高性能 TCP & UDP 通信框架

—— HP-Socket v3.3-20150101

Bruce Liang

前 言

HP-Socket 是一套通用的高性能 TCP/UDP 通信框架，包含服务端组件、客户端组件和 Agent 组件，广泛适用于各种不同应用场景的 TCP/UDP 通信系统，提供 C/C++、C#、Delphi、E（易语言）、Java、Python 等编程语言接口。HP-Socket 对通信层实现完全封装，应用程序不必关注通信层的任何细节；HP-Socket 提供基于事件通知模型的 API 接口，能非常简单高效地整合到新旧应用程序中。

为了让使用者能方便快速地学习和使用 HP-Socket，迅速掌握框架的设计思想和使用方法，特此精心制作了大量 Demo 示例（如：PUSH 模型示例、PULL 模型示例、性能测试示例以及其它编程语言示例）。HP-Socket 目前运行在 Windows 平台，将来会实现跨平台支持。

◆ 通用性

- HP-Socket 的唯一职责就是接收和发送字节流，不参与应用程序的协议解析等工作。
- HP-Socket 与应用程序通过接口进行交互，并完全解耦。任何应用只要实现了 HP-Socket 的接口规范都可以无缝整合 HP-Socket。

◆ 易用性

- 易用性对所有通用框架都是至关重要的，如果太难用还不如自己重头写一个来得方便。因此，HP-Socket 的接口设计得非常简单和统一。
- HP-Socket 完全封装了所有底层通信细节，应用程序不必也不能干预底层通信操作。通信连接被抽象为 Connection ID，Connection ID 作为连接的唯一标识提供给应用程序来处理不同的连接。

◆ 高性能

HP-Socket 作为底层的通用框架，性能是关键指标，绝对不能成为系统的瓶颈。HP-Socket 在设计上充分考虑性能、使用场景、复杂性和易用性等因素，作出以下几点设计决策：

- **Client 组件：**基于 Event Select 通信模型，在单独线程中执行通信操作，避免与主线程或其他线程相互干扰。每个组件对象管理一个 Socket 连接。
- **Server 组件：**基于 IOCP 通信模型，并结合缓存池、私有堆（Private Heap）等技术，支持超大规模连接，在高并发场景下实现高效内存管理。
- **Agent 组件：**对于代理服务器或中转服务器等应用场景，服务器自身也作为客户端向其它服务器发起大规模连接，一个 Agent 组件对象同时可管理多个 Socket 连接；Agent 组件与 Server 组件采用相同的技术架构，可以用作代理服务器或中转服务器的客户端部件。

◆ 伸缩性

应用程序可以根据不同的容量要求、通信规模和资源状况等现实场景调整 HP-Socket 的各项性能参数（如：工作线程的数量、缓存池的大小、发送模式和接收模式等），优化资源配置，在满足应用需求的同时不必过度浪费资源。

目 录

1 概 述

1.1 整体架构

HP-Socket 完全封装了底层通信细节，并为应用程序提供一套简单易用的并且与底层通信完全无关的 API 接口，使应用程序获得高性能、高伸缩性通信的同时，免除处理通信细节的负担。HP-Socket 的 API 接口模型如图 1.1-1 所示：

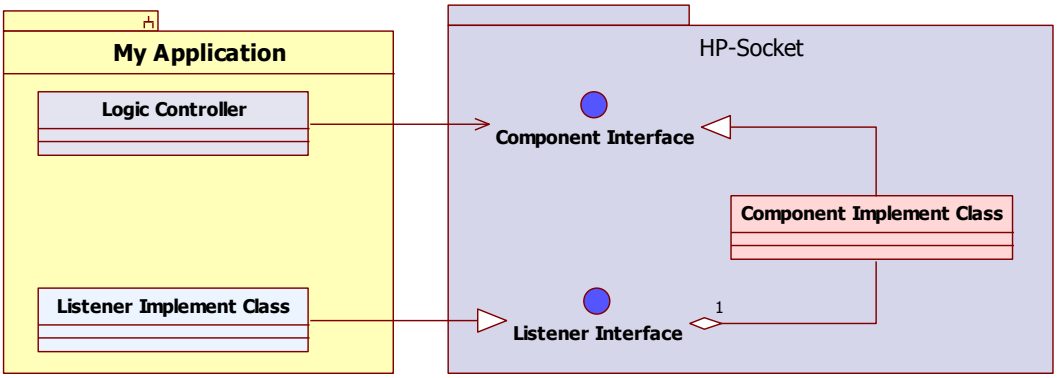


图 1.1-1 HP-Socket API 接口模型

HP-Socket 定义了组件接口（如：ITcpServer / IUdpClient）、组件实现类（如：CTcpServer / CUdpClient）和监听器接口（如：ITcpServerListener / IUdpClientListener），其中：

- ✧ **组件接口：**声明组件操作方法，应用程序创建组件对象后通过该接口来使用组件
- ✧ **组件实现类：**实现组件接口，执行实际通信处理工作，并向监听器报告通信事件
- ✧ **监听器接口：**声明组件的通信事件回调方法

每个组件对象都会关联一个监听器对象（监听器对象的实现类由应用程序定义），当组件对象触发一个通信事件时会调用监听器对象相应的回调方法，应用程序在回调方法中处理应用业务逻辑。图 1.1-2 以 TCP Agent 为例展示了组件与应用程序的交互：

应用程序首先创建监听器对象和 TCP Agent 对象，创建 TCP Agent 对象时传入监听器对象，把 TCP Agent 对象与监听器对象关联起来。TCP Agent 对象创建完毕后，应用程序调用 TCP Agent 接口方法操作 TCP Agent 对象（如：Start / Connect / Send / Stop 等）。当 TCP Agent 对象触发通信事件时，会调用监听器对象的回调方法（如：*OnConnect* / *OnSend* / *OnReceive* / *Onclose* 等）通知应用程序。

注意：监听器对象的异步回调方法是在组件的通信线程中执行的，因此回调方法不应执行耗时较长的业务逻辑代码，同时要注意多线程同步问题，也应尽量避免使用锁。

HP-Socekt 通过设置“接收策略”和“Connection Extra”能协助应用程序巧妙地避免多线程同步和锁导致的复杂性和性能问题。

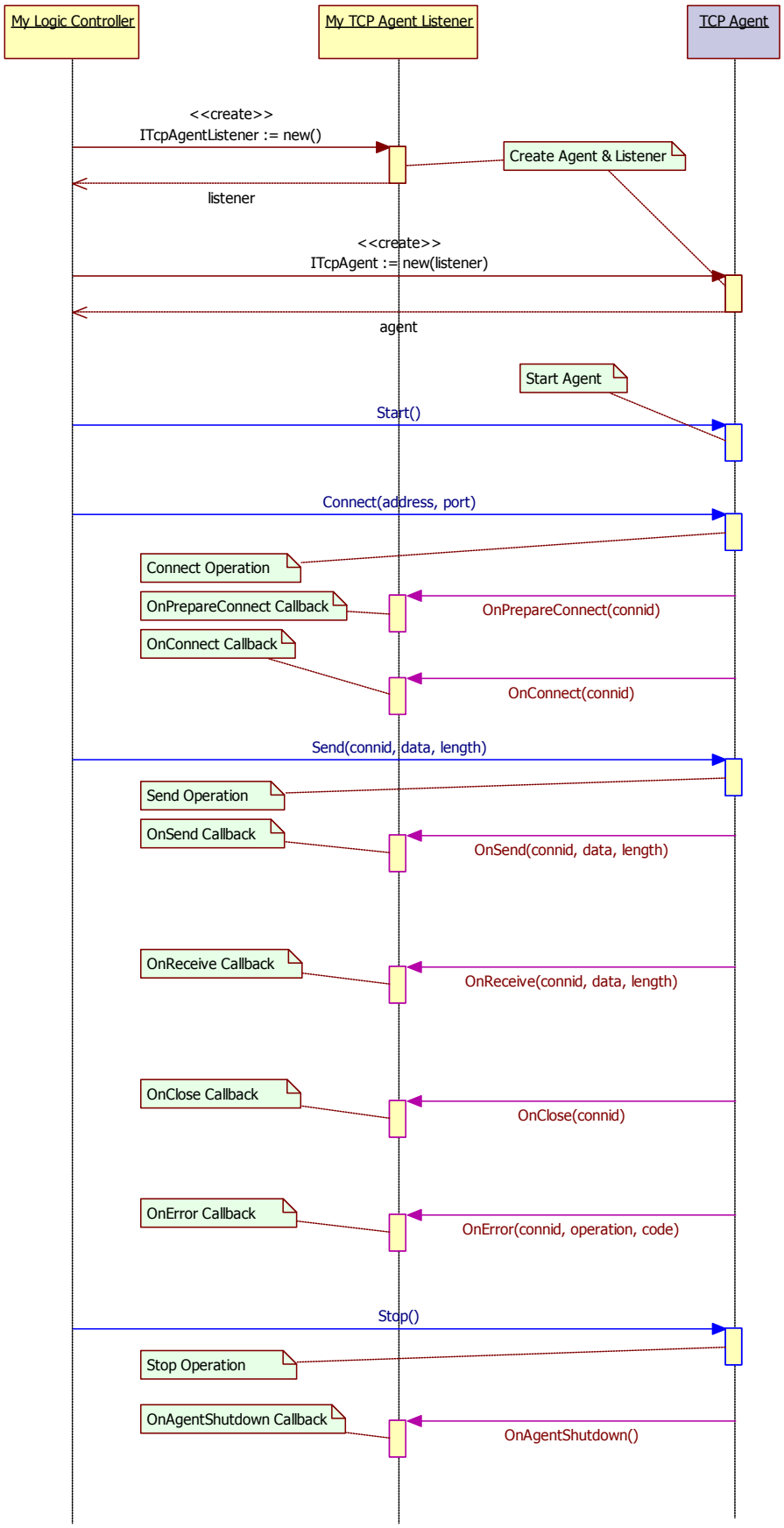


图 1.1-2 TCP Agent 与应用程序交互示例

1.2 组件分类

HP-Socket 包含 9 个组件，可根据通信角色（Client / Server）、通信协议（TCP / UDP）和接收模型（PUSH / PULL）进行归类。表 1.2-1 列出了所有组件的名称、接口、监听器接口、实现类及其分类：

名 称	组件接口 监听器接口	实现类	通信 角色	通信 协议	接收 模型
TCP Server	*ITcpServer* *ITcpServerListener*	CTcpServer	Server	TCP	PUSH
TCP Pull Server	*ITcpPullServer* *ITcpServerListener*	CTcpPullServer	Server	TCP	PULL
UDP Server	*IUdpServer* *IUdpServerListener*	CUdpServer	Server	UDP	PUSH
TCP Agent	*ITcpAgent* *ITcpServerListener*	CTcpAgent	Client	TCP	PUSH
TCP Pull Agent	*ITcpPullAgent* *ITcpAgentListener*	CTcpPullAgent	Client	TCP	PULL
TCP Client	*ITcpClient* *ITcpClientListener*	CTcpClient	Client	TCP	PUSH
TCP Pull Client	*ITcpPullClient* *ITcpClientListener*	CTcpPullClient	Client	TCP	PULL
UDP Client	*IUdpClient* *IUdpClientListener*	CUdpClient	Client	UDP	PUSH
UDP Cast	*IUdpCast* *IUdpCastListener*	CUdpCast	Client	UDP	PUSH

表 1.2-1 组件分类

- ✓ Agent 组件本质上是 Client 组件，一个 Agent 对象能同时管理多个客户端连接
- ✓ 根据实际使用场景，HP-Socket 中只实现了基于 TCP 的 Agent 组件
- ✓ Cast 组件是为组播和广播而设计的 UDP 组件，可认为是一种特殊的 Client 组件
- ✓ 基于 TCP 的组件都分别提供 PUSH 和 PULL 两种接收模型

1.3 组件接口

Server、Agent 和 Client 的组件接口定义如图 1.3-1 — 1.3-3 所示，组件接口定义了组件提供的所有操作方法。其中，PULL 模型接口多重继承于 IPullSocket / IPullCliet 接口，它提供了 *Fetch*(*dwConnID*, *pData*, *iDataLength*) 方法，让应用程序从组件中拉取数据。

图 1.3-1 Server 组件接口

图 1.3-2 Agent 组件接口

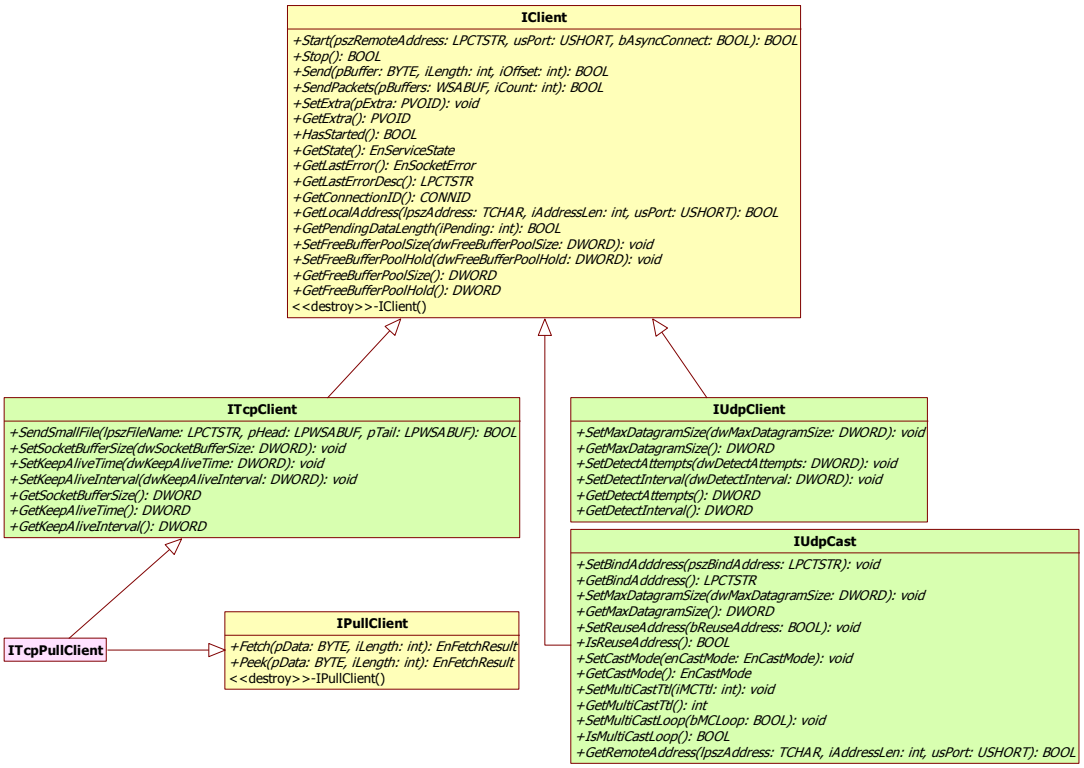


图 1.3-3 Client 组件接口

1.4 监听器接口

Server、Agent 和 Client 的监听器接口定义如图 1.4-1 — 1.4-3 所示：

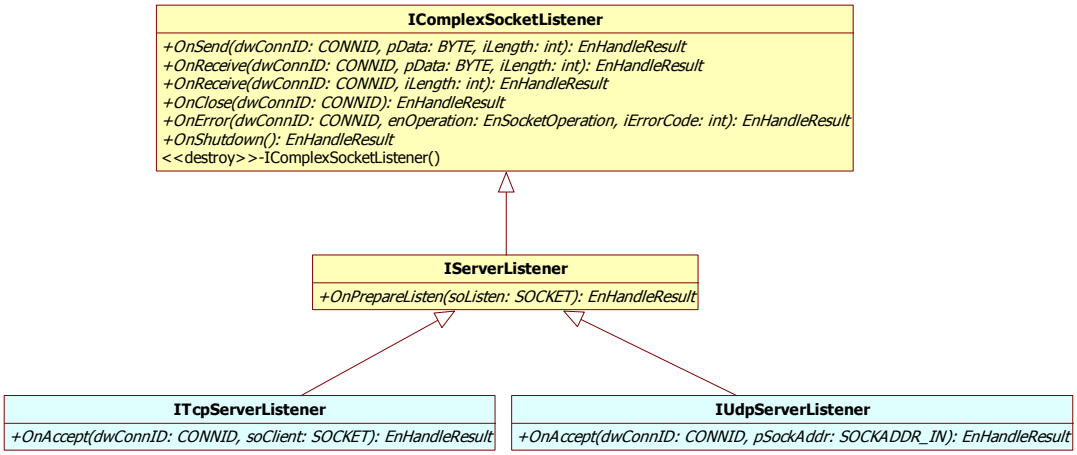


图 1.4-1 Server 监听器接口

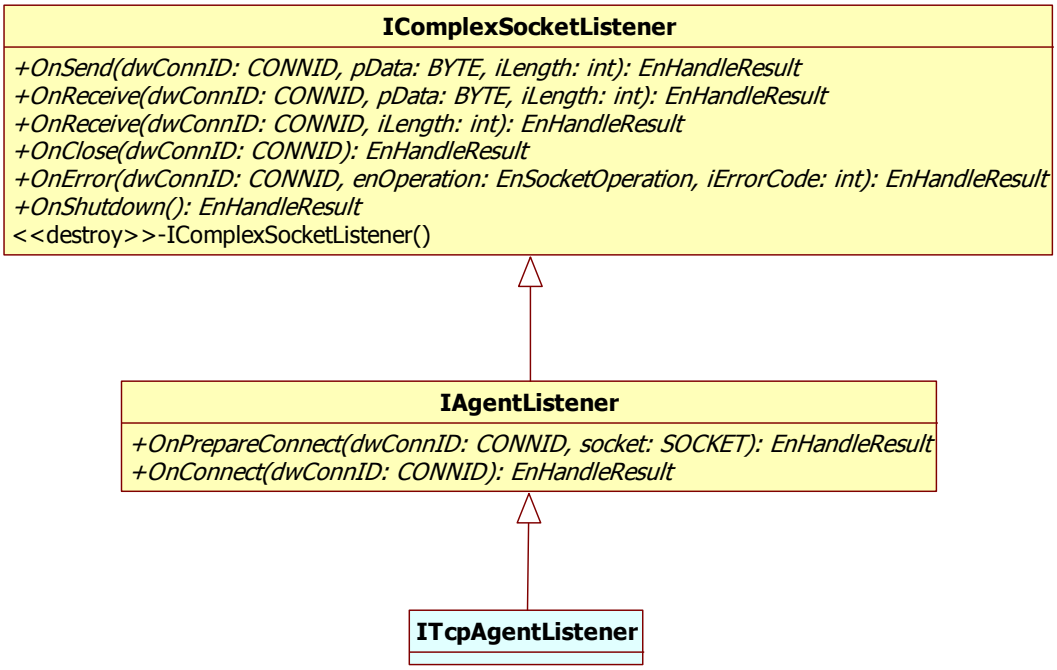


图 1.4-2 Agent 监听器接口

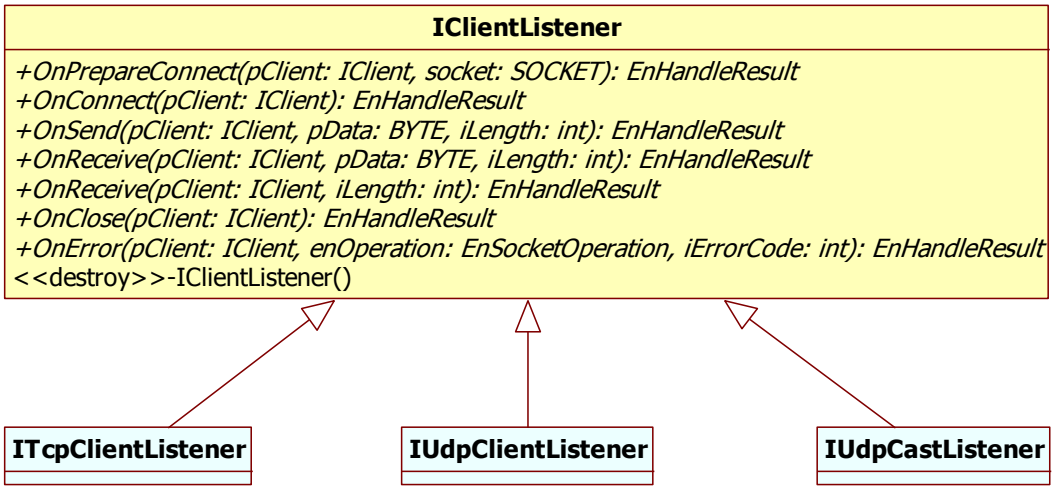


图 1.4-3 Client 监听器接口

HP-Socket 没有为 PUSH 和 PULL 模型组件定义单独的监听器接口，它们使用相同的监听器接口，区别在于：PUSH 模型组件接收到数据时会触发监听器对象的 *OnReceive*(*dwConnID*, *pData*, *iLength*) 事件，而 PULL 模型组件接收到数据时会触发监听器对象的 *OnReceive*(*dwConnID*, *iLength*) 事件。事件的含义如表 1.4-1 所示：

接 口	事 件	说 明
IComplexSocketListener	*OnSend*(*dwConnID*, *pData*, *iLength*)	数据已发送
	OnReceive(*dwConnID*, *pData*, *iLength*)	数据到达（PUSH）

	OnReceive(*dwConnID*, *iLength*)	数据到达（PULL）
	OnClose(*dwConnID*)	连接关闭
	OnError(*dwConnID*, *enOperation*, *iErrorCode*)	通信错误
	OnShutdown()	关闭通信组件
IServerListener	*OnPrepareListen*(*soListen*)	准备监听
ITcpServerListener	*OnAccept*(*dwConnID*, *soClient*)	接受连接请求
IUdpServerListener	*OnAccept*(*dwConnID*, *pSockAddr*)	接受连接请求
IAgentListener	*OnPrepareConnect*(*dwConnID*, *socket*)	准备连接
	OnConnect(*dwConnID*)	完成连接
IClientListener	*OnPrepareConnect*(*pClient*, *socket*)	准备连接
	OnConnect(*pClient*)	完成连接
	OnSend(*pClient*, *pData*, *iLength*)	数据已发送
	OnReceive(*pClient*, *pData*, *iLength*)	数据到达（PUSH）
	OnReceive(*pClient*, *iLength*)	数据到达（PULL）
	OnClose(*pClient*)	连接关闭
	OnError(*pClient*, *enOperation*, *iErrorCode*)	通信错误

表 1.4-1 监听器接口事件

监听器事件回调方法的返回值类型为 *EnHandleResult*：

-

<<enumeration>>
EnHandleResult

+HR_OK
+HR_IGNORE
+HR_ERROR

- ✓ **HR_OK** ：成功处理
- ✓ **HR_IGNORE** ：忽略处理
- ✓ **HR_ERROR** ：处理失败

注意：当 OnReceive / OnPrepareListen / OnAccept / OnPrepareConnect / OnConnect 事件回调方法返回 HR_ERROR 时，组件会立即中断连接。

2 框架详述

2.1 关键概念

2.1.1 接收模型

如表 1.2-1 所示，HP-Socket 的 TCP 组件支持 PUSH 和 PULL 两种接收模型：

- PUSH 模型：组件接收到数据时会触发监听器对象的 *OnReceive(dwConnID, pData, iLength)* 事件，把数据“推”给应用程序。
- PULL 模型：组件接收到数据时会触发监听器对象的 *OnReceive(dwConnID, iTotalLength)* 事件，告诉应用程序当前已经接收到多少数据，应用程序检查数据的长度，如果满足需要则调用组件的 *Fetch(dwConnID, pData, iDataLength)* 方法把需要的数据“拉”出来。

两种模型的比较如图 2.1.1-1 所示：

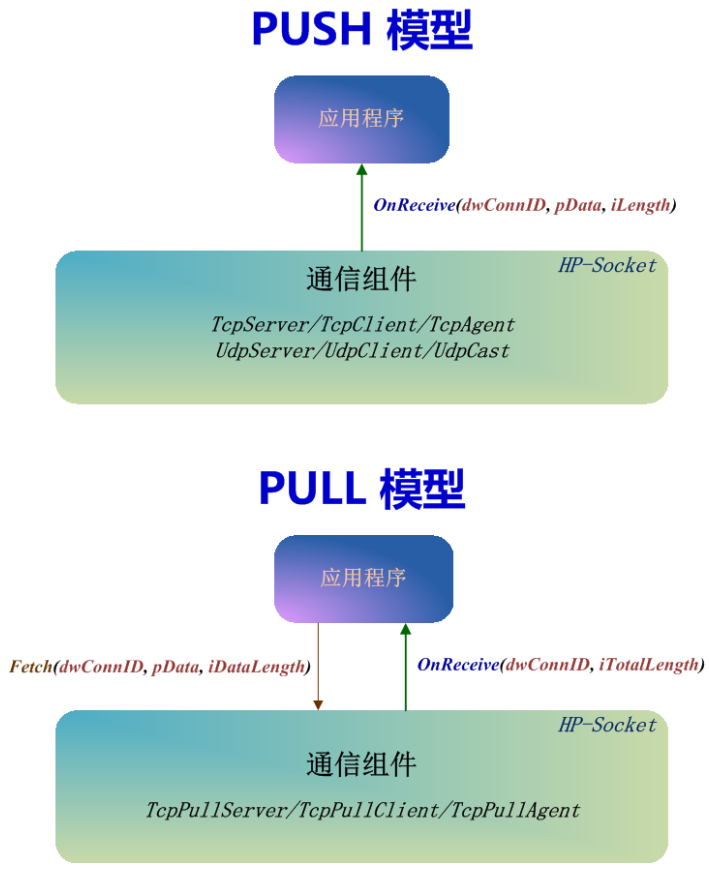
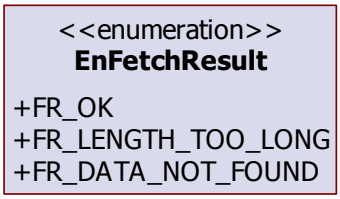


图 2.1.1-1 PUSH / PULL 接收模型

PUSH 模型组件触发监听器对象的 *OnReceive(dwConnID, pData, iLength)* 事件时，应用程序需要立即处理接收到的数据，如：粘包处理、协议解析等。组件不会对应用层的数据处理工作提供任何协助。

PULL 模型组件触发监听器对象的 *OnReceive(dwConnID, iTotalLength)* 事件时，应用程序根据应用层协议检测接收到的数据长度（*iTotalLength*）是否满足处理条件，选择性地进行处理。当 *iTotalLength* 小于当前期望的长度时可以忽略本次事件；当 *iTotalLength* 大于或等于当前期望的长度时，循环调用组件的 *Fetch(dwConnID, pData, iDataLength)* 方法把需要的数据拉取出来，直到剩余的数据长度小于当前期望的长度。

Fetch(dwConnID, pData, iDataLength) 方法返回值类型为 *EnFetchResult*：



- ✓ **FR_OK** ：成功拉取
- ✓ **HR_LENGTH_TOO_LONG** ：拉取的长度超过实际数据长度
- ✓ **HR_DATA_NOT_FOUND** ：没有数据可拉取，可能连接已被关闭

注意：只有当 Fetch(dwConnID, pData, iDataLength)方法返回 FR_OK 时，数据才会被拉取出来。另外，PULL 模型组件还提供 Peek(dwConnID, pData, iDataLength)方法用于窥探接收缓冲区，该方法不会移除缓冲区数据。

PULL 模型适用于完全清楚应用层协议，并且应用层协议可以根据当前数据包得知下一个数据包长度的场景。典型的场景如 Head + Body，Head 长度固定，第一个数据包为 Head，通过 Head 得知 Body 的长度，接收完 Body 之后下一个数据包一定为 Head。

PUSH 和 PULL 模型相关示例请参考《示例 Demo》章节。

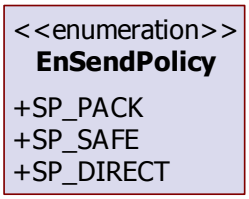
注意：通过 PULL 模型与应用层协议的相互配合，使得应用程序可以免除粘包处理和分拆包工作，从而减少应用程序的负担。

2.1.2 发送策略

对于 IClient 系列组件，当应用程序调用组件的 *Send()*、*SendPackets()*、*SendSmallFile()* 方法发送数据时，组件内部会把数据缓存起来，在适当的时机再发送出去。

对于 IServer 和 IAgent 系列组件，当应用程序调用组件的 *Send()*、*SendPackets()*、*SendSmallFile()* 方法发送数据时，根据不同的发送策略会有不同的处理方式。

（发送策略通过 SetSendPolicy(enSendPolicy) 方法进行设置）



✓ **SP_PACK** ：打包策略（默认）

尽量把多个发送操作的数据组合在一起发送，增加传输效率

✓ **SP_SAFE** ：安全策略

尽量把多个发送操作的数据组合在一起发送，并尽量避免缓冲区溢出

✓ **SP_DIRECT** ：直接策略

对每一个发送操作都直接投递，适用于负载不高但要求实时性较高的场合

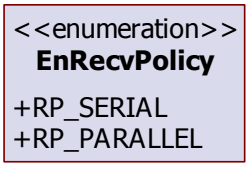
对于 SP_PACK 和 SP_SAFE 策略，组件内部会缓存待发送的数据，应用程序可以调用组件的 *GetPendingDataLength*(*dwConnID*, *iPending*) 方法获取指定连接的未发出数据量，实现流量控制；对于 SP_DIRECT 策略，组件不会缓存待发送数据，如果要实现流量控制则需要比较组件通过 *Send*()、*SendPackets*()、*SendSmallFile*() 方法提交的数据量与 *OnSend*(*dwConnID*, *pData*, *iLength*) 事件报告的实际发出的数据量。

2.1.3 接收策略

对于 IClient 系列组件，每个组件对象都在单独的通信线程执行所有通信工作，*OnReceive* 与 *Onclose* / *OnError* 事件不可能同时触发，因此不必担心在处理 *OnReceive* 事件时某些共享数据被 *Onclose* / *OnError* 事件处理代码意外修改或释放的情形。

对于 IServer 和 IAgent 系列组件，由于有多个通信线程同时工作，对于同一连接，在触发 *OnReceive* 事件时可能同时触发了 *Onclose* / *OnError* 事件，因此，共享数据存在被意外修改或释放的可能。为了应对这个问题，HP-Socket 提供了以下接收策略：

（接收策略通过 SetRecvPolicy(enRecvPolicy) 方法进行设置）



✓ **RP_SERIAL** ：串行策略（默认）

对于单个连接，顺序触发 OnReceive 和 OnClose / OnError 事件。降低应用程序处理的复杂度，增强安全性；但同时损失一些并发性能。

✓ **RP_PARALLEL** ：并行策略

对于单个连接，同时收到 OnReceive 和 OnClose / OnError 事件时，会在不同的通信线程中同时触发这些事件，使并发性能得到提升，但应用程序需要考虑在 OnReceive 的事件处理代码中，某些公共数据可能被 OnClose / OnError 的事件处理代码修改或释放的情形，程序代码逻辑会变得复杂，处

理不好时将会产生代码缺陷。

除非有充足的理由并且完全能避免 RP_PARALLEL 策略所带来的隐患，否则不建议应用程序使用 RP_PARALLEL 策略。

注意：对于 HP-Socket 的所有组件（IServer / IAgent / IClient），当连接触发了 OnClose 或 OnError 事件时，均表示连接已被关闭。并且 OnClose 或 OnError 事件只会触发一次，也就是说：如果触发了 OnError 事件则不会再触发 OnClose 事件或第二个 OnError 事件。

2.1.4 连接方式

HP-Socket 所有组件的通信过程都是异步的，如：调用组件的 *Send()* 方法会立即返回，稍后监听器会接收到 *OnSend()* 事件获知发送了多少数据，或者会接收到 *OnError()* 事件获知发送失败原因。

但 HP-Socket 的 IClient 和 IAgent 组件向服务器发起连接的过程可以是同步或异步的。同步是指组件的连接方法（IClient - *Start()*，IAgent - *Connect()*）等到建立连接成功或失败了再返回（返回 TRUE 或 FALSE）。

异步是指组件的连接方法 *Start()* / *Connect()* 会立即返回，如果 *Start()* / *Connect()* 返回成功（TRUE）则稍后会接收到 *OnConnect()* 或 *OnError()* 事件，收到前者则说明连接成功，收到后者则说明连接失败。注意：如果 *Start()* / *Connect()* 返回失败（FALSE）则稍后不一定能接收到 *OnError()* 事件。因此，对于异步连接也必须检查 *Start()* / *Connect()* 的返回值，当返回失败（FALSE）则立即可以断定连接失败。

- **IClient 建立连接方法：**

BOOL Start(pszRemoteAddress, usPort, bAsyncConnect = FALSE)

参数 *bAsyncConnect* 指示是否采用异步连接方式（默认：*FALSE*），如果 *Start()*方法返回失败可以调用组件的 *GetLastError()* 和 *GetLastErrorDesc()* 方法获取错误代码和错误描述。如果 *Start()*方法返回成功可以调用组件的 *GetConnectionID()* 方法获取当前连接的 Connection ID。

注意：IUdpCast 组件的 Star()方法忽略 bAsyncConnect 参数。

- **IAgent 建立连接方法：**

BOOL Start(pszBindAddress = nullptr, bAsyncConnect = FALSE)
BOOL Connect(pszRemoteAddress, usPort, pdwConnID = nullptr)

Start() 方法启动 IAgent 组件并指定连接方式，参数 *bAsyncConnect* 指示是否采用异步连接方式（默认：*FALSE*），如果 *Start()* 方法返回失败可以调用组件的 *GetLastError()* 和 *GetLastErrorDesc()* 方法获取错误代码和错误描述。注意：*Start()* 方法在整个通信周期中只需调用 1 次。

Connect() 方法与指定服务器建立连接，参数 *pdwConnID* 用来获取本连接的 Connection

ID（默认：*nullptr*，不获取），如果 *Connect()* 方法返回失败可以调用 Windows API 函数 ***::GetLastError()*** 获取 Windows 错误代码。

无论是同步或异步连接，成功完成连接的过程中都会先后触发监听器的两个事件：

- ✓ ***OnPrepareConnect(dwConnID, socket)***
- ✓ ***OnConnect(dwConnID)***

其中 *OnPrepareConnect(dwConnID, socket)* 在发起连接前触发，*socket* 是本地 SOCKET 句柄，可以在该事件中通过 *setsockopt()* / *WSAIoctl()* 等方法设置 SOCKET 选项。*OnConnect(dwConnID)* 则在连接建立成功后触发。

2.1.5 连接绑定

对于 IClient 系列组件，一个组件对象对应一个 Connection ID 和一个通信连接，因此很容易把通信连接与应用层数据关联起来。应用程序与组件交互时，直接通知组件处理数据即可（如：*Send(pData, iLength)*）。如图 2.1.5-1 所示：

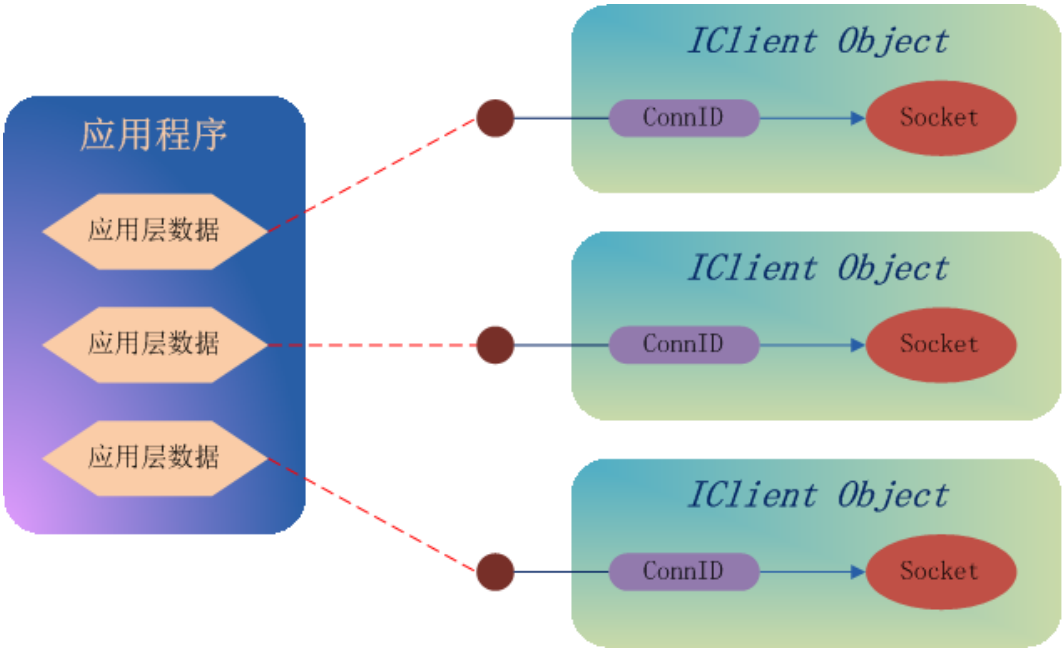


图 2.1.5-1 IClient 组件连接示意图

注意：对于 IClient 组件，可以通过 SetExtra() / GetExtra() 方法绑定、获取附加数据。

对于 IServer 和 IAgent 系列组件，一个组件对象管理多个通信连接，HP-Socket 把通信连接抽象为 Connection ID，应用程序与组件交互时需要指定 Connection ID 标识来告知组件处理哪个连接的数据（如：*Send(dwConnID, pData, iLength)* ）。如图 2.1.5-2 示：

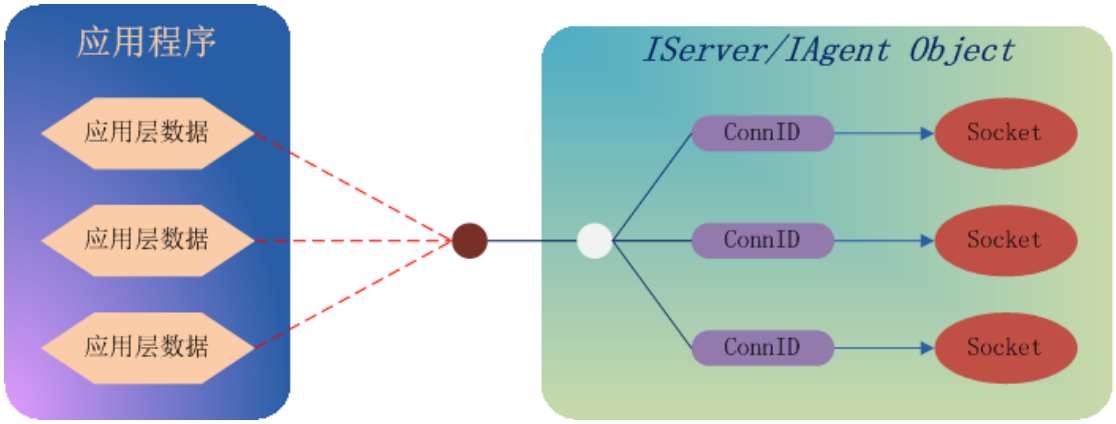


图 2.1.5-2 IServer/IAgent 组件连接示意图

应用程序为了建立 Connection ID 与应用层数据的对应关系通常需要维护一张映射表（如：*map*<*CONNID*, *TMyAppData**>），从而不但增加了应用程序的负担；另外，由于运行在多线程环境下，对映射表的读写操作需要进行同步处理，从而降低了应用程序的并发性能。

HP-Socket 为 IServer 和 IAgent 系列组件提供以下方法组绑定 Connection ID 和应用层数据，尽量避免让应用程序维护映射表。

- ***BOOL SetConnectionExtra(CONNID dwConnID, PVOID pExtra)***
- ***BOOL GetConnectionExtra(CONNID dwConnID, PVOID* ppExtra)***

通常的应用情景如下：

1) 在 *OnAccept()* / *OnConnect()* 事件中调用 *SetConnectionExtra*(*dwConnID*, *pExtra*) 把 Connection ID 和应用层数据进行绑定。
2) 在 *OnReceive()* / *OnSend()* 事件中调用 *GetConnectionExtra*(*dwConnID*, *ppExtra*) 取出与 Connection ID 绑定的应用层数据，执行相应业务逻辑处理。
3) 在 *OnClose()* / *OnError()* 事件中取消 Connection ID 和应用层数据的绑定，清除应用层数据并释放资源。

注意：如果组件的接收策略为RP_PARALLEL，由于OnReceive 事件与Onclose / OnError 事件可能同时触发，请慎用这种绑定方法（参考：《接收策略》）。

2.2 Server 组件

2.2.1 接口描述

图 2.2.1-1 Server 组件接口

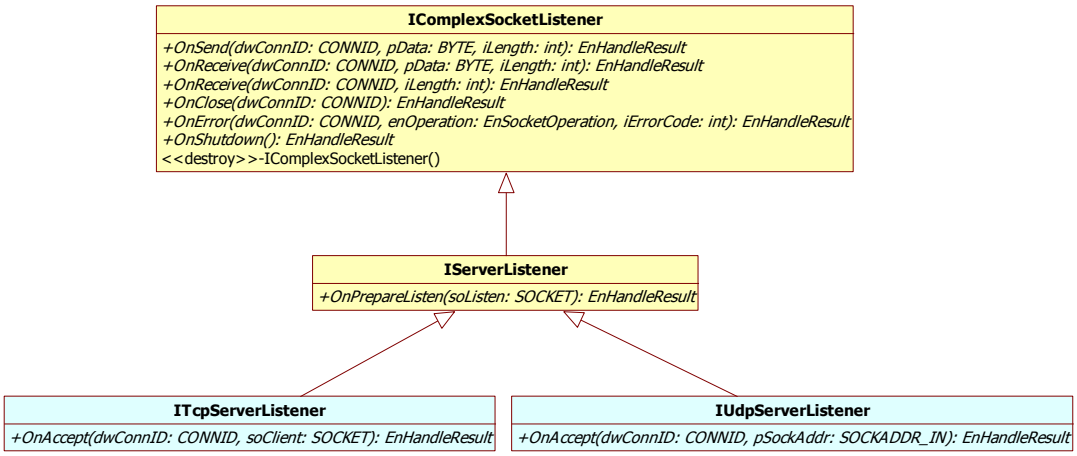


图 2.2.1-2 Server 监听器接口

Server 组件接口的继承层次结构如图 2.2.1-1 所示，其中，*ITcpServer* 和 *IUdpServer* 继承于 *IServer*，*ITcpPullServer* 则继承于 *ITcpServer*。主要接口方法如表 2.2.1-1 所示，其它接口方法请参考 *Src/SocketInterface.h* 文件的相关注释：

组件接口	操作方法	描 述
IServer	*Start()*	启动组件
	Stop()	关闭组件
	Send()	发送数据
	SendPackets()	发送多组数据
	Disconnect()	断开连接
	DisconnectLongConnections()	断开长连接
	DisconnectSilenceConnections()	断开静默连接
	HasStarted()	检查通信组件是否已启动
	GetState()	获取通信组件当前状态
	GetConnectionCount()	获取连接数
	GetConnectPeriod()	获取连接时长
	GetSilencePeriod()	获取静默时长
	GetAllConnectionIDs()	获取所有连接的 CONNID
	GetRemoteAddress()	获取某个连接的远程地址
	GetListenAddress()	获取监听 Socket 的地址
	GetLastError()	获取最近一次失败操作的错误代码
	GetLastErrorDesc()	获取最近一次失败操作的错误描述
	SetWorkerThreadCount()	设置工作线程数
ITcpServer	*SendSmallFile()*	发送小文件
	SetSocketListenQueue()	设置监听 Socket 的等候队列大小
	SetAcceptSocketCount()	设置 Accept 预投递数量
	SetSocketBufferSize()	设置通信数据缓冲区大小
	SetKeepAliveTime()	设置心跳检测包发送间隔

	SetKeepAliveInterval()	设置心跳检测重试包发送间隔
ITcpPullServer	*Fetch()*	拉取数据
	Peek()	窥探数据
IUdpServer	*SetMaxDatagramSize()*	设置数据报文最大长度
	SetDetectAttempts()	设置检测重试次数
	SetDetectInterval()	设置检测包发送间隔

表 2.2.1-1 Server 组件接口

Server 监听器接口的继承层次结构如图 2.2.1-2 所示，其中，*ITcpServerListener* 和 *IUdpServerListener* 继承于 *IServerListener*，接口回调事件如表 2.2.1-2 所示：

监听器接口	回调事件	描 述
IServerListener	*OnPrepareListen()*	准备监听 绑定监听地址前触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH） 接收到数据时触发
	OnReceive() [PULL]	数据到达（PULL） 接收到数据时触发
	OnClose()	连接关闭 连接正常关闭时触发
	OnError()	通信错误 连接非正常关闭时触发
	OnShutdown()	关闭通信组件 通信组件停止后触发
ITcpServerListener	*OnAccept()*	接受连接请求 客户端连接请求到达时触发
IUdpServerListener	*OnAccept()*	接受连接请求 客户端连接请求到达时触发

表 2.2.1-2 Server 监听器接口

2.2.2 工作流程

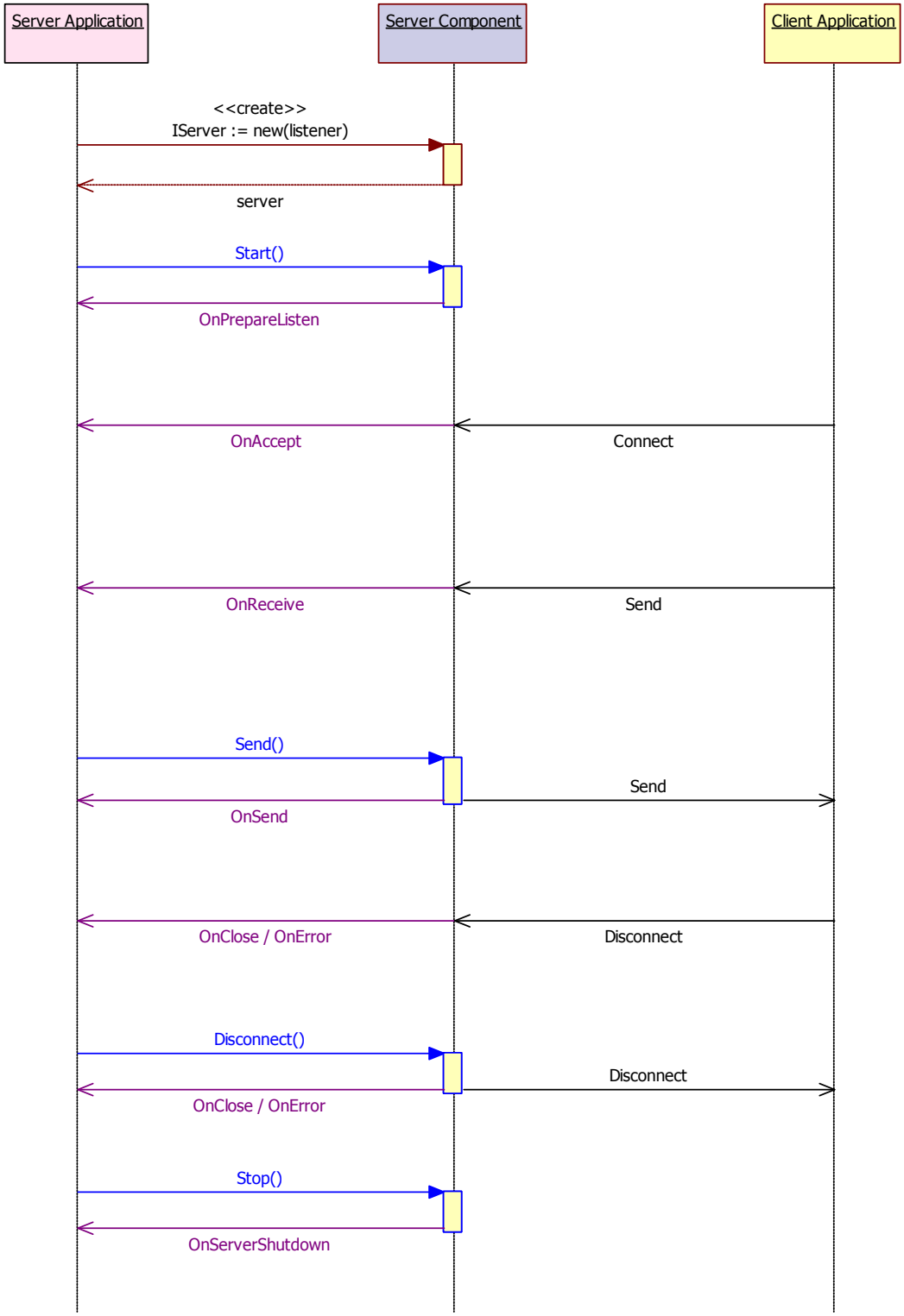


图 2.2.2-1 Server 工作流程

图 2.2.2-1 展示了服务端、客户端应用程序与 Server 组件的交互流程：

- 服务端应用程序调用 *Start()* 方法启动 Server 组件，如果调用成功则返回 TRUE 并收到 *OnPrepareListen* 事件。
- 客户端应用程序向服务端应用程序发起连接请求时，服务端应用程序将收到 *OnAccept* 事件。
- 客户端应用程序向服务端应用程序发送数据时，服务端应用程序将收到 *OnReceive* 事件。
- 服务端应用程序调用 *Send()* 方法向客户端应用程序发出数据后，服务端应用程序将收到 *OnSend* 事件。
- 断开连接时，服务端应用程序将收到 *OnClose* 或 *OnError* 事件。
- 服务端应用程序调用 *Stop()* 方法关闭 Server 组件，如果调用成功则返回 TRUE 并收到 *OnShutdown* 事件。

2.3 Agent 组件

2.3.1 接口描述

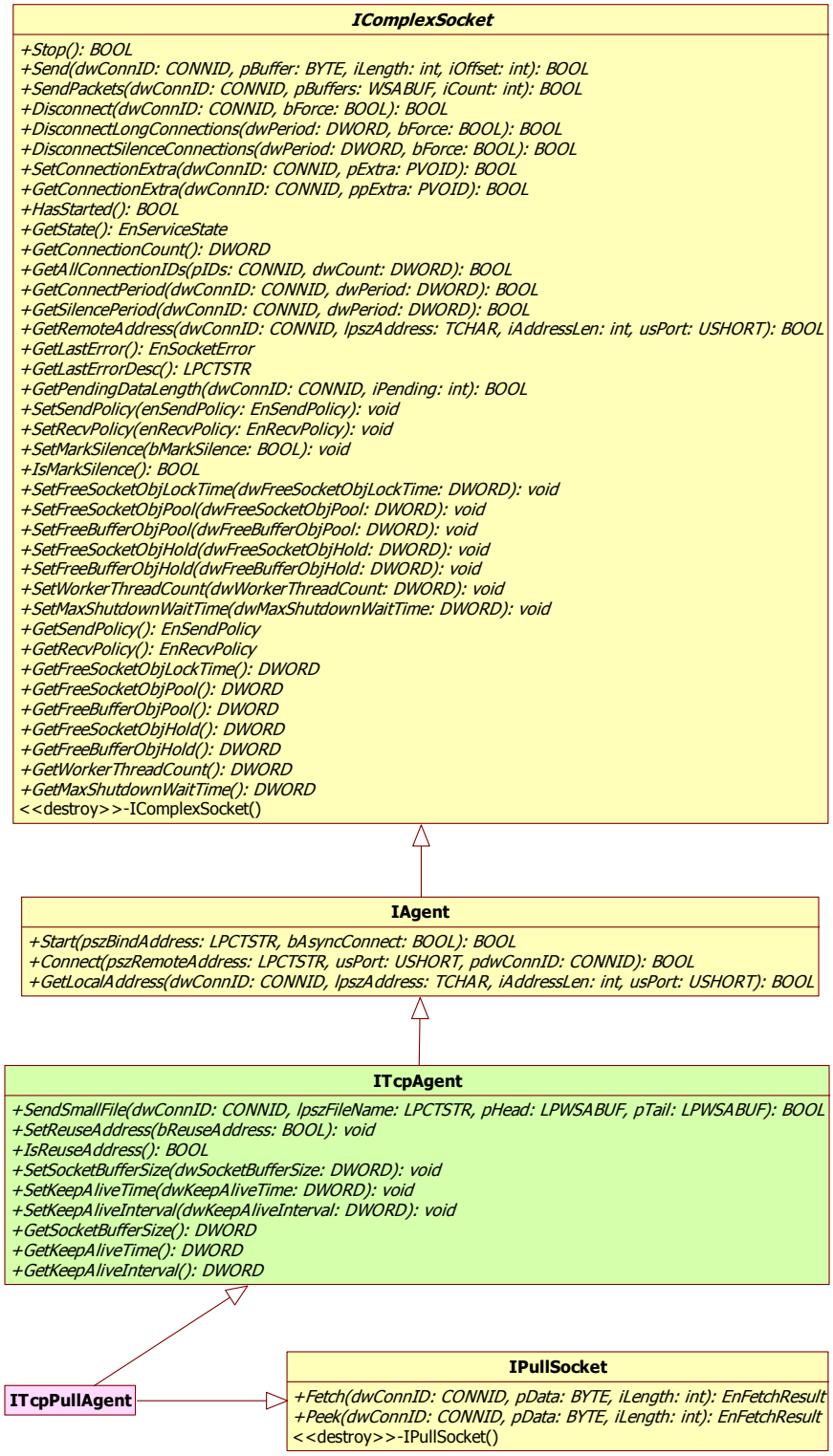


图 2.3.1-1 Agent 组件接口

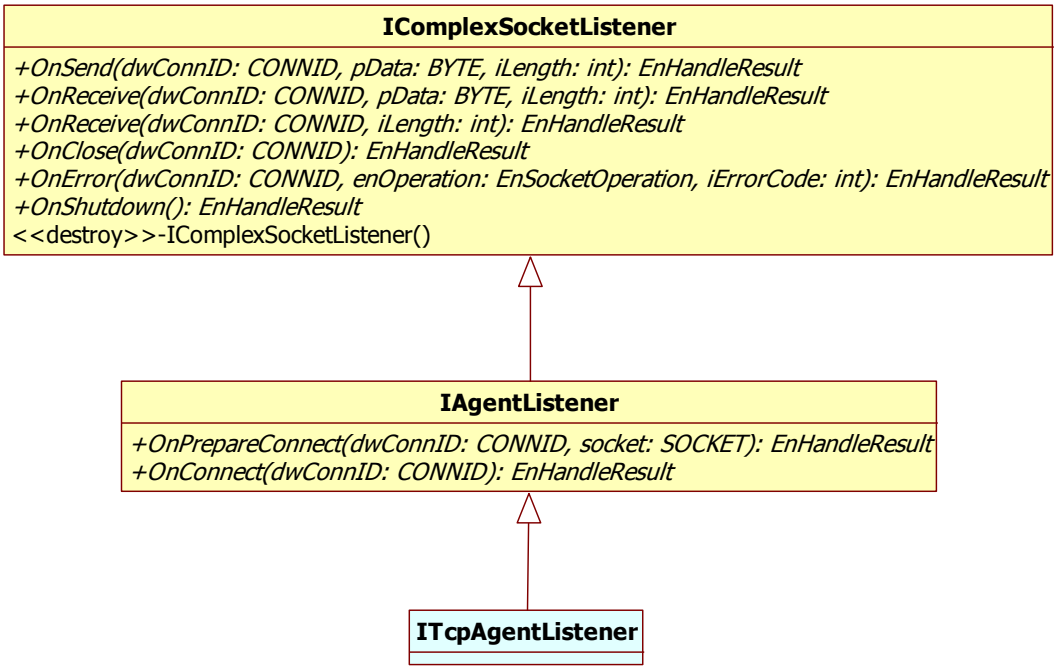


图 2.3.1-2 Agent 监听器接口

Agent 组件接口的继承层次结构如图 2.3.1-1 所示，其中，*ITcpAgent* 继承于 *IAgent*，*ITcpPullAgent* 则继承于 *ITcpAgent*。主要接口方法如表 2.3.1-1 所示，其它接口方法请参考 *Src/SocketInterface.h* 文件的相关注释：

组件接口	操作方法	描 述
IAgent	*Start()*	启动组件
	Stop()	关闭组件
	Connect()	连接服务器
	Send()	发送数据
	SendPackets()	发送多组数据
	Disconnect()	断开连接
	DisconnectLongConnections()	断开长连接
	DisconnectSilenceConnections()	断开静默连接
	HasStarted()	检查通信组件是否已启动
	GetState()	获取通信组件当前状态
	GetConnectionCount()	获取连接数
	GetConnectPeriod()	获取连接时长
	GetSilencePeriod()	获取静默时长
	GetAllConnectionIDs()	获取所有连接的 CONNID
	GetRemoteAddress()	获取某个连接的远程地址
	GetLocaleAddress()	获取某个连接的本地地址
	GetLastError()	获取最近一次失败操作的错误代码
	GetLastErrorDesc()	获取最近一次失败操作的错误描述

	SetWorkerThreadCount()	设置工作线程数
ITcpAgent	*SendSmallFile()*	发送小文件
	SetReuseAddress()	设置是否启用地址重用机制
	SetSocketBufferSize()	设置通信数据缓冲区大小
	SetKeepAliveTime()	设置心跳检测包发送间隔
	SetKeepAliveInterval()	设置心跳检测重试包发送间隔
ITcpPullAgent	*Fetch()*	拉取数据
	Peek()	窥探数据

表 2.3.1-1 Agent 组件接口

Agent 监听器接口的继承层次结构如图 2.3.1-2 所示，其中，*ITcpAgentListener* 继承于 *IAgentListener*，接口回调事件如表 2.3.1-2 所示：

监听器接口	回调事件	描 述
IAgentListener	*OnPrepareConnect()*	准备建立连接 建立连接前触发
	OnConnect()	成功建立连接 成功建立连接后触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH） 接收到数据时触发
	OnReceive() [PULL]	数据到达（PULL） 接收到数据时触发
	OnClose()	连接关闭 连接正常关闭时触发
	OnError()	通信错误 连接非正常关闭时触发
	OnShutdown()	关闭通信组件 通信组件停止后触发
ITcpAgentListener		

表 2.3.1-2 Agent 监听器接口

2.3.2 工作流程

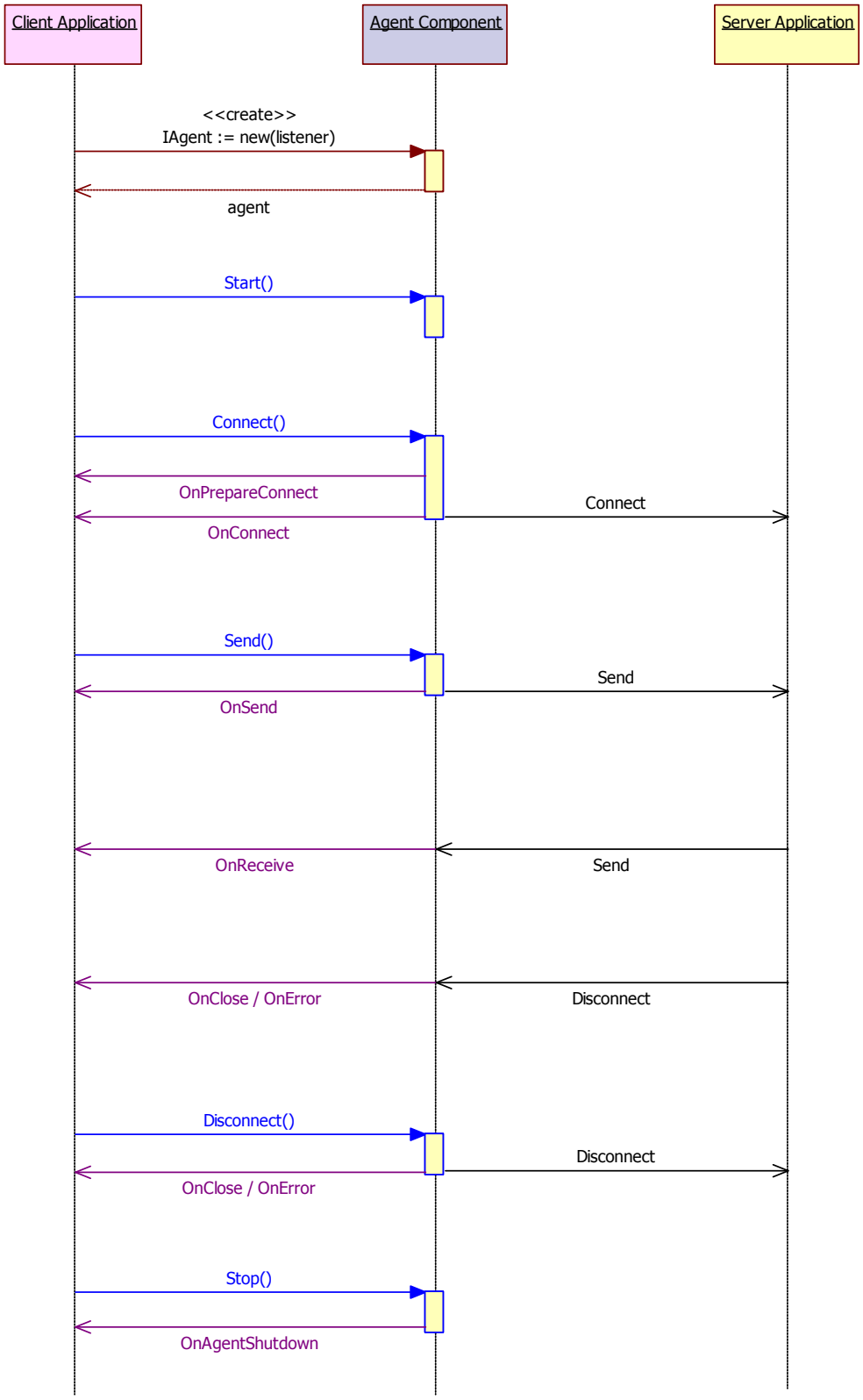


图 2.3.2-1 Agent 工作流程

图 2.3.2-1 展示了服务端、客户端应用程序与 Agent 组件的交互流程：

- 客户端应用程序调用 *Start()* 方法启动 Agent 组件，如果调用成功则返回 TRUE。
- 客户端应用程序调用 *Connect()* 方法向服务端应用程序发起连接请求，如果连接成功则返回 TRUE 并且会先后接收到 *OnPrepareConnect* 和 *OnConnect* 事件。
- 客户端应用程序调用 *Send()* 方法向服务端应用程序发出数据后，客户端应用程序将收到 *OnSend* 事件。
- 服务端应用程序向客户端应用程序发送数据时，客户端应用程序将收到 *OnReceive* 事件。
- 断开连接时，客户端应用程序将收到 *OnClose* 或 *OnError* 事件。
- 客户端应用程序调用 *Stop()* 方法关闭 Agent 组件，如果调用成功则返回 TRUE 并收到 *OnShutdown* 事件。

2.4 Client 组件

2.4.1 接口描述

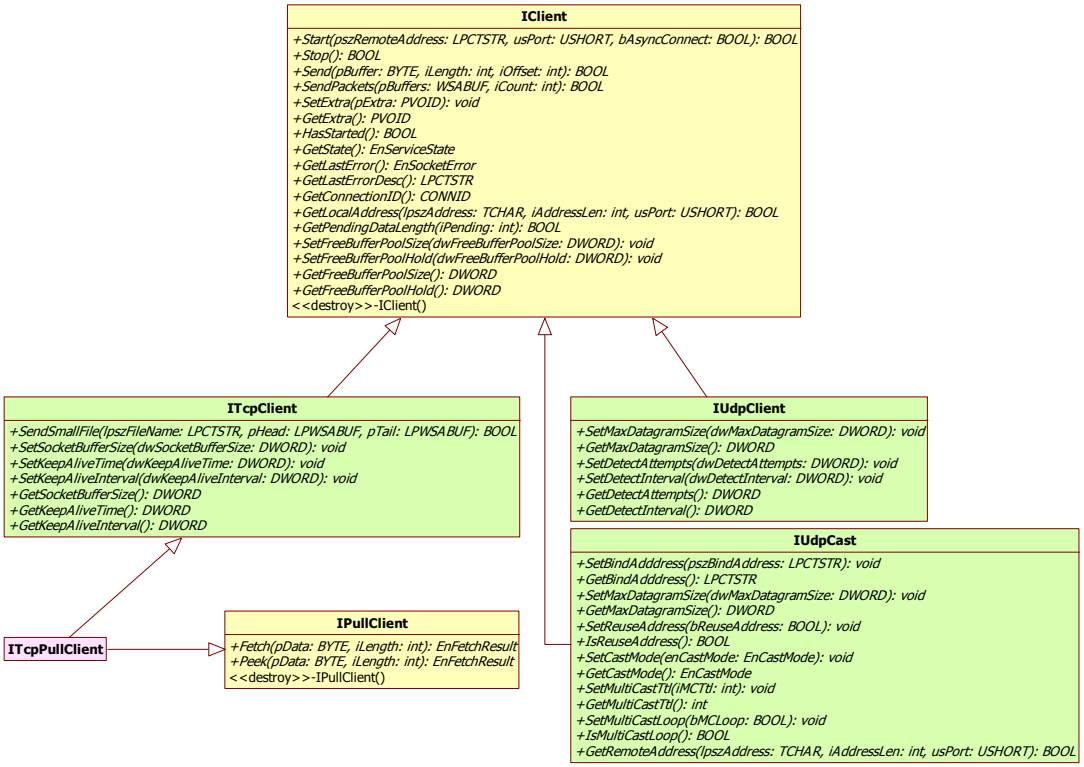


图 2.4.1-1 Client 组件接口

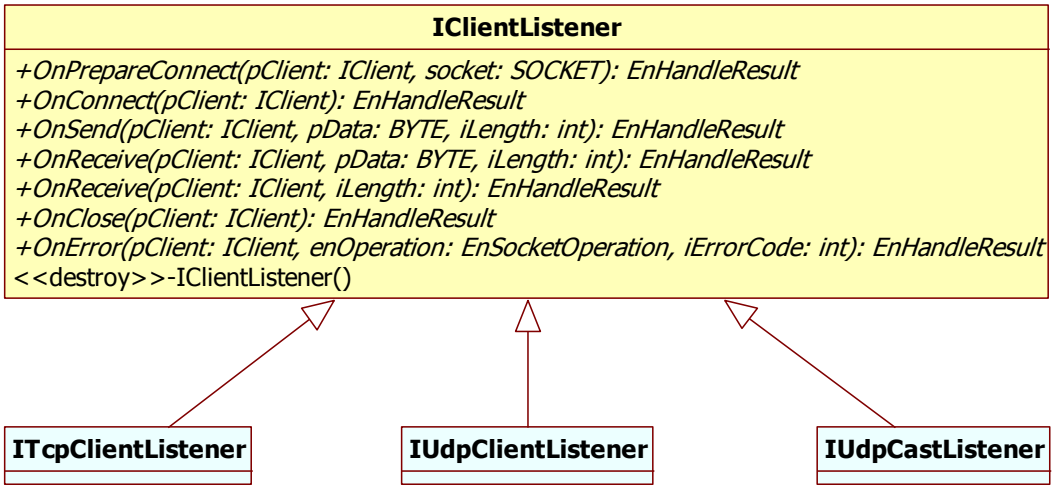


图 2.4.1-2 Client 监听器接口

Client 组件接口的继承层次结构如图 2.4.1-1 所示，其中，*ITcpClient*、*IUdpClient* 和 *IUdpCast* 继承于 *IClient*，*ITcpPullClient* 则继承于 *ITcpClient*。主要接口方法如表 2.4.1-1 所示，其它接口方法请参考 *Src/SocketInterface.h* 文件的相关注释：

组件接口	操作方法	描 述
IClient	*Start()*	启动组件
	Stop()	关闭组件
	Connect()	连接服务器
	Send()	发送数据
	SendPackets()	发送多组数据
	HasStarted()	检查通信组件是否已启动
	GetState()	获取通信组件当前状态
	GetConnectionID()	获取该组件对象的 CONNID
	GetLocaleAddress()	获取连接的本地地址
	GetLastError()	获取最近一次失败操作的错误代码
	GetLastErrorDesc()	获取最近一次失败操作的错误描述
ITcpClient	*SendSmallFile()*	发送小文件
	SetSocketBufferSize()	设置通信数据缓冲区大小
	SetKeepAliveTime()	设置心跳检测包发送间隔
	SetKeepAliveInterval()	设置心跳检测重试包发送间隔
ITcpPullClient	*Fetch()*	拉取数据
	Peek()	窥探数据
IUdpClient	*SetMaxDatagramSize()*	设置数据报文最大长度
	SetDetectAttempts()	设置检测重试次数
	SetDetectInterval()	设置检测包发送间隔
IUdpCast	*SetBindAddress()*	设置绑定地址
	SetMaxDatagramSize()	设置数据报文最大长度

	SetReuseAddress()	设置是否启用地址重用机制
	SetCastMode()	设置传播模式（组播或广播）
	SetMultiCastTtl()	设置组播播报文的 TTL
	SetMultiCastLoop()	设置是否启用组播环路
	GetRemoteAddress()	获取当前数据包的远程地址

表 2.4.1-1 Client 组件接口

Client 监听器接口的继承层次结构如图 2.4.1-2 所示，其中，*ITcpClientListener* 和 *IUdpClientListener* 继承于 *IClientListener*，接口回调事件如表 2.4.1-2 所示：

监听器接口	回调事件	描 述
IClientListener	*OnPrepareConnect()*	准备建立连接 建立连接前触发
	OnConnect()	成功建立连接 成功建立连接后触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH） 接收到数据时触发
	OnReceive() [PULL]	数据到达（PULL） 接收到数据时触发
	OnClose()	连接关闭 连接正常关闭时触发
	OnError()	通信错误 连接非正常关闭时触发
ITcpClientListener		
IUdpClientListener		
IUdpCastListener		

表 2.4.1-2 Client 监听器接口

2.4.2 工作流程

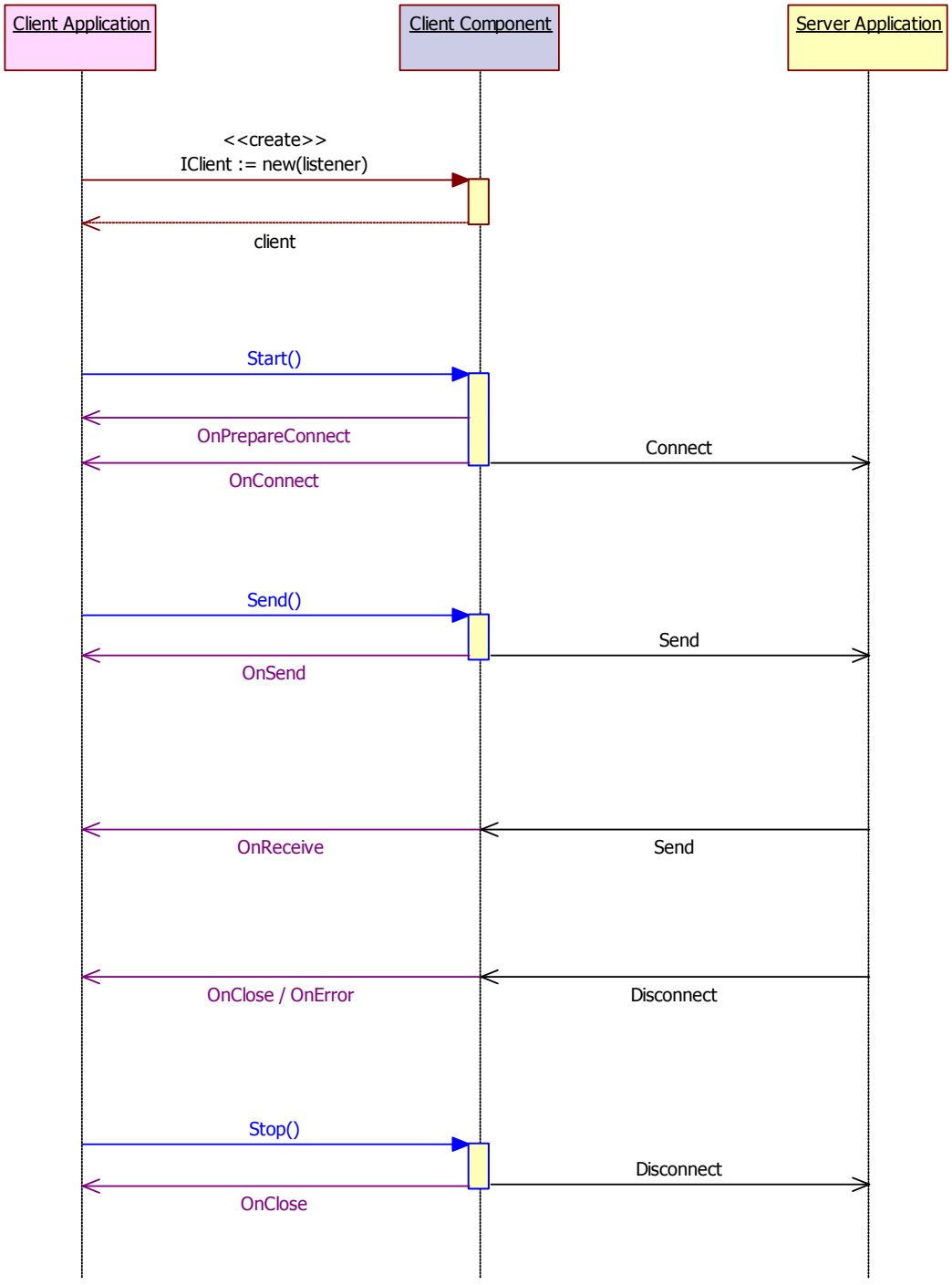


图 2.4.2-1 Client 工作流程

图 2.4.2-1 展示了服务端、客户端应用程序与 Client 组件的交互流程：

- 客户端应用程序调用 *Start()* 方法向服务端应用程序发起连接请求，如果连接成功

则返回 TRUE 并且会先后接收到 *OnPrepareConnect* 和 *OnConnect* 事件。

- 客户端应用程序调用 *Send()* 方法向服务端应用程序发出数据后，客户端应用程序将收到 *OnSend* 事件。
- 服务端应用程序向客户端应用程序发送数据时，客户端应用程序将收到 *OnReceive* 事件。
- 断开连接时，客户端应用程序将收到 *OnClose* 或 *OnError* 事件。
- 客户端应用程序调用 *Stop()* 方法关闭 Client 组件，如果调用成功则返回 TRUE 并收到 *OnClose* 事件。

3 使用方式

HP-Socket 支持 MBCS 和 Unicode 字符集，支持 32 位和 64 位应用程序。可以通过源代码、DLL 或 LIB 方式使用 HP-Socket。HP-Socket 发布包中已经提供了 HPSocket DLL 和 HPSocket4C DLL，如果希望以 LIB 方式使用 HP-Socket 则需要自己编译。

3.1 源代码

HP-Socket 依赖于 *Common/Src* 目录下的一些公共代码。所以，通过源代码方式使用 HP-Socket 时需要把 HP-Socket 的 *Src* 目录和 *Common/Src* 目录下的相应代码文件加入到工程项目（参考：TestEcho / TestEcho-UDP 示例 Demo）。

注意：通过源代码方式或 LIB 方式使用 HP-Socket，当需要更新或升级 HP-Socket 时必须重新编译应用程序。

3.2 HPSocket DLL

HPSocket DLL 导出 C++ 编程接口，是 C++ 程序使用 HP-Socket 的首选方式。HPSocket DLL 通过 HP-Socket 发布包中的 *DLL/HPSocketDLL* 工程项目编译生成，输出以下 DLL：

- ✓ ***Bin\x86\HPSocket.dll*** ：（32 位/MBCS/Release）
- ✓ ***Bin\x86\HPSocket_D.dll*** ：（32 位/MBCS/Debug）
- ✓ ***Bin\x86\HPSocket_U.dll*** ：（32 位/Unicode/Release）
- ✓ ***Bin\x86\HPSocket_UD.dll*** ：（32 位/Unicode/Debug）
- ✓ ***Bin\x64\HPSocket.dll*** ：（64 位/MBCS/Release）
- ✓ ***Bin\x64\HPSocket_D.dll*** ：（64 位/MBCS/Debug）
- ✓ ***Bin\x64\HPSocket_U.dll*** ：（64 位/Unicode/Release）
- ✓ ***Bin\x64\HPSocket_UD.dll*** ：（64 位/Unicode/Debug）

使用 HPSocket DLL 时需要把 *Src/HPSocket.h*、*Src/SocketInterface.h* 以及 DLL 对应的*.*lib* 文件加入到工程项目。*Src/HPSocket.h* 除了导出组件的创建、销毁方法和组件接口外，还定义了各组件的智能指针（如：*CTcpServerPtr* / *CTcpClientPtr*），通过这些智能指针可以更方便地使用 HP-Socket 组件。（参考：TestEcho-Pull / TestEcho-PFM 示例 Demo）。

通过 HPSocket DLL 方式使用 HP-Socket，当需要更新或升级 HP-Socket 时，如果 DLL 接口发生变化则必须重新编译应用程序；如果 DLL 接口没有改变则直接替换 DLL 即可，不需要重新编译应用程序。

3.3 HPSocket4C DLL

HPSocket4C DLL 导出 C 编程接口，提供给 C 语言或其它编程语言使用 HP-Socket。HPSocket4C DLL 通过 HP-Socket 发布包中的 *DLL/HPSocketDLL4C* 工程项目编译生成，输出以下 DLL：

- ✓ ***Bin\x86\HPSocket4C.dll*** ：（32 位/MBCS/Release）
- ✓ ***Bin\x86\HPSocket4C_D.dll*** ：（32 位/MBCS/Debug）
- ✓ ***Bin\x86\HPSocket4C_U.dll*** ：（32 位/Unicode/Release）
- ✓ ***Bin\x86\HPSocket4C_UD.dll*** ：（32 位/Unicode/Debug）
- ✓ ***Bin\x64\HPSocket4C.dll*** ：（64 位/MBCS/Release）
- ✓ ***Bin\x64\HPSocket4C_D.dll*** ：（64 位/MBCS/Debug）
- ✓ ***Bin\x64\HPSocket4C_U.dll*** ：（64 位/Unicode/Release）
- ✓ ***Bin\x64\HPSocket4C_UD.dll*** ：（64 位/Unicode/Debug）

使用 HPSocket4C DLL 时需要把 *Src/HPSocket4C.h* 以及 DLL 对应的*.*lib* 文件加入到工程项目。（参考：TestEcho-4C 示例 Demo）。

通过 HPSocket4C DLL 方式使用 HP-Socket，当需要更新或升级 HP-Socket 时，如果 DLL 接口发生变化则必须重新编译应用程序；如果 DLL 接口没有改变则直接替换 DLL 即可，不需要重新编译应用程序。

注意：HPSocket DLL 和 HPSocket4C DLL 均导出以下几个 Socket 辅助方法：

- ✧ ***DWORD SYS_GetLastError()*** ：封装 API 函数 *GetLastError()*
- ✧ ***int SYS_WSAGetLastError()*** ：封装 API 函数 *WSAGetLastError()*
- ✧ ***int SYS_SetSocketOption()*** ：封装 API 函数 *setsockopt()*
- ✧ ***int SYS_GetSocketOption()*** ：封装 API 函数 *getsockopt()*
- ✧ ***int SYS_IoctlSocket()*** ：封装 API 函数 *ioctlsocket()*
- ✧ ***int SYS_WSAIoctl()*** ：封装 API 函数 *WSAIoctl()*

3.4 其它编程语言使用 HPSocket

HP-Socket 发布包中提供了 C#、E 和 Java 编程语言的 SDK，应用程序可以通过 SDK 方式使用 HP-Socket；对于其它没有提供 SDK 的编程语言（如：Delphi、Python），可以通过导入 HPSocket4C DLL 的方式使用 HP-Socket。

C#、E、Delphi 和 Java 使用 HP-Socket 的例子请参考 HP-Socket 发布包中 *Demo/Other Languages Demo* 目录下的示例 Demo。

4 附 录

4.1 示例 Demo

项目名称	使用组件	引用方式	描 述
HttpProxy	*TCP Server* *TCP Agent*	DLL	HTTP 代理服务器
TestEcho	*TCP Server* *TCP Client*	源代码	Echo 服务端和客户端
TestEcho-4C	*TCP PULL Server* *TCP PULL Client*	4C DLL	Echo 服务端和客户端
TestEcho-Agent (Agent-4C)	*TCP PULL Agent*	4C DLL	Echo 客户端
TestEcho-Agent (Agent-PFM)	*TCP Agent*	源代码	Echo 性能测试客户端
TestEcho-Agent (Agent-PULL)	*TCP PULL Agent*	DLL	Echo 客户端
TestEcho-PFM	*TCP Server* *TCP Client*	DLL	Echo 性能测试服务端和客户端
TestEcho-Pull	*TCP PULL Server* *TCP PULL Client*	DLL	Echo 服务端和客户端
TestEcho-UDP	*UDP Server* *UDP Client*	源代码	Echo 服务端和客户端
TestEcho-UDP-PFM	*UDP Server* *UDP Client*	DLL	Echo 性能测试服务端和客户端
TestUDPCast	*UDP Cast*	源代码	组播（广播）网络成员

表 4.1-1 示例 Demo

4.2 FAQ

- ***Q*：Connection ID 的长度是多少字节？数值溢出怎么办？**
  - ***A***：Connection ID 是一个从 1 开始顺序递增的整数值，当溢出时会重新从 1 开始递增。Connection ID 在 32 位程序中是 4 字节，在 64 位程序中是 8 字节，理论上在 32 位程序中存在溢出的可能，但不必过于担心，因为极少有“*创建了 40 亿次连接后第一个连接还没断开*”的场景。

- ***Q*：可以在通信线程中调用 *Start() / Stop()* 吗？**
  - ***A***：不能。由于监听器事件（*OnReceive / OnClose / OnError* 等）通常都在通信线程中被触发，因此不能在监听器事件处理代码中调用 *Start() / Stop()* 控制方法。

- ***Q*：如何断开超长连接？**
  - ***A***：所谓超长连接是指连接时长超过正常时长的连接。Server 和 Agent 组件提供 *DisconnectLongConnections()* 方法断开所有超长连接，也提供 *GetConnectPeriod()* 方法用来获取某个连接的时长。

- ***Q*：如何断开静默连接？**
  - ***A***：所谓静默连接是长时间没有数据交互的指连接。Server 和 Agent 组件提供 *DisconnectSilenceConnections()* 方法断开所有静默连接，也提供 *GetSilencePeriod()* 方法用来获取某个连接的静默时间。注意：当组件开启了静默标记时上述两个方法才有效，可在组件启动前调用“*SetMarkSilence(TRUE)*”方法来开启静默标记。静默标记默认不开启。

- ***Q*：断线重连该如何实现？**
  - ***A***：既然不能在事件处理代码中调用 *Start() / Stop()* 控制方法进行重连，那么可以选择以下方法实现：
    1) 启动一个监测线程或定时器，定期调用组件对象的 *GetState()* 方法检查组件对象的状态，如果状态为 *SS_STOPED* 则执行重连。
    2) 启动一个监测线程，在组件的 OnClose / OnError 事件中向监测线程发送重连通知（Event）激活监测线程，监测线程循环调用组件对象的 *GetState()* 方法检查组件对象的状态，直到状态为 *SS_STOPED* 则执行重连。
    3) 使用窗口消息机制结合 *::PostMessage() / ::PostThreadMessage()* API 函数替代 2) 中的监测线程和通知（Event）。

- ***Q*：HP-Socket 有心跳检测机制吗？**
  - ***A***：有（UdpCast 组件除外）。TCP 组件使用 TCP 协议内置的心跳检测机制，UDP 组件通过互发 0 字节数据包实现心跳检测：
    1) TCP 心跳检测：*SetKeepAliveTime() / SetKeepAliveInterval()*，单位 - 毫秒
    2) UDP 心跳检测：*SetDetectInterval() / SetDetectAttempts()*，单位 - 秒

- ***Q*：为什么 HP-Socket 的 UDP 组件与我的 UDP 程序通信经常会断开连接？**
  - ***A***：HP-Socket 的 UDP 服务端和客户端组件默认都开启了心跳检测机制，与第三方

UDP 程序通信时，有两种选择：

1) 调用 *SetDetectInterval(0)* / *SetDetectAttempts(0)* 关闭 UDP 组件的心跳检测机制。
2) 你自己的程序实现与 HP-Socket 的 UDP 心跳检测握手。具体方法是：
   - 使用 IUdpClient 作为客户端：当服务端接收到 0 字节的 UDP 心跳数据包时立刻回复一个 0 字节的握手包。
   - 使用 IUdpServer 作为服务端：客户端需要定期向服务端发送 0 字节的 UDP 心跳数据包。

- ***Q*：HP-Socket 的 TCP 组件是否实现了粘包处理？**
  - ***A*：没有。但可以通过 PULL 模型与应用层协议的相互配合，使得应用程序可以免除粘包处理和分拆包工作。**

- ***Q*：多个线程同时发送数据时会不会造成发送方或接收方发送数据包乱序？**
  - ***A*：不会。对于发送方，HP-Socket 会确保每个 *Send()*方法调用所发出的数据都是完整有序的，不会受其它 *Send()* 方法干扰；对于接收方，HP-Socket 对同一连接不会同时触发 *OnReceive* 多个事件，因此，接收方也不会发生数据包乱序的情形。**

- ***Q*：多个通信组件能共享同一个监听器对象吗多个？**
  - ***A*：可以。根据实际的使用场景来确定是否应该用一个监听器对象监听多个组件的通信事件。**