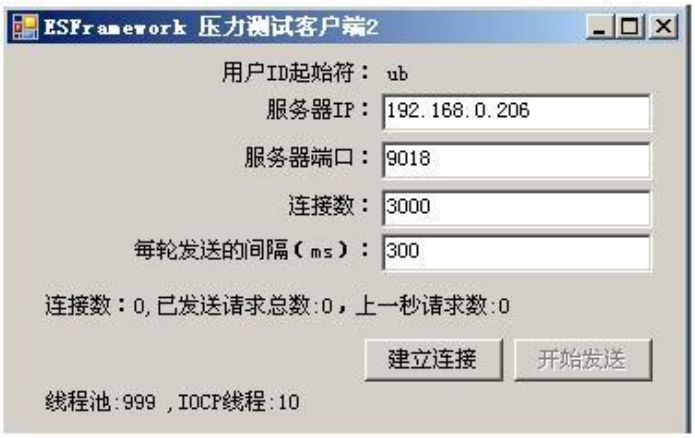
连接总数 6000，每轮发送间隔 300ms

将方案一启动的各程序全部关掉，重头再来一次。

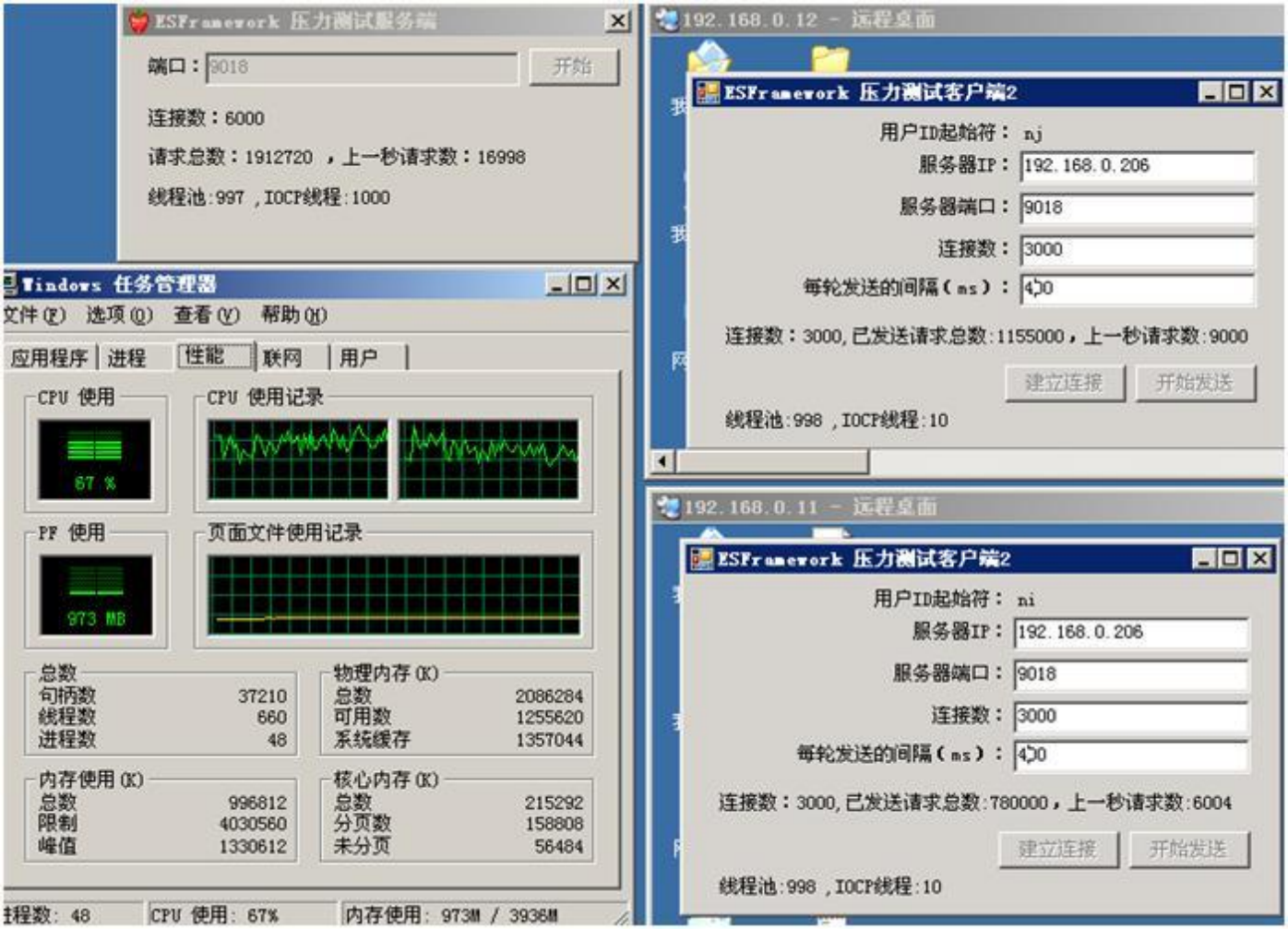
（1）在作为服务器的 PC 上启动服务端。



（2）在作为客户端的两台 PC 上分别运行一个客户端实例。每个客户端实例设定连接数 3000，每轮发送的间隔为 400。



（3）如此，服务端的总的连接数为 6000，以下是运行一段时间后的截图：



在该测试过程中，服务端的每秒处理的消息数量在 14000~18000 之间波动，而 CPU 持续在 65%以上。

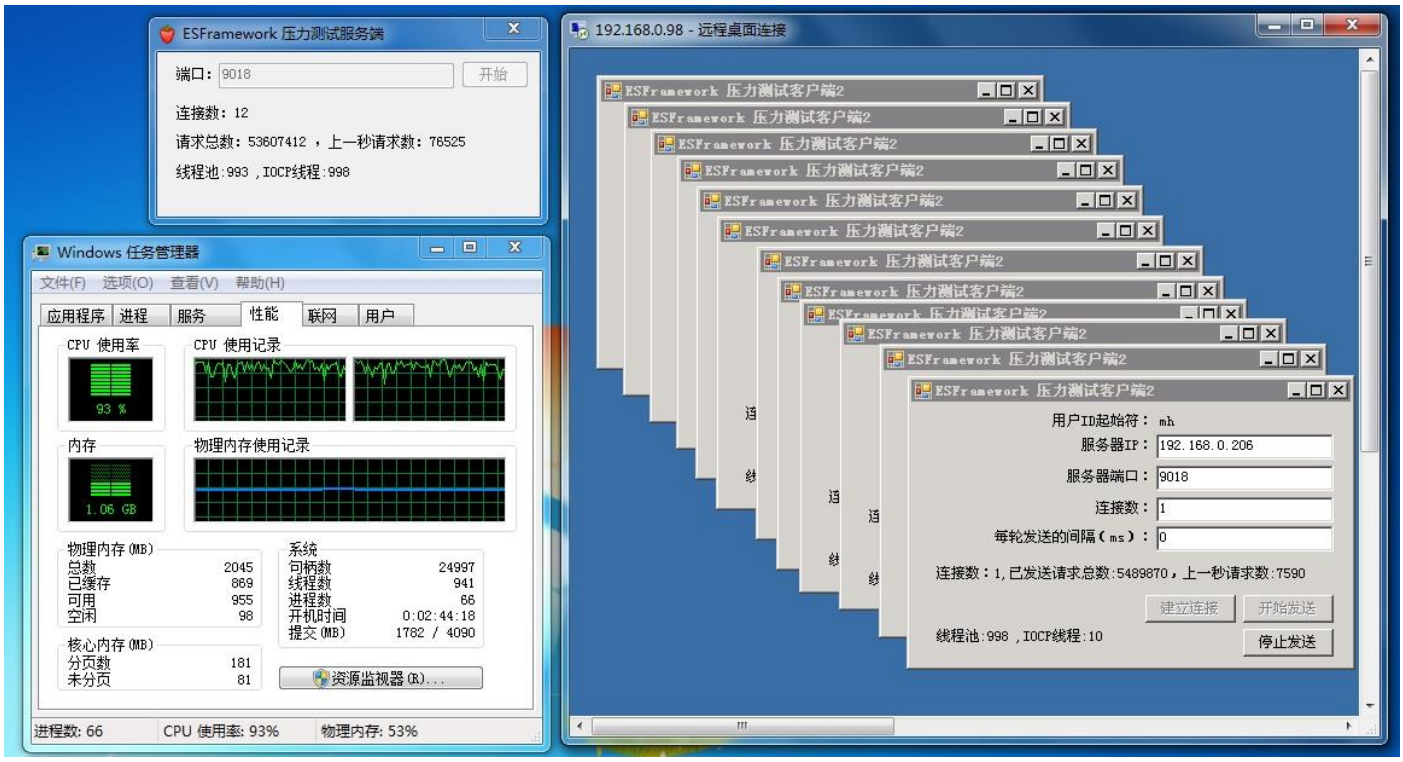
测试方案三：

我们最后测试一种极端的情况，那就是极少的连接数，极小的发送间隔。连接总数十多个，每轮发送间隔 0ms。

（1）在作为服务器的 PC 上启动服务端。

（2）在作为客户端的一台 PC 上逐个启动客户端，每个客户端设定连接数为 1，发送间隔为 0ms，这样，当增加到 12 个时，服务端的 CPU 几乎始终维持在 90%以上。

运行一段时间后的截图如下所示：



在该测试过程中，服务端的每秒处理的消息数量在 70000~80000 之间波动，而 CPU 持续在 90%以上。【方案三测试于 2012.04.01 所新增】

四．测试结论

第三种方案是比较极端的，在现实场景中很少碰到，但是作为一个极端的测试还是有必要的。一般服务器都要承载至少数千用户同时在线，所以我们主要讨论大连接数的测试方案。

对于大连接数测试，除了前两种方案的测试以外，我们还进行了其它方案的测试，比如设置更小的连接数（至少 1000）和更小的发送间隔（至少 5ms），或设置更大的连接数和更大的发送间隔。测试反映，这台作为服务器的 PC 能承载的最佳并发连接数在 4000 左右，此时，服务器的吞吐量可以达到最大（每秒处理 30000 个左右的消息）。当连接数进一步增加时，吞吐量会降下来。最佳并发连接数和最大吞吐量的值与服务器机器的配置密切相关。

五．测试程序

本文末会提供测试程序的压缩包下载。在压缩包内附带了测试的服务端和客户端程序，有兴趣的朋友可以在自己的服务器上做更多的策略测试。在自己运行测试时，要注意以下几点：

（1）服务端最好运行在一台单独的机器上。如果客户端也运行在服务端所在的机器上，则会严重地影响服务端的吞吐量。

（2）合理地设置客户端的连接数和发送时间间隔。

（3）客户端运行的机器的操作系统对 tcp 连接数可能有最大值限制（有的是三千多），如果遇到这种系统，

而单个客户端实例的连接数的设定又大于这个限制值，则会导致后续的 tcp 连接失败。若发生这种情况，请关闭客户端并重启，然后设定较小的连接数，再次测试。可以多开几个客户端实例，来增加连接数。

（4）如果服务器上有防火墙软件，可能会影响测试结果，最好关闭服务器上的防火墙进行测试。

（5）如果是在 Internet 上测试，请确保服务端带宽和客户端的带宽都足够大。

ESFramework 4.0 性能测试程序下载。