HP-Socket

高性能网络通信框架

Version - 5.5.2

Bruce Liang

2019-05-20

前 言

HP-Socket 是一套通用的高性能 TCP/UDP/HTTP 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/HTTP 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、PACK 模型示例、性能测试示例以及其它编程语言示例）。HP-Socket 目前支持 Windows 和 Linux 平台。

✧ 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从头写一个来得方便。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。
- HP-Socket 提供 PUSH / PULL / PACK 等接收模型，应用程序可以灵活选择以手工方式、半自动方式或全自动方式处理封解包，PULL / PACK 接收模型在降低封解包处理复杂度的同时能大大减少出错几率。

✧ 高性能

- **Server 组件：**基于 IOCP / EPOLL 通信模型，并结合缓存池、私有堆等技术实现高效内存管理，支持超大规模、高并发通信场景。
- **Agent 组件：**Agent 组件实质上是 Multi-Client 组件，与 Server 组件采用相同的技术架构。一个 Agent 组件对象可同时建立和高效处理大规模 Socket 连接。
- **Client 组件：**基于 Event Select / POLL 通信模型，每个组件对象创建一个通信线程并管理一个 Socket 连接，适用于小规模客户端场景。

✧ 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：

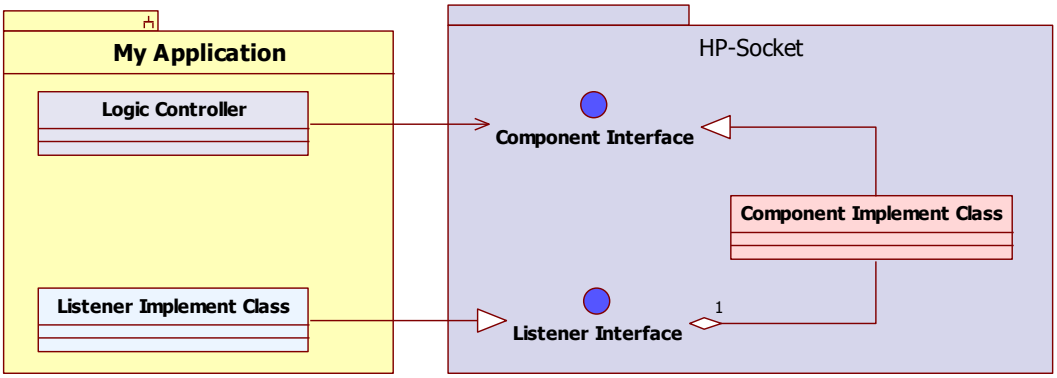


图 1.1-1 HP-Socket API 接口模型

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“连接绑定”能协助应用程序巧妙地避免由于多线程同步和互斥锁等导致的复杂性和性能问题。

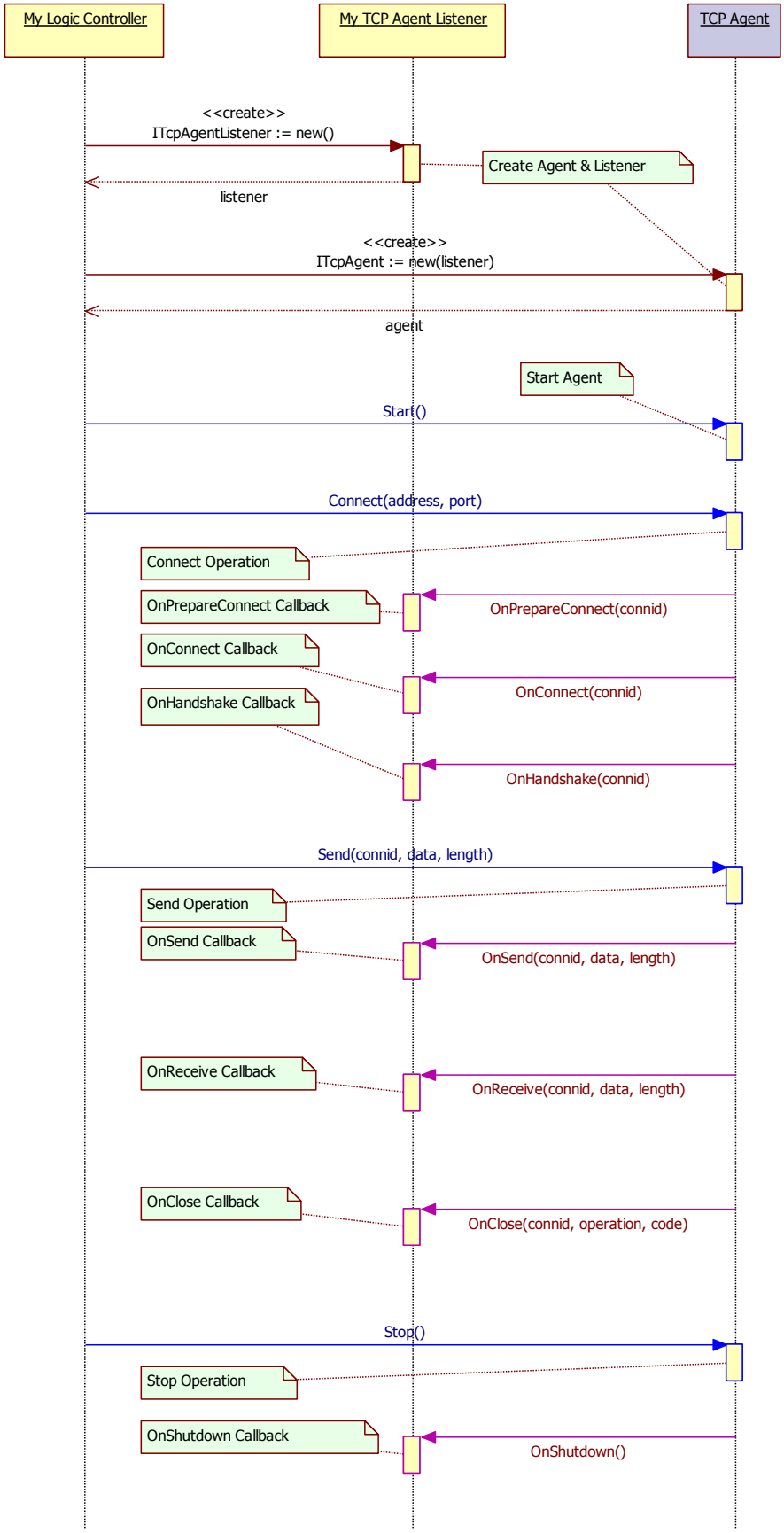


图 1.1-2 TCP Agent 与应用程序交互示例

1.2 组件分类

HP-Socket 包含 31 个组件（其中 9 个 SSL 组件将在第 3 章详细讲解，8 个 HTTP 组件将在第 4 章详细讲解），可根据通信角色（Client / Server）、通信协议（TCP / UDP/HTTP）和接收模型（PUSH / PULL / PACK）进行归类。表 1.2-1 列出了 TCP / UDP 组件的名称、接口、监听器接口、实现类及其分类：

Name	Component Intface Listener Intface	Implement Class	Role	Protocol	Recv Model
TCP Server	*ITcpServer* *ITcpServerListener*	CTcpServer	Server	TCP	PUSH
TCP Pull Server	*ITcpPullServer* *ITcpServerListener*	CTcpPullServer	Server	TCP	PULL
TCP Pack Server	*ITcpPackServer* *ITcpServerListener*	CTcpPackServer	Server	TCP	PACK
UDP Server	*IUdpServer* *IUdpServerListener*	CUdpServer	Server	UDP	PUSH
UDP ARQ Server	*IUdpArqServer* *IUdpServerListener*	CUdpArqServer	Server	UDP	PUSH
TCP Agent	*ITcpAgent* *ITcpServerListener*	CTcpAgent	Client	TCP	PUSH
TCP Pull Agent	*ITcpPullAgent* *ITcpAgentListener*	CTcpPullAgent	Client	TCP	PULL
TCP Pack Agent	*ITcpPackAgent* *ITcpServerListener*	CTcpPackAgent	Client	TCP	PACK
TCP Client	*ITcpClient* *ITcpClientListener*	CTcpClient	Client	TCP	PUSH
TCP Pull Client	*ITcpPullClient* *ITcpClientListener*	CTcpPullClient	Client	TCP	PULL
TCP Pack Client	*ITcpPackClient* *ITcpClientListener*	CTcpPackClient	Client	TCP	PACK
UDP Client	*IUdpClient* *IUdpClientListener*	CUdpClient	Client	UDP	PUSH
UDP ARQ Client	*IUdpArqClient* *IUdpClientListener*	CUdpArqClient	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 和 HTTP 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。

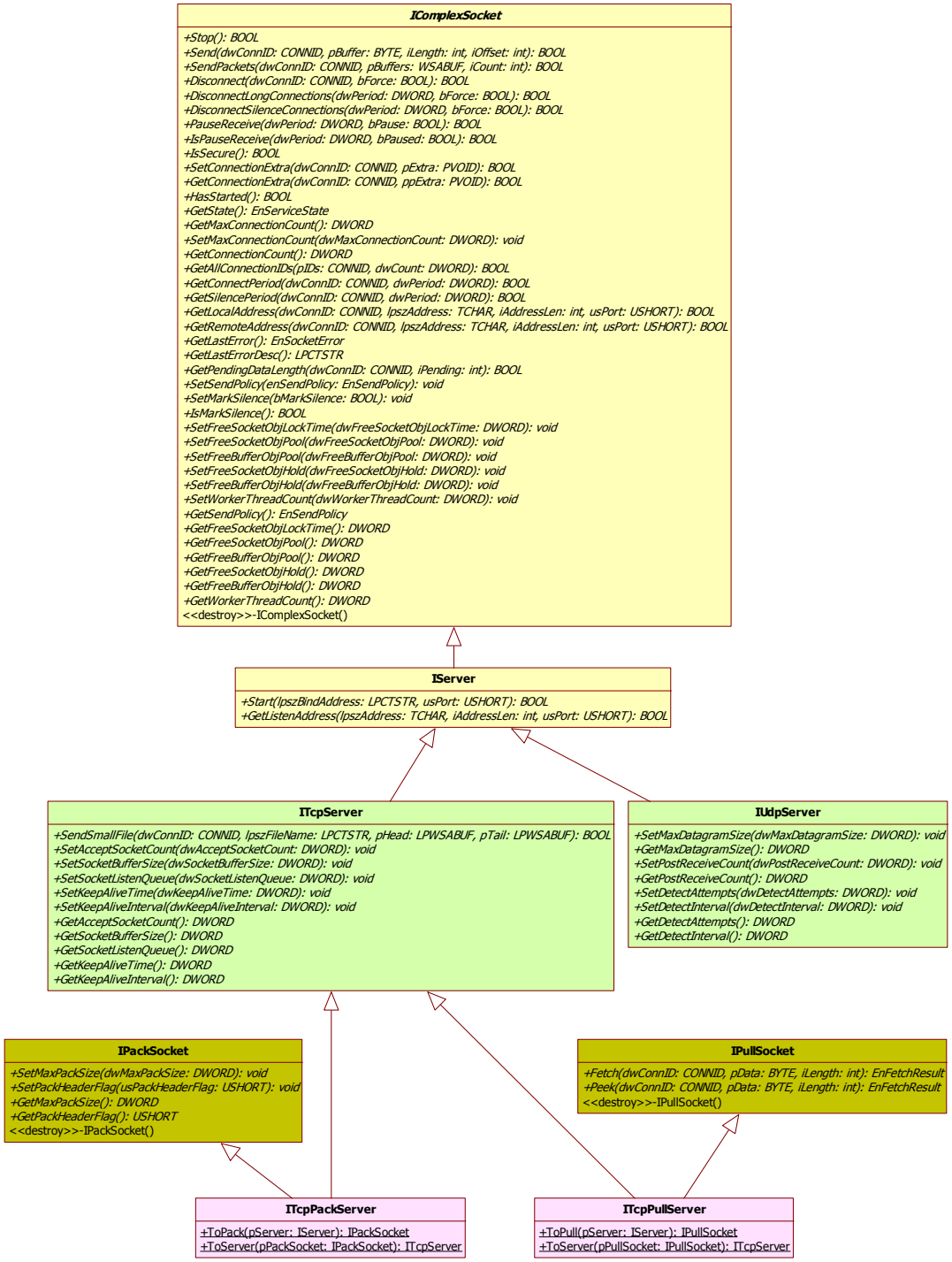


图 1.3-1 Server 组件接口

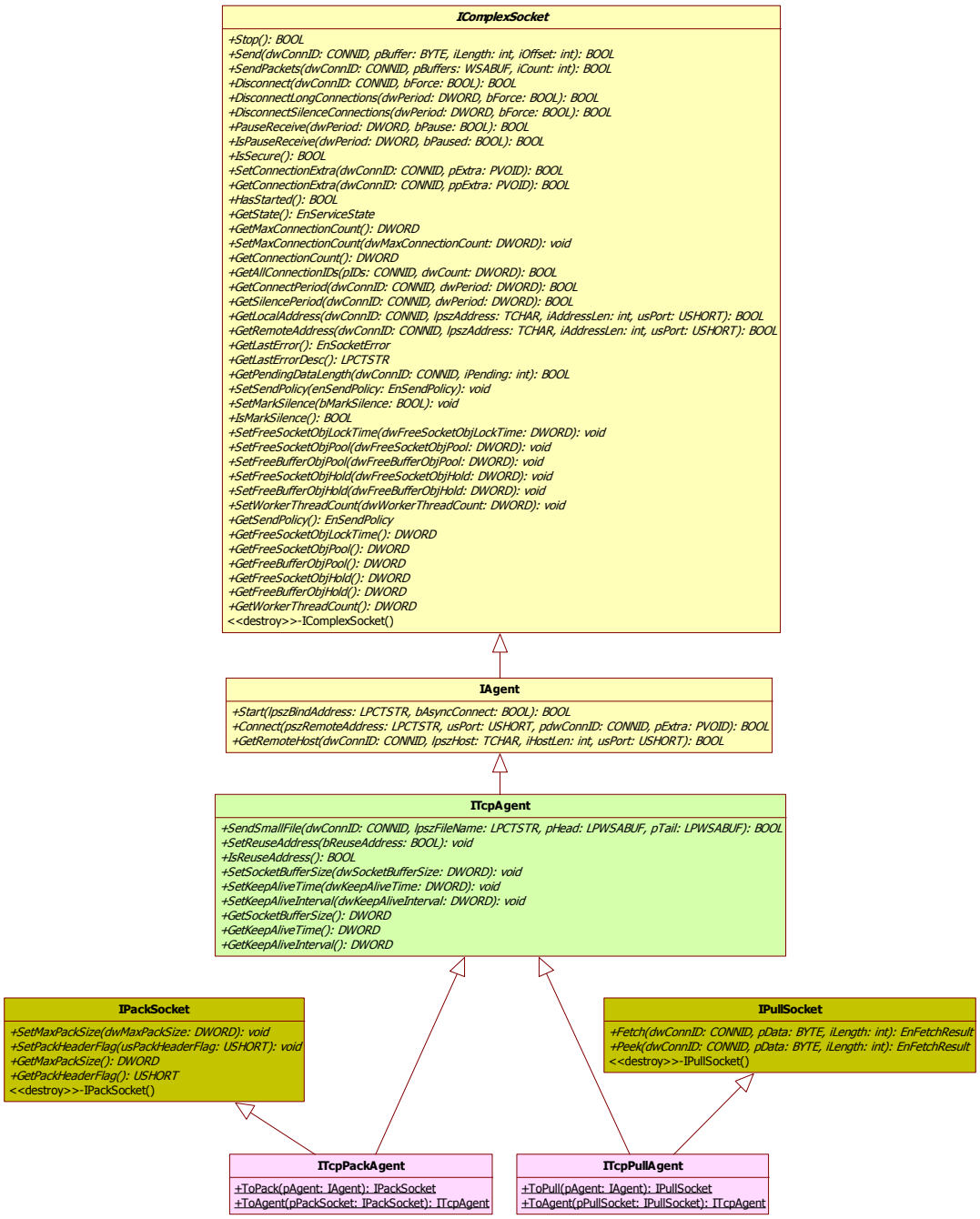


图 1.3-2 Agent 组件接口

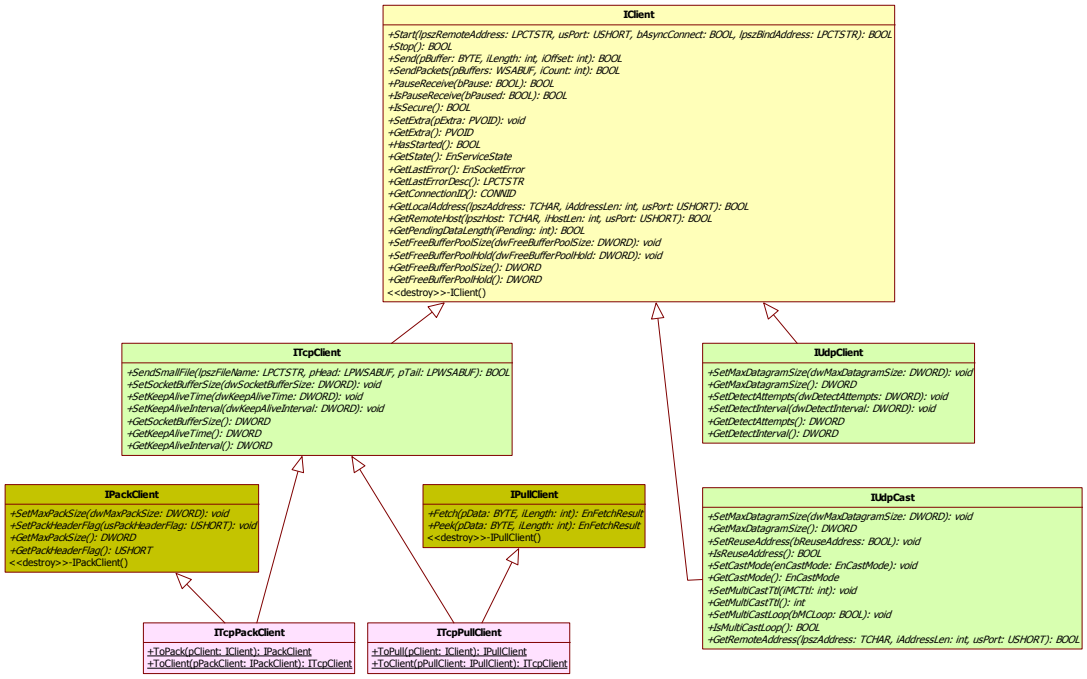


图 1.3-3 Client 组件接口

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：

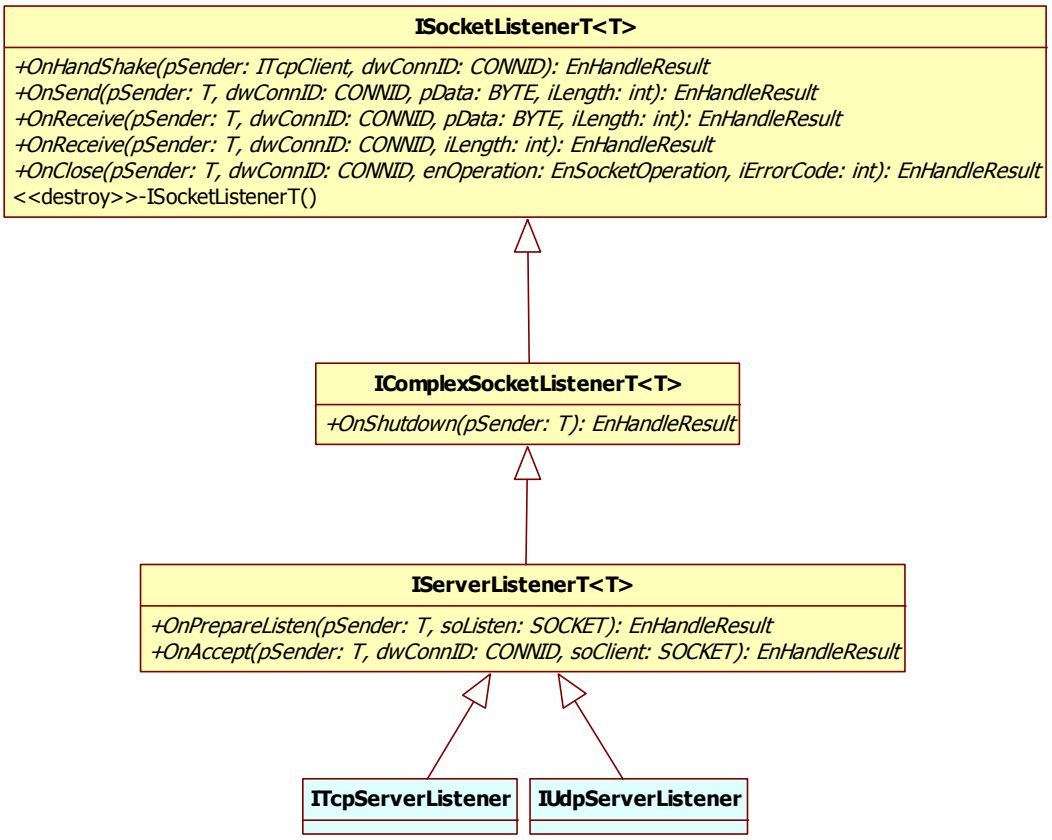


图 1.4-1 Server 监听器接口

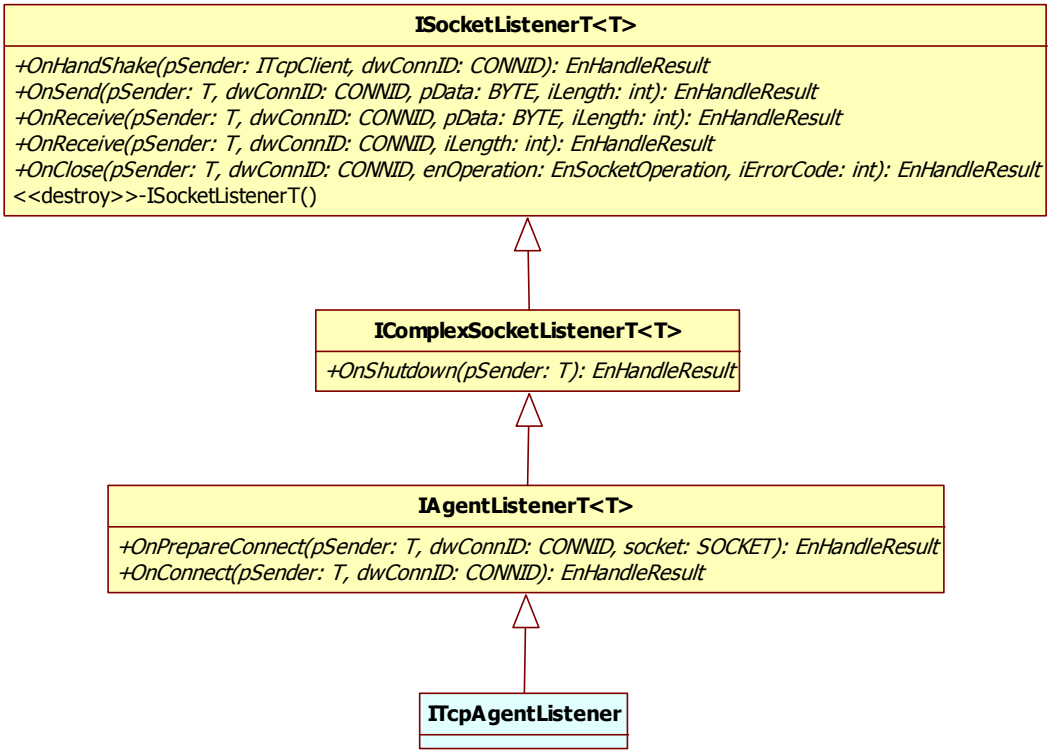


图 1.4-2 Agent 监听器接口

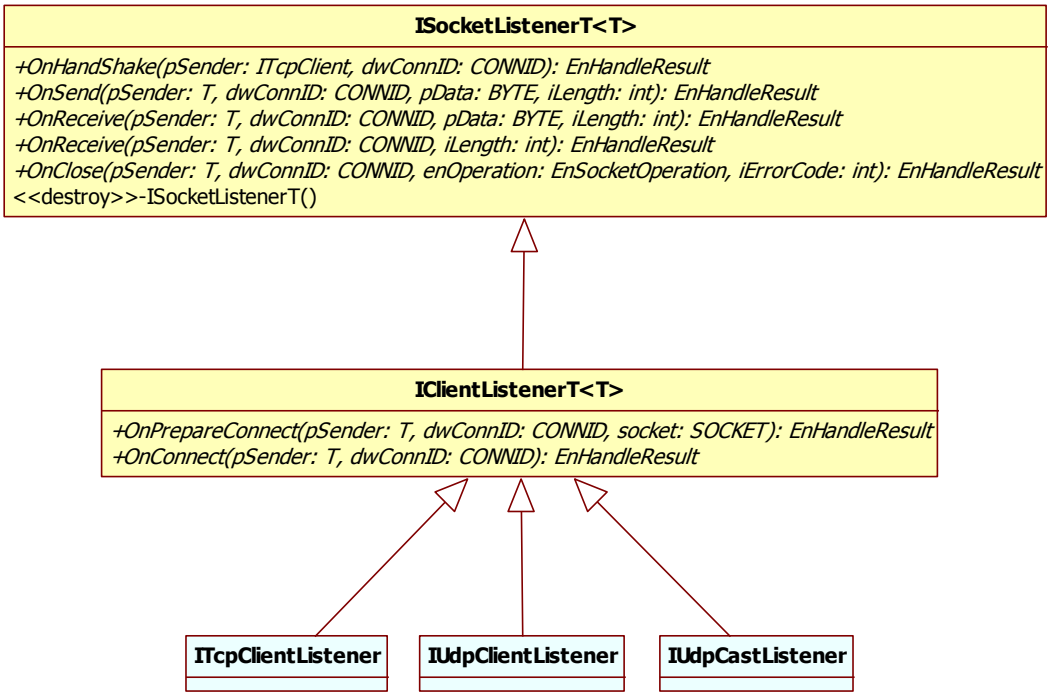


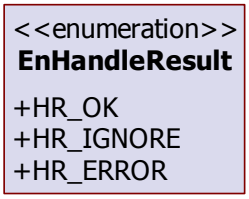
图 1.4-3 Client 监听器接口

HP-Socket 没有为 PUSH / PULL / PACK 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 和 PACK 模型组件接收到数据时会触发监听器对象的 *OnReceive*(*pSender*, *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*(*pSender*, *dwConnID*, *iLength*) 事件。事件的含义如表 1.4-1 所示：

接 口	事 件	说 明
ISocketListenerT	*OnHandShake*(*pSender*, *dwConnID*)	握手完成
	OnSend(*pSender*, *dwConnID*, *pData*, *iLength*)	数据已发送
	OnReceive(*pSender*, *dwConnID*, *pData*, *iLength*)	数据到达（PUSH）
	OnReceive(*pSender*, *dwConnID*, *iLength*)	数据到达（PULL）
	OnClose(*pSender*, *dwConnID*, *enOperation*, *iErrorCode*)	连接关闭
IComplexSocketListenerT	*OnShutdown*(*pSender*)	关闭组件
IServerListenerT	*OnPrepareListen*(*pSender*, *soListen*)	准备监听
	OnAccept(*pSender*, *dwConnID*, *soClient*)	接受连接
IAgentListenerT	*OnPrepareConnect*(*pSender*, *dwConnID*, *socket*)	准备连接
	OnConnect(*pSender*, *dwConnID*)	完成连接
IClientListenerT	*OnPrepareConnect*(*pSender*, *dwConnID*, *socket*)	准备连接
	OnConnect(*pSender*, *dwConnID*)	完成连接

表 1.4-1 监听器接口事件

监听器事件回调方法返回值的类型为 *EnHandleResult*：



- ✓ **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 / OnHandShake 事件回调方法返回 HR_ERROR 时，组件会立即中断连接。

为了使非 SSL 组件和 SSL 组件的处理流程一致，HP-Socket v4.0.x 开始，非 SSL 组件也会触发 OnHandShake 事件。因此，当组件接收到 OnHandShake 事件即说明连接已建立，并可以开始通信。

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、PULL 和 PACK 三种接收模型：

- PUSH 模型：组件接收到数据时会触发监听器对象的 *OnReceive*(*pSender*, *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*(*pSender*, *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。
- PACK 模型：PACK 模型系列组件是 PUSH 和 PULL 模型的结合体，应用程序不必处理分包（如：PUSH）与数据抓取（如：PULL），组件保证每个 OnReceive 事件都向应用程序提供一个完整数据包。

三种模型的比较如图 2.1.1-1 所示：

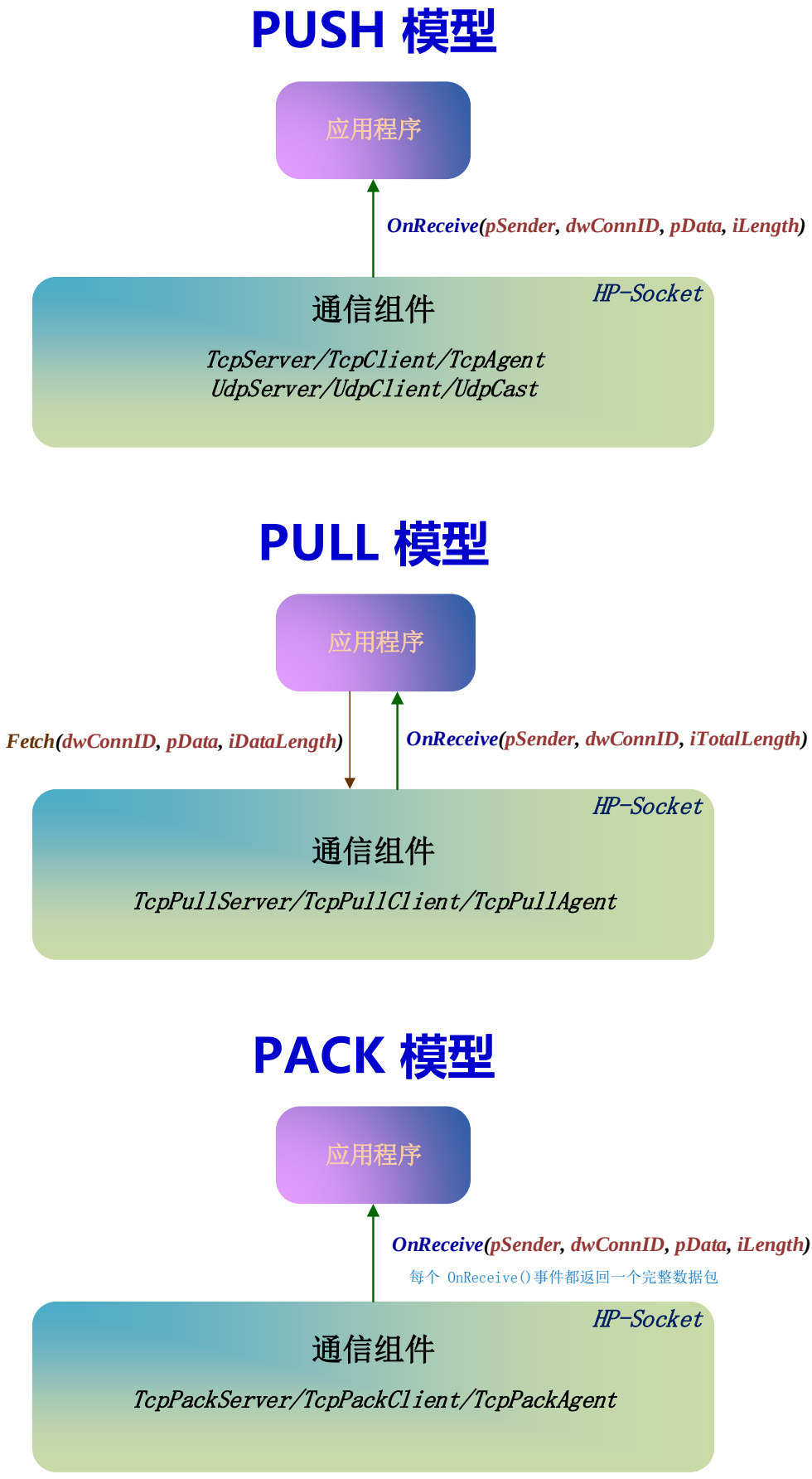
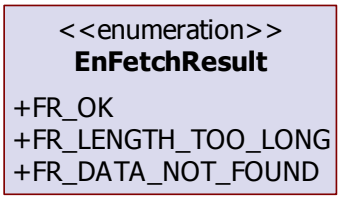

PUSH 模型
应用程序
OnReceive(pSender, dwConnID, pData, iLength)
HP-Socket
通信组件
TcpServer/TcpClient/TcpAgent
UdpServer/UdpClient/UdpCast
PULL 模型
应用程序
Fetch(dwConnID, pData, iDataLength)
OnReceive(pSender, dwConnID, iTotalLength)
HP-Socket
通信组件
TcpPullServer/TcpPullClient/TcpPullAgent
PACK 模型
应用程序
OnReceive(pSender, dwConnID, pData, iLength)
每个 OnReceive()事件都返回一个完整数据包
HP-Socket
通信组件
TcpPackServer/TcpPackClient/TcpPackAgent

图 2.1.1-1 接收模型

- PUSH 模型组件触发监听器对象的 *OnReceive*(*pSender*, *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*(*pSender*, *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*) 方法返回值的类型为 *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。

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。

- PACK 模型组件触发监听器对象的 *OnReceive*(*pSender*, *dwConnID*, *pData*, *iLength*) 事件时，会保证 *pData* 是一个完整的数据包。PACK 模型组件会对应用程序发送的每个数据包自动加上 4 字节（32 位）的包头，组件接收到数据时根据包头信息自动分包，每个完整数据包通过 *OnReceive* 事件发送给应用程序。

PACK 包头格式：

XXXXXXXXXX YYYYYYYYYYYYYYYYYYYY

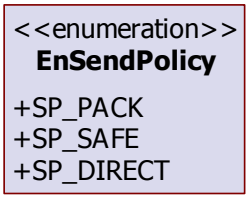
前 10 位 X 为包头标识位，用于数据包校验。有效包头标识取值范围 0 ~ 1023（0x3FF），当包头标识等于 0 时不校验包头。后 22 位 Y 为长度位，记录包体长度。有效数据包最大长度不能超过 4194303（0x3FFFFF）字节，默认长度限制为：262144（0x40000）字节。应用程序可以通过 *SetPackHeaderFlag()* 和 *SetMaxPackSize()* 分别设置包头标识与最大包长限制。

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。
  注：SP_DIRECT 通常与 TCP_NODELAY Socket 选项配合使用来获得最低延时。TCP_NODELAY Socket 选项可以通过 HP-Socket 导出函数 SYS_SSO_NoDelay() 设置。

对于 SP_PACK 和 SP_SAFE 策略，组件内部会缓存待发送的数据。另外，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。

注意：基于 Linux 的 Socket 模型特点，Linux 平台的通信组件不支持发送策略设置，所有 Linux 通信组件的发送策略均为 SP_PACK。也就是说，SetSendPolicy(enSendPolicy) 方法对 Linux 通信组件无效。

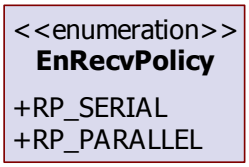
2.1.3 接收策略

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* 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 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* 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 v3.3 及其之前的版本提供了以下两种接收策略（*通过 SetRecvPolicy(enRecvPolicy) 方法设置*）：

-



- ✓ **RP_SERIAL** ：串行策略（默认）
  对于单个连接，顺序触发 OnReceive 和 OnClose 等事件。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 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 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。

HP-Socket v3.4 版本开始，废弃了 RP_PARALLEL 接收策略。HP-Socket 保证除了 OnSend 事件以外其它所有事件都是线程安全的。

注意：对于 HP-Socket 的所有组件（IServer / IAgent / IClient），当连接触发了 OnClose 事件时，表示连接已被关闭。并且 OnClose 事件只会触发一次，也就是说：同一个连接不可能收到两个或多个 OnClose 事件。

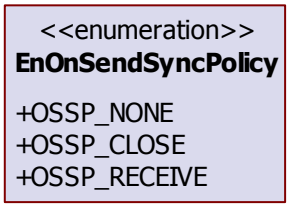
HP-Socket v3.3 及其之前版本中提供了 OnClose 和 OnError 两个事件，分别表示正常关闭和异常关闭连接。HP-Socket v3.4 开始，这两个事件已合并为一个事件，应用程序可根据 OnClose 事件的 iErrorCode 参数判断是正常关闭还是异常关闭。

2.1.4 OnSend 事件同步策略

HP-Socket v5.4.2 版本开始，IServer 和 IAgent 系列组件支持对 *OnSend* 事件设置同步策略。

（OnSend 事件同步策略通过 SetOnSendSyncPolicy(enSyncPolicy) 方法进行设置）

-



- ✓ **OSSP_NONE** ：不同步（默认）
  不同步 OnSend 事件，可能同时触发 OnReceive 和 OnClose 事件。
- ✓ **OSSP_CLOSE** ：同步 OnClose
  只同步 OnClose 事件，可能同时触发 OnReceive 事件。
- ✓ **OSSP_RECEIVE** ：同步 OnReceive
  （只用于 TCP 组件）同步 OnReceive 和 OnClose 事件，不可能同时触发 OnReceive 或 OnClose 事件。

OnSend 事件对于一般应用程序来说意义不大，因此采用默认同步策略 OSSP_NONE 即可，这种情况下 *OnSend* 事件不是线程安全的，在处理 *OnSend* 事件的过程中可能会同时触发 *OnReceive* 或 *OnClose* 事件；OSSP_CLOSE 同步策略则会确保在处理 *OnSend* 事件的过程中不可能触发 *OnClose* 事件；OSSP_RECEIVE 同步策略则会确保在处理 *OnSend* 事件的过程中不可能触发 *OnReceive* 或 *OnClose* 事件。

注意：基于 Linux 的 Socket 模型特点，Linux 平台的通信组件不支持 OnSend 事件同步策略设置，所有 Linux 版本的 IServer 和 IAgent 通信组件的 OnSend 事件同步策略均为 OSSP_CLOSE。也就是说，SetOnSendSyncPolicy(enSyncPolicy) 方法对 Linux 通信组件无效。

2.1.5 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 *OnClose()* 事件可获知发送失败原因。

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）。

异步连接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 *OnClose()* 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 *OnClose()* 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(lpszRemoteAddress, usPort, bAsyncConnect = TRUE,
        lpszBindAddress = nullptr, usLocalPort = 0)
```

参数 *bAsyncConnect* 指示是否采用异步连接方式（默认：*TRUE*），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(lpszBindAddress = nullptr, bAsyncConnect = TRUE)
BOOL Connect(lpszRemoteAddress, usPort, pdwConnID = nullptr,
             pExtra = nullptr, usLocalPort = 0)
```

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：*TRUE*），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），参数 *pExtra* 设置“连接绑定”数据（默认：*nullptr*，不设置），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；如果设置了 *pExtra*，这时也要手工释放。

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(pSender, dwConnID, socket)***
✓ ***OnConnect pSender, dwConnID)***

其中 *OnPrepareConnect(pSender, 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(pSender, dwConnID)* 则在连接建立成功后触发。

2.1.6 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：

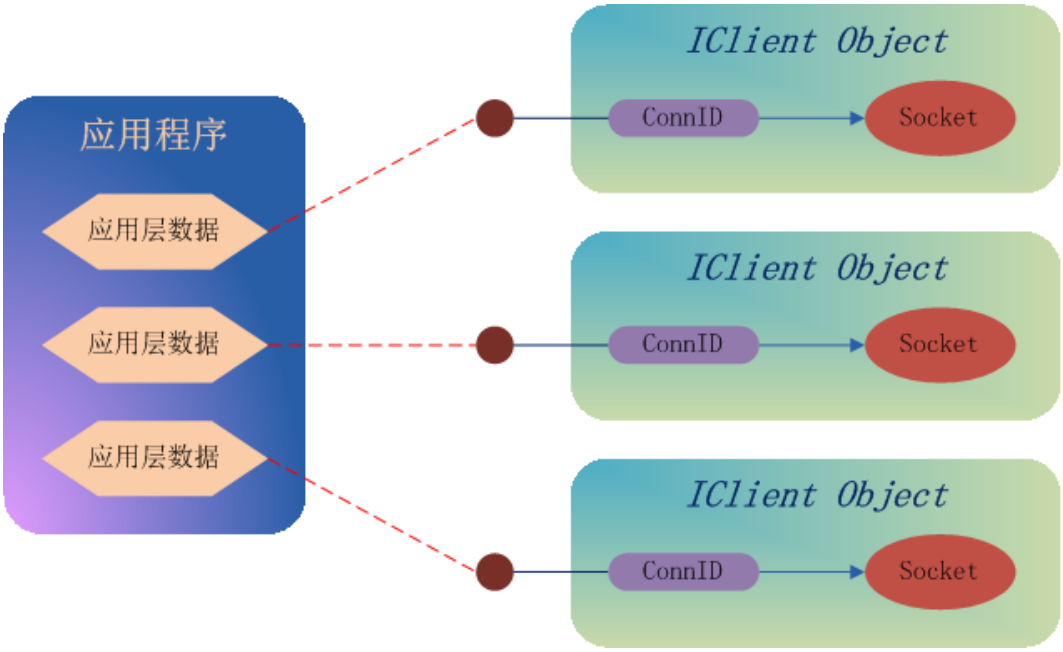


图 2.1.5-1 IClient 组件连接示意图

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 来告知组件处理哪个连接（如：*Send(dwConnID, pData, iLength)* ）。如图 2.1.5-2 示：

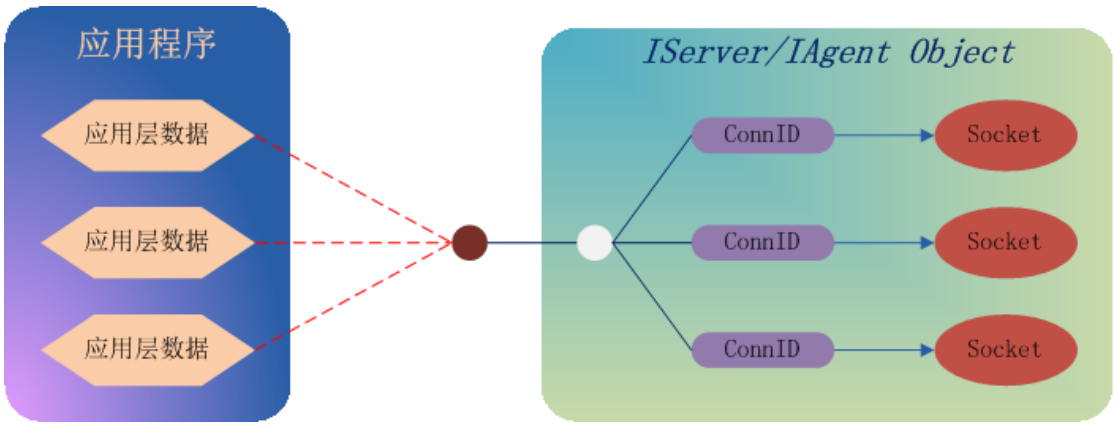


图 2.1.5-2 IServer/IAgent 组件连接示意图

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()* 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：由于 HP-Socket 已经确保了 OnReceive() / OnClose() 等事件的线程安全，因此应用程序可以放心使用连接绑定机制，不用担心同步问题。

2.2 Server 组件

2.2.1 接口描述

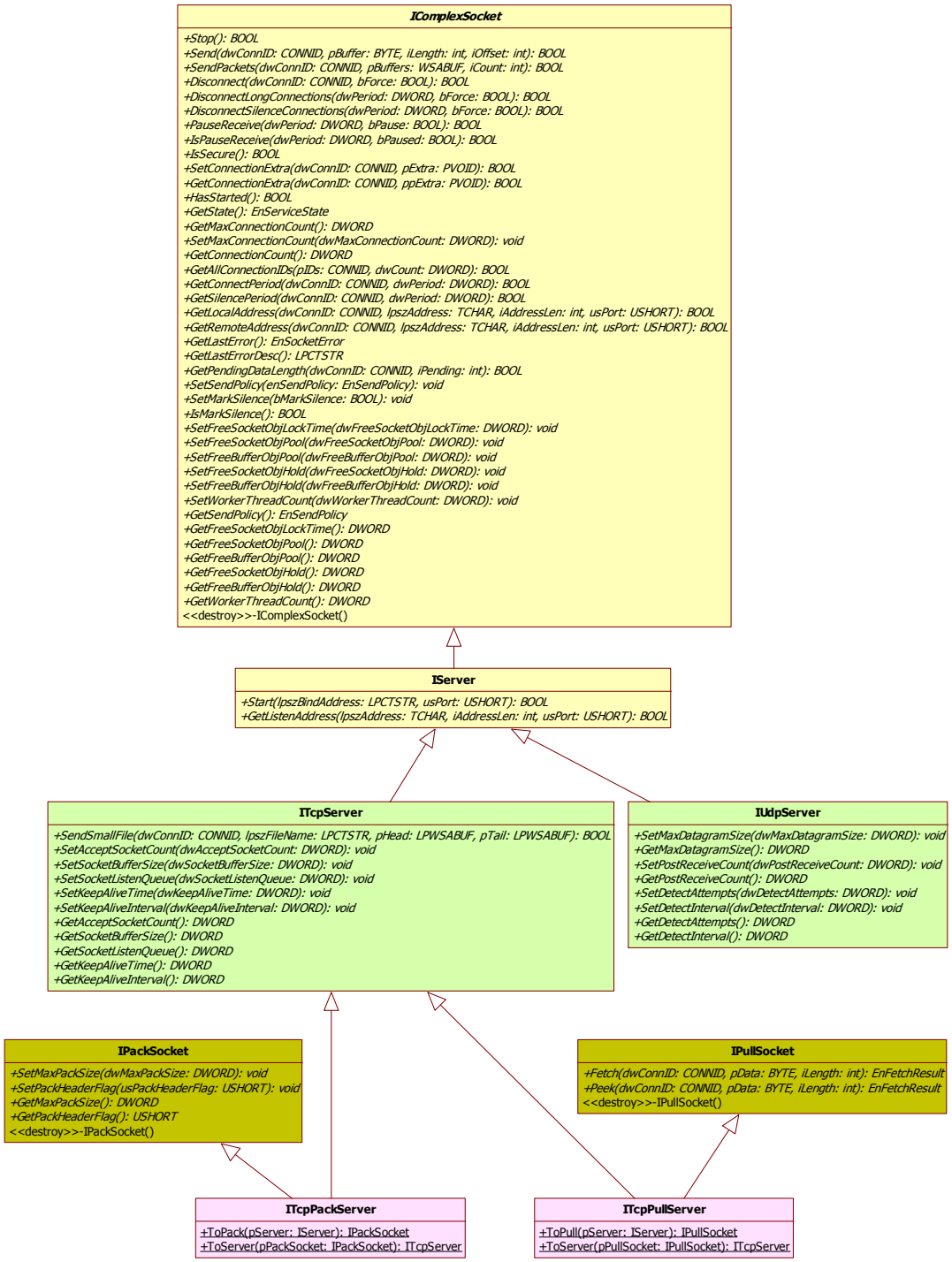


图 2.2.1-1 Server 组件接口

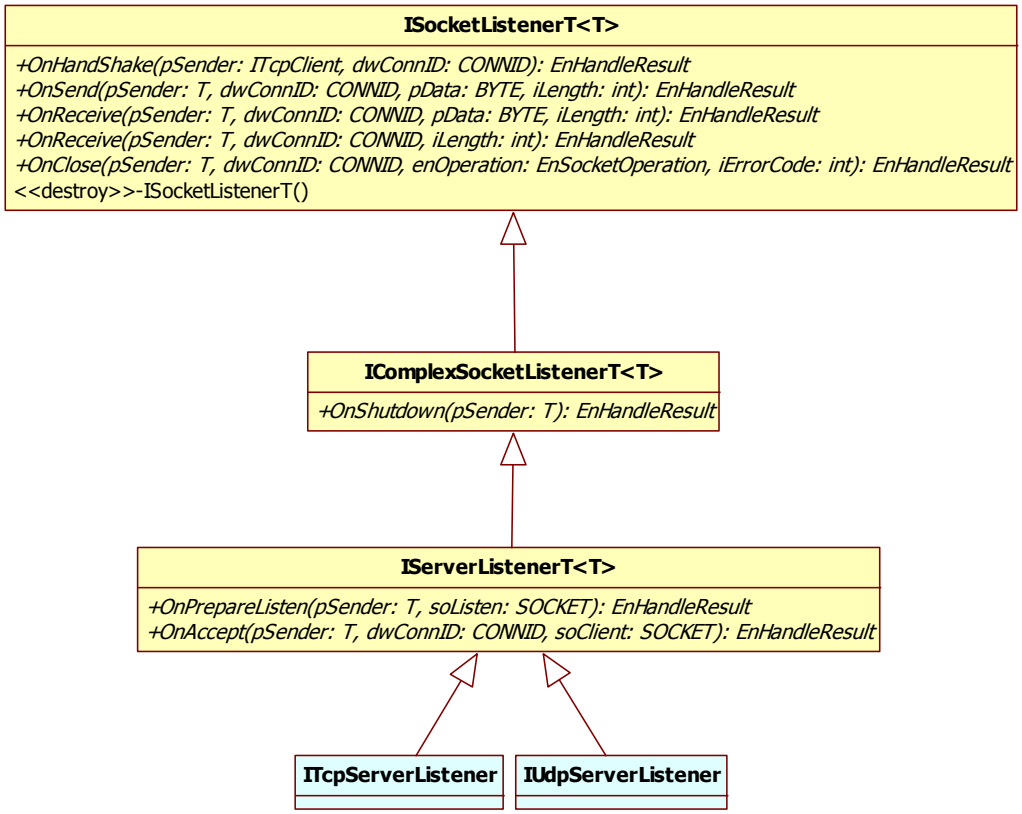


图 2.2.1-2 Server 监听器接口

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* 和 *ITcpPackServer* 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
	PauseReceive()	暂停接收数据
	IsConnected()	检测是否有效连接
	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

	GetLocalAddress()	获取某个连接的本地地址
	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()	设置工作线程数量
	SetMaxConnectionCount()	设置最大连接数量
ITcpServer	*SendSmallFile()*	发送小文件
	SetSocketListenQueue()	设置监听 Socket 的等候队列大小
	SetAcceptSocketCount()	Windows：设置 Accept 预投递数量 Linux：设置 EPOLL 事件最大数量
	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
ITcpPackServer	*SetMaxPackSize()*	设置最大包长限制
	SetPackHeaderFlag()	设置包头校验标识
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
IUdpArqServer	*SetMaxMessageSize()*	设置 ARQ 数据报文最大长度
	SetHandShakeTimeout()	设置 ARQ 握手超时时间

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
ISocketListenerT	*OnHandShake()*	握手完成 握手完成时触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH / PACK） 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 连接正常或异常关闭时触发
IComplexSocketListenerT	*OnShutdown()*	关闭通信组件 通信组件停止后触发
IServerListenerT	*OnPrepareListen()*	准备监听

		绑定监听地址前触发
	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

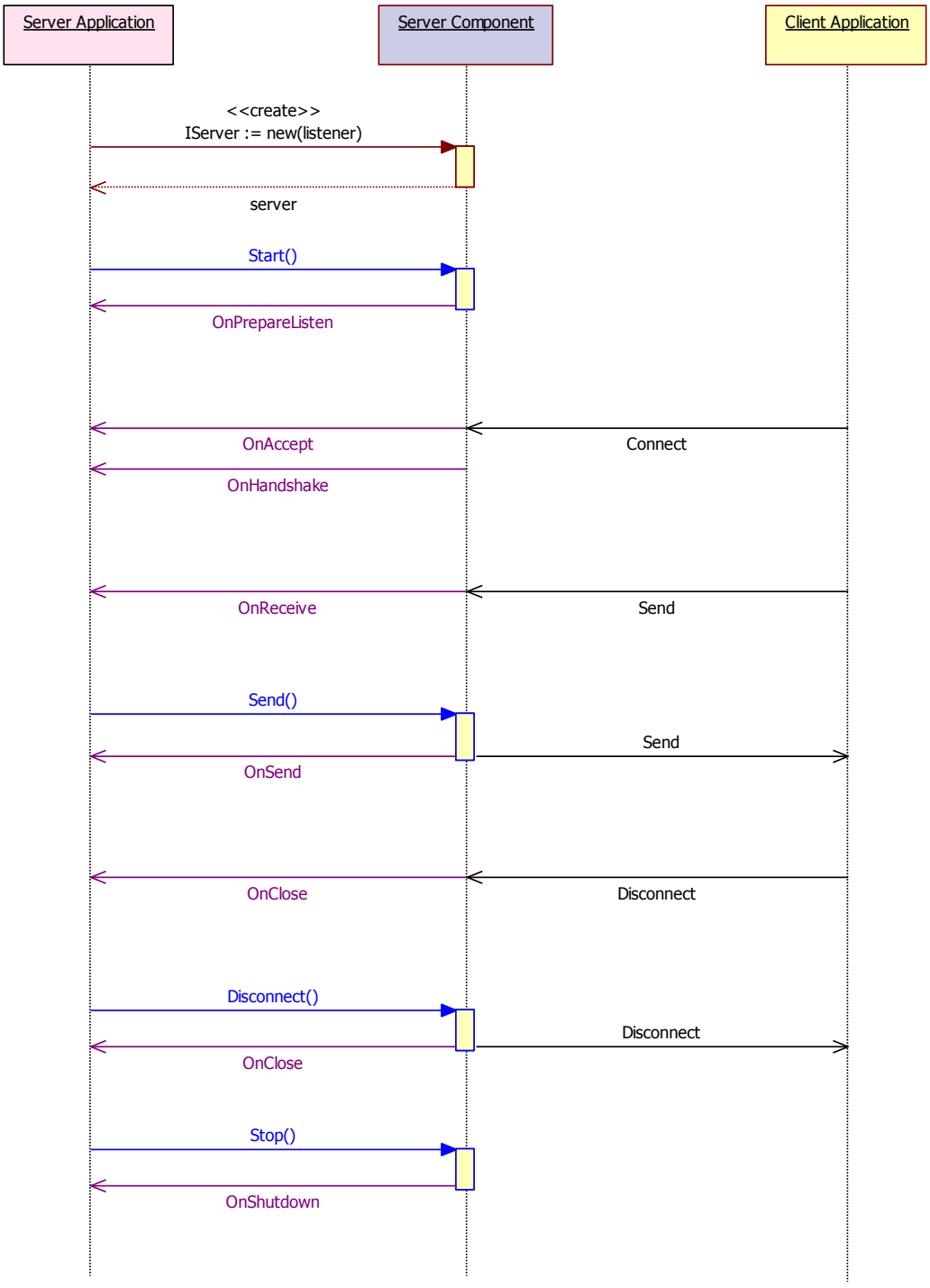


图 2.2.2-1 Server 工作流程

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* 和 *OnHandshake* 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* 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

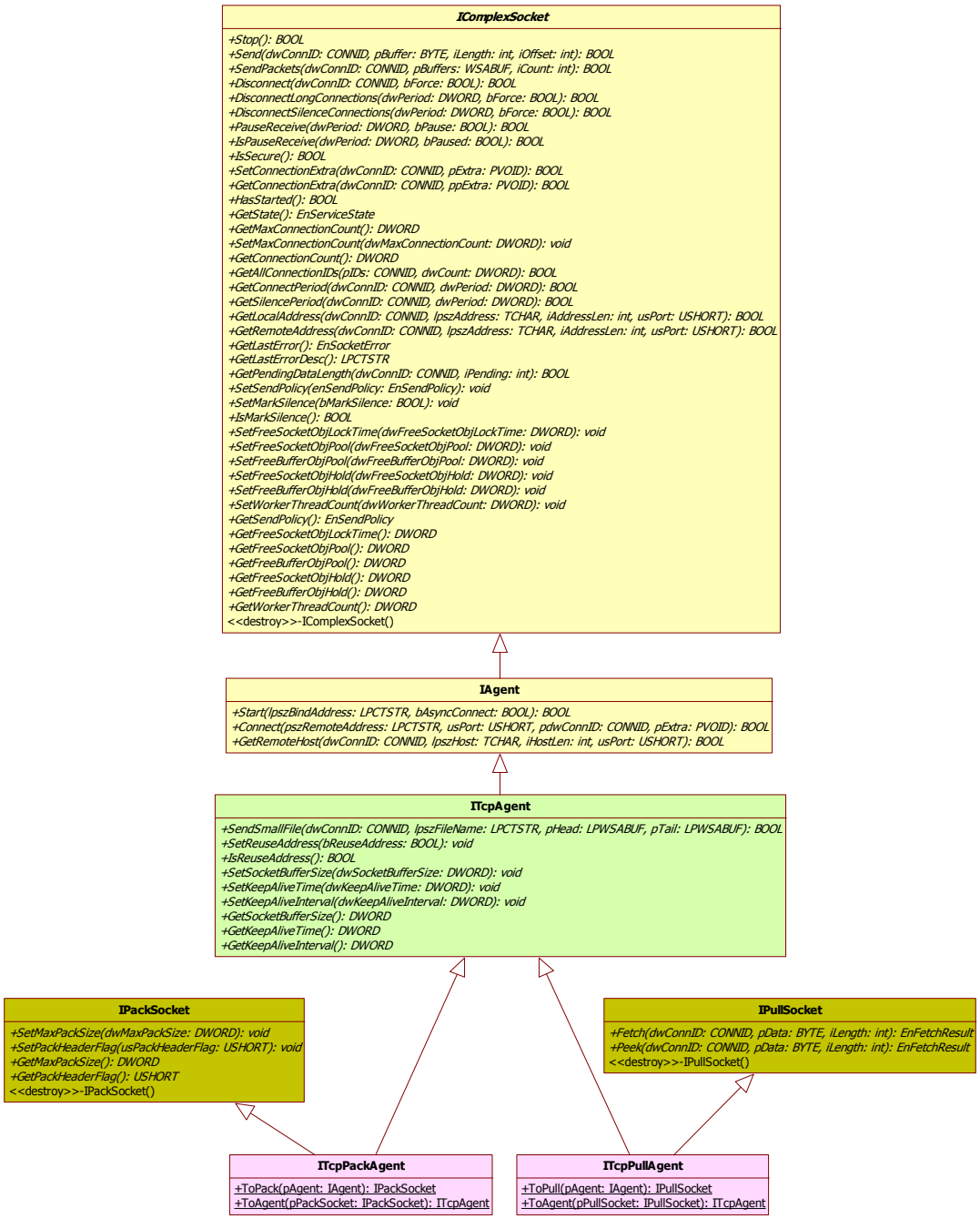


图 2.3.1-1 Agent 组件接口

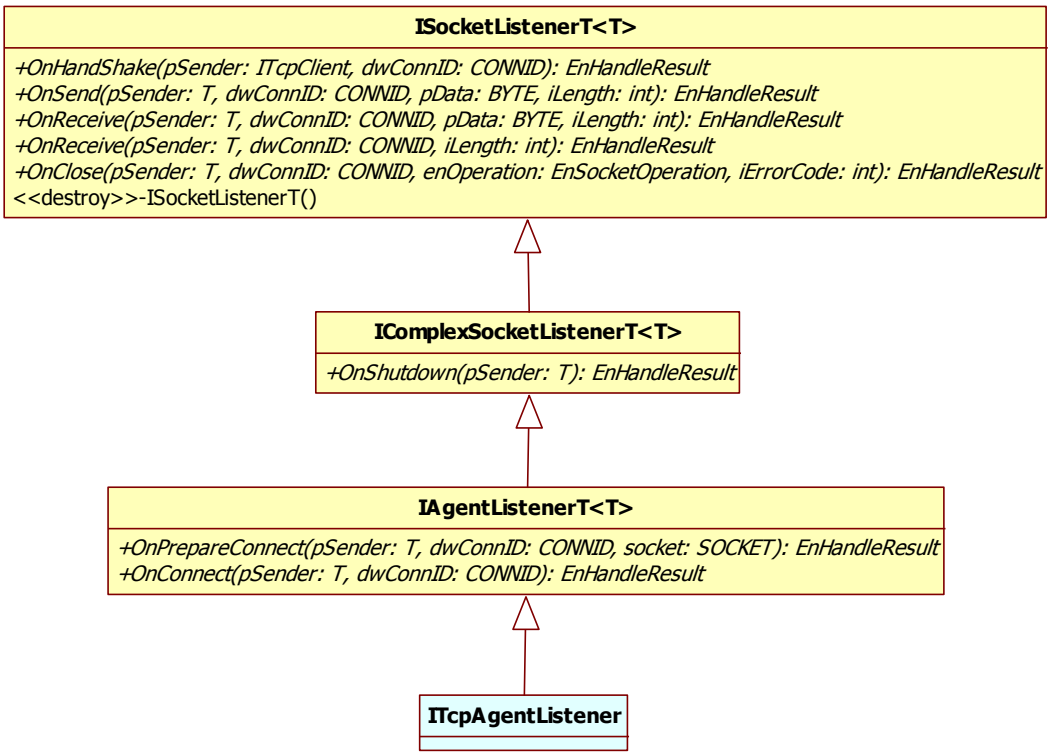


图 2.3.1-2 Agent 监听器接口

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* 和 *ITcpPackAgent* 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
	PauseReceive()	暂停接收数据
	IsConnected()	检测是否有效连接
	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

	GetLocalAddress()	获取某个连接的本地地址
	GetRemoteHost()	获取某个连接的远程主机
	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()	设置工作线程数量
	SetMaxConnectionCount()	设置最大连接数量
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
ITcpPackAgent	*SetMaxPackSize()*	设置最大包长限制
	SetPackHeaderFlag()	设置包头校验标识

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
ISocketListenerT	*OnHandShake()*	握手完成 握手完成时触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH / PACK） 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 连接正常或异常关闭时触发
IComplexSocketListenerT	*OnShutdown()*	关闭通信组件 通信组件停止后触发
IAgentListenerT	*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

表 2.3.1-2 Agent 监听器接口

2.3.2 工作流程

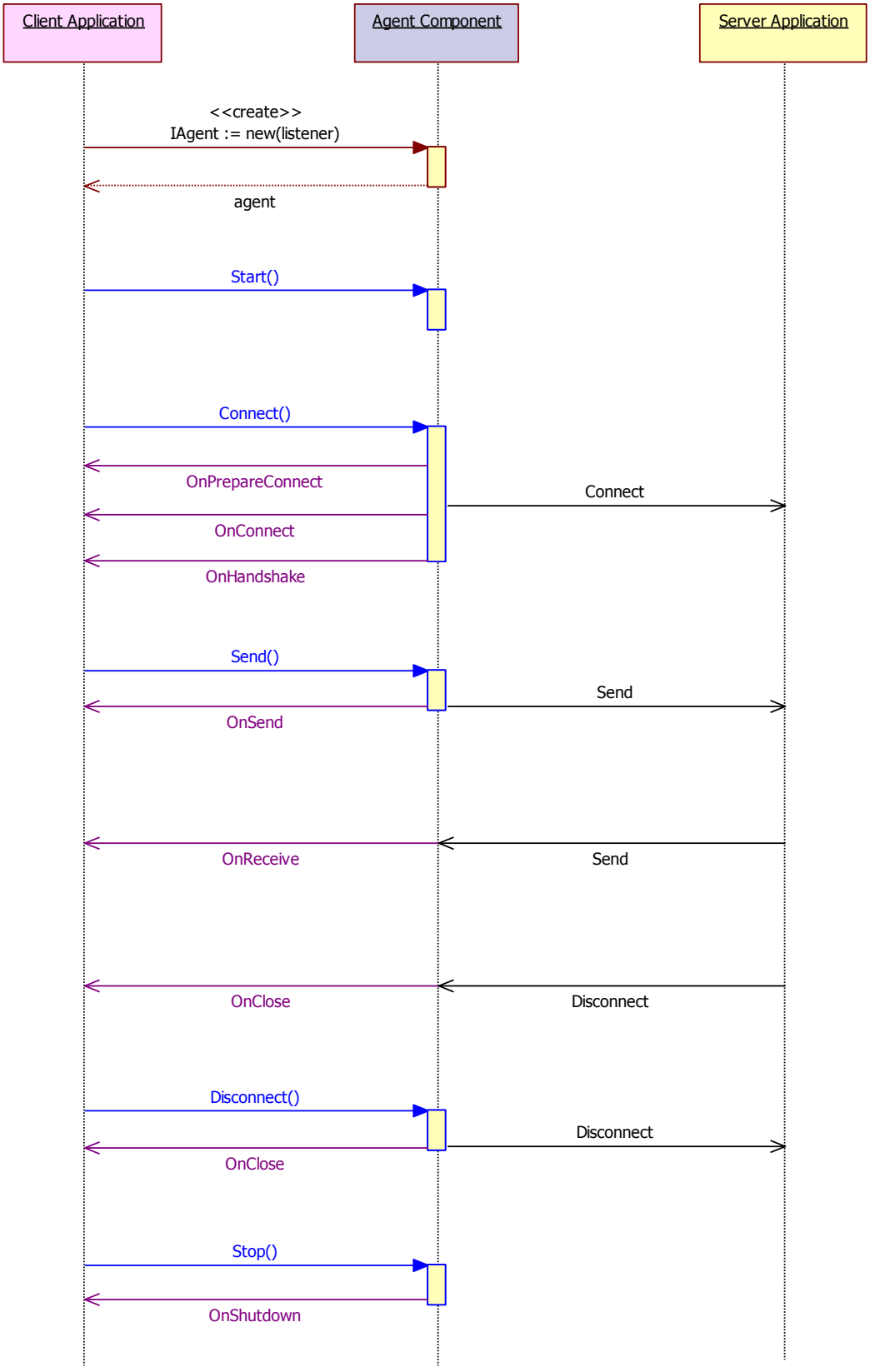


图 2.3.2-1 Agent 工作流程

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*、*OnConnect* 和 *OnHandshake* 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* 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

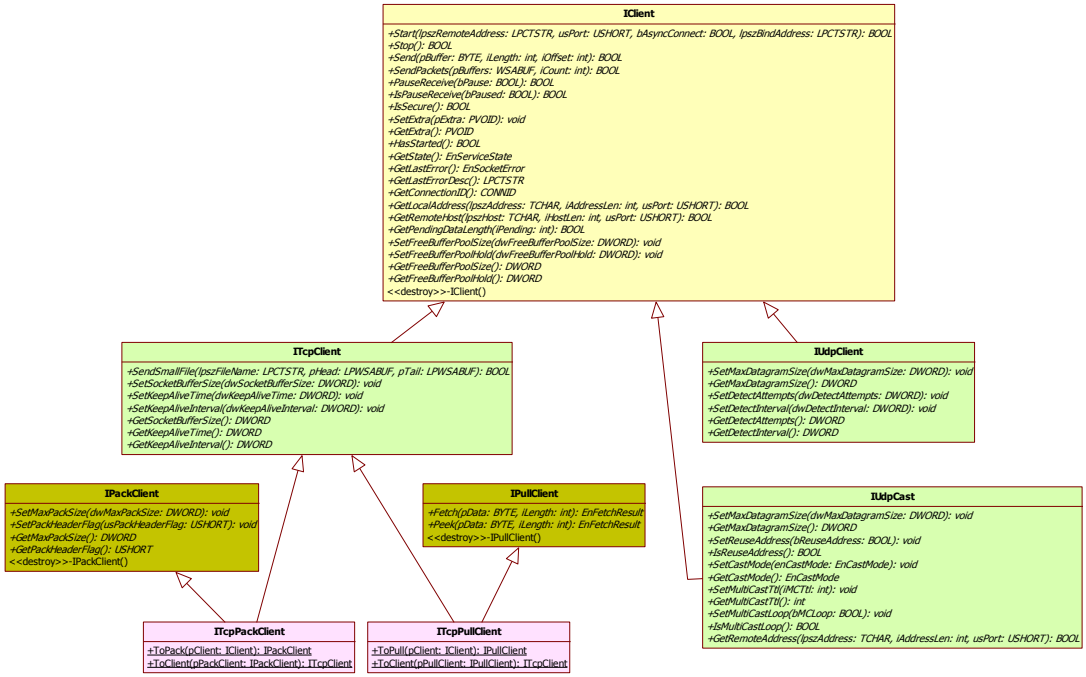


图 2.4.1-1 Client 组件接口

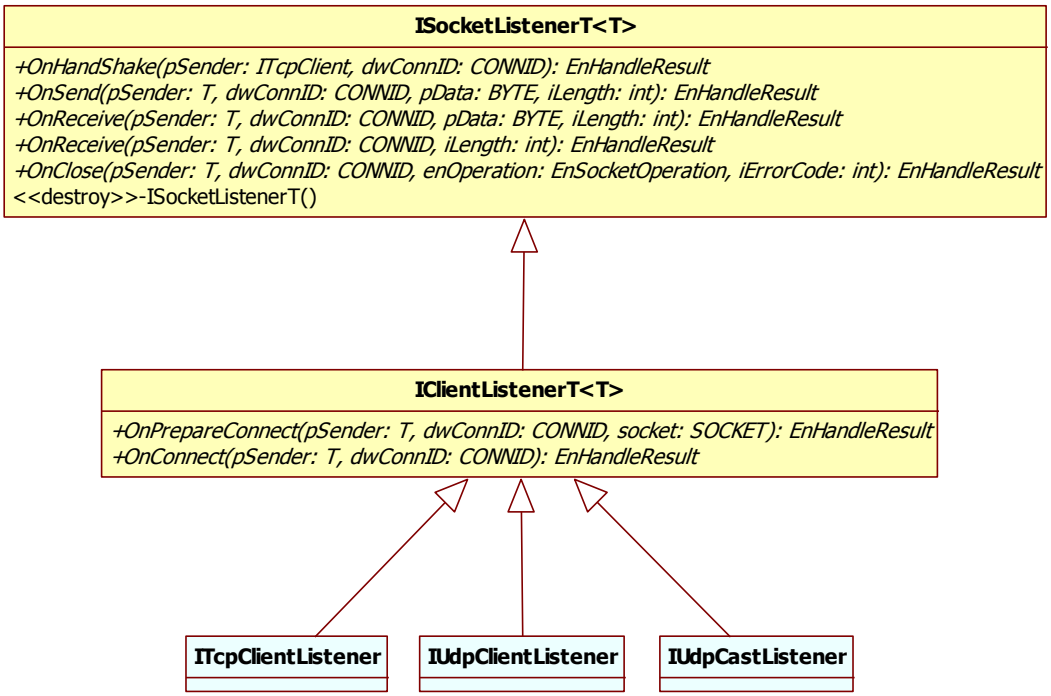


图 2.4.1-2 Client 监听器接口

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* 和 *ITcpPackClient* 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
	PauseReceive()	暂停接收数据
	IsConnected()	检测是否有效连接
	HasStarted()	检查通信组件是否已启动
	GetState()	获取通信组件当前状态
	GetConnectionID()	获取该组件对象的 CONNID
	GetLocalAddress()	获取连接的本地地址
	GetRemoteHost()	获取连接的远程主机
	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
ITcpPackClient	*SetMaxPackSize()*	设置最大包长限制
	SetPackHeaderFlag()	设置包头校验标识
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
IUdpArqClient	*SetMaxMessageSize()*	设置 ARQ 数据报文最大长度
	SetHandShakeTimeout()	设置 ARQ 握手超时时间
IUdpCast	*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()	设置组播报文 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
ISocketListenerT	*OnHandShake()*	握手完成 握手完成时触发
	OnSend()	数据已发送 数据发送成功后触发
	OnReceive() [PUSH]	数据到达（PUSH / PACK） 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 连接正常或异常关闭时触发
IClientListenerT	*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

表 2.4.1-2 Client 监听器接口

2.4.2 工作流程

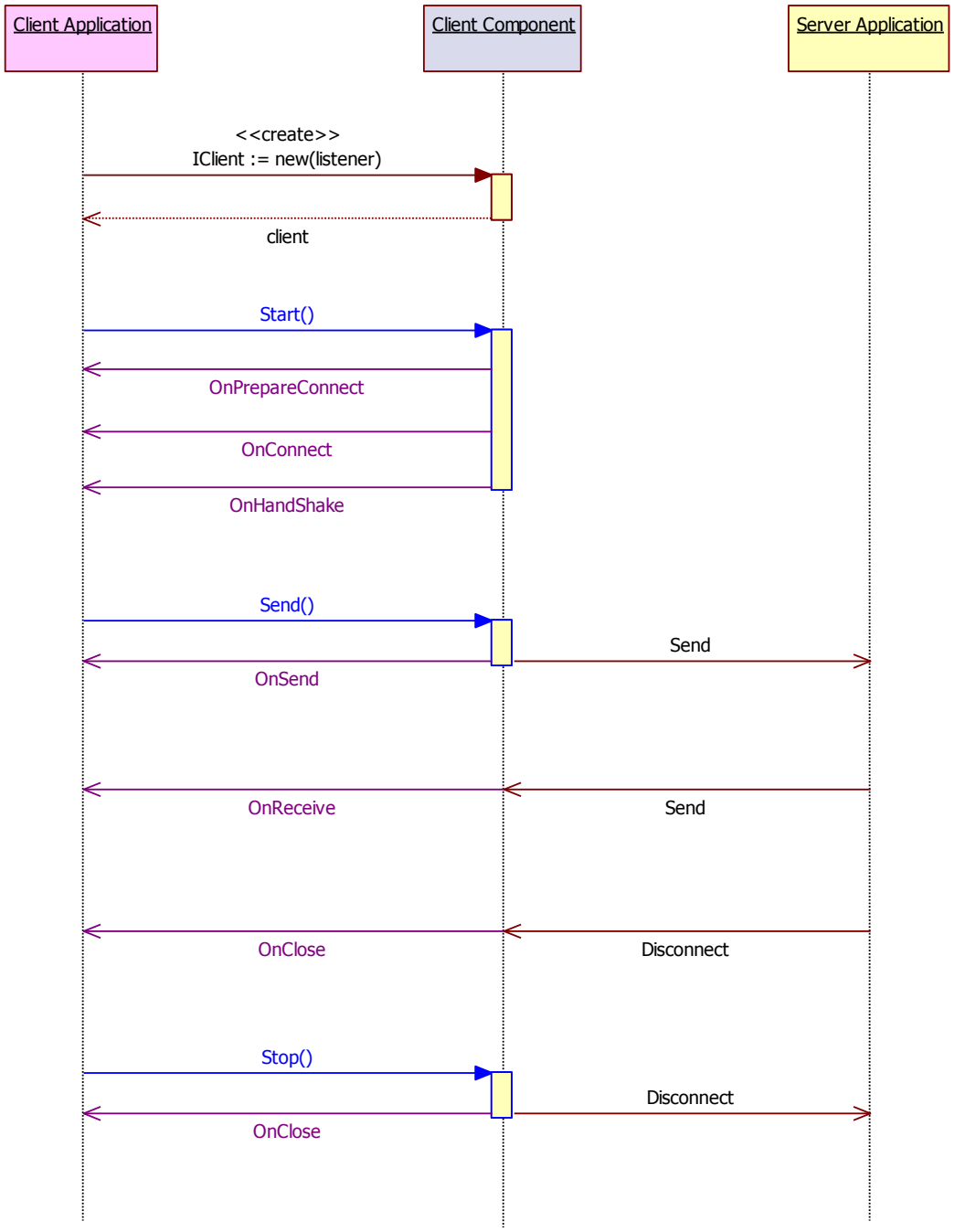


图 2.4.2-1 Client 工作流程

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*、*OnConnect* 和 *OnHandshake* 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* 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 SSL

3.1 组件接口

HP-Socket v3.5.x 版本开始，所有 TCP 组件全面支持 SSL。SSL 组件与对应的非 SSL 组件实现相同的接口，它们的使用方式也一致。表 3.1-1 列出了所有 SSL 组件的名称、接口、监听器接口、实现类及其分类：

Name	Component Intface Listener Intface	Implement Class	Role	Protocol
SSL Server	*ITcpServer* *ITcpServerListener*	CSSLServer	Server	PUSH
SSL Pull Server	*ITcpPullServer* *ITcpServerListener*	CSSLPullServer	Server	PULL
SSL Pack Server	*ITcpPackServer* *ITcpServerListener*	CSSLPackServer	Server	PACK
SSL Agent	*ITcpAgent* *ITcpServerListener*	CSSLAgent	Client	PUSH
SSL Pull Agent	*ITcpPullAgent* *ITcpAgentListener*	CSSLPullAgent	Client	PULL
SSL Pack Agent	*ITcpPackAgent* *ITcpServerListener*	CSSLPackAgent	Client	PACK
SSL Client	*ITcpClient* *ITcpClientListener*	CSSLClient	Client	PUSH
SSL Pull Client	*ITcpPullClient* *ITcpClientListener*	CSSLPullClient	Client	PULL
SSL Pack Client	*ITcpPackClient* *ITcpClientListener*	CSSLPackClient	Client	PACK

表 3.1-1 组件分类

各 SSL 组件的层次结构如图 3.1-1 所示，所有 SSL 组件都继承于对应的 TCP 组件：

- ***CSSLServer*** >> *CTcpServer*
- ***CSSLAgent*** >> *CTcpAgent*
- ***CSSLClient*** >> *CTcpClient*

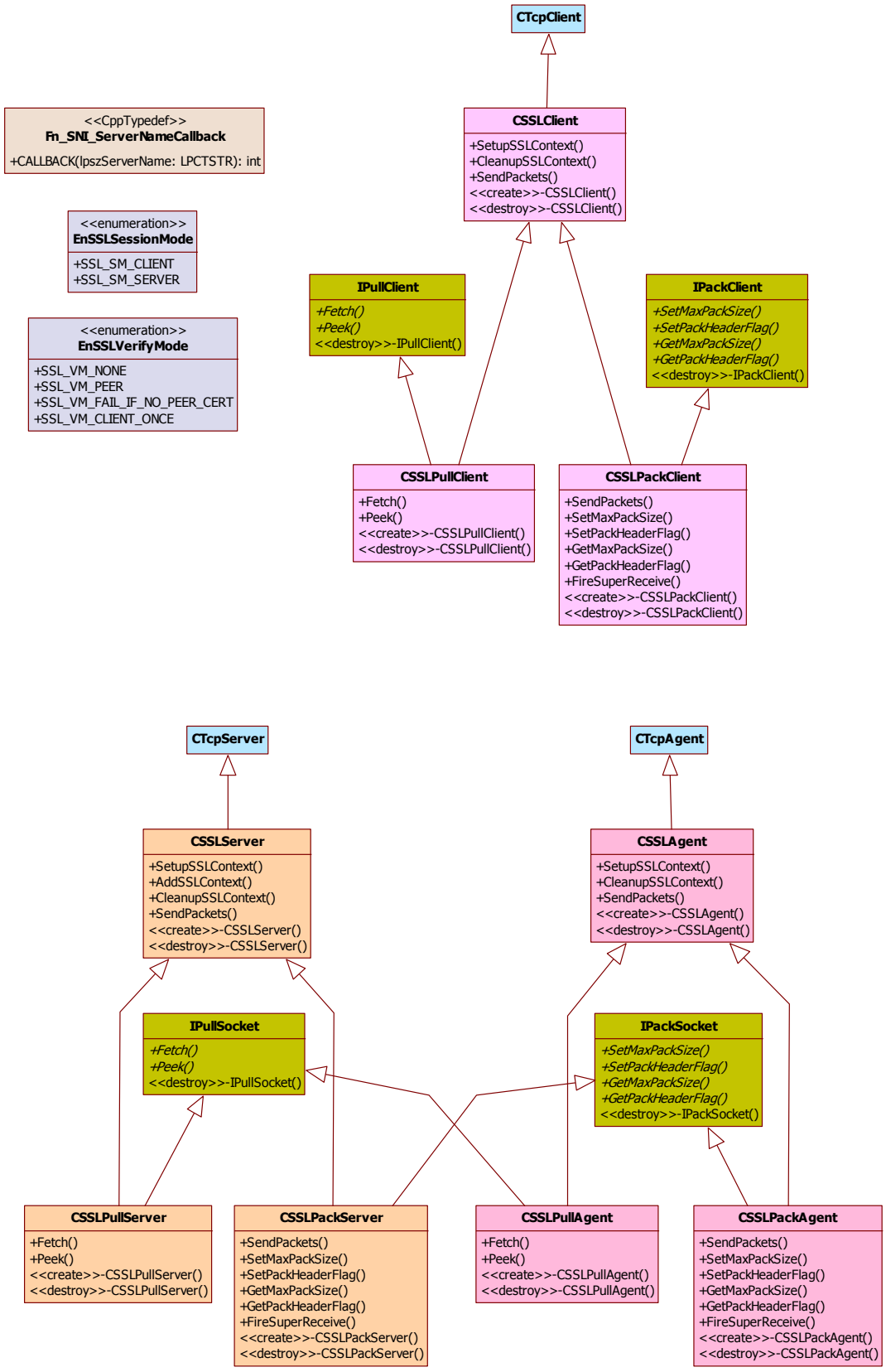

CTcpClient
<<CppTypedef>>
Fn_SNI_ServerNameCallback
+CALLBACK(lpszServerName: LPCTSTR): int
CSSLClient
+SetupSSLContext()
+CleanupSSLContext()
+SendPackets()
<<create>>-CSSLClient()
<<destroy>>-CSSLClient()
<<enumeration>>
EnSSLSessionMode
+SSL_SM_CLIENT
+SSL_SM_SERVER
IPullClient
+Fetch()
+Peek()
<<destroy>>-IPullClient()
IPackClient
+SetMaxPackSize()
+SetPackHeaderFlag()
+GetMaxPackSize()
+GetPackHeaderFlag()
<<destroy>>-IPackClient()
<<enumeration>>
EnSSLVerifyMode
+SSL_VM_NONE
+SSL_VM_PEER
+SSL_VM_FAIL_IF_NO_PEER_CERT
+SSL_VM_CLIENT_ONCE
CSSLPullClient
+Fetch()
+Peek()
<<create>>-CSSLPullClient()
<<destroy>>-CSSLPullClient()
CSSLPackClient
+SendPackets()
+SetMaxPackSize()
+SetPackHeaderFlag()
+GetMaxPackSize()
+GetPackHeaderFlag()
+FireSuperReceive()
<<create>>-CSSLPackClient()
<<destroy>>-CSSLPackClient()
CTcpServer
CTcpAgent
CSSLServer
+SetupSSLContext()
+AddSSLContext()
+CleanupSSLContext()
+SendPackets()
<<create>>-CSSLServer()
<<destroy>>-CSSLServer()
CSSLAgent
+SetupSSLContext()
+CleanupSSLContext()
+SendPackets()
<<create>>-CSSLAgent()
<<destroy>>-CSSLAgent()
IPullSocket
+Fetch()
+Peek()
<<destroy>>-IPullSocket()
IPackSocket
+SetMaxPackSize()
+SetPackHeaderFlag()
+GetMaxPackSize()
+GetPackHeaderFlag()
<<destroy>>-IPackSocket()
CSSLPullServer
+Fetch()
+Peek()
<<create>>-CSSLPullServer()
<<destroy>>-CSSLPullServer()
CSSLPackServer
+SendPackets()
+SetMaxPackSize()
+SetPackHeaderFlag()
+GetMaxPackSize()
+GetPackHeaderFlag()
+FireSuperReceive()
<<create>>-CSSLPackServer()
<<destroy>>-CSSLPackServer()
CSSLPullAgent
+Fetch()
+Peek()
<<create>>-CSSLPullAgent()
<<destroy>>-CSSLPullAgent()
CSSLPackAgent
+SendPackets()
+SetMaxPackSize()
+SetPackHeaderFlag()
+GetMaxPackSize()
+GetPackHeaderFlag()
+FireSuperReceive()
<<create>>-CSSLPackAgent()
<<destroy>>-CSSLPackAgent()

图 3.1-1 组件层次结构

3.2 SSL 运行环境

SSL 组件在启动通信前需要初始化 SSL 环境参数，通信完毕时需要清理 SSL 运行环境。HP-Socket v4.x.x 及其之前的版本使用全局唯一 SSL 运行环境，所有 SSL 组件都共享该环境，并通过 *HP_SSL_Initialize()* / *HP_SSL_AddServerContext()* / *HP_SSL_Cleanup()* / *HP_SSL_IsValid()* 等全局函数操作该全局 SSL 环境，HP-Socket v5.0.x 版本开始，每个 SSL 组件使用独立的 SSL 运行环境，因此上述函数已被删除。取而代之，SSL 组件提供相应的实例方法来操作自身的 SSL 运行环境：

✧ **初始化 SSL 环境参数：**

BOOL SetupSSLContext(enSessionMode, iVerifyMode = SSL_VM_NONE, lpszPemCertFile = nullptr, lpszPemKeyFile = nullptr, lpszKeyPasswod = nullptr, lpszCAPemCertFileOrPath = nullptr, [fnServerNameCallback = nullptr])

参数 *iVerifyMode* 指定 SSL 验证模式；参数 *lpszPemCertFile*、*lpszPemKeyFile*、*lpszKeyPasswod* 和 *lpszCAPemCertFileOrPath* 分别指定证书文件、私钥文件、私钥密码和 CA 证书文件/目录；参数 *fnServerNameCallback* 指定 SNI 回调函数指针，该参数只用于 HTTPS 服务端。参数详细说明请参考 *Src/HPSocket-SSL.h* 或 *Src/HPSocket4C-SSL.h* 头文件。初始化成功返回 TRUE，失败返回 FALSE，初始化失败可通过 *SYS_GetLastError()* 获取错误代码。

✧ **增加 SNI 主机证书（SSL Server 组件）：**

int AddSSLContext (iVerifyMode, lpszPemCertFile, lpszPemKeyFile, lpszKeyPasswod = nullptr, lpszCAPemCertFileOrPath = nullptr)

仅用于 SSL Server 组件。参数 *iVerifyMode* 指定 SSL 验证模式；参数 *lpszPemCertFile*、*lpszPemKeyFile*、*lpszKeyPasswod* 和 *lpszCAPemCertFileOrPath* 分别指定证书文件、私钥文件、私钥密码和 CA 证书文件/目录。执行成功返回 SNI 主机证书对应的索引，该索引用于在 SNI 回调函数中定位 SNI 主机；失败返回-1，可通过 *SYS_GetLastError()* 获取错误代码。

注意：应用程序一般在初始化 SSL 运行环境时调用 AddSSLContext() 加载所有 SNI 主机证书，但一些特殊应用可能要动态加载 SNI 主机证书，此时需要对调用 AddSSLContext() 的代码段进行同步处理，避免重复加载相同证书。

✧ **清理 SSL 环境：**

void CleanupSSLContext()

组件停止通信（调用 *Stop()*）时会自动清理 SSL 环境，因此，应用程序只需调用 *SetupSSLContext()* 初始化组件的 SSL 环境参数，而不需要手工调用本函数。

✧ **清理线程局部 SSL 环境资源（全局函数）：**

void HP_SSL_RemoveThreadLocalState()

任何一个操作 SSL 的线程，在退出时都需要清理线程的局部环境 SSL 资源，主线程和 HP-Socket 工作线程在通信结束时会自动清理线程局部环境 SSL 资源。因此，一般情况下不必手工调用本函数；特殊情况下，当自定义线程参与 HP-Socket 通信操作（如：通信组件发送策略为 SP_DIRECT 并且该自定义线程调用了 *Send()* 方法发送数据）并检查到 SSL 内存泄漏时，需在每次停止组件时在该自定义线程调用本函数。

3.3 SSL 握手

SSL 组件（包括 Https 组件）默认情况下在建立连接后会立刻开始 SSL 握手（*OnHandShake* 事件会紧接着 *OnConnect* / *OnAccept* 事件触发）。但在某些场景下需要在启动 SSL 通信前执行一些前置操作，例如：通过代理服务器与目标服务器通信就是其中一个典型场景。在此场景中，必须与代理服务器建立连接后才能开始 SSL 通信。

HP-Socket v5.4.2 版本开始，提供了对手工启动 SSL 握手的支持。调用 SSL 组件的 *SetSSLAutoHandShake(FALSE)* 方法把组件设置为手工握手模式，在执行完前置操作后调用 *StartSSLHandShake()* 启动 SSL 通信。

✧ **手工启动 SSL 握手：**

BOOL StartSSLHandShake(dwConnID)

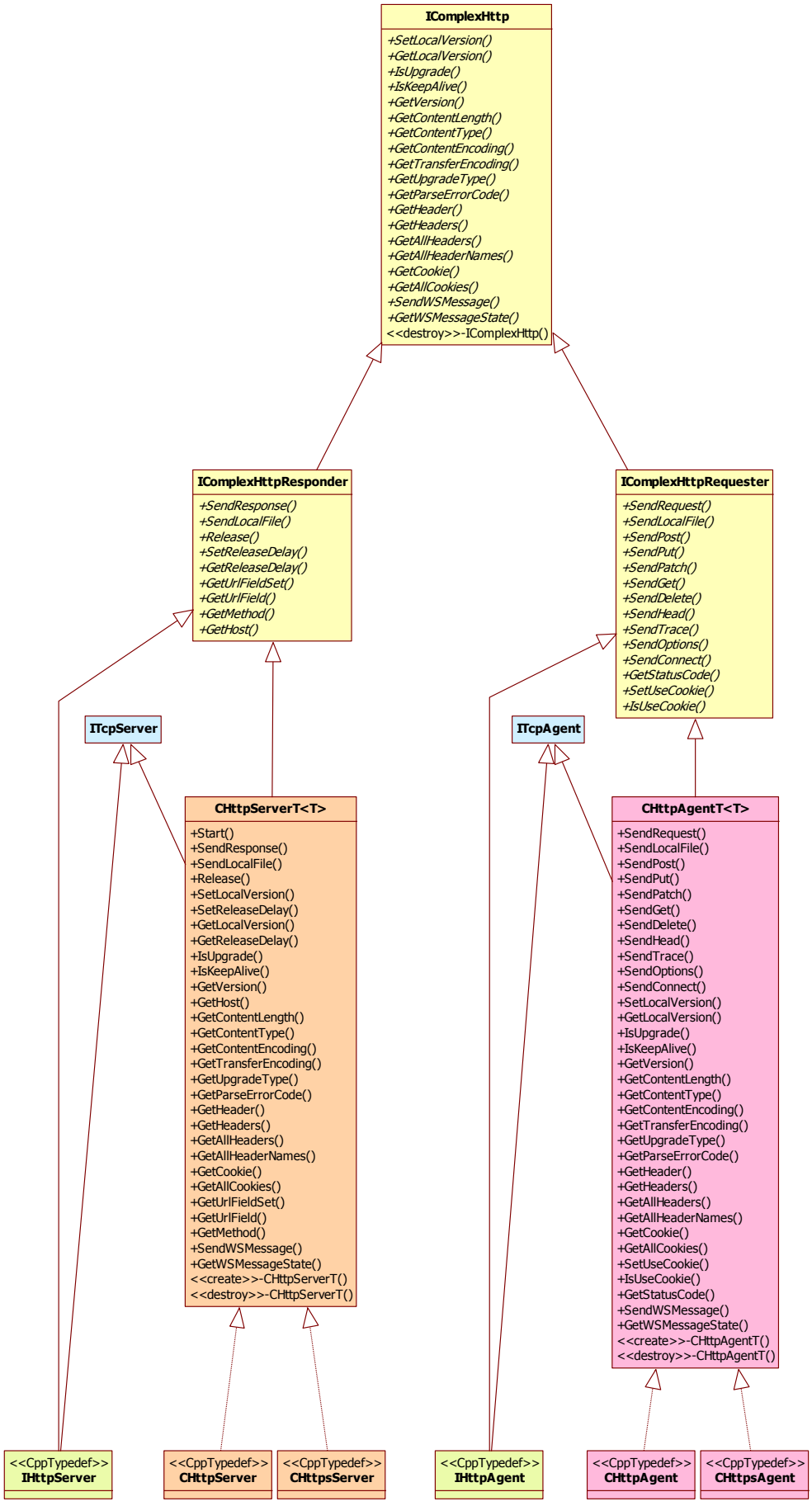
4 HTTP

4.1 组件接口

HP-Socket v4.0.x 版本开始加入 HTTP 组件。HTTP 组件继承于相应的 TCP 组件并增加 HTTP 相关操作方法； HTTP 组件监听器也继承于相应的 TCP 组件监听器并提供 HTTP 相关通信事件。表 4.1-1 列出了所有 HTTP 组件的名称、接口、监听器接口、实现类及 TCP 组件父类：

Name	Component Intface Listener Intface	Implement Class	Role	Base Class
Http Server	*IHttpServer* *IHttpServerListener*	CHttpServer	Server	CTcpServer
Https Server	*IHttpServer* *IHttpServerListener*	CHttpsServer	Server	CSSLServer
Http Agent	*IHttpAgent* *IHttpAgentListener*	CHttpAgent	Client	CTcpAgent
Https Agent	*IHttpAgent* *IHttpAgentListener*	CHttpsAgent	Client	CSSLAgent
Http Client	*IHttpClient* *IHttpClientListener*	CHttpClient	Client	CTcpClient
Https Client	*IHttpClient* *IHttpClientListener*	CHttpsClient	Client	CSSLClient
Http Sync Client	*IHttpSyncClient* *IHttpClientListener*	CHttpSyncClient	Client	CTcpClient
Https Sync Client	*IHttpSyncClient* *IHttpClientListener*	CHttpsSyncClient	Client	CSSLClient

表 4.1-1 组件分类


IComplexHttp
+SetLocalVersion()
+GetLocalVersion()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+SendWSMessage()
+GetWSMessageState()
<<destroy>>-IComplexHttp()
IComplexHttpResponder
+SendResponse()
+SendLocalFile()
+Release()
+SetReleaseDelay()
+GetReleaseDelay()
+GetUrlFieldSet()
+GetUrlField()
+GetMethod()
+GetHost()
IComplexHttpRequester
+SendRequest()
+SendLocalFile()
+SendPost()
+SendPut()
+SendPatch()
+SendGet()
+SendDelete()
+SendHead()
+SendTrace()
+SendOptions()
+SendConnect()
+GetStatusCode()
+SetUseCookie()
+IsUseCookie()
ITcpServer
ITcpAgent
CHttpServerT<T>
+Start()
+SendResponse()
+SendLocalFile()
+Release()
+SetLocalVersion()
+SetReleaseDelay()
+GetLocalVersion()
+GetReleaseDelay()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetHost()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+GetUrlFieldSet()
+GetUrlField()
+GetMethod()
+SendWSMessage()
+GetWSMessageState()
<<create>>-CHttpServerT()
<<destroy>>-CHttpServerT()
CHttpAgentT<T>
+SendRequest()
+SendLocalFile()
+SendPost()
+SendPut()
+SendPatch()
+SendGet()
+SendDelete()
+SendHead()
+SendTrace()
+SendOptions()
+SendConnect()
+SetLocalVersion()
+GetLocalVersion()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+SetUseCookie()
+IsUseCookie()
+GetStatusCode()
+SendWSMessage()
+GetWSMessageState()
<<create>>-CHttpAgentT()
<<destroy>>-CHttpAgentT()
<<CppTypedef>>
IHttpServer
<<CppTypedef>>
CHttpServer
<<CppTypedef>>
CHttpsServer
<<CppTypedef>>
IHttpAgent
<<CppTypedef>>
CHttpAgent
<<CppTypedef>>
CHttpsAgent

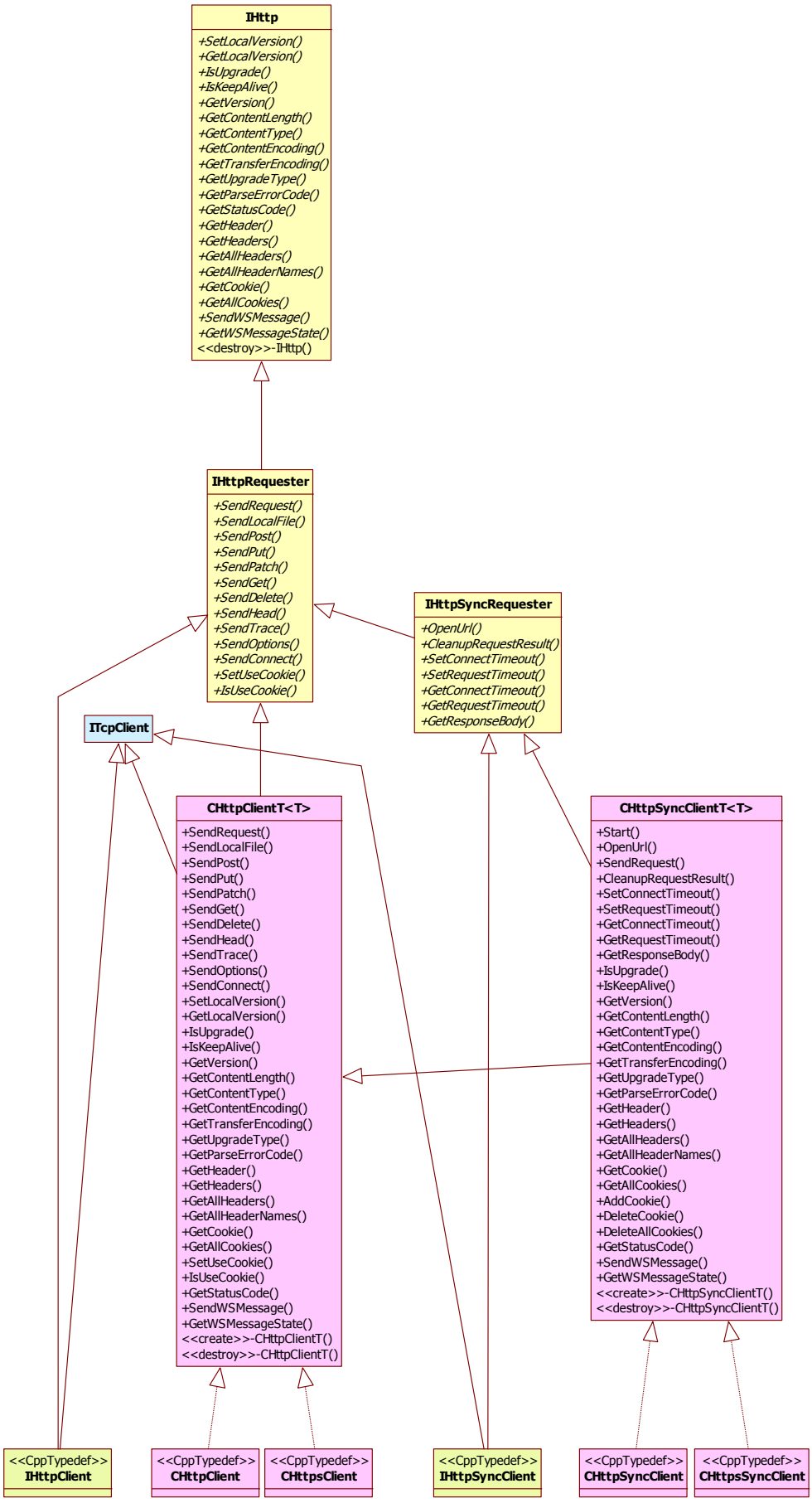

IHttp
+SetLocalVersion()
+GetLocalVersion()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetStatusCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+SendWSMessage()
+GetWSMessageState()
<<destroy>>-IHttp()
IHttpRequester
+SendRequest()
+SendLocalFile()
+SendPost()
+SendPut()
+SendPatch()
+SendGet()
+SendDelete()
+SendHead()
+SendTrace()
+SendOptions()
+SendConnect()
+SetUseCookie()
+IsUseCookie()
IHttpSyncRequester
+OpenUrl()
+CleanupRequestResult()
+SetConnectTimeout()
+SetRequestTimeout()
+GetConnectTimeout()
+GetRequestTimeout()
+GetResponseBody()
ITcpClient
CHttpClientT<T>
+SendRequest()
+SendLocalFile()
+SendPost()
+SendPut()
+SendPatch()
+SendGet()
+SendDelete()
+SendHead()
+SendTrace()
+SendOptions()
+SendConnect()
+SetLocalVersion()
+GetLocalVersion()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+SetUseCookie()
+IsUseCookie()
+GetStatusCode()
+SendWSMessage()
+GetWSMessageState()
<<create>>-CHttpClientT()
<<destroy>>-CHttpClientT()
CHttpSyncClientT<T>
+Start()
+OpenUrl()
+SendRequest()
+CleanupRequestResult()
+SetConnectTimeout()
+SetRequestTimeout()
+GetConnectTimeout()
+GetRequestTimeout()
+GetResponseBody()
+IsUpgrade()
+IsKeepAlive()
+GetVersion()
+GetContentLength()
+GetContentType()
+GetContentEncoding()
+GetTransferEncoding()
+GetUpgradeType()
+GetParseErrorCode()
+GetHeader()
+GetHeaders()
+GetAllHeaders()
+GetAllHeaderNames()
+GetCookie()
+GetAllCookies()
+AddCookie()
+DeleteCookie()
+DeleteAllCookies()
+GetStatusCode()
+SendWSMessage()
+GetWSMessageState()
<<create>>-CHttpSyncClientT()
<<destroy>>-CHttpSyncClientT()
<<CppTypedef>> IHttpClient
<<CppTypedef>> CHttpClient
<<CppTypedef>> CHttpsClient
<<CppTypedef>> IHttpSyncClient
<<CppTypedef>> CHttpSyncClient
<<CppTypedef>> CHttpsSyncClient

图 4.1-1 HTTP 组件层次结构

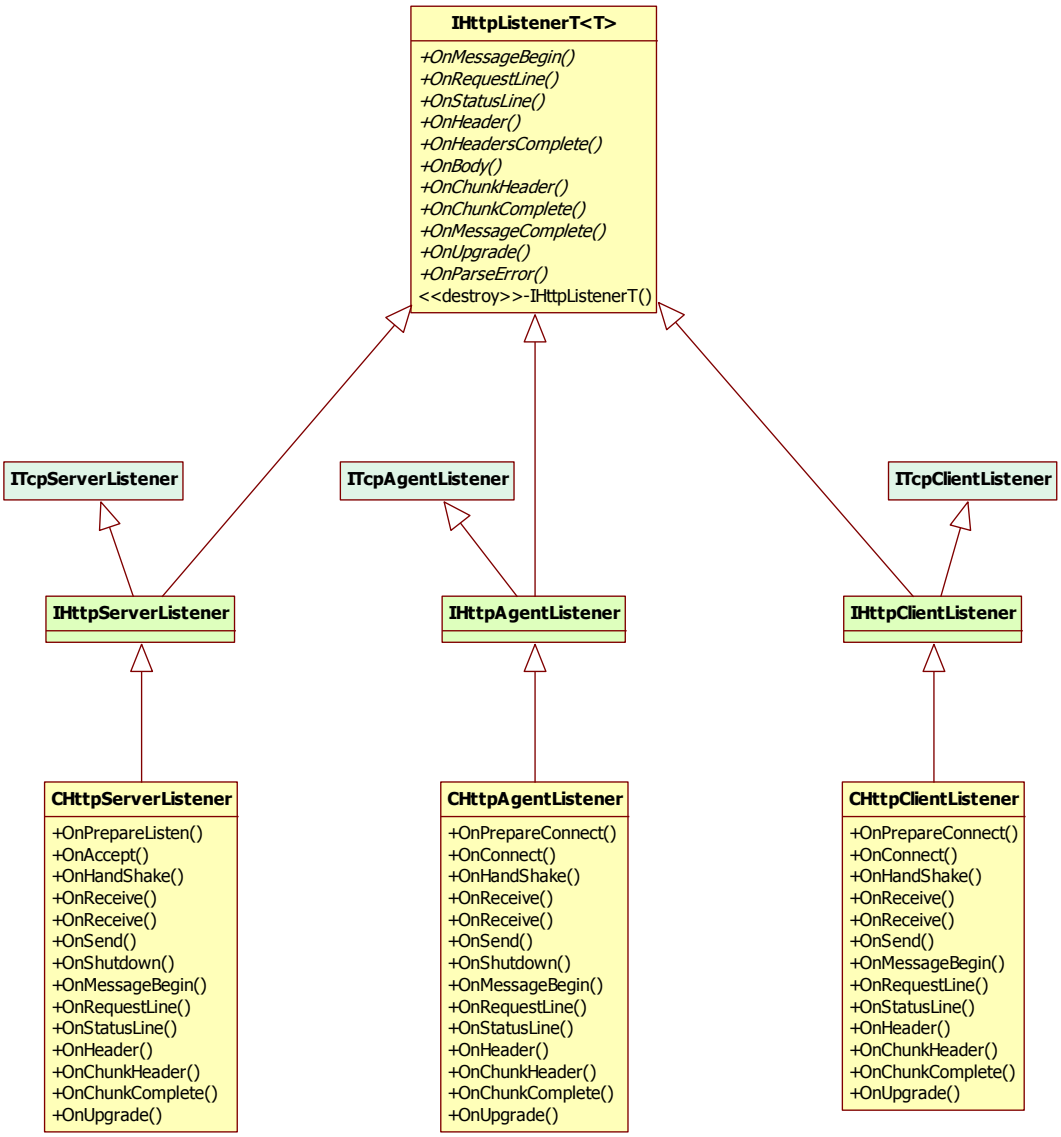
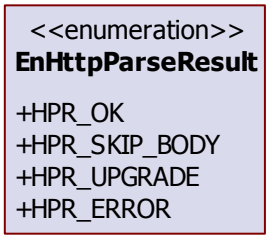


图 4.1-2 HTTP 组件监听器层次结构

4.2 HTTP 监听器事件

HTTP 组件监听器接口继承于 IHttpListenerT 和相应的 TCP 组件监听器接口。IHttpListenerT 的 HTTP 事件可以理解为 TCP 组件监听器 *OnReceive* 事件的分解。HTTP 事件返回值的类型为 EnHttpParseResult：

-



- ✓ **HPR_OK** ：解析成功，继续执行
- ✓ **HPR_SKIP_BODY** ：跳过当前请求 BODY，完成本次请求
  （仅用于 *OnHeadersComplete* 事件）
- ✓ **HPR_UPGRADE** ：升级协议，完成本次请求，并且不再进行后续 HTTP 解析
  （仅用于 *OnHeadersComplete* 事件）
- ✓ **HPR_ERROR** ：解析错误，终止解析，断开连接

✧ **开始解析事件：**

EnHttpParseResult OnMessageBegin(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID

✧ **请求行解析完成事件*（仅用于 HTTP 服务端）*：**

EnHttpParseResult OnRequestLine(pSender, dwConnID, lpszMethod, lpszUrl)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *lpszMethod* -- 请求方法名
- ✓ *lpszUrl* -- 请求行中的 URL 域

✧ **状态行解析完成事件*（仅用于 HTTP 客户端）*：**

EnHttpParseResult OnStatusLine(pSender, dwConnID, usStatusCode, lpszDesc)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *usStatusCode* -- HTTP 状态码
- ✓ *lpszDesc* -- 状态描述

✧ **请求头事件：**

EnHttpParseResult OnHeader(pSender, dwConnID, lpszName, lpszValue)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *lpszName* -- 请求头名称
- ✓ *lpszValue* -- 请求头值

✧ **请求头完成事件：**

EnHttpParseResult OnHeadersComplete(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID

✧ **BODY 报文事件：**

EnHttpParseResult OnBody(pSender, dwConnID, pData, iLength)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *pData* -- 数据缓冲区
- ✓ *iLength* -- 数据长度

✧ **Chunked 报文头事件：**

EnHttpParseResult OnChunkHeader(pSender, dwConnID, iLength)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *iLength* -- Chunked 报文体数据长度

✧ **Chunked 报文结束事件：**

EnHttpParseResult OnChunkComplete(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID

✧ **完成解析事件：**

EnHttpParseResult OnMessageComplete(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID

✧ **升级协议事件：**

EnHttpParseResult OnUpgrade(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *enUpgradeType* -- 协议类型

✧ **解析错误事件：**

EnHttpParseResult OnParseError(pSender, dwConnID, iErrorCode, lpszErrorDesc)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *iErrorCode* -- 错误代码
- ✓ *lpszErrorDesc* -- 错误描述

✧ **Web Socket 数据包头事件：**

EnHandleResult OnWSMessageHeader(pSender, dwConnID, bFinal, iReserved, iOperationCode, lpszMask, ullBodyLen)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *bFinal* -- 是否结束帧
- ✓ *iReserved* -- RSV1/RSV2/RSV3 各 1 位
- ✓ *iOperationCode* -- 操作码：0x0 - 0xF
- ✓ *lpszMask* -- 掩码（nullptr 或 4 字节掩码，如果为 nullptr 则没有掩码）
- ✓ *ullBodyLen* -- 消息体长度

✧ **Web Socket 数据包体事件：**

EnHandleResult OnWSMessageBody(pSender, dwConnID, pData, iLength)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID
- ✓ *pData* -- 消息体数据缓冲区
- ✓ *iLength* -- 消息体数据长度

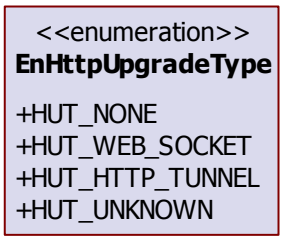
✧ **Web Socket 数据包完成事件：**

EnHandleResult OnWSMessageComplete(pSender, dwConnID)

- ✓ *pSender* -- 事件源对象
- ✓ *dwConnID* -- 连接 ID

OnHeadersComplete、*OnBody*、*OnMessageComplete* 和 *OnParseError* 是最基本的 4 个 HTTP 事件，所有应用程序都要对这些事件进行处理，其他事件则根据应用程序的实际情况进行处理。例如：如果应用程序不用处理 Chunked 报文则可忽略 *OnChunkHeader* 和 *OnChunkComplete* 事件；如果应用程序不会进行协议升级则可忽略 *OnUpgrade* 事件。

HP-Socket 支持 Web Socket 和 Http Tunnel 协议升级，协议升级时会触发 *OnUpgrade* 事件，该事件的 *enUpgradeType* 参数指示升级类型：



- ✓ **HUT_WEB_SOCKET** ：Web Socket
- ✓ **HUT_HTTP_TUNNEL** ：HTTP Tunnel

当完成协议升级后不再进行 HTTP 解析，也不会触发常规 HTTP 事件。如果升级类型为 Web Socket，触发 *OnWSMessageHeader*、*OnWSMessageBody* 和 *OnWSMessageComplete* 事件处理 Web Socket 数据；如果升级类型为 HTTP Tunnel，后续接收的所有数据都被看作为 TCP 数据，触发 *OnReceive* 事件。

注意：

Sync Client：同步 HTTP 客户端组件（CHttpSyncClient 和 CHttpsSyncClient）内部会处理所有事件，因此，它们不需要绑定监听器（构造方法的监听器参数传入 null）；如果绑定了监听器则可以跟踪组件的通信过程。

4.3 Cookie 管理

HP-Socket v4.2.x 版本开始，为 HTTP 客户端组件（HttpClient、HttpAgent）提供进程级别的 Cookie 管理器，管理器实现了标准 HTTP Cookie 功能，支持 *Max-Age*、*expires*、*httpOnly*、*secure* 等。

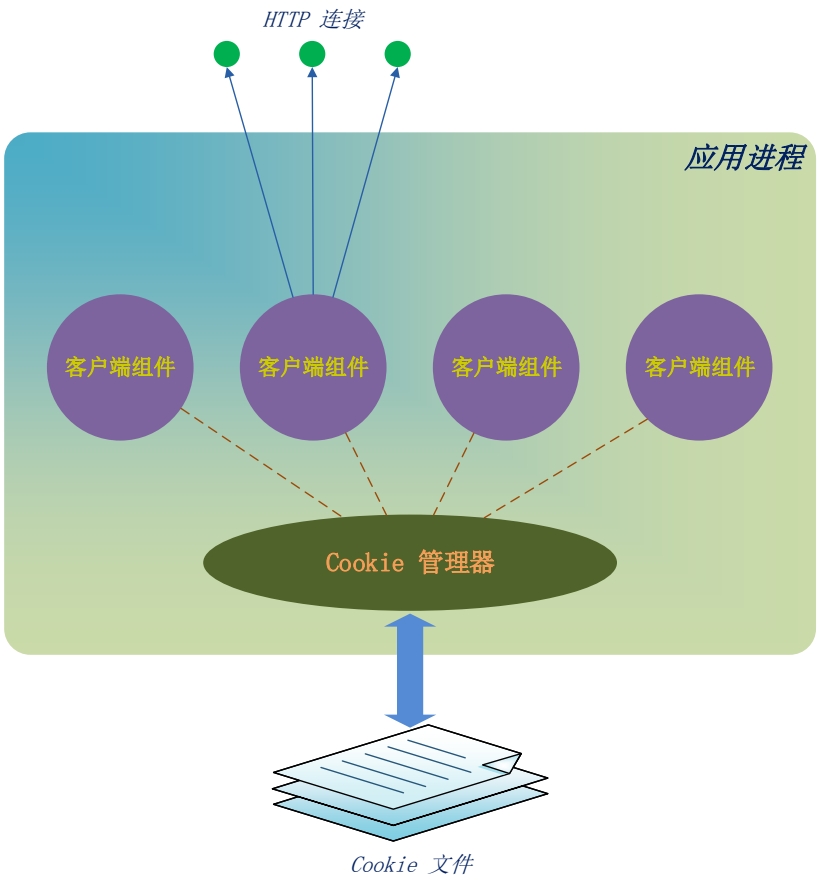


图 4.3-1 Cookie 管理器

如图 4.3-1 所示，管理器在内存中维护所有 HTTP 客户端组件产生的 Cookie，可在不同连接、不同组件对象间共享 Cookie，并支持 Cookie 序列化与反序列化。

客户端组件默认设置为使用 Cookie，只要客户端组件设置为使用 Cookie，它接收到的 Cookie 会自动存入管理器；当发送 HTTP 请求时会自动从管理器中加载 Cookie。

如果组件设置为不使用 Cookie（设置方法：***SetUseCookie(FALSE)***），它接收到的 Cookie 不会被解析，不会存入管理器；发送 HTTP 请求时也不会从管理器中加载 Cookie。

注意：管理器内部使用读写锁控制 Cookie 存取，高并发下会带来一些性能损失。如果确定组件不会用到 Cookie，可设置为不使用 Cookie。

HP-Socket 提供一些管理函数操作管理器，也提供 Cookie 辅助函数便于应用程序处理 Cookie：

✧ **从文件加载 Cookie：**

BOOL HP_HttpCookie_MGR_LoadFromFile(lpszFile, bKeepExists)

- ✓ *lpszFile* -- 文件
- ✓ *bKeepExists* -- 是否保留管理器中原有的 Cookie

✧ **保存 Cookie 到文件：**

BOOL HP_HttpCookie_MGR_SaveToFile(lpszFile, bKeepExists)

- ✓ *lpszFile* -- 文件
- ✓ *bKeepExists* -- 是否保留文件中原有的 Cookie

✧ **清理 Cookie：**

BOOL HP_HttpCookie_MGR_ClearCookies(lpszDomain, lpszPath)

- ✓ *lpszDomain* -- 域，为空则表示所有域
- ✓ *lpszPath* -- 路径，为空则表示所有路径

✧ **清理过期 Cookie：**

BOOL HP_HttpCookie_MGR_RemoveExpiredCookies(lpszDomain, lpszPath)

- ✓ *lpszDomain* -- 域，为空则表示所有域
- ✓ *lpszPath* -- 路径，为空则表示所有路径

✧ **设置 Cookie：**

BOOL HP_HttpCookie_MGR_SetCookie(lpszName, lpszValue, lpszDomain, lpszPath, iMaxAge, bHttpOnly, bSecure, enSameSite, bOnlyUpdateValueIfExists)

- ✓ *lpszName* -- 名称
- ✓ *lpszValue* -- 值
- ✓ *lpszDomain* -- 域
- ✓ *lpszPath* -- 路径
- ✓ *iMaxAge* -- 生命周期：> 0 –> 存活秒数；= 0 –> 立刻删除，< 0 –> 到应用程序结束
- ✓ *bHttpOnly* -- 是否有 HttpOnly 属性
- ✓ *bSecure* -- 是否有 secure 属性
- ✓ *enSameSite* -- SameSite 属性：0 –> 无；1 –> Strict；2 –> LAX
- ✓ *bOnleUpdateValueIfExists* -- 如果 Cookie 已存在是否只更新 Cookie 值

✧ **删除 Cookie：**

BOOL HP_HttpCookie_MGR_DeleteCookie(lpszDomain, lpszPath, lpszName)

- ✓ *lpszDomain* -- 域
- ✓ *lpszPath* -- 路径
- ✓ *lpszName* -- 名称

✧ **设置是否允许第三方 Cookie：**

void HP_HttpCookie_MGR_SetEnableThirdPartyCookie(bEnableThirdPartyCookie)

- ✓ *bEnableThirdPartyCookie* -- TRUE –> 允许；FALSE –> 禁止

✧ **检查是否允许第三方 Cookie：**

BOOL HP_HttpCookie_MGR_IsEnableThirdPartyCookie()

- ✓

✧ **Cookie expires 字符串转换为整数：**

BOOL HP_HttpCookie_HLP_ParseExpires(lpszExpires, ptmExpires)

- ✓ *lpszExpires* -- expires 字符串
- ✓ *ptmExpires* -- expires 整数指针

✧ **整数转换为 Cookie expires 字符串：**

BOOL HP_HttpCookie_HLP_MakeExpiresStr(lpszBuff, piBuffLen, tmExpires)

- ✓ *lpszBuff* -- 字符串缓冲区
- ✓ *piBuffLen* -- 缓冲区长度
- ✓ *tmExpires* -- expires 整数

✧ **生成 Cookie 字符串：**

BOOL HP_HttpCookie_HLP_ToString(lpszBuff, piBuffLen, lpszName, lpszValue, lpszDomain, lpszPath, iMaxAge, bHttpOnly, bSecure, enSameSite)

- ✓ *lpszBuff* -- 字符串缓冲区
- ✓ *piBuffLen* -- 缓冲区长度
- ✓ *lpszName* -- 名称
- ✓ *lpszValue* -- 值
- ✓ *lpszDomain* -- 域
- ✓ *lpszPath* -- 路径
- ✓ *iMaxAge* -- 生命周期：> 0 –> 存活秒数；= 0 –> 立刻删除，< 0 –> 到应用程序结束
- ✓ *bHttpOnly* -- 是否有 HttpOnly 属性
- ✓ *bSecure* -- 是否有 secure 属性
- ✓ *enSameSite* -- SameSite 属性：0 –> 无；1 –> Strict；2 –> LAX

✧ **获取当前 UTC 时间：**

__time64_t HP_HttpCookie_HLP_CurrentUTCTime()

✓

✧ **Max-Age -> expires：**

__time64_t HP_HttpCookie_HLP_MaxAgeToExpires(iMaxAge)

✓ *iMaxAge* -- 生命周期：> 0 –> 存活秒数；= 0 –> 立刻删除，< 0 –> 到应用程序结束

✧ **expires -> Max-Age：**

int HP_HttpCookie_HLP_ExpiresToMaxAge(tmExpires)

✓ *tmExpires* -- expires 整数

4.4 启动 HTTP 通讯

Http 组件（包括 Https 组件）默认情况下在建立连接后会立刻开始 HTTP 通信（*OnReceive* 事件会分解为一系列 HTTP 通信事件）。但在某些场景下需要在启动 HTTP 通信前执行一些前置操作，例如：通过 SOCKS 代理服务器与目标服务器通信就是其中一个典型场景。在此场景中，必须与 SOCKS 代理服务器建立连接后才能开始 HTTP 通信。

HP-Socket v5.4.3 版本开始，提供了对手工启动 HTTP 通信的支持。调用 HTTP 组件的 *SetHttpAutoStart(FALSE)* 方法把组件设置为手工启动 HTTP 通信模式，在执行完前置操作后调用 *StartHttp()* 启动 HTTP 通信。

✧ **手工启动 HTTP 通信：**

BOOL StartHttp(dwConnID)

5 ARQ UDP

HP-Socket v5.5.x 版本开始，提供 ARQ UDP（自动重传请求 UDP、可靠 UDP）组件：IUdpArqServer 和 IUdpArqClient。在低带宽、高延时或丢包严重的网络环境中，可以提供比 TCP 更高效的网络传输性能。

5.1 组件接口

ARQ UDP 组件 IUdpArqServer / IUdpArqClient 的操作方法和监听器接口与常规 UDP 组件 IUdpServer / IUdpClient 一致，只在常规 UDP 组件的基础上增加了若干与 ARQ 通信相关的参数设置、获取方法。

Name	Component Intface Listener Intface	Implement Class	Role	Base Class
UDP Server	*IUdpServer* *IUdpServerListener*	CUdpServer	Server	
UDP Client	*IUdpClient* *IUdpClientListener*	CUdpClient	Client	
UDP Cast	*IUdpCast* *IUdpCastListener*	CUdpCast	Client	
UDP ARQ Server	*IUdpArqServer* *IUdpServerListener*	CUdpArqServer	Server	CUdpServer
UDP ARQ Client	*IUdpArqClient* *IUdpClientListener*	CUdpArqClient	Client	CUdpClient

表 5.1-1 UDP 组件分类

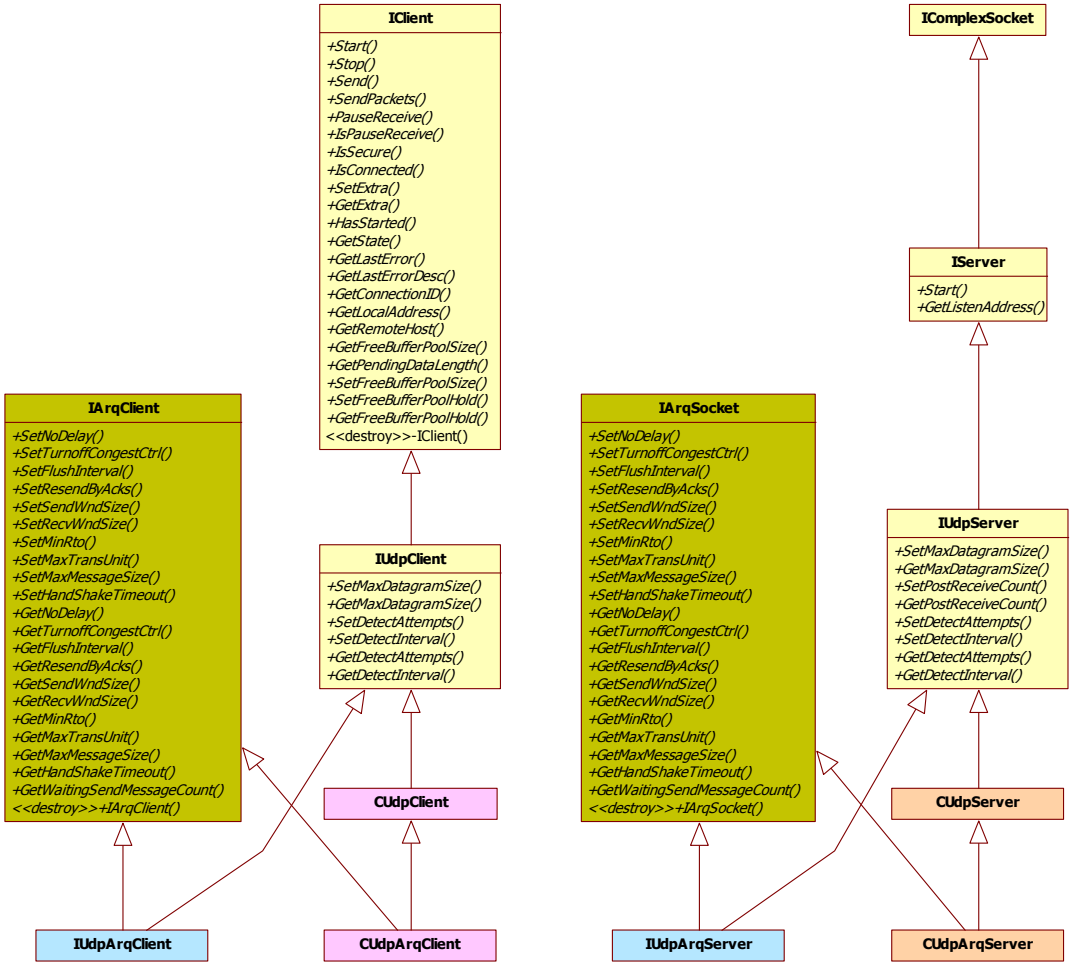


图 5.1-1 UDP 组件层次结构

5.2 握手协议

HP-Socket 使用 KCP 实现 ARQ 通信，但通信协商方式与常规 KCP 不同，常规 KCP 需要通信双方使用相同的会话 ID，HP-Socket 使用自身的通信握手协议实现通信协商。通信双方可以使用各自独立的会话 ID。

如图 5.2-1 所示，HP-Socket 的 ARQ 组件在开始通信之前需要进行通信握手。当 IUdpArqServer 组件接收到 *OnAccept* 事件或者 IUdpArqClient 组件接收到 *OnConnect* 事件时即开始 ARQ 握手，握手方式为定时向对方发送握手报文并处理对方发过来的握手报文，直至握手完成或超时终止。握手完成会接收到 *OnHandshake* 事件；握手超时会接收到 *OnClose* 事件，错误代码为 *ERROR_TIMEOUT*。

协商报文格式：

MM C F XXXX YYYY

- ✓ 报文长度 ：12 字节

- ✓ 第 1-2 字节 ：固定魔数 0xBB4F
- ✓ 第 3 字节 ：命令类型，目前只使用了握手命令 0x01
- ✓ 第 4 字节 ：命令标识，0x00 - 未完成，0x01 - 已完成
- ✓ 第 5-8 字节 ：本方会话 ID
- ✓ 第 9-12 字节 ：对方会话 ID，如未收到对方会话 ID 则用 0x00000000 填充

注意：IUdpArqServer 通过客户端“IP 地址 + 端口 + 会话 ID”标识一个远程连接，在客户端断开又重连的情形下，服务端可能未收到前一个连接的断开通知。因此客户端需要确保前后两个连接的“IP 地址 + 端口 + 会话 ID”不会完全相同。最简单的做法是让前后两个连接使用不同的会话 ID。

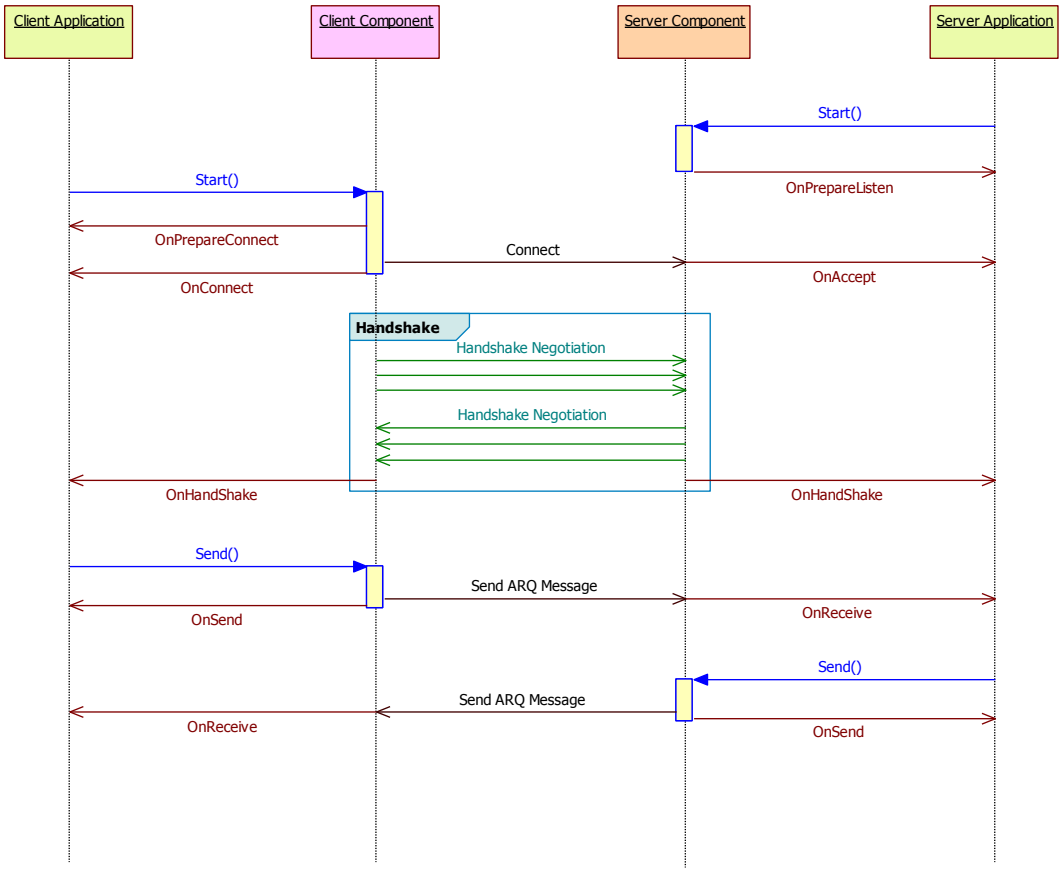


图 5.2-1 ARQ 通信过程

5.3 配置参数

- ✧ ***SetNoDelay(BOOL)***
  是否开启 nodelay 模式（默认：FALSE，不开启）
- ✧ ***SetTurnoffCongestCtrl(BOOL)***
  是否关闭拥塞控制（默认：FALSE，不关闭）
- ✧ ***SetFlushInterval(DWORD)***
  数据刷新间隔（毫秒，默认：60）

- ***SetResendByAcks(DWORD)***
  快速重传 ACK 跨越次数（默认：0，关闭快速重传）
- ***SetSendWndSize(DWORD)***
  发送窗口大小（数据包数量，默认：128）
- ***SetRecvWndSize(DWORD)***
  接收窗口大小（数据包数量，默认：512）
- ***SetMinRto(DWORD)***
  最小重传超时时间（毫秒，默认：30）
- ***SetMaxTransUnit(DWORD)***
  最大传输单元（默认：0，与 UDP 参数 SetMaxDatagramSize(DWORD) 一致）
- ***SetMaxMessageSize(DWORD)***
  最大数据包大小（默认：4096）
- ***SetHandShakeTimeout(DWORD)***
  握手超时时间（毫秒，默认：5000）

除 *SetMaxMessageSize(DWORD)*和 *SetHandShakeTimeout(DWORD)*之外，其他配置参数均与 KCP 相关，请参考 KCP 对相关参数的说明。

6 线程池

6.1 组件接口

HP-Socket v5.4.x 版本开始，提供线程池组件 IHPThreadPool，协助用户实现通信逻辑与业务逻辑分离，提高应用程序的整体执行效率。IHPThreadPool 提供以下主要操作方法。这些方法成功返回 TRUE，失败返回 FALSE，失败可通过 *SYS_GetLastError()* 获取系统错误代码。

✧ 启动线程池

BOOL Start(dwThreadCount = 0, dwMaxQueueSize = 0, enRejectedPolicy = TRP_CALL_FAIL, dwStackSize = 0)

- *dwThreadCount* 线程数量，（默认：0）
  - >0 ： *dwThreadCount*
  - =0 ： (CPU 核数 * 2 + 2)
  - <0 ： (CPU 核数 * (-*dwThreadCount*))
- *dwMaxQueueSize* 任务队列最大容量（默认：0，不限制）
- *enRejectedPolicy* 任务拒绝处理策略
  - TRP_CALL_FAIL ：（默认）立刻返回失败
  - TRP_WAIT_FOR ：等待（直到成功、超时或线程池关闭等原因导致失败）
  - TRP_CALLER_RUN ：调用者线程直接执行
- *dwStackSize* 线程堆栈空间大小（默认：0 -> 操作系统默认）

✧ 关闭线程池

在规定时间内关闭线程池组件，如果工作线程在最大等待时间内未能正常关闭，会尝试强制关闭，这种情况下很可能会造成系统资源泄漏。

BOOL Stop(dwMaxWait = INFINITE)

- *dwMaxWait* 最大等待时间（毫秒，默认：INFINITE，一直等待）

✧ 提交任务

BOOL Submit(fnTaskProc, pvArg, dwMaxWait = INFINITE)

- *fnTaskProc* 任务处理函数
- *pvArg* 任务参数
- *dwMaxWait* 任务提交最大等待时间（毫秒，仅对 TRP_WAIT_FOR 类型线程池生效，默认：INFINITE，一直等待）。

** *SYS_GetLastError()* 错误码 *ERROR_DESTINATION_ELEMENT_FULL* 表示任务队列已满。

✧ 提交 Socket 任务

BOOL Submit(pTask, dwMaxWait = INFINITE)

- *pTask* 任务参数
- *dwMaxWait* 任务提交最大等待时间（毫秒，仅对 TRP_WAIT_FOR 类型线程池生效，默认：INFINITE，一直等待）
  ** *SYS_GetLastError()* 错误码 *ERROR_DESTINATION_ELEMENT_FULL* 表示任务队列已满。

注意：pTask 由 HP_Create_SocketTaskObj( )函数创建，当 Submit(pTask)提交成功时线程池负责自动销毁 pTask 对象；当提交失败时应用程序需要手工调用 HP_Destroy_SocketTaskObj( )销毁 pTask 对象。

✧ 动态调整线程池大小

BOOL AdjustThreadCount(dwNewThreadCount)

- *dwNewThreadCount* 线程数量
  >0 ： *dwNewThreadCount*
  =0 ： (CPU 核数 * 2 + 2)
  <0 ： (CPU 核数 * (-*dwNewThreadCount*))

7 Linux

HP-Socket for Linux 提供的 API 接口与 Windows 版本一致，但实现代码则完全独立。HP-Socket for Linux 使用了 C++14 标准的新特性，需要 GCC 5.x 以上版本的编译器进行编译。

注意：HP-Socket for Linux 编译运行要求：

1) ***Linux 内核版本：2.6.32 及以上***
2) ***GCC 版本：5.x 及以上***
3) ***依赖库：librt、libdl、libpthread***

7.1 编译

HP-Socket 发布包已提供 x86 和 x64 平台下的编译好的二进制库文件和示例 Demo 可执行文件（*编译环境：Linux 2.6.32、GCC 6.3.1*）。另外，也可以通过以下两种方式使用 HP-Socket 源码自行编译：

1、 ***Visual C++ for Linux Development 插件：***HP-Socket 发行包中提供了 HP-Socket 及其示例 Demo 的 Visual Studio 项目工程，安装配置好 *Visual C++ for Linux Development* 插件后即可在 Visual Studio 中编译 HP-Socket。

2、 ***compile.sh：***可以使用 HP-Socket 发行包中提供的 *compile.sh* 编译脚本编译 HP-Socket 库文件。注：*compile.sh* 只编译 HP-Socket 库，不会编译示例 Demo。

```
$ ./compile.sh -h
Usage: compile.sh [...O.P.T.I.O.N.S...]
----------------------+-----------------------------------------------------
  -d|--with-debug-lib : compile debug libs (default: true)
  -j|--use-jemalloc   : use jemalloc in release libs
                      : (x86/x64 default: true, ARM default: false)
  -u|--udp-enabled    : enable UDP components (default: true)
  -t|--http-enabled   : enable HTTP components (default: true)
  -s|--ssl-enabled    : enable SSL components (default: true)
  -z|--zlib-enabled   : enable ZLIB related functions (default: true)
  -i|--iconv-enabled  : enable ICONV related functions (default: true)
  -c|--compiler       : compiler (default: g++)
  -e|--clean          : clean compilation intermediate temp files
  -r|--remove         : remove all compilation target files
  -v|--version        : print hp-socket version
  -h|--help           : print this usage message
----------------------+-----------------------------------------------------
```

图 7.1-1 compile.sh 编译脚本

7.2 安装

发布包提供的 *install.sh* 脚本用于安装（或卸载）HP-Socket，它会安装（或卸载）当前平台的库文件进行。*install.sh* 脚本支持以下参数：

```
$ sudo ./install.sh -h
Usage: install.sh [...O.P.T.I.O.N.S...]
-----------------+-----------------------------------------------------
  -p|--prefix    : install/uninstall path (default: /usr/local)
  -l|--libdir    : lib dir (x86/ARM default: 'lib', x64 default: 'lib64')
  -d|--with-demo : install demos or not (default: false)
  -u|--uninstall : execute uninstall operation from install path
  -v|--version   : print hp-socket version
  -h|--help      : print this usage message
-----------------+-----------------------------------------------------
```

图 7.2-1 install.sh 安装脚本

注意：compile.sh 编译脚本依赖发行包的 script/、src/、include/、dependent/ 目录；install.sh 安装脚本依赖发行包的 script/、include/、lib/ 目录，如果要安装示例 Demo 的执行文件，还需要依赖 demo/Release/ 目录。

7.3 Android NDK

HP-Socket 提供了 Android NDK 库文件构建脚本 *build-android-ndk.sh*（*Windows 平台：build-android-ndk.bat*）,安装配置好 NDK 后执行 *build-android-ndk.sh* 命令进行库文件构建，构建脚本默认会构建当前 NDK 支持的所有 ABI 的动态库和静态库，并把构建目标库文件输出到 *lib/android-ndk/* 目录。如有特殊需要请为构建脚本提供相应的命令行参数。

（执行默认构建）

```
$ cd HP-Socket/Linux
$ ./build-android-ndk.sh
```

7.3.1 ABIs

构建脚本默认会构建当前 NDK 支持的所有非过时 ABI 的库文件，你可以为构建脚本提供 *APP_ABI* 命令行参数构建特定 ABI 的库文件。

（只构建 armeabi-v7a 和 x86 ABI 库文件）

```
$ ./build-android-ndk.sh APP_ABI=armeabi-v7a,x86
```

7.3.2 功能特性开关

构建脚本默认会构建所有功能特性（*UDP / SSL / HTTP / ZLIB / ICONV*），你可以为构建脚本提供 *__XXX_DISABLE=true* 参数来移除某些功能特性。

- *_UDP_DISABLED=true* ：移除 UDP
- *_SSL_DISABLED=true* ：移除 SSL
- *_HTTP_DISABLED=true* ：移除 HTTP
- *_ZLIB_DISABLED=true* ：移除 ZLIB
- *_ICONV_DISABLED=true* ：移除 ICONV

（移除 SSL 和 ICONV）

```
$ ./build-android-ndk.sh _SSL_DISABLED=true _ICONV_DISABLED=true
```

注意：如果在构建库时移除了某些功能特性，那么在编译使用该库的应用程序时需要定义相应的宏。例如：在构建库时移除了 SSL 和 ICONV，在编译应用程序时需要定义 _SSL_DISABLED 和 _ICONV_DISABLED 宏。

7.3.3 其他选项

构建脚本 *build-android-ndk.sh* 只是对 Android NDK 命令 *ndk-build* 的简单包装，构建脚本支持 *ndk-build* 的所有命令行参数。*ndk-build* 的详细说明请参考官方文档。

以下示例演示如下构建选项：

- ✓ 构建 armeabi-v7a 和 x86_64 ABI 库文件
- ✓ 移除 UDP、ZLIB 和 ICONV 功能特性
- ✓ 库文件输出到 lib/android-ndk/ 目录
- ✓ obj 中间文件输出到 lib/android-ndk/obj 目录

```
$ ./build-android-ndk.sh APP_ABI=armeabi-v7a,x86_64 \
                         _UDP_DISABLED=true \
                         _ZLIB_DISABLED=true \
                         _ICONV_DISABLED=true \
                         NDK_LIBS_OUT=./lib/android-ndk \
                         NDK_OUT=./lib/android-ndk/obj
```

8 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 发行包中已经提供了 HPSocket DLL 和 HPSocket4C DLL。

注意：HP-Socket v5.2.x 开始，发行包不再分别提供 SSL 和非 SSL 库文件，发行包中提供的库文件包含 SSL 和 HTTP 组件，如果想去除 SSL 或 HTTP 组件，可以分别定义 _SSL_DISABLED 或 _HTTP_DISABLED 宏重新编译。

8.1 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）。

8.2 静态库

HP-Socket 发行包中的 *Project/HPSocketLIB* 和 *Project/HPSocketLIB4C* 工程项目用于编译 HPSocket LIB 和 HPSocket4C LIB，输出目录为 *Bin\x86(x64)\static*。如果需要可以自己编译。静态库与动态库的使用方式一致，请参考后续章节。

（静态库引用方式参考：TestEcho-SSL-4C / TestEcho-SSL-PFM 示例 Demo）。

注意：如果工程项目使用 HPSocket LIB 或 HPSocket4C LIB，需要在工程属性中定义预处理宏 -> HPSOCKET_STATIC_LIB。

8.3 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 发行包中的 *Project/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）。

HPSocket DLL 包含 SSL 组件和非 SSL 组件，如果需要用到 SSL 组件则需要把 *Src/HPSocket-SSL.h*、*Src/SocketInterface.h* 以及 DLL 对应的**.lib* 文件加入到工程项目。（参考：TestEcho-SSL-Pack 示例 Demo）。

通过 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。

8.4 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 发行包中的 *Project/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）。

HPSocket4C DLL 包含 SSL 组件和非 SSL 组件，如果需要用到 SSL 组件则需要把 *Src/HPSocket4C-SSL.h* 以及 DLL 对应的**.lib* 文件加入到工程项目。（参考：TestEcho-SSL-Pack 示例 Demo）。

通过 4C 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。

8.5 其它编程语言使用 HPSocket

HP-Socket 发行包中提供了 C# 和易语言 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、Java），可以通过导入 HPSocket4C DLL 的方式使用 HP-Socket。

C#、Delphi 和易语言使用 HP-Socket 的例子请参考 HP-Socket 发行包中 *Demo/Other Languages Demo* 目录下的示例 Demo。

9 附 录

9.1 示例 Demo

9.1.1 Windows 示例

项目名称	使用组件	引用方式	描 述
HttpProxy (Server-1)	*TCP Server* *TCP Agent*	DLL	HTTP 代理服务器（TCP 实现）
HttpProxy (Server-2)	*HTTP Server* *TCP Agent*	DLL	HTTP 代理服务器（HTTP 实现）
TestEcho	*TCP Server* *TCP Client*	SRC	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*	SRC	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-Pack	*TCP PACK Server* *TCP PACK Client*	4C DLL DLL	Echo 服务端和客户端
TestEcho-UDP	*UDP Server* *UDP Client*	SRC	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*	SRC	组播（广播）网络成员
TestEcho-SSL	*SSL Server* *SSL Client*	SRC	Echo 服务端和客户端
TestEcho-SSL-4C	*SSL PULL Server* *SSL PULL Client*	4C LIB	Echo 服务端和客户端
TestEcho-SSL-Pack	*SSL PACK Server* *SSL PACK Client*	4C DLL DLL	Echo 服务端和客户端
TestEcho-SSL-PFM	*SSL Server*	LIB	Echo 性能测试服务端和客户端

	SSL Agent		
TestEcho-Http	*HTTP Server* *HTTP Client* *HTTP Sync Client*	SRC	Echo 服务端和客户端
TestEcho-Http-4C	*HTTP Server* *HTTP Client* *HTTP Sync Client*	4C LIB 4C DLL 4C DLL	Echo 服务端和客户端
TestEcho-ARQ	*UDP ARQ Server* *UDP ARQ Client*	SRC	Echo 服务端和客户端
TestEcho-ARQ-PFM	*UDP ARQ Server* *UDP ARQ Client*	DLL	Echo 性能测试服务端和客户端

表 7.1-1 Windows 示例

9.1.2 **Linux** 示例

项目名称	使用组件	引用方式	描 述
testecho	*TCP Server* *TCP Agent* *TCP Client*	SRC	Echo 服务端和客户端
testecho-pfm	*TCP Server* *TCP Agent* *TCP Client*	SRC	Echo 性能测试服务端和客户端
testecho-pull	*TCP PULL Server* *TCP PULL Agent* *TCP PULL Client*	SRC	Echo 服务端和客户端
testecho-pack	*TCP PACK Server* *TCP PACK Agent* *TCP PACK Client*	SRC	Echo 服务端和客户端
testecho-udp	*UDP Server* *UDP Client* *UDP Cast*	SRC	Echo 服务端和客户端 组播（广播）网络成员
testecho-udp-pfm	*UDP Server* *UDP Client*	SRC	Echo 性能测试服务端和客户端
testecho-lib	*TCP PULL Server* *TCP PULL Agent* *TCP PULL Client*	SO 4C SO 4C SO	Echo 服务端和客户端
testecho-ssl	*SSL Server* *SSL Agent* *SSL Client*	SRC	Echo 服务端和客户端
testecho-ssl-pfm	*SSL Server* *SSL Agent* *SSL Client*	SO	Echo 性能测试服务端和客户端

testecho-ssl-pull	*SSL PULL Server* *SSL PULL Agent* *SSL PULL Client*	SO	Echo 服务端和客户端
testecho-ssl-pack	*SSL PACK Server* *SSL PACK Agent* *SSL PACK Client*	4C SO	Echo 服务端和客户端
Testecho-http	*HTTP Server* *HTTP Agent* *HTTP Client* *HTTP Sync Client*	SRC	Echo 服务端和客户端
Testecho-http-4c	*HTTP Server* *HTTP Agent* *HTTP Client* *HTTP Sync Client*	4C SO	Echo 服务端和客户端
testecho-arq	*UDP ARQ Server* *UDP ARQ Client*	SRC	Echo 服务端和客户端
testecho-arq-pfm	*UDP ARQ Server* *UDP ARQ Client*	A	Echo 性能测试服务端和客户端

表 7.1-2 Linux 示例

9.2 辅助函数

- ✧ ***LPCTSTR HP_GetSocketErrorDesc()*** ： 获取 HPSocket 错误码对应描述
- ✧ ***DWORD SYS_GetLastError()*** ： 封装 API 函数 *GetLastError() / errno*
- ✧ ***LPCSTR SYS_GetLastErrorStr()*** [L] ： 封装 API 函数 *strerror()*
- ✧ ***int SYS_WSAGetLastError()*** [W] ： 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()*** [W] ： 封装 API 函数 *WSAIoctl()*
- ✧ ***int SYS_fcntl_SETFL()*** [L] ： 封装 API 函数 *fcntl()*，设置 F_SETFL
- ✧ ***int SYS_SSO_ Block()*** [L] ： 设置 FD 选项：O_NONBLOCK
- ✧ ***int SYS_SSO_NoDelay()*** ： 设置 socket 选项：TCP_NODELAY
- ✧ ***int SYS_SSO_DontLinger()*** ： 设置 socket 选项：SO_DONTLINGER
- ✧ ***int SYS_SSO_Linger()*** ： 设置 socket 选项：SO_LINGER
- ✧ ***int SYS_SSO_RecvBuffSize()*** ： 设置 socket 选项：SO_RCVBUF
- ✧ ***int SYS_SSO_SendBuffSize()*** ： 设置 socket 选项：SO_SNDBUF
- ✧ ***int SYS_SSO_ReuseAddress()*** ： 设置 socket 选项：SO_REUSEADDR
- ✧ ***BOOL SYS_GetSocketLocalAddress()*** ： 获取 socket 本地地址
- ✧ ***BOOL SYS_GetSocketRemoteAddress()*** ： 获取 socket 远程地址
- ✧ ***ULONG SYS_EnumHostIPAddresses()*** ： 枚举主机 IP 地址
- ✧ ***BOOL SYS_FreeHostIPAddresses()*** ： 释放主机 IP 地址结构体
- ✧ ***BOOL SYS_IsIPAddress()*** ： 检查字符串是否符合 IP 地址格式
- ✧ ***BOOL SYS_GetIPAddress()*** ： 通过主机名获取 IP 地址
- ✧ ***BOOL SYS_NToH64()*** ： 64 位网络字节序转主机字节序
- ✧ ***BOOL SYS_HToN64()*** ： 64 位主机字节序转网络字节序
- ✧ ***BOOL SYS_CodePageToUnicode()*** [W] ： CP_XXX -> UNICODE
- ✧ ***BOOL SYS_UnicodeToCodePage()*** [W] ： UNICODE -> CP_XXX
- ✧ ***BOOL SYS_CharsetConvert()*** [L] ： Charset-A -> Charset-B
- ✧ ***BOOL SYS_GbkToUnicode()*** ： GBK -> UNICODE
- ✧ ***BOOL SYS_UnicodeToGbk()*** ： UNICODE -> GBK
- ✧ ***BOOL SYS_Utf8ToUnicode()*** ： UTF8 -> UNICODE
- ✧ ***BOOL SYS_UnicodeToUtf8()*** ： UNICODE -> UTF8
- ✧ ***BOOL SYS_GbkToUtf8()*** ： GBK -> UTF8
- ✧ ***BOOL SYS_Utf8ToGbk()*** ： UTF8 -> GBK
- ✧ ***int SYS_Compress()*** ： ZLib 压缩
- ✧ ***int SYS_CompressEx()*** ： 高级 ZLib 压缩
- ✧ ***int SYS_Uncompress()*** ： ZLib 解压
- ✧ ***int SYS_UncompressEx()*** ： 高级 ZLib 解压
- ✧ ***int SYS_GuessCompressBound()*** ： 推测压缩结果长度
- ✧ ***int SYS_GZipCompress()*** ： GZip 压缩
- ✧ ***int SYS_GZipUncompress()*** ： GZip 解压
- ✧ ***int SYS_GZipGuessUnompressBound()*** ： 推测 Gzip 解压结果长度

- ***int SYS_GuessBase64EncodeBound()*** ：计算 Base64 编码后长度
- ***int SYS_GuessBase64DecodeBound()*** ：计算 Base64 解码后长度
- ***int SYS_Base64Encode()*** ：Base64 编码
- ***int SYS_Base64Decode()*** ：Base64 解码
- ***int SYS_GuessUrlEncodeBound()*** ：计算 URL 编码后长度
- ***int SYS_GuessUrlDecodeBound()*** ：计算 URL 解码后长度
- ***int SYS_UrlEncode()*** ：URL 编码
- ***int SYS_UrlDecode()*** ：URL 解码
- ***LPBYTE SYS_Malloc()*** ：分配内存
- ***LPBYTE SYS_Realloc()*** ：重新分配内存
- ***VOID SYS_Free()*** ：释放内存

9.3 FAQ

- ***Q-01*：Connection ID 的生成规则是什么？数值溢出怎么办？**
  - ***A***：Connection ID 在 32 位程序中是 4 字节，在 64 位程序中是 8 字节。Client 组件和 Agent/Server 组件有不同的生成规则：
    1) Client 组件：当 Connection ID 溢出时会重新从 1 开始递增。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。
    2) Agent/Server 组件：Connection ID 取值范围：1 - N*256，其中 N 为最大连接数。HPSocket 通过内部算法合理分配一个安全随机的 Connection ID。

- ***Q-02*：可以在事件处理函数中调用 *Start() / Stop()* 吗？**
  - ***A***：不可以。由于监听器事件（*OnReceive / OnClose* 等）通常都在通信线程中被触发，*Stop()* 方法需要等待通信线程结束，这样会导致自己等等自己结束的死循环，因此不能在监听器事件处理代码中调用 *Start() / Stop()* 控制方法。

- ***Q-03*：可以在事件处理函数中更新用户界面吗？**
  - ***A***：不可以。事件处理函数由 Socket IO 线程触发，如果在事件处理函数中更新用户界面会急剧降低应用程序性能并且很容易造成死锁，应该使用其他方法异步更新用户界面。

- ***Q-04*：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-05*：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。HP-Socket v3.5.x 及后续版本默认开启静默标记。

- ***Q-06*：断线重连该如何实现？**
  - ***A***：Agent 组件可以在接收到断线通知事件（*OnClose*）时立刻发起 *Connect()* 调用进行重连；Client 组件则不能接收到断线通知事件（*OnClose*）时立刻调用 *Start()* 方法进行重连。因此，Client 组件可以选择以下方法实现重连：
    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* 事件中向监测线程发送断线重连通知（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-07*：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
       超时时间计算公式：*KeepAliveTime* + (*KeepAliveInterval* * ***N***)
       其中 *N* 为固定值：WinXP 以下系统 *N*=5；Win7 以上系统 *N*=10
    2) UDP 心跳检测：*SetDetectInterval()* / *SetDetectAttempts()*，单位 - 毫秒
       超时时间计算公式：*DetectInterval* * (*DetectAttempts* + ***X***)
       其中 *X* 可能为 0~1 之间的任意值
    3) 对于 Server 或 Agent 组件，可以通过“断开静默连接”方式间接实现心跳检测。如：使用定时器或独立线程定时调用 *DisconnectSilenceConnections()* 方法断开静默连接。

- ***Q-08*：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-09*：HP-Socket 的 TCP 组件是否处理了粘包？**
  - ***A*：三种选择：**
    1) PUSH 模型：应用程序手工处理粘包。
    2) PULL 模型：与应用层协议配合，半自动处理粘包。
    3) PACK 模型：通信组件自动处理粘包。

- ***Q-10*：HP-Socket 如何与第三方 Socket 应用通信？**
  - ***A*：根据 HP-Socket 组件接收模型分别处理：**
    1) PUSH 模型：与应用层协议无关，可以直接通信。
    2) PULL 模型：与对端协商应用层协议。
    3) PACK 模型：对端需遵守 HP-Socket PACK 的数据包格式。

- ***Q-11*：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多个 *OnReceive* 事件，因此，接收方的数据包也不会发生乱序。**

- ***Q-12*：多个通信组件能共享同一个监听器对象吗？**

➢ ***A：***可以。监听器回调事件的 *pSend* 参数标识当前通信组件。

- ***Q-13：*HP-Socket 如何实现流量控制？**

➢ ***A：***可以通过数据接收和数据发送两个方面实现流量控制：

1) 数据接收：调用 *PauseReceive()* 方法暂停或恢复数据接收。
2) 数据接收：调用 *GetPendingDataLength()* 方法获取堆积的未发出数据量，控制数据发送速度。

- ***Q-14：*HP-Socket 如何设置代理服务器？**

➢ ***A：***根据不同组件类型和代理服务器类型有不同的设置方式：

1) TCP 组件设置 SOCKS 代理：通过 SOCKS 协议连接到代理服务器后即可开始正常数据通信。
2) SSL 组件设置 SOCKS 代理：调用 *SetSSLAutoHandShake(FALSE)* 方法把组件设置为“手工启动 SSL 握手”模式，通过 SOCKS 协议连接到代理服务器后调用 *StartSSLHandShake()*方法启动 SSL 握手，开始正常数据通信。
3) Http/Https 组件设置 SOCKS 代理：调用 *SetHttpAutoStart(FALSE)* 方法把组件设置为“手工启动 HTTP 通信”模式，通过 SOCKS 协议连接到代理服务器后调用 *StartHttp()*方法启动 HTTP 通信。
4) Http 组件设置 HTTP 代理：不必做任何额外工作，只需把连接地址设置为代理服务器地址，发送请求时指定“Host”请求头为目标服务器地址。
5) Https 组件设置 HTTP Tunnel 代理：调用 *SetSSLAutoHandShake(FALSE)* 方法把组件设置为“手工启动 SSL 握手”模式，向代理服务器发起“CONNECT”请求，请求成功后调用 *StartSSLHandShake()*方法启动 SSL 握手，开始正常数据通信。

- ***Q-15：*如何减少库文件的体积？**

➢ ***A：*** HP-Socket 发布包提供的库文件包含了所有组件和功能特性，如果不需要某些组件或功能特性，可以在 *src/HPTypeDef.h* 中定义相应编译指示宏重新编译库文件（注意：修改后的 *src/HPTypeDef.h* 需要拷贝到 *include/hpsocket/ HPTypeDef.h*）：

1) _UDP_DISABLED ：排除 UDP 组件。
2) _SSL_DISABLED ：排除 SSL 组件（包括 Https 组件）。
3) _HTTP_DISABLED ：排除 HTTP 组件（包括 Https 组件）。
4) _ZLIB_DISABLED ：排除 zlib 相关功能函数。
5) _ICONV_DISABLED ：排除 iconv 相关功能函数（只对 Linux 有效）。

- ***Q-16：*关闭 Server 或 Agent 时，一直卡在 *Stop()*方法里面，Why？**

➢ ***A：***几种可能：

1) 在事件处理函数中调用 *Stop()* 方法（参考：*Q-02*）。
2) 一个或多个通信线程都被死锁了，导致 *Stop()* 方法一直等不到所有通信线程结束。如果是所有通信线程都被死锁还会伴随另一种现象：组件不能接收和处理任何通信请求。
3) 如果有些机器能顺利执行 *Stop()* 方法有些机器不能，可能是 Winsock 协议栈被破坏了。解决方法：以管理员身份在命令行工具中执行：“*netsh winsock reset*”，重启机器。以后少上不良网站，卸载不良软件。