Indy In Depth

译者注：

本人利用业余时间本着学习的目的，学习并翻译此 Indy 官方文档。只限于学习交流，禁止用于任何商业目的。Demo 等可以在 Indy 的官网 http://www.atozedsoftware.com/index.en.aspx 获得。第一部分----介绍 和 第二部分 ----- 技术支持 和 最后两个部分 ------福利资料及作者介绍 未翻译。

翻译时译者尽量保持原版排版，由于此文档年代久远，很多代码在 delphi 新版本中已经不能使用或者需要一定的修改方能使用，对于知道怎么修改的，译者已经直接进行了修改，对于一些有疑问或问题的，译者用不同颜色的字做了标注。由于译者能力有限，翻译中难免存在一定的问题，敬请联系我指正，谢谢！

转发请注明译者或者保留这一页。

联系方式：
あの夏の白い云
QQ：350645157

目录

3 Socket 介绍

3.1 综述

这篇文章是篇 socket(TCP/IP socket)概念的介绍。我并不打算完整包含全部 socket 概念；只是以一个入门的程度来教读者关于哪种 socket 编程可以便利的通信。

有些必须要先说明的概念。只要有可能，我就会用现实世界你很可能明白的概念来类比：电话系统。

3.2 TCP/IP

TCP/IP 是传输控制协议(Transmission Control Protocol/Internet Protocol) 的简称。

TCP/IP 可以代表许多东西。它常用来当做"catch all"使用，更经常的，他代表的是网络协议本身。

3.3 Client 客户端

客户端是发起一个连接的程序。典型的，客户端一次只会和一个服务器对话。如果一个过程需要和多个服务器对话，它会创建多个客户端。

类比于打电话，一个客户端就像打电话的那个人。

3.4Server 服务器

服务器是一个答复请求的程序。典型的客户端同时答复许多客户端的请求。然而每一个从服务器到客户端的连接都是个单独的接口(socket)。

类比于打电话，服务器就像接电话的人，服务器被特别的安装使得它可以处理电话。这就类似于客户服务中心通过一堆操作员来处理成千上万的电话一样。

3.5 IP 地址

每个在 TCP/IP 网络上的电脑都有个独一无二的地址。一些电脑可能有不止一个地址。一个 IP 地址是一个 32 位数字并且常用句点来表示，比如 192.168.0.1. 每个部分都代表 32 位地址的一个字节。

IP 地址就像电话号码。然而就像一个地方（如住所，公司）可以有一个或多个电话号码，单个机器也可能有超过一个属于它的 IP 地址。有多 IP 地址的机器被叫做 multi-homed。

为和在特定地点的人对话，一个连接企图通过拨打那个地方的电话号码来发起(电话拨出，被拨打电话的地方的电话铃响了)。另一端的人于是可以决定接或者不接电话。

通常一个 IP 地址简称为 IP。

3.6 Port 端口

一个端口是一个整形值，定义了客户端在某个 IP 地址具体想连接的是哪个应用或者服务。

一个端口很像一个电话分机。打个电话会让你联系到某地，但在 TCP/IP 中，每个地方都有个分机。没有默认的分机，就如住宅电话，因此连接服务器时除了 IP 地址外，还必须指定一个端口。

当一个应用服务器开始接收请求，它开始在某个端口监听。这就是有时应用或协议通过字口(word port)来可交换使用的原因。当一个客户端想和服务器通讯，他必须先知道应用在哪(IP 地址/电话号码)，和应用在监听哪个端口(分机)。

特别的，应用拥有固定的端口于是不管它在哪台机器上跑，端口都是固定的。比如 HTTP(Web)使用 80 端口，FTP 使用 21 端口。当你想获得一个 Web 页，你只需要知道那台电脑的 IP 地址就行，因为你知道 HTTP 使用的是 80 端口。

1024 以下的端口号是预留的，只有在你通信或者实现一个已知的拥有预留端口的协议时才能使用。大部分受欢迎的协议使用预留端口。

3.7 Protocol 协议

Protocol 这个词来自 Greek protocollon。protocollon 是黏在手写的书上的一张纸，描述它的内容。

在 TCP/IP 条款中，协议用来描述某事工作机理。但它更常用来指代两个东西：

1.接口(socket)的类型
2.一个高级命令协议。

当指代 socket 时，协议说明 socket 是那个 socket。常见的 socket 类型有 TCP,UDP 和 ICMP。

当指代高等协议时，它指实现需要功能的函数的命令和返回值。这类协议被称为 RFC's。例子如 HTTP，FTP 和 SMTP。

3.8 Socket 接口

这篇文章里的所有 socket 都指的是 TCP/IP。一个 socket 是一个 IP 地址，一个端口号和一个协议的总称。一个 socket 也是两个程序间的虚拟通信电缆。这两个程序可以是本地(local)或异地(remote)的。

一个 socket 就像一次交谈的电话连接。为了交谈，你必须先打电话，然后对面要接电话，否则没有连接被建立。

3.9 Host Name 主机名

Host names 是 IP 地址的人可读的别名。比如 www.nevrona.com. 每个主机名都有个对应的 IP 地址.比如 www.nevrona.com 解析为 208.225.207.130。

主机名不仅为了给予人类方便，还为了使得电脑能够在不丢失潜在客户端的前提下更换 IP 地址。

主机名就像人们或公司名，人或者公司可以更换号码，但我们仍然可以联系到他们。

3.10 DNS

DNS 是 *域名服务 Domain Name Service* 的简称。

DNS 把主机名翻译成 IP 地址的服务。为建立连接，必须使用个 IP 地址，所以 DNS 被首先用来查找 IP 地址。

为了打电话,你必须用手机号码拨打电话。你无法拨人的名字。如果你没有某人的电话号码，或者它变了，你会在电话薄上查找他的电话号码或者打给查号台。因此，DNS 就是因特网的电话薄或查号台。

3.11 TCP

TCP 是 *传输控制协议 Transmission Control Protocol* 的简称。

TCP 有时也代表 流协议(stream protocol)。TCP/IP 包含许多协议和通信方式。最常见的是 TCP 和 UDP。TCP 是个基于连接(connection-based)的协议，意思是，你在发和接收数据前必须连接。TCP 保证数据在连接上的传输接收及其准确性。TCP 还保证数据按发送顺序到达。大部分使用 TCP/IP 的东西用 TCP 来传输。

TCP 连接就像用电话来进行次电话会议。

3.12 UDP

UDP 是 *用户数据报协议 User Datagram Protocol* 的简称。

UDP 是用来发数据电报的，它不用连接。UDP 能传输"轻量"数据包给主机而不用先连接到另一个主机。UDP 不保证数据包裹的到达，也不保证会按发送顺序到达。当发送个 UDP 数据包时，是作为一整个块发送的。因此，数据包大小决不能超过你 TCP/IP 栈(stack)所定义的大小。

因为这些原因，许多人觉得 UDP 基本没用。实际上不是这样的。许多流协议(streaming protocols)，比如 RealAudio 使用的就是 UDP。

注意："streaming"很容易和"stream"连接搞混，后者是 TCP。当你看到这些名词时，你应该先根据内容确定下他们到底指的是哪个意思。

UDP 数据包的可靠性依赖于网络的可靠性和饱和度。UDP 数据包也常用于在局域网(LAN)上跑的应用，因为局域网很可靠。在因特网上传输的 UDP 数据包通常也很可靠，可以在改错后使用，或者更常在添写后使用。然而传输并不是在任何网络上都能保证——所以不要默认你的数据必然到达。

因为 UDP 不会确认发送，它也不保证到达。如果你给另一个主机发送 UDP 数据包，你无法得知数据包是否真的传到了。栈(stack)不会，也没有办法，确认到达，因此即使数据包没有

达到目的地也无法报错。如果你需要这些信息，你需要从异地服务器发回些类似于回执的东西。

UDP 就像用老式的手机(传呼机之类的)给某人发了个短信，你知道你发了信息，但你不知道他收到了没。可能不存在，也可能不在服务区，可能没开机，也有可能坏掉了。并且数据网 1 络也有可能丢失信息。除非他给你回了消息，不然你根本不知道他到底收到了没。还有，如果你发了多条短信，有可能它们会按不同顺序到达。

3.13 ICMP

ICMP 是 *控制报文协议 Internet Control Message Protocol* 的简称。

ICMP 是控制及维护协议。特别的，你用不到 ICMP 的。特别的，它是用来与路由器和其他网络设备通信用的。ICMP 允许节点(node)分享 IP 状态和错误信息。ICMP 是给 PING, TRACEROUTE 和其他类似的协议的使用的。

3.14 HOSTS

HOSTS 是一个包含本地主机查询表的文本文件/系统文件。当栈企图将一个主机名解析为对应的 IP 地址时，它先查询 HOSTS 文件。如果查到匹配的入口(entry)，它会使用那个入口。如果没有查到，它就会使用 DNS 。

这是一个 HOSTS 文件的示例：

```
# 这是个示例 HOSTS 文件
caesar          192.168.0.4    # Server computer
augustus        192.168.0.5    # Firewall computer
```

主机名和 IP 地址由空格或者 TAB 符分隔。注释也可以通过在文本前加#号来表示。

HOSTS 也可以用来篡改入口，或者重写 DNS 入口。HOSTS 文件在小的没有 DNS 服务器的局域网上经常被使用。HOSTS 文件也常用来重写主机 IP 地址以便调试。你不需要读取 HOSTS 文件；栈会在需要解析域名时透明地为你处理这些细节。

3.15 SERVICES

SERVICES 文件和 HOSTS 文件很类似。不同的是，它不是把主机名解析为 IP 地址，而是解析服务名为对应的端口号。

以下是一个 SERVICES 文件的一部分。你可以在你的电脑上查看完整的文件，或者取得 RFC 1700。RFC 1700 包含已经赋值的保留端口号：

```
echo 7                  /tcp
echo 7                  /udp
discard                 9/tcp           sink null
discard                 9/udp           sink null
```

```
systat          11/tcp      users           #Active users
systat          11/tcp      users           #Active users
daytime         13/tcp
daytime         13/udp
qotd            17/tcp      quote           #Quote of the day
qotd            17/udp      quote           #Quote of the day
chargen         19/tcp      ttytst source   #Character
                                            # generator
chargen         19/udp      ttytst source   #Character
                                            # generator
ftp-data        20/tcp                      #FTP data
ftp             21/tcp                      #FTP control
telnet          23/tcp
smtp            25/tcp      mail            #Simple Mail
                                            # Transfer Protocol
```

每个入口的格式是：

<service name> <port number>/<protocol> [aliases...] [#<comment>]

你不需要读 SERVICES 文件;栈会帮你做这些事的。SERVICES 文件是由栈中的特定函数读取的；然而，大部分程序不调用这些函数，自然也忽视他们的值。比如许多 FTP 程序默认 21 端口而不用栈去查询'ftp'入口；

一般你不会修改这个文件。然后还是有些程序添加入口进文件并且真的会用它。于是你可以更改这个入口去让这些程序去使用另一个端口。比如 Interbase。Interbase 创建如下入口：

```
gds_db          3050/tcp
```

你可以更改这个入口让 Interbase 使用另一个端口。尽管最好别这么做，但这是个好的练习。当你写 socket 应用特别是服务器时应该考虑这个事情。让客户端使用栈在 SERVICES 文件中查询值，特别是非标准协议的值，也是个好的练习。如果没找到入口，应该使用个默认值。

3.16 Localhost(Loopback 本地环回接口) 本地主机

LOCALHOST 就好像 Delphi 中的"Self"，或者 C++中的"this"。LOACLHOST 指应用正在执行的那台电脑。这是个 Loopback 地址，并且有个物理 IP 地址 127.0.0.1。如果在客户端中使用 127.0.0.1，它总是环回并在同一台电脑上查找服务器

这在调试中很有用。则可以用来和任何在你电脑上运行的服务连接。如果你有个本地 web 服务器，你可以直接写 127.0.0.1 ，这样就不用知道电脑的 IP 地址了，也不需要每个开发者在测试脚本中去修改它。

3.17 Ping

Ping 是一个用来确认某个主机是否可以由本地电脑连接的协议。Ping 常被用于性能诊断。

Ping 可以像一个命令行程序一样使用。用法是：

ping < host name or IP>

以下是一个成功 Ping 的输出的例子：

如果一个主机连接不上，输出会类似于这个：

C:\>ping 192.168.0.200

Pinging 192.168.0.200 with 32 bytes of data:

Request timed out.
Request timed out.
Request timed out.
Request timed out.

Ping statistics for 192.168.0.200:
Packets: Sent = 4, Received = 0, Lost = 4 (100% loss),

3.18 TraceRoute 路由跟踪

TCP/IP 数据包并不直接从一个主机传到另一个。数据包的传输就像屋到屋的车辆运输。特别的，车必须换道，TCP/IP 数据包一般就在一条路上传输。每次数据包在"枢纽"换道，它就经过了个节点。通过获得节点清单换句话就是数据包在主机间传递经过的"枢纽"，你就可以确定它的路径。这在检测主机无法连接的原因时很有用。

TraceRoute 就像命令行一样使用。TraceRoute 列出从你电脑到指定主机所要经过的节点清单，以及每条路用时多久。这些时间可以用来确定瓶颈。TraceRoute 还可以确定传输失败时成功处理你数据包的最后一个路由器。TraceRoute 是用来进一步诊断 Ping 发现的问题的。

下例是成功 TraceRoute 的输出的例子：

C:\>tracert www.atozedsoftware.com
Tracing route to www.atozedsoftware.com [213.239.44.103]
over a maximum of 30 hops:

1	<1 ms	<1 ms	<1 ms	server.mshome.net [192.168.0.1]
2	54 ms	54 ms	50 ms	102.111.0.13
3	54 ms	51 ms	53 ms	192.168.0.9
4	55 ms	54 ms	54 ms	192.168.5.2
5	55 ms	232 ms	53 ms	195.14.128.42
6	56 ms	55 ms	54 ms	cosmos-e.cytanet.net [195.14.157.1]
7	239 ms	237 ms	237 ms	ds3-6-0-cr02.nyc01.pccwbtn.net [63.218.9.1]
8	304 ms	304 ms	303 ms	ge-4-2-cr02.ldn01.pccwbtn.net [63.218.12.66]
9	304 ms	307 ms	307 ms	linx.uk2net.com [195.66.224.19]
10	309 ms	302 ms	306 ms	gw12k-hex.uk2net.com [213.239.57.1]
11	307 ms	306 ms	305 ms	pop3 [213.239.44.103]

Trace complete.

3.19 LAN 局域网

LAN 是 *局域网 Local Area Network* 的简称。

LAN 是一个模糊的概念，会随着网络配置的变化而变化。然而，局域网一般指由 Ethernet(或者有时指 Token Ring 及其他)、hubs 和 switches 连接的所有系统，不包含其他由桥或路由器勾住的 LAN。所以 LAN 由在网络交通中不需要通过桥或路由器连接的所有地方组成。

把 LAN 想象成一个城市的表面街道，桥和路由器是连接城市的的高速公路。

3.20 WAN 广域网

WAN 是*广域网(Wide Area Network)* 的简称。

WAN 指的是用桥和路由器连接起来 LAN 形成的一个更大的网络。

再次用到城市的例子，WAN 由通过高速路和洲际公路连接起来许多城市(LAN)组成。因特网本身也被定义为一个 WAN。

3.21 IETF

IETF (Internet Engineering Task Force) 是一个提供因特网操作，稳定性和解析的技术支持的开放社区。IETF 就像开源软件开发团队一样工作。网址为 http://www.ietf.org/。

3.22 RFC

RFC 是 Request for Comments 的简称。

RFC's 是 IETF 的官方文档，它用来描述和详细说明因特网的协议。RFC's 由数字来区分，比如 RFC 822。

因特网上有许多包含 RFC 文档的镜像服务器。推荐一个允许搜索的：

http://www.rfc-editor.org/

RFC 编辑(以上网址的)这样描述 RFCs：

> The Requests for Comments (RFC) document series is a set of technical and organizational notes about the Internet (originally the ARPANET), beginning in 1969. Memos in the RFC series discuss many aspects of computer networking, including protocols, procedures, programs, and concepts, as well as meeting notes, opinions, and sometimes humor.

3.23 Thread 线程

一个线程就是一个程序的执行路径。许多程序只有一个线程。然而也可以创建多个线程。

在多 CPU 的系统里，线程可以分配给不同的 CPU 来获得更快的运算速度。

在只有一个 CPU 的系统中，多线程仍然可以通过优先级(preemption)来执行。

优先级通过给每个线程一个很小的时间块(time slice)来使得 CPU 执行多任务。因此线程实际上看起来好像是被不同 CPU 单独执行的。

3.24 Fork

Unix 至今不支持多线程。它使用 forking。在多线程中，单独的执行线被建立，但它和其他线程存在于同一个程序中，因此也分享同样的内存空间。Forking 导致一个程序本身的"分离"。一个新程序启动，然后句柄传递给它。

Forking 不如多线程有效率，当时确实有一些优势。Forking 更加的稳定。而且在很多时候 Forking 更易于编程。

因为内核的使用和支持，Forking 在 Unix 中很常见，但多线程很新。

3.25 Winsock

Winsock 是 *Windows Sockets* 的简称。

Winsock 是个定义好并文档化的标准 API，为编写网络协议服务。它最常被用来编程 TCP/IP，但是也可以用来编程 Novell(IPX/SPX)和其他新网络协议。Winsock 是 Win32 自带的动态链接库(DLL)。

3.26 Stack 栈

专有名词 Stack 指代操作系统中实现网络及给开发者提供网络连接 API 的层。

在 Windows 中 Winsock 就是栈的实现。

3.27 Network Byte Order

不同的操作系统用不同的顺序存储数值数据。一些电脑用最小有效字节优先(LSB)来存储数值，然而其他的最大有效字节优先(MSB)。当用到网络时，并不总能知道连接的另一端的电脑用什么顺序。为了解决这个问题，有个标准的数值存储及网络传输顺序叫做 Network Byte Order。Network Byte Order 是应用在传输二进制数值时固定顺序。

4 Indy 介绍

4.1 Then Indy Way

Indy 就是为多线程而生的。用 Indy 建立客户端和服务器的方式和建立 Unix 服务器和客户端的方式很类似，除了 Indy 更加简单，因为你有 Indy 和 Delphi。而 Unix 应用却几乎不用抽象层而直接调用栈。

特别地，Unix 服务器有一个及以上搜寻客户端请求的监听程序。对每个需要服务的客户端，它会分支(fork)出一个新程序来分别服务。因为每个程序都只要服务一个客户端，编程十分简单。每个程序都在自己的 security context 里运行，security context 可以由监听器或过程基于证书，身份认证或其他方式设定。

Indy 服务器的工作方式很类似。Windows 对 fork 的支持不如 Unix，但是它对线程的支持很好。Indy 服务器为每个客户端链接分配一个线程。

Indy 服务器设置一个分离于主线程的监听器线程。监听器线程监听客户端的请求。为每个应答的请求，它产生一个新线程去服务客户端。然后新线程内的相应事件被触发。

4.2 Indy Methodology

Indy 和你可能熟悉的那些 socket 组件不同。如果你没用过其他 socket 组件，你会发现 Indy 很简单，因为 Indy 用你希望的方式运行。如果你用过其他 socket 组件，请忘掉你学过的，那些知识只会妨碍你，导致你犯些错误的默认。

几乎全部的其他组件都是用非阻塞(non-blocking)的调用和异步运作。他们要求你答复事件，设置状态机，并且常使用等待循环(wait loops)。

比如，用其他组件时，当你发起连接，你必须等待连接事件触发，或者循环到某个属性通知你已经连接上了。用 Indy 时，你几乎只用调用 Connect 函数，然后等待返回值。如果成功，它会在完成任务后返回。如果失败，它会引发异常。

操作 Indy 非常像操作文件。Indy 允许你把所有代码放在同个地方，而不是分离到不同的事件中。还有，大部分人觉得 Indy 使用起来简单的多。Indy 也设计的对线程支持良好。如果你使用 Indy 实现一些事情时出了问题，一步步返回，就像文件一样处理它。

4.3 How Indy is Different Indy 的独特之处

- Indy 使用阻塞的 socket API。
- Indy 不依赖于事件。Indy 有用于获得信息的事件，但那不是强制的。
- Indy 为多线程而设计。可 Indy 没有多线程也能用。
- Indy 编程时使用顺序编程设计(sequential programming)。
- Indy 高度抽象。大部分 socket 组件没有有效的把程序员从 socket 中分离出来。大部分 socket 组件没让用户远离复杂的栈反而让他们去完成 Delphi / C++ Builder 封装。

4.4 客户端概述

Indy 被设计为提供高度抽象。程序员不需要明白 TCP/IP 栈的复杂的实现细节。

一个典型的 Indy 客户端部分长成这样：

```
with IndyClient do begin
Host := 'postcodes.atozedsoftware.com'; // Host to call
Port := 6000; // Port to call the server on
Connect; Try
// Do your communication
finally Disconnect; end;
end;
```

4.5 服务器概述

Indy 服务器组件创建了一个分离于程序主线程的监听器线程。监听器线程监听进入的客户端请求。为每个应答的请求产生一个新线程去服务那个客户端。然后新线程内的相应事件被触发。

4.6 Threading

Threading 是使用多线程去实现其功能的程序。Indy 在服务器的实现中广泛地使用多线程，多线程对客户端也很有用。

非阻塞的 socket 也可以成为线程，但他们需要其他处理并且阻塞 socket 的优势也就丢失了。

5 Blocking vs. Non-Blocking 阻塞和非阻塞

5.1 Programming Models 编程模型

在 Windows 下的 socket 编程有两个编程模型，阻塞和非阻塞。有时，他们也被叫做同步(阻塞)和异步(非阻塞)。这篇文档通篇都在使用专有名词 阻塞 和 非阻塞。

在 Unix 中只支持阻塞模型。

5.2 其他模型

实际上还有些其他的已经实现的模型。包括完成端口(completion ports)，和重叠 I/O(overlapped I/O) 。然而使用这些模型要用明显更多的代码并且它们是特别为高级服务器应用保留的。

还有，这些模型不跨平台并且在不同平台的实现方式和实现程度不同。

Indy 10 包含对这些其他模型的支持。

5.3 阻塞

Indy 使用阻塞 socket 调用。阻塞调用很像一个文件的读写。当你读数据或者写数据时，直到操作完成，函数才会返回。不同的是，socket 中，操作可能会消耗更久因为数据可能不能立即读写。一个读或写操作的速度无法超过网络或模型接收和传输数据的速度。

使用 Indy 时，连接一个接口只要简单的调用 Connect 方法，然后等待它返回。如果 Connect 成功，它会在连接后返回。如果 Connect 方法失败，它会引发相应的异常

5.4 非阻塞

非阻塞 socket 使用事件机制。调用被触发后，当他们完成或者需要被注意时，引发事件。

比如，当尝试连接一个 socket 时，你必须调用 Connect 方法。Connect 方法在 Socket 连接前就立即会返回值。当 socket 连接上了，一个事件会发生。这要求通信逻辑被分成许多程序，或者使用轮询循环(polling loops)。

5.5 Winsock 的历史

一开始是 Unix。这里指 Berkely Unix。Berkely Unix 有个标准 socket API ，这 API 后来被 Unix 爱好者们采用并传播。

后来是 Windows，最终，某人觉得能在 Windows 上编程 TCP/IP 是个好主意。于是，他们移植了大受欢迎的 Unix socket API 。这让已存的大量 Unix 代码很轻松地移植到了 Windows。

5.6 Blocking is not Evil 阻塞并不烂

阻塞 sockets 在错误的前提下已经被不断诽谤了很久。和大部分人所相信的相反，阻塞 socket 并不烂。

当 Unix socket API 刚被移植到 Windows 时，它被命名为 Winsock。Winsock 是 "Windows Sockets" 的简称。在移植中，一个问题很快出现了。在 Unix 中，fork 十分正常。Forking 和多线程很相似，但是使用分别的程序而不是分别的线程。Unix 客户端和服务器为每个 socket fork 一个程序。这些程序独立的使用阻塞 socket 运行。

Window3.1 无法有效率的 fork 因为它缺乏多任务优先级和其他必要的支持。Window3.1 也不支持多线程。使用阻塞 socket 导致用户界面的冻结。又因为 Windows 3.1 使用协作地多线程，这也导致系统上的所有程序也都无法响应了。因为这极其不受欢迎，于是非阻塞式的插件被添加进了 WinSock 来让有缺陷的 Windows 3.x 在使用 WinSock 时不会冻结整个系统。然而这些插件需要一种完全不同的编程 socket 的方式。微软和其他人发现诽谤阻塞式 socket 比提起 Windows 3.1 的缺陷对他们最有利。

随着 Windows NT 和 Windows 95 的引入，Windows 有了多任务优先级和多线程的能力。但在那时，大部分程序员已经坚信阻塞式 socket 很烂了。桥被烧毁了，于是对阻塞 socket 的诽谤还在继续。

事实上，阻塞式 API 是 Unix 唯一支持的 API。

一些插件被安在 Unix 上来支持非阻塞 socket。然而这些插件的工作方式与 Windows 中的非阻塞插件十分不同。这些插件在所有 Unix 系统中没有统一标准并且没有广泛使用。阻塞 socket 仍被 Unix 中几乎所有实例使用，并且使用还会继续。

阻塞 socket 还有其他优点。比如阻塞 socket 对线程十分友好，很安全及其他好处。

5.7 Pros of Blocking 阻塞的优点

1. **易于编程** — 阻塞 socket 很易于编程。所有的用户代码都在一个地方，并且顺序排列。
2. **跨平台** — 由于 Unix 使用阻塞 socket，便携式代码很好写。Indy 依靠这个事实来实现它的单源代码跨平台能力。其他跨平台的 socket 组件通过内部使用阻塞调用来模拟非阻塞行为。
3. **多线程工作良好** — 由于阻塞 socket 的使用按照自然顺序，他们是可继承的封装，因此很适合多线程。
4. **不依赖消息** — 非阻塞 socket 依赖于 window 信息系统。当在多线程中使用时，分别的信息序列要被创建。当不在多线程中使用时，处理多链接时很容易发生瓶颈情况。

5.8 Cons of Blocking 阻塞的缺点

客户端用户界面"冻结" — 阻塞 socket 调用后直到完成任务才会返回。当这调用发生在应用的主线程中时，主线程无法处理用户界面消息。这导致用户界面的"冻结"。冻结的发生是由于直到阻塞 socket 调用返回控制应用程序的句柄前，更新，重绘和其他消息都无法被处理。

5.9 TIdAntiFreeze

Indy 有一个特殊的组件以透明的解决用户界面的冻结问题。应用中 TIdAntiFreeze 实例的存在使得主线程中阻塞调用的使用不会让用户界面冻结。TIdAntiFreeze 之后在 depth 中被覆盖的更广。

TIdAntiFreeze 的使用使得阻塞 socket 保留了所有优点的同时没有了明显的缺点。

5.10 Pros of Non-Blocking 非阻塞的优点

1. 客户端无用户界面"冻结" — 因为用户代码响应事件，Windows 在 socket 事件间拥有控制权。因此，Windows 也可以答复其他 Windows 消息。
2. 可以不用多线程的多任务 — 单线程可以处理许多 socket。
3. 许多 socket 是轻量级的 — 因为许多 socket 可以被处理而不需要多线程，内存和 CPU 的占用通常很少。

5.11 Cons of Non-Blocking 非阻塞的缺点

编程更难 — 非阻塞要求使用轮询或者事件。由于轮询很没效率，事件更常用。使用事件要求用户代码分成许多子过程因此需要状态追踪。这导致代码易于出 bug 并且难以维护。

5.12 Comparison 对比

如果你熟悉 Indy 以及它的方法你也许想跳过。如果你以前有编程 socket 的经验，这部分对你会很有用。

对那些没有编程 socket 经历的人，Indy 使用起来很简单和自然。当时对于那些有编程 socket 经历的人，Indy 是个大障碍。这是应该 Indy 的运作方式不同。这不是说其他的方式就是错的，只是 Indy 很特别。想用编程其他 socket 库的方式去编程 Indy 就好像想用使用火炉一样的方式来用微波烹饪，结果会是灾难性的甚至会爆炸。

如果你之前使用过其他 socket 库，请记住这个简单而微妙的建议：

忘了你学过的！

然而这说起来比做起来难，因为习惯成自然。为了强调区别，一个抽象的对照会用个例子来

展示。为了抽象概念以及切断联系你原本的思考 socket 的模式，我用文件来代替。这个文档默认你知道怎么读和写文件。(换句话说，VB 程序员不适合这文档)。

5.13 Files vs. Sockets

文件和 socket 的最大不同是文件的存取快的多。然而文件的存取也不是总很快。软盘，网络硬盘，备用存储和分级存储管理系统也常回应缓慢。

5.14 Scenario

我会设定一个简单的场景来简单的写数据到文件。尽管程序做这些很简单，这里是示例的细节。

1. 打开文件
2. 写数据
3. 关闭文件

5.15 Blocking File Write 写阻塞文件

一个阻塞文件的写操作看起来是这样的：

```
procedure TForm1.Button1Click(Sender: TObject);
var
  s: string;
begin
  s := 'Indy Rules the (Kudzu) World !' + #13#10;
  try
      // Open the file
      with TFileStream.Create( 'c:\temp\test.dat', fmCreate) do try
          // Write data to the file
          WriteBuffer(s[1], Length(s));
      // Close the file
      finally Free; end;
  end;
end;
```

如你看到的，它与之前的提纲一步一步一一对应，就如注释一样。代码是顺序执行的并且读起来很好理解。

5.16 Non-Blocking File Write 写非阻塞文件

实际上没有写非阻塞文件这种事(可能要除了重叠 I/O，但那已经超出了文件范畴并且代码更难看)，但是假设有的话，那看起来就会像下面这样。File1 是个图像非阻塞文件组件，已经被拖拽到窗体中了。

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  File1.Filename := 'd:\temp\test.dat' ;
  File1.Open;
end;
procedure TForm1.File1OnOpen(Sender: TObject);
var
  i: integer;
begin
  FWriteData := 'Hello World!' + #13#10;
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
end;
procedure TForm1.File1OnWrite(Sender: TObject);
var
  i: integer;
begin
  i := File1.Write(FWriteData);
  Delete(FWriteData, 1, i);
  if Length(FWriteData) = 0 then begin
      File1.Close;
  end;
end;
procedure TForm1.File1OnClose(Sender: TObject);
begin
  Button1.Enabled := True;
end;
```

花些时间去尝试理解发生了什么。如果你用过非阻塞 socket，你会很快理解的，以下是大纲：

1. Button1Click 被调用并打开文件。当 Open 方法立即返回值，文件还没有打开并还没法存取。
2. 当文件已经被打开并可以存取后 OnOpen 事件被触发，一个写文件的企图被执行，但是数据可能还没有被接受。Write 方法会返回接受了多少字节。剩余的数据被保存然后之后再重试。
3. 当文件准备好写更多数据时 OnWrite 事件被触发，Write 方法使用剩下的数据再次尝试。
4. 第三步继续重复，直到所有数据都被 Write 方法接受。当所有数据都被接受，Close 方法被调用。然而文件还没有关闭。
5. OnClose 事件被触发。文件现在关闭了。

5.17 File Write Comparison 写文件对比

两个例子都只写了数据。非阻塞例子中读写数据会更加复杂，当时在阻塞的例子中只是加了一行代码。

对非阻塞的例子，只是打开，写数据，然后关闭文件就需要：

- 3 个 File1 个事件
- 1 个窗体成员变量

非阻塞版本更加复杂和困难去理解。如果选择的话，任何人都会选择非阻塞的那个。大部分 C++程序员都是受虐狂，两个都不会选，因为他们都太简单了。于是几乎所有 socket 控件功能都是用非阻塞模型。

5.18 Just Like Files

使用 Indy 就像使用文件。事实上 Indy 常更简单因为 Indy 有一堆读写方法而不是只有几个。和文件等同的 Indy 例子看起来是这样的。

```
with IndyClient do begin
  Connect; Try
      IOHandler.WriteLn('Hello World.');
  finally Disconnect; end;
end;
```

如你所见，Indy 真的很像进行文件操作。Connect 代替了 Open，Disconnect 代替了 Close。如果你如对待文件一样对待 socket，你会发现使用 Indy 太简单了。

6 客户端介绍

6.1 基础客户端

一个基本 Indy 客户端的形式是这样的：

```
with IndyClient do begin
  Host := 'test.atozedsoftware.com' ;
  Port := 6000;
  Connect; Try
      // Read and write data here
  finally Disconnect; end;
end;
```

Host 和 Port 也可以在设计时使用属性浏览器设定。这是使用 Indy 写客户端所需的最少代码。以下是建立客户端的最低标准：

1. 设置 Host 属性。
2. 设置 Port 属性。这只在没有默认端口时需要。大部分协议提供一个默认端口。
3. 连接。
4. 传输数据。包括读和写。
5. 断连。

6.2 Handling Exceptions 异常处理

Indy 的异常处理就和文件一样。如果在调用 Indy 方法的过程中发生了错误，相应的异常会触发。为了处理这些异常，代码应该用 try..finally 或 try..except 语句块合理的封装。

没有 OnError 事件，所以别去找。如果你用过其他 socket 库的话这可能看起来很怪，但是想想 TFileStream。TFileStream 没有 OnError 事件，它简单的在有错误时引发异常。Indy 运行行为完全相同。

就像所有的文件打开都要对应文件关闭，Indy 中所有的 Connect 调用都应该对应个 Disconnect 调用。基本的 Indy 客户端的开头结尾应该是这样的：

```
Client. Connect; try
// Perform read/write here
finally Client.Disconnect; end;
```

很容易区分 Indy 异常和其他 VCL 异常，因为所有的 Indy 异常都从 EIdException 继承。如果你想要与区分的处理 Indy 异常和 VCL 异常，可以按下面的示例来做。

注意：为了使用 EIdException 你得在 uses 语句中添加 IdException。

```
try
  Client. Connect; try
      // Perform read/write here
  finally Client.Disconnect; end;
except
  on E: EIdException do begin
      ShowMessage('Communication Exception: ' + E.Message);
  end else begin
      ShowMessage('VCL Exception: ' + E.Message);
  end;
```

```
end;
```

如果在调用 Connect 方法中发生错误。它会在抛出相应异常前自我清理。因此，try 在 Connect 之后而不是之前。但是如果在数据传输中发生了异常，只会产生 exception 异常。Socket 会仍在连接状态。你自己要负责重试你的操作或者断连。在上面的框架中没有多余操作，不管有没有发生错误，socket 都会在最后断连。

为了处理连接中发生的错误并把他们从其他通讯错误中分离出来，你的代码可以是这种形式：

```
try
  IdTCPClient1.Connect; try
      try
          // Do your communications here
      finally IdTCPClient1.Disconnect; end;
  except
      on E: EIdException do begin
          ShowMessage('An network error occurred during communication: '
          + E.Message);
      end;
      on E: Exception do begin
          ShowMessage('An unknown error occurred during communication: '
          + E.Message);
      end;
  end;
except
  on E: EIdException do begin
      ShowMessage('An network error occurred while trying to connect: '
      + E.Message);
  end;
  on E: Exception do begin
      ShowMessage('An unknown error occurred while trying to connect: '
      + E.Message);
  end;
end;
```

这段代码不止检测了连接中发生的异常，还从这异常中区分了通讯时发生的异常。还更进一步区分了 Indy 异常和非 Indy 异常。

6.3 Exceptions are not Errors 异常并非错误

许多开发者被教育或者默认异常就是错误。其实不是这样，如果是这样的话 Borland 就会命名他们为 Errors 而不是 Exceptions。

异常是不正常的东西。软件开发中，异常是发生并更改正常程序流的东西。

异常被用来实现 Delphi 中的错误，因此大部分异常是错误。然而还有异常比如 EAbort 不是错误。Indy 也定义了很多不是错误的异常。这些非错误异常继承自 EIdSilentException 因此也很好从错误和其他异常中分辨出来。

想要更多示例，请看 EIdConnClosedGracefully

6.4 TIdAntiFreeze （VCL 中有，FMX 中没有）

Indy 有个特别的组件显式地解决用户界面冻结。应用中 TIdAntiFreeze 实例的存在使得主线

程中的阻塞调用不会导致用户界面冻结。

TIdAntiFreeze 工作原理是内部调用栈的暂停函数来使得消息在暂停期间能够传递。Indy 的外部调用仍然是阻塞的，因此代码就像不存在 TIdAntiFreeze 一样继续工作。

因为用户界面冻结只是源于主线程中的阻塞调用，TIdAntiFreeze.只影响主线程中的 Indy 调用。如果一个应用在多线程中使用 Indy，TIdAntiFreeze 并没有用。用了也只会影响主线程中的调用。

使用 TIdAntiFreeze 某种程度上会降低 socket 通信速度。可以通过 TIdAntiFreeze 的属性来设置给予应用多大的优先级。使用 TIdAntiFreeze 会降低 socket 通讯速度的原因是主线程可以处理消息。因此必须注意不要让太多时间消耗在消息事件中。这包括大部分用户界面事件比如 OnClick，OnPaint, OnResize 等。因为非阻塞 socket 本身依赖于消息，这个问题再非阻塞 socket 中很重要。因为 Indy 可以使用 TIdAntiFreeze，程序员有完全的控制权。

6.5 Demo - Postal Code Client 示例—邮编客户端

这是个查询邮编协议的客户端示例。协议很简单，我们假设服务器已经预定义好来。这章不讲服务器。

邮政编码查询就是用给定的邮编来查询城市和州(在美国是 zip code)。包含服务器的例子程序包含了美国邮编的数据。美国邮政编码是五位数字。

这个示例的服务器在之后会有。

6.5.1 Postal Code Protocol 邮编协议

邮编协议很简单。它只包含两个命令：

- Lookup <post code 1> <post code 2> ...
- Quit

一个示例通信长这样：

```
Server: 204 Post Code Server Ready.
Client: lookup 16412
Server: 200 Ok
Server: 16412: EDINBORO, PA
Server: .
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
Client: quit
Server: 201-Paka!
Server: 201 4 requests processed.
```

这个服务器成功连接后回复句问候语。问候语和答复包含声明状态的三位数字。在后面的部分会说更多细节。

在问候之后，服务器就准备好了接受请求。如果收到了 Lookup 命令，服务器就会答复一个

相关地点的清单。每行答复通过一个句点结束。客户端爱提交多少命令就可以提交多少直到客户端提交 Quit 命令来声明它准备断连了。

6.5.2 Code Explanation

邮编客户端包含两个 buttons，一个 listbox 和一个 memo 。一个按钮是用来清空结果，另一个是用来从 memo 取值并给服务器发送请求。结果会呈现在 listbox 里。

在一个正常的应用中，用户将有指定主机，端口甚至代理配置信息的权利。然而为了示范，主机已经被在设计时设定为 127.0.0.1 (自己) 和端口 6000。

当用户点击 Lookup 按钮，以下事件会执行：

```
procedure TformMain.butnLookupClick(Sender: TObject);
var
  i: integer;
begin
  butnLookup.Enabled := False; try
      lboxResults.Clear;
      with Client do begin
          Connect; try
              // Read the welcome message
              GetResponse(204);
              lboxResults.Items.AddStrings(LastCmdResult.Text);
              lboxResults.Items.Add( '');
              // Submit each zip code and read the result
              for i := 0 to memoInput.Lines.Count - 1 do begin
                  SendCmd('Lookup ' + memoInput.Lines[i], 200);
                  Captured(lboxResults.Items);
                  lboxResults.Items.Add( '');
              end;
              SendCmd('Quit', 201);
          finally Disconnect; end;
      end;
  finally butnLookup.Enabled := True; end;
end;
```

这里的 Indy 命令解释的很简略，因为后面会很详细的讲。

代码一开始先停用 Button，这样当一个命令执行中时，用户就没法尝试发送另一个 Lookup 了。你可能会觉得这不可能发生，因为按钮点击事件是由消息驱动的。但是这个例子有一个 TIdAntiFreeze ，它调用了 Application.ProcessMessages ，于是界面能够重绘，另一个点击事件也能发生。因此，你得额外的注意用户行为。

拖一个 TIdTCPClient (Client)到设计界面，例子然后就能连接到服务器并等待欢迎信息了。GetResponse 读取答复然后把答复作为结果返回。在这里返回结果被弃用了，但 GetResponse 验证了答复的数字是 204。如果服务器答复了一个不同的代码，就会引发异常。一些情况下服务器可能会答复不同的代码，比如太忙了或者在维护中。

对用户输入了每个邮政编码，例子都会发送个 lookup 命令给服务器，然后等待 200 号答复。SendCmd 发送命令然后用第二个参数调用 GetResponse(这是是 200)。如果 SendCmd 成功了，例子会调用 Capture，Capture 会读回复到一行中发现了一个句点"."。因为例子每条命令只提交一个邮编，一条答复也应该只有一行，或者当邮编无效时什么也没有。

最终例子发送了 Quit 命令，等待 201 答复以知道服务器同意了，然后断连。使用 Quit 命令是个好主意，这样两边都知道要断连了。

7 UDP

7.1 Overview 综述

UDP (User Datagram Protocol)用户数据报协议是用来发数据电报，不用连接。UDP 能够无连接的发送轻量级数据包给主机。UDP 数据包不保证到达目的地，也可能不按发报的顺序到达。当发送一个 UDP 数据包时，它是成块发送的。因此，你决不能发超过你的 TCP/IP 栈的最大尺寸的数据包。

因为这些原因，许多人觉得 UDP 基本没用。事实上不是这样。许多流协议，比如 Real Audio 使用 UDP。

注意："streaming"很容易和"stream"连接搞混，后者是 TCP。当你看到这些名词时，你应该先根据内容确定下他们到底指的是哪个意思。

7.2 Reliability 可靠性

UDP 数据包的可靠性依赖于网络的可靠性和饱和度。UDP 数据包也常用于在局域网(LAN)上跑的应用，因为局域网很可靠。在因特网上传输的 UDP 数据包通常也很可靠，可以在改错后使用，或者更常在添写后使用。添改(Interpolation)是在基于之前或者之后收到的数据包推测出发生了数据丢失然后发生的。然而传输并不是在任何网络上都能保证——所以不要默认你的数据必然到达。

因为 UDP 不会确认发送，它也不保证到达。如果你给另一个主机发送 UDP 数据包，你无法得知数据包是否真的传到了。栈(stack)不会，也没有办法，确认到达，因此即使数据包没有达到目的地也无法报错。如果你需要这些信息，你需要从异地服务器发回些类似于回执的东西。

UDP 就像用老式的手机(传呼机之类的)给某人发了个短信，你知道你发了信息，但你不知道他收到了没。可能不存在，也可能不在服务区，可能没开机，也有可能坏掉了。并且数据网络也有可能丢失信息。除非他给你回了消息，不然你根本不知道他到底收到了没。还有，如果你发了多条短信，有可能它们会按不同顺序到达。

另一个现实世界中和 UDP 相似的例子是邮寄服务。你可以发送它，但你不知道它会不会到达目的地，它可能在路上的任何地方丢失或者在到达前损坏。

7.3 Broadcasts 广播

UDP 有一个很有用的特殊能力。这个能力就是可以发广播。广播的意思是发送一条消息，但是可以被许多人接收到。这和群播(multicasting)不是一回事。群播是一个订阅模型，在模型中，接受者订阅并且被添加到了分发列表中。通过广播，一个消息在网络上传播，任何在收听的人都可以接收到而不用先订阅。

群播就好像分发报纸。只有那些订阅了的人才能收到报纸。广播就像无线电台信号。任何有收音机的人都可以调到特定的电台去收听。用户不用通知收听的电台。

一个特定的广播地址可以由触发触发者和子网掩码(subnet mask)的 IP 地址计算出来。然而，大部分情况下使用 255.255.255.255 最简单，它会广播的尽可能远。

然而几乎所有路由器都被设计地默认过滤掉广播信息。换句话说，信息不会传播过 桥和外部路由器，广播会被限制在本地局域网。

7.4 Packet Sizes 数据包大小

大部分操作系统允许 32K 甚至 64K 的 UDP 数据包大小。然而一般路由器的限制较大。UDP 数据包只能和最大允许尺寸一样大，也就是所有要经过的路由器和网络设备允许的最大大小。没办法预测这个大小。

7.5 Confirmations

7.5.1 Overview 综述

在局域网环境中 UDP 十分可靠。然而在广域网或者因特网中你就得实现许多确认计划(confirmation schemes)。

7.5.2 Acknowledgements

在 Acknowledgements 系统中，每个数据包都被收到数据的人确认。如果发出后在给定时间没有收到确认，就会重发。

因为"确认"本身可能丢失，每个包裹都要有个独一无二的身份。一般这个身份是个简单的序列号。于是接收数据的接受者可以过滤掉重复的数据包，同样的，"确认"信息也可以说明它们到底收到了哪一个数据包。

7.5.3 Sequencing 排序

数据包可以由一个序列号区分。这个数字可以让接收者用来确认数据包是否丢失。然后就可以请求重发特定的丢失的数据包，或者有时比如音频可以根据邻近的数据包添改数据来最佳猜测丢失的数据或者简单的无视掉丢失数据。

这种丢失数据包的行为可以在使用 real audio 或者移动数字电话时听出来。在某些系统中你几乎听不出来跳跃或者中断。在其他数据包很小并且系统添写的情况下你可能听到一声"worbling"。

7.6 TIdUDPClient

TIdUDPClient 是基础 UDP 客户端，用来发送 UDP 数据包给其他地方。最常使用的方法是 Send，Send 使用 Host 和 Port 属性来发送 UDP 数据包。它接受一个字符串作为参数。

还有个 SendBuffer 方法和 Send 方法的任务相同，除了它接受一个 Buffer 和一个 Size 作为参数。

TIdUDPClient 也可以作为某种服务器来独立地等待和接收 进入的 UDP 数据包。

7.7 TIdUDPServer

因为 UDP 是无连接的，所以 IdUDPServer 和 TIdTCPServer 的运作方式很不一样。TIdUDPServer 没有任何和 TIdSimpleServer 相似的模式，但因为 UDP 是无连接的，TIdUDPClient 确实只使用 listening 方法。

TIdUDPServer 在激活后建立了一个监听线程来监听到达的 UDP 数据包。为每个接收到的 UDP 数据包，TIdUDPServer 会在主线程或者监听线程中触发 OnUDPRead 事件，这取决于 ThreadedEvent 的值。

当 ThreadedEvent 是 False，OnUDPRead 事件会在主线程中触发。但 ThreadedEvent 是 True，OnUDPRead 会在监听线程中触发。

不管 ThreadedEvent 是 True 还是 False，它的执行都会阻塞更多消息的接收，因此 OnUDPRead 事件必须快速的执行。

7.8 UDP Example - RBSOD

7.8.1 Overview

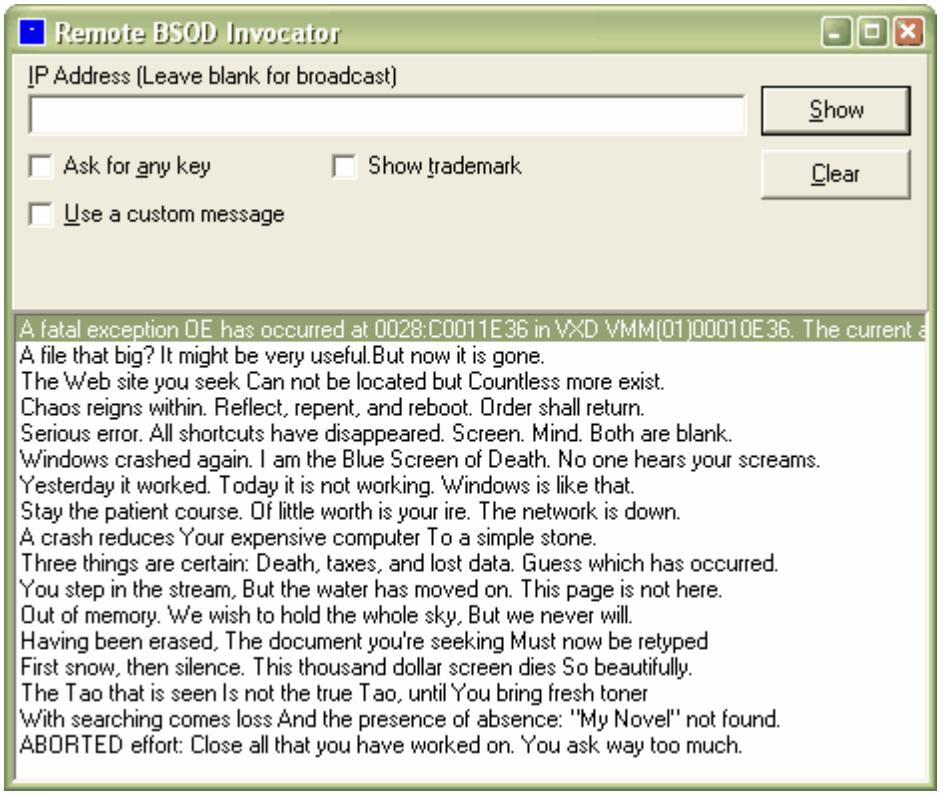
这个例子会提供一个 UDP 客户端和服务器的例子。这例子也是个能用来在许多企业环境中恶作剧的有用程序。然而一定要小心使用它。

例子的名字是 Remote BSOD invocator 简称 RBSOD。RBSOD 可以用来在同事(或者敌人)的电脑上触发假的蓝屏错误信息(BSOD)。触发的蓝屏错误不是真的，也不会导致人们丢失数据。然而却会吓他们一跳。蓝屏错误同样也可以远程清理。

RBSOD 包含两个程序。在异地机器运行的服务器，以及控制及触发蓝屏错误报告的客户端。

程序有许多选项。根据你的选择，蓝屏可以弄的和真的一样，或者它可以弄的跟开玩笑一样出现。即使是开玩笑一样出现，用户都要一会来反应过来，并且一开始会吓一跳。

RBSOD 客户端长得像这样：



提供了以下选择：

IP 地址：

指定你想要引发蓝屏错误报告的 IP 地址或者主机名。前提是异地电脑上已经安装并正在运行服务器。

如果这个为空，信息就会被在本地子网中广播(大部分情况下是你的本地局域网),然后所有安装并运行服务器的电脑都会显示蓝屏。

Message 提示信息

Message 是展现在用户蓝屏错误报告上的错误信息。默认信息是：

A fatal exception OE has occurred at 0028:C0011E36 in VXD VMM(01)00010E36. The current application will be terminated.

文本是一段真实蓝屏错误报告，所以在用户看来特别真实。

为了有意思，许多日本俳句(一种特别的日本诗体裁)也包含在内。有许多有意思的句子可以选择。这两句是我的最爱：

Windows crashed again. I am the Blue Screen of Death. No one hears your screams.

Three things are certain: Death, taxes, and lost data. Guess which has occurred.

想象下当你的同事意思到他们被捉弄后的表情。但是在你捉弄你戴博特类型的老板或者 VB 程序员的时候小心点。他们可能意识不到这是一个恶作剧。

这些信息在 messages.dat 里，你也可以添加你自己的句子。

Use a Custom Message 使用定制消息

如果这个选项被勾选了，另一个文本框会出现，你可以输入定制消息。这个选项在提供交互性或者实质的蓝屏错误信息时很有用。比如，如果你的老板某天穿了件俗气的衣服，你可能会引发这条消息。

Ugly brown suit error. I cannot continue in such company.

Show Any Key

这个选项可以用来添加恶作剧的幽默感。BSOD 最初会提示用户"Press any key to continue _"。这就像一般的蓝屏错误报告。然而如果这个选项被勾选了，在他们按了一个键后，会继续用闪烁文本提示他们"Not that key, press the NY key!"。

这个选项肯定会让德伯特老板和 VB 程序员忙活好一阵去找任何按键。这个在就要出差去机场前使用最好，或者当你想让他们忙活一阵的时候。

Show Trademark 显示商标

如果这个选项勾选了，在屏幕的底部中间会显示以下很小但是可读的文本：

* The BSOD is a trademark of the Microsoft Corporation.

Show

Show 按钮会在异地电脑引发蓝屏错误报告。

Clear

Clear 按钮可以用来遥控清除已经引发的蓝屏错误报告。一般情况下你会让用户自己清除蓝屏，但有时候你可能想要遥控的清除。

7.8.2 Server

我们先看眼服务器

安装

服务器被命名为 svchost 而不是 RBSODServer 或其他敏感的名字。这样你在别的电脑安装并隐藏它就容易的多。svchost 是个一般的可执行窗口，它经常有多份拷贝同时执行。如果你现在看看任务管理器，你很可能看到四条这个可执行程序。. 由于你会在单独路径存放这个特别版本，它不会妨碍主窗体，而会在任务管理器中表现的很平常。

服务器没有任何窗体并且不会出现在系统任务托盘上。如果你想要停止它，使用任务管理器选择登记为当前用户登录的 svchost。系统 svchosts 会登记为 SYSTEM。选定后，你就可以停止任务了。

安装程序时，一般只需要简单的拷贝编译后的服务器到目标电脑(为了方便部署，编译时不要有 packages)然后运行它。它会停留在内存中直到用户重启他们的电脑。在他们重启后，程序不会重新加载。如果你想要程序自动启动，你可以添加它到用户启动组中或者使用注册表来偷偷启动。

Source Code 源代码

服务器包含两个单元，Server.pas 和 BSOD.pas。 BSOD.pas 包含一个用来展现蓝屏的窗体。BSOD.pas 不包含任何 Indy 代码因此这里不列出来。

Server.pas 是主程序窗体并且包含单个 UDP 服务器。port 属性已经被设置为 6001，active 属性设置为了 True。当应用运行时，它会立即开始监听端口 6001 来的 UDP 数据包。

就如前面讨论的一样，UDP 就像传呼机。因此接受数据并不需要连接。数据包简单的作为一个整体到达。为每个到达的 UDP 数据包，UDP 服务器都会触发 OnUDPRead 事件。不需要其他事件来实现一个 UDP 服务器。当 OnUDPRead 事件被触发时，完整的数据包接收完成并且可以使用了。

OnUDPRead 事件传递三个参数：

1. ASender:TObject - 这是触发这个事件的组件。这只在有多个共享同个方法作为事件的 UDPServers 时有用。这很少使用或者需要。(译者注:XE10 中是 `AThread: TIdUDPListenerThread`，下面代码做相应修改)。
2. AData:TStream - 这是主要的参数，它包含一个数据包。UDP 数据包可能包含文本以及(/或者)二进制数据。因此 Indy 用 stream 代表他们。为了获得数据，简单的使用 TStream 类的 read 方法。(译者注:XE10 中是 `AData: TIdBytes`，下面代码做相应修改)。
3. ABinding: TIdSocketHandle - 这对使用绑定对象去接收数据包的高级选择很有用。只有在你创建了绑定对象的时候才有用。

RBSOD 服务器的 OnUDPRead 事件看起来是这样；

```
procedure TformMain.IdUDPServer1UDPRead(AThread: TIdUDPListenerThread;
  const AData: TIdBytes; ABinding: TIdSocketHandle);
var
  LMsg: string;
begin
  if Length(AData) = 0 then begin
    formBSOD.Hide;
  end else begin
    LMsg := BytesToString(AData);
    formBSOD.ShowBSOD(Copy(LMsg, 3, MaxInt)
     , Copy(LMsg, 1, 1) = 'T'
     , Copy(LMsg, 2, 1) = 'T');
  end;
end;
```

注意，有一个 if 语句来简称大小(size)是否为 0.发送和接收空 UDP 数据包是合法的。这是用来通知服务器去清除蓝屏的。

如果大小不是 0，数据被使用 TStream.ReadBuffer 读进局部字符串。

UDP 服务器不为每个数据包使用单独线程，所以 OnUDPRead 事件顺序发生。默认地，OnUDPRead 在主线程中触发，所以窗体和其他界面控件可以安全的存取。

7.8.3 Client 客户端

RBSOD 客户端甚至比服务器还简单。RBSOD 客户端包含一个窗体：Main.pas。Main.pas 包含

许多事件但大部分都和用户界面有关，所以你自己理解去吧。

RBSOD 客户端中和 Indy 有关的代码摘录在这里，这是从 Show 按钮的 OnClick 事件中摘出来的。

```
IdUDPClient1.Host := editHost.Text;
IdUDPClient1.Send(
 iif(chckShowAnyKey.Checked, 'T', 'F')
 + iif(chckTrademark.Checked, 'T', 'F')
 + s);
```

第一行设置了 UDP 数据包要发给的主机。端口在设计时就已经用属性浏览器设置为服务器的值 6001。

下一行使用 Send 方法来发送 UDP 数据包。因为 UDP 是无连接的，数据要不然就用多个数据包发送，要不然打包到单个数据包中。如果发送多个数据包，开发者就要负责去定位和重装配它们。这并不像看上去那么简单，所以除非你发的数据太多了，把数据打包会简单的多。

传递个 Send 的参数会在调用时立即作为 UDP 数据包发送，所有你想要发送的数据必须在单次调用中传递。

Indy 也包含一个重载的 SendBuffer 方法来用 buffer 发送数据。

在 RBSOD 里，协议简单的包含两个字符来说明是否显示商标，是否显示 Any Key，已经真正要展示的消息。

RBSOD 客户端中唯一的别的 Indy 代码在 Clear 按钮中的 OnClick 事件中，它和先前的摘要几乎一样。

8 Reading and Writing 读写

Indy 提供了许多方法来满足多种多样地读写需要。这些方法包含等待，检查状态，和 poll 的方法。

这些方法全部都属于 TIdTCPConnection 类。这是每个服务器连接的类型，还是 TIdTCPClient 的父类。也就是说你在服务器和客户端中都可以使用全部这些方法。

许多人只熟悉一些读写方法。甚至，许多人从不用核心客户端，只用协议客户端，导致可能不熟悉任何的核心方法。

记住，Indy 是阻塞式的，所以不要寻找事件来通知你何时 request 操作完成了。方法指导 request 任务完成了才会返回。如果 request 任务无法完成，就会引发异常。

这篇文档并不是个有所有方法的百科全书。要想知道所有方法那你应该查阅 Indy 帮助文件。这个部分是为了让你熟悉许多方法。

(译者注：许多方法现在要在 IOHander 或 socket 中使用如 TIdTCPClient.IOHander.Alldata)

8.1 Read Methods 读的方法

8.1.1 AllData

AllData 阻塞并收集进入的数据直到链接被中断。然后它会返回收集到的数据作为结果。AllData 对一些协议比如 Whols 很有用，因为 Whols 在返回数据后通过断连来作为数据结束的标记。AllData 在内存中缓存数据所以不应该用来存取太大的数据。

8.1.2 Capture

```
procedure Capture(ADest: TStream; const ADelim: string = '.';const AIsRFCMessage:
Boolean = True); overload;

procedure Capture(ADest: TStream; out VLineCount: Integer; const ADelim: string
= '.'; const AIsRFCMessage: Boolean = True);overload;

procedure Capture(ADest: TStrings; const ADelim: string = '.';const AIsRFCMessage:
Boolean = True); overload;

procedure Capture(ADest: TStrings; out VLineCount: Integer; const ADelim: string
= '.'; const AIsRFCMessage: Boolean = True);overload;
```

Capture 方法有许多重载形式。总之，Capture 会读数据读到自己在某行中读到了特定的分隔符。

8.1.3 CurrentReadBuffer

```
function CurrentReadBuffer: string;
```

CurrentReadBuffer 返回所有当前在 Indy 的内部接收缓存中的数据。在返回数据前，如果连接着，CurrentReadBuffer 还会试着从已连接的 socket 中读取任何能读的数据。调用 CurrentReadBuffer 会清空内部缓存。

如果没有数据，那就会返回个空字符串。

8.1.4 InputBuffer

```
property InputBuffer: TIdManagedBuffer read FInputBuffer;
```

InputBuffer 是一个 TIdManagedBuffer 实例对象的引用。InputBuffer 是 Indy 的内部接收缓存。TIdManagedBuffer 确实有许多高级方法，但正常情况下，用户不需要调用他们。

8.1.5 InputLn

```
function InputLn(const AMask: String = ''; AEcho: Boolean = True; ATabWidth:
Integer= 8; AMaxLineLength: Integer = -1): String;
```

InputLn 从服务器中读取一行并用退格符标记尾部。如果 AMask 被指定，每个接收到的字符发送给 peer 时都会被 AMask 替代，而不是真正读到的字符。InputLn 在你获得你不想展示给用户的数据的时候十分有用，比如密码输入时。

8.1.6 ReadBuffer

```
procedure ReadBuffer(var ABuffer; const AByteCount: Longint);
```

ReadBuffer 是用来将数据直接读到一个指定的内存缓存中的。如果在 Indy 内部缓存中没有足够的数据，ReadBuffer 将从 socket 中读数据。

8.1.7 ReadCardinal

```
function ReadCardinal(const AConvert: boolean = true): Cardinal;
```

ReadCardinal 从连接中读取一个无符号 32 位序列数，可以选择调整网络字符序。

8.1.8 ReadFromStack

```
function ReadFromStack(const ARaiseExceptionIfDisconnected: Boolean = True;
ATimeout: Integer = IdTimeoutDefault);
```

ReadFromStack 用来填充 Indy 的内部接收缓存。一般来说终端用户不该调用这个函数，除非他们正在实现一个新的不使用其他现存的读方法的读方法，或者他们正使用 InternalBuffer 属性和内部接收缓存直接交互。

8.1.9 ReadInteger

```
function ReadInteger(const AConvert: boolean = true): Integer;
```

ReadInteger 从连接中读取一个 32 位有符号整形数，可选择网络字符序。

8.1.10 ReadLn

```
function ReadLn(ATerminator: string = LF; const ATimeout: Integer =
IdTimeoutDefault; AMaxLineLength: Integer = -1): string; virtual;
```

ReadLn 从连接中读取数据读到发现终结符，或者超过 timeout 时间，或者达到最大单行长度。

8.1.11 ReadLnWait

```
function ReadLnWait(AFailCount: Integer = MaxInt): string;
```

ReadLnWait 函数就像有异常报告的 ReadLn，它直到发现 非\空 行时才会返回。在 AFailCount 行读到时也会返回。

8.1.12 ReadSmallInt

```
function ReadSmallInt(const AConvert: Boolean = true): SmallInt;
```

ReadSmallInt 从连接中读取一个小整形，可选网络字符序。

8.1.13 ReadStream

```
procedure ReadStream(AStream: TStream; AByteCount: LongInt = -1; const
AReadUntilDisconnect: boolean = false);
```

ReadStream 读取数据到流中。可以指定一个特定的字节数量，从流中读取，或者流可以一直读到断连。

8.1.14 ReadString

```
function ReadString(const ABytes: Integer): string;
```

ReadString 读取特定数量的字节到一个字符串中并把数据作为结果返回。

8.1.15 ReadStrings

```
procedure ReadStrings(var AValue: TStrings; AReadLinesCount: Integer = -1);
```

ReadStrings 通过换行符从连接中读取指定数量的字符串。如果字符串数量没有指定，那就会先从连接中读取一个 32 位整数，然后使用那个整数。

8.1.16 WaitFor

```
function WaitFor(const AString: string): string;
```

WaitFor 从连接中读取数据直到特定的字符。

8.2 Read Timeouts 读取时限

TIdTCPConnection(所有的 TCP 客户端，以及服务器使用的连接都从 TIdTCPConnection 继承)拥有一个叫做 ReadTimeout 的属性。ReadTimeout 指定要求的时限，单位是毫秒。默认属性值是 IdTimeoutInfinite。这个设置会使读取时限的功能无效。

Timeout 并不是一个完成的工作单元的时限。它是指一个空闲时限。意思是说，如果超过了 ReadTimeout 属性指定的时间然而却没有数据可以写，那一个 EIdReadTimeout 异常就会被抛出。

经常的，网络连接很慢或者无法传输数据，但然而保持着有效连接。这种情况下，连接可能慢的根本无法使用而成为服务器的累赘并且对客户端没用。

为了解决这个问题，Indy 通过使用 TIdTCPConnection 的 ReadTimeout 属性实现了读取时限。ReadTimeout 默认为 0，也就是弃用了读取时限。为激活读取时限就要指定一个以毫秒为单位的值。

在任何读取调用中，如果超过指定的读取时限还没有数据从连接中读到，一个 EIdReadTimeout 异常就会被引发。时限不只应用于在时限内就接收到的数据的数据请求，

而是所有数据。那就是说，如果你请求读取 100 字节的数据，读取时限设置为 1000 毫秒(1 秒)，读取操作仍然有可能超过 1 秒。只有在 1 秒内一个字节都没被收到，EIdReadTimeout 异常才会被引发。

8.3 Write Methods 写方法

8.3.1 SendCmd

```
function SendCmd(const AOut: string; const AResponse: SmallInt = -1): SmallInt;
overload;
function SendCmd(const AOut: string; const AResponse: Array of SmallInt): SmallInt;
overload;
```

SendCmd 被用来发送文本命令并解析一个 RFC 风格的数字答复。

8.3.2 Write

```
procedure Write(AOut: string);
```

Write 是 Indy 里最基础的输出方法。Write 发送一个 AOut 参数给连接。Write 并不用任何方式改变 AOut 参数。

8.3.3 WriteBuffer

```
procedure WriteBuffer(const ABuffer; AByteCount: Longint; const AWriteNow:
Boolean =False);
```

WriteBuffer 允许直接写内存缓存。如果 AWriteNow 为 True，那如写缓存正在使用中，它将被旁通。

8.3.4 WriteCardinal

```
procedure WriteCardinal(AValue: Cardinal; const AConvert: Boolean = True);
```

WriteCardinal 往链接中写一个无符号 32 位序列数，可选地调整网络字节序。

8.3.5 WriteHeader

```
procedure WriteHeader(AHeader: TStrings);
```

WriteHeader 往链接中写一个 TStrings 对象并依次转换'=' 为 ':' 。WriteHeader 在写完 TStrings 对象后还会写一个空行。

8.3.6 WriteInteger

```
procedure WriteInteger(AValue: Integer; const AConvert: Boolean = True);
```

WriteInteger 血一个有符号 32 位整数到链接中。可选择网络字符序。

8.3.7 WriteLn

```
procedure WriteLn(const AOut: string = '');
```

WriteLn 和 Write 的功能基本一样，除了它在 AOut 参数后面还加了一个换行符(EOL)。

8.3.8 WriteRFCReply

```
procedure WriteRFCReply(AReply: TIdRFCReply);
```

WriteRFCReply 写一个 RFC 风格 数值+文本 答复 使用指定的 TIdRFCReply 对象。

8.3.9 WriteRFCStrings

```
procedure WriteRFCStrings(AStrings: TStrings);
```

WriteRFCStrings 用 RFC 格式写指定的 TStrings 对象并在每一行最后都加上一个 '.' ；来作为结束符。

8.3.10 WriteSmallInt

```
procedure WriteSmallInt(AValue: SmallInt; const AConvert: Boolean = True);
```

WriteSmallInt 向链接中写一个小整数，可选网络字符序。

8.3.11 WriteStream

```
procedure WriteStream(AStream: TStream; const AAll: Boolean = True; const
AWriteByteCount: Boolean = False; const ASize: Integer = 0);
```

WriteStream 向链接中写指定流对象。 WriteStream 包含许多参数来指定写流对象的哪一个部分，并可以选择写入链接的字符总数。

8.3.12 WriteStrings

```
procedure WriteStrings(AValue: TStrings; const AWriteLinesCount: Boolean =
False);
```

WriteStrings 向链接中写一个 TStrings ，它和 ReadStrings 是成对的。

8.3.13 WriteFile

```
function WriteFile(AFile: String; const AEnableTransferFile: Boolean = False):
Cardinal;
```

WriteFile 是个直接往连接中写文件的函数。WriteFile 使用操作系统优化来将文件写入 socket ，这比简单的使用一个 TFileStream 加上 SendStream ，有更好的表现。

8.4 Write Buffering 写缓存

TCP 必须用数据包来发送数据。数据包的大小是可变的，但一般是稍微比 1KB 大一点。但是，如果 TCP 等待整个数据包的数据，由于数据并没有发送在许多情况下答复是无法返回的。所以 TCP 可以使用整个被请求的数据包并且在发送前并不需要整个数据包，一个叫做 Nagle Coalescing 的算法会被使用。Nagle Coalescing 在内部缓存数据直到到达应该有的数据包大小，或者一个内部计算的时间超过了。这时间范围通常很小，是以毫秒为单位的。

发送许多小块的数据可能会导致算法混乱或导致它发送太多的数据包。因为每个数据包也有间接开销(?overhead)，这浪费了带宽并降低了传输速度。

数据可以在字符串或其他东西中缓存并立即发送，然而这会提高你代码的要求并很可能改变你发送数据的方式，因为在缓存中并不是所有的写方法都可用。这也会使你的代码更复杂并提高内存使用率。

替代地，你可以使用 Indy 的写缓存特征。通过使用写缓存，你可以让 Indy 来缓存开放的数据并允许你一般可以使用 Indy 的所有写方法。

为了开始写缓存，调用 WriteBufferOpen(译者注:原文中为 OpenWriteBuffer,这里改为 XE10 中的方法,后面也直接修改)并定义一个缓存大小(属性 SendBufferSize)。然后你就可以调用所有的 Indy 的写函数，并且所有输出都被缓存直到到达缓存大小。每次到达缓存大小，缓存就会被写入链接然后清空。如果没有定义缓存大小，所有数据都会被缓存，直到手动刷新。

有许多写缓存方法来管理缓存。

WriteBufferClear 清除当前的缓存并保持缓存开启。WriteBufferFlush 刷新当前内容并保持缓存开启。

为结束写缓存，调用 WriteBufferCancel 或 WriteBufferClose。WriteBufferClose 写完所有剩余的数据然后结束写缓存，但 WriteBufferCancel 直接关闭缓存而不会传输剩下的数据。

8.5 Work Transactions 工作事件

Work Transactions 被用来定义工作单元。他们被叫做 Work Transactions 是由于他们可能被封装，并且读写操作也可能同时地被分离。Work Transactions 一般被用作进度条展示或者传输状态展示。

Transactions 在 Indy 中被用多种方式预定义，比如 TIdHTTP.Get, WriteStream。作为个用户，你也可以通过 BeginWork, DoWork 和 EndWork 定义里自己的 transactions。

8.5.1 OnWork Events

OnWork 事件包含三个事件并被用来交流 工作事件 的状态。这些事件是: OnWorkBegin, OnWork and OnWorkEnd。

当一个工作事件开始，OnWorkBegin 事件就会被触发。OnWorkBegin 事件声明 Sender, WorkMode, 和 WorkCount(XE10 中 AWorkCountMax)。Sender 是工作事件当前应用的链接。WorkMode 声明这是一个读工作事件还是写工作事件。读和写工作事件可以同时发生，并且

工作事件可以被封装。WorkCount 指明工作事件的大小。在许多工作事件中，大小无法被预先知道并且在这种情况下，WorkCount 的值为 0。否则，WorkCount 声明了字节的数量。通常，这个事件是用来准备进度条。

然后，一系列 OnWork 事件被触发。OnWork 事件声明 Sender, WorkMode, 和 当前 WorkCount(XE10 中 AWorkCountMax)。这个事件对于更新进度条很有用。

当工作事件完成了，OnWorkEnd 事件被触发。OnWorkEnd 甚至只声明了 Sender 和 WorkMode。这个事件对标志进度条完成很有用。

8.5.2 Managing Your Own Work Transactions

管理自己的工作事务

你也可以通过调用 BeginWork, DoWork 和 EndWork 创建你自己的工作事务。参数和之前那些事件一样。如果你封装了工作事务，Work 调用就会自动管理封装。

为实现一个工作事务，首先调用 BeginWork 并传递工作事务的大小如果知道的话。然后调用 DoWork 来更新进度条。最后结束时调用 EndWork。

9 Detecting Disconnects 探测断链

因为Indy天然是阻塞的而且它的事件只与状态相关,没有事件去提示一次提早的中断连接。当一次过早断链发生时，如果一个读或写调用正在进行，一个异常就会被引发并能被捕捉。如果没有进行中的读或写调用，就不会引发异常直到下一次读或写调用。

的确有一个 OnDisconnected 事件，然而这并不是你想的那样。OnDisconnected 事件只有在 Disconnect 方法被调用后才会触发。这不是一个告诉你过早断链事件的事件。

9.1 Saying Good Bye 说再见

在 TCP 命令基础协议中，最简单的方式就是说个再见。这只有在正常链接的情况下才会被探测到，但确是个很大的帮助。

为了说再见，协议使用 QUIT 命令。当客户端准备断链，它会先向服务器发送一个 QUIT 命令，而不是简单的直接断链。然后它会等待一个答复，然后自己中断 socket。

这是个好行为，它允许双方都能合适的断链。不这样做就像打电话的时候一方突然挂断了。你不知道发生了什么，然后只能猜测，然后也挂断电话。

不同的是服务器可能有成千上万的客户端。如果他们都不通知服务器就断链，服务器将会有许多"死"链接等着解决。

9.2 Do you really need to know? 你真需要知道吗

许多程序员立刻质疑这个事实,争辩说他们需要在过早断链时立刻得知。你可能听过这句话：“如果一棵树在森林中倒下了，当时并没有人在场并听到，那它发出了声音了吗？”所以，如果一个 socket 断链了，并且它并不是正在存取中，它是否关闭了真的重要吗？大部分情况下，回答是不重要。如果一个 socket 过早断链并无法存取了，没有有异常会引发，没有事件会触发直到需要存取 socket。

这可能看起来很怪，但是这就和我们正在存取一个文件的情况十分相似。想象力你打开了软盘上的一个 excel 电子数据表。你正在操作文件，但是它被存在了内存中，并且 excel 不正在当前内存中读写。在过程中有时你忘了你开启了电子数据表，然后你直接移走了软盘。之后你继续操作你的电子数据表。一切都没问题，直到你要保存你的更改。只有那个时候你才会收到错误。尽管 Excel 在那个软盘上打开了一个文件，Excel 在你移走软盘的时候并不会收到一个"EAnIdiotRemovedTheFloppyDis-kException"错误。

9.3 I need to know now! 我要立刻知道

注意到异常可能不会立即引发。比如，如果你拔掉了网线，异常并不会立即发生，而会过一分钟或者更久。TCP/IP 是由美国军方设计来作为额外的网络协议来抵抗核打击。因为它的设计，网络将从链接另一端等待超时，并尝试重连。你想立即探测到断链是违背 TCP/IP 的设计的。这种探测可以实现，但理解为什么默认不实现这个功能 对你是十分重要。你对这的理解会帮助你去合适的实现断链立刻探测。

如果你考虑它的话，大部分情况下你将会发现这个行为是可接受甚至需要的(不立刻通知断链)。然而有些情况下，知道是否链接丢失是十分重要的。想象你正在实现心跳监控器和远程监控的接口的情况。这是一种你确是想非常非常快地知道是否链接丢失的情况。

如果你需要即时的断链通知，你需要在协议中实现这种探测。

9.3.1 Keep Alives

Keep alives 是立即探测断连的一种方法。Keep alives 是这样实现的：链接的一端按照一定的间隔发送一条消息比如 NOOP(No Operation 无操作)到另一端并等待一个答复。如果在规定时间内没有收到答复，那链接就会被认为有问题并且中断。时限一般十分短并且由正在使用的协议来确定。如果你正在实现到一个心跳监控器的链接，定义的时间区间应该比你如果在实现个啤酒瓶温度监控的时候定义的短。

实现一个 Keep Alives 系统还会获得一个额外的优点。在许多情况下，即使程序停止答复和停止对请求的服务，一个 TCP 链接还是会保持有效。在另一种情况下，程序可能会继续服务请求但是由于一些问题，可能速率十分慢，或者降低了网络带宽。Keep Alives 将会探测到这种情况，并标记其为不合适使用，即使链接仍然有效。

Keep Alives 可能会以有规律的间隔发送，或者根据协议，只在需要检测状态的时候发送。

9.3.2 Pings

Pings 的实现和 Keep Alives 很相似，除了它们并不答复请求。Pings 按一定的时间间隔从链接的一端发送到另一端。一端成为了广播员，另一端成为了监视器。如果监视器从上一次 ping 到达开始计算，已经经过了一定的时间还没有新的 ping 到达，链接就会被认为不适合并且终结。

Pings 可以被当做一种起搏或者心跳，如果它停止或者减慢了，那就是有问题了。

9.4 EIdConnClosedGracefully

许多 Indy 用户得被由 Indy 服务器引发的 EIdConnClosedGracefully 异常弄的很火，特别是 HTTP 和其他服务器。EIdConnClosedGracefully 是一个表示链接已经被由另一端故意关闭的异常。这和坏链不同，坏链会导致一个连接重置错误。如果另一端已经关闭了链接，然后 socket 再被读或写入，EIdConnClosedGracefully 将被 Indy 引发。这和企图读或写一个在你不知道的情况下被关闭的文件的情况很类似。

在某些情况下，这是个真的异常，你的代码需要处理它。在其他情况下(特别是服务器)这是一个处理协议的正常部分，Indy 会为你处理这个异常。即使 Indy 捕获了它，当在 IDE 中运行时，调试器会先被引发。你可以简单的按 F9 来继续，Indy 将会处理异常，但是调试中的不断的停止也很烦人。当 Indy 捕获到异常时，你的用户并不会在程序中看到一个异常，除非它在 IDE 中运行。

9.4.1 Introduction 介绍

9.4.2 Why Does This Exception Occur in Servers?

当客户端已连接到服务器后，一般有两种方式来处理断链：

1. 相互协定(Mutual Agreement) - 通过一方接收到要断连的提示(另一端可选地告知已收到)，两端都同意断链，然后两边都准确的断连。
2. 单方断连(Single Disconnect) - 断连，然后让另一端注意。

用相互协定的方法时，双方都知道什么时候断连并且都准确的断连。大部分会话协议(conversational protocols)比如 Mail(邮件)，News(新闻)用这种方式断连。但客户端准备断连时，它给服务器发送个命令来告知它将要断连。服务器答复一个已知要断连的回执，然后服务器和客户端两方都断连。这种情况下 EIdConnClosedGracefully 异常不应该引发，如果一方引发了异常，那其实就是个错误，应该解决掉。

在相互协定的某些情况下不会发送任何命令，当时双否都知道什么时候另一方会断连。经常是，客户端连接，提交一个命令，从服务器接收一个答复，然后就断连。这时尽管客户端没有明确的指令告知断连，但协议已经说明了链接应该在答复一个命令后被断掉。一些时间协议(time protocols)就是例子。

在单方断连中，一方就是直接断连了。另一方要自己去探测到链接中断然后采取合理的行为来终结会话。在实现使用这种方式断连的协议的时候，你会常常见到 EIdConnClosedGracefully 异常。这是个异常，但是 Indy 知道并且会为你处理它。whois protocol 就是一个例子。客户端连接到服务器然后发送一个包含域名的字符串作为请求。服务器然后发送答复当答复结束时断连。除了正常的断连，没有发送给客户端其他信号。

HTTP 协议对两个方式都允许，这就是为什么 HTTP 服务器看到 EIdConnClosedGracefully 异常十分正常了。HTTP 1.0 的运作方式和 WHOIS 协议十分相似，HTTP1.0 中服务器在答复客户端的请求后简单的直接断连。于是客户端每次发送请求都得重新建立连接。

HTTP1.1 允许单个连接发送多请求。然而却没有命令来通知断连。客户端和服务器在一次请求后的任何时候都可以断连。当客户端断连了然而服务器仍在接收请求，一个 EIdConnClosedGracefully 异常就会引发。在 HTTP1.1 的大多数情况下，客户端是断连的那个。当服务器实现部分 HTTP1.1 的时候，它会断连，但是不支持 keep alive。

9.4.3 Why is it an Exception? 为什么这是个异常？

许多用户曾觉得应该通过返回特定的返回值而不是一个抛出异常来通知这种断链的情况。然而这在这种情况下是个错误的方式。

EIdConnClosedGracefully 异常是从核心函数中引发的，然而当这个函数被调用时通常是在许多方法内层的。EIdConnClosedGracefully 实际上是个异常，而且在大部分情况下需要被最外层的函数捕获。最合适的处理方式是个异常。

9.4.4 Is it an Error? 这是个错误吗？

并不是所有异常都是错误。许多开发者被教或者默认所有的异常都是错误。然而并不是这样的，这就是为什么他们叫做异常而不是错误。

Delphi 和 C++ Builder 使用异常来简洁地处理错误。然而除了错误，异常还有其他用法。EAbort 异常是一个非错误的异常的例子。像这类异常被用来修饰标准程序流并通讯信息给外层函数来捕获。Indy 也用这种方式运用异常。

9.4.5 When is it an Error? 什么时候它是个错误

当 EIdConnClosedGracefully 异常在客户端中被引发时，它是个错误并且你应该捕获并处理这个异常。

在服务器中他是个异常。然而有时这是个错误，有时不是。对许多协议，这异常是协议正常功能的一部分。因为这个一般性行为，如果你不在服务器代码中捕获 EIdConnClosedGracefully 异常，Indy 将会。它将会标志链接被关闭然后停止赋值给链接的线程。除非你自己处理这个异常，否则 Indy 会处理它并自动为你采用合适的动作。

9.4.6 Simple Solution 简单处理

因为 EIdConnClosedGracefully 异常是特定服务器的一般性异常，它从 EIdSilentException 异常继承。在 Debugger Options(调试器选项)中的 Language Exceptions 标签(Tools Menu 中)，你可以添加 EIdSilentException 到忽略异常的列表中。在添加进列表之后，异常将仍然在代码中发生并且被处理，但是调试器将不会停止程序来调试它。

10 Implementing Protocols 实现协议

Indy 实现了全部的常见协议已经许多没那么常用的协议。还有你需要自己实现协议的情况。最常见的实现一个协议的理由是你需要为没有现存协议的一些东西定制。

第一步是理解将要被实现的协议。有三个基础类型的协议：

1. **标准(Standard) -** 这是指那些网络标准的协议。为了明白协议，只要简单的发现 RFC's 和协议的联系。RFC's 将会详细说明协议。
2. **定制(Custom)** - 定制协议是用在当没有现存协议满足需要的功能的时候。在后文将会讲到怎么创建一个定制协议。
3. **Vendor -** 这指的是那些和专有系统或者设备交互的协议。Vendor 协议是 vendor 作为一个定制协议开发的协议，你与他交互。如果你在实现 Vendor 协议的话，我祝你好运，因为他们大部分都是由硬件设计者实现的，那些设计者差不多只在大学上过堂 C++的课，还没有任何实现协议的经验。

大部分协议都是会话式的(conversational)而且是纯文本，除非你有合理理由，你的全部协议也应该是这样的。在后面的部分中有两个专有名词。

建立个客户端或服务器的第一步是理解协议。对于标准协议，只要阅读合适的 RFC 就行了。对于定制协议，必须建立个协议。

大部分协议都是会话式的(conversational)而且是纯文本。会话式的是指，给一个命令，答复一个状态值以及一个可能的数据。领域很局限的那些协议常常不是会话式的，但仍然常识纯文本。纯文本让协议更容易调试，以及与不同的编程语言和操作系统接口。

状态码传统上是个三位数字。没有哪个标准定义了状态码或者他们的惯例，但是实际上许多协议遵从以下惯例：

- 1xx - 消息的
- 2xx - 成功
- 3xx - 临时性错误
- 4xx - 永久性错误
- 5xx - 内部错误

典型地，每个错误都对应一个特定的数字，但是有一些协议重复使用数字并且为每个命令提供独一无二的数字答复而不是为每个协议。

10.1 Protocol Terminology 协议专有名词

在谈论协议的实现之前，我会说并解释一些专有名词：

10.1.1 Plain Text 纯文本

Plain text 定义所有命令和答复以七位 ASCII 格式。尽管用二进制答复数据是很常见的，几乎所有的协议命令和答复是纯文本的。除非有一个更好的理由，否则就不该用二进制来发送命令或答复。

使用纯文本能让调试和测试很简单。它也确保了操作系统和编程语言间的内部可操作性。

10.1.2 Command 命令

命令是在客户端和服务器间传输以请求信息或进行某些操作的文本字符串。命令的例子比如：HELP，QUIT，LIST。

命令也可能包含些可选参数，由空格分开。

比如：

```
GET CHARTS.DAT
```

GET 是个命令，CHARTS.DAT 是个参数。在这个例子中，只指定了一个参数，但是命令可能会有许多参数。一般的，每个参数都由一个空格分开，这和在 DOS 窗口或 command shell 中输命令很相似。

命令常常是英文编码的。这可能看起来很有偏见，但是必须要选一个共同点，英语不管在技术世界还是在商业世界中都是最通用的语言。如果不使用英语的话，协议就没有那么有用了。

命令本地化的一个例子就在 Microsoft Office 中。Microsoft Office 可以被自动使用 OLE automation。然而微软本地化了命令的名字和对象的属性。这就是说，如果你使用美国版本的 Microsoft Office Automation 写一个应用，你的软件在法国就会无法运行，因为在法国的用户使用的是法国版本的 Microsoft Office。

10.1.3 Reply 答复

一个 reply 就是一个用来答复一个被提交的命令的简短答复。reply 包含了是否成功执行了请求命令的状态信息，以及有时还包含了一些数据。

比如，如果提交了命令 *GET customer.dat*，可能就会返回一条答复：*200 OK*，说明命令被理解了并且将被执行。答复一般只有一行，但是也可以有很多行。

但是和命令不同的是，reply 的正文部分可能本地化为其他语言，只要它遵从 7 位 ASCII。这是因为协议本身依赖于数字部分，文本部分是展现给终端用户的或者为了调试。

10.1.4 Response 答复

一个 response 是对命令的答复的数据部分。Responses 是可选地并且并不是所有的命令都会有 Response。Responses 发生在 reply 被发送之后，这样命令的发送者就知道是否会有一个 Response，以及 Responses 的格式。

Responses 可能是文本或二进制。如果是文本的，通常是 RFC Response 格式。

10.1.5 Conversations 会话

大部分协议是会话式的。有一些简单的不是，但他们通常仍然是纯文本的。

会话是说协议的命令结构是这样的：

1. 给命令
2. 返回 状态 reply
3. 可选的数据 response

根据邮政编码查询的那个例子，会话长得像这样：

```
Client: lookup 37642 77056
Server: 200 Ok
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

下面把会话分成独立的部分：

命令：

```
Client: lookup 37642 77056
```

Reply：

```
Server: 200 Ok
```

Response：

```
Server: 37642: CHURCH HILL, TN
Server: 77056: HOUSTON, TX
Server: .
```

这里每一项后面部分都会详细讲。

10.2 RFC Definitions

这部分定义的专有名词实际上并没有标准。但是这些专有名词都基于个几乎所有 RFC 基础协议都遵循的"准标准"

一个明显的例外是 POP3。为什么设计者决定把它设计的和其它 RFC 都不一样的原因是个秘密。POP3 协议实际上限制更多并且并没有提供与协议有关的更多的功能。它是个完全多余的东西。

IMAP4 是继承 POP3 的协议，它在不遵循标准方面比 POP3 有过之而无不及，这真尴尬。IMAP4 并没有被广泛使用，POP3 仍然是标准的邮件协议。

除了这两个明显的例外之外，文本协议都坚持已存的"准标准"。这个"准标准"是发送命令，replies，然后 response 的标准方式。

10.2.1 RFC Status Codes　RFC 状态码

RFC 状态码是指这种形式：

```
XXX Status Text
```

XXX 是从 100 - 599 的数字。

这三位数字包含了 reply 的意思并被用来在运行时探测一条命令的输出结果。通常，紧跟着的可选的文本是用来调试时给用户看的。在这种情况下 response 常常是英文的但是当使用的是 7 位 ASCII 码时可以被本地化。在某些数据很短的情况下，数据就直接跟在可选文本里了。这种情况下，数据必须还是 7 位 ASCII，但是协议本身决定了语言和格式限制。在大部分情况下，它是数据和语言中立的。举一个例子，如命令是"TIME"，在返回结果的可选文本里是"15:30"。

一个 RFC reply 看起来可能是这样：

```
404 No file exists
```

404 是数字 response ，"No file exists" 是可选文本信息。在这时程序实际上只知道 404，由于协议中明确定义了 404 是这个意思。文本信息几乎只是为了调试，记录日志以及展现给用户看。文本信息可能会随着实现的变化而变化并且可能会被本地化为其他语言。无法合适地由 7 位 ASCII 码展现的语言必须被音译为英文字符。

状态码的数字部分可以被赋值为任何意思。但是常见的会话遵循如下规则：

```
1xx - Informational      信息
2xx - Success            成功
3xx - Temporary Error    暂时性错误
4xx - Permanent Error    永久性错误
5xx - Internal Error     内部错误
```

数字通常是独一无二的，但并不总是这样。那是说，如果你赋值 201 为 "文件没有找到"，许多命令可能会收到 201 reply 并且总是同个意思。在一些罕见的情况下，数字的意思依赖于提交的命令。那是说，对于每种数字答复，每个命令都对每个数字 reply 赋予了特别地意思。

以 00 结尾的数字代码，如 100,200…… 是为没有特定意义的通用答复保留的。200 通常简单地意思为"OK"。

状态码也可能有多行来包含更大的文本答复。这种情况下，答复的多行的每一行在数字码后都包含一个破折线，除了最后一行是个空格。

多行状态码的例子：

```
400-Unknown Error in critical system
400-The server encountered an error and has no clue what caused it.
400-Please contact Microsoft technical support, or your local
400-tarot card reader who may be more helpful.
400 Thank you for using our products!
```

10.2.1.1 Examples

这有些从 HTTP 协议中提取的例子状态码。正如你看到的，他们与第一个数字的分类一致：

```
200 Ok
302 Redirect
404 Page not found
500 Internal Error
```

如果你看到了 *500 Internal Error*，有可能你在使用 Microsoft IIS。

10.2.2 RFC Reply

一条 RFC reply 是条作为 RFC 状态码返回的 reply。

10.2.3 RFC Response

RFC response 是个由一个句点的单行来结束的文本 response。如果数据包含了包含单个句点的一行，这行为了传输目的被转换为两个句点，并在接收的时候被转换回去。

当返回的数据量未知的时候，RFC responses 非常常见。它被 HTTP，Mail，News 和其他协议使用。

支持 RFC responses 的 Indy 方法是 Capture(为了接收)和 WriteStrings(为了发送)。

10.2.4 RFC Transactions RFC 事件

RFC Transactions 是个包含命令，reply，和可选 response 的会话，都是以 RFC 格式的。命令句柄和 Indy 的其他部分是围绕 RFC Transactions 建立的。

例子事件：

```
GET File.txt
201 File follows
Hello,

  Thank you for your request, however we cannot grant your
request for funds for researching as you put it in your
application "A better mouse trap".

  Thank you, and please do not give up.
.
```

10.3 TIdReplyRFC(原文中是 TIdRFCReply,XE10 中改为这)

TIdReplyRFC 帮助发送和接收 RFC Replies。TIdReplyRFC 有三个主要属性：NumericCode, Text 和 TextCode。NumericCode 和 Code(原文中是 TextCode)是相互专有的。Code 是个处理如 POP3 和 IMAP4 的协议的属性。

为生成一个 reply，设置 NumericCode(或者 Code)属性以及可选地文本到 Text 属性中。Text 是 TStrings 类型的以允许多行答复。

TIdReplyRFC 有用来写合适的格式化 replies 的方法，也用来转变文本为一个 TIdReplyRFC 实例。

TIdReplyRFC 是用来发送答复给命令的，还用在 TIdTCPServer(子类 TIdcmdTCPServer 中)的 ExceptionReply (原文 ReplyException)， ReplyUnknownCommand (原文 ReplyUnknown)和 Greeting 属性中。

10.4 ReplyTexts

reply 中的数字代码对每个错误都是特定的。实例的 HTTP 协议使用 404 表示"Resource not found(未找到资源)"。许多不同的命令都允许以 404 作为一个错误来返回，但是 404 总是代表同个错误。为了避免每次每次为 404 错误都复制次文本，TIdcmdTCPServer 也有一个 ReplyTexts 属性。

ReplyTexts 是一个 TIdReplyRFC 实例的 collection，可以在运行时或设计时管理。ReplyTexts 是用来维持一个对应每个数字代码的文本列表的。当在 TCPServer 中使用的 TIdReplyRFC 只有数字代码却没有文本时，Indy 会查询 ReplyTexts 中对应的那项并使用它的文本。

所以与其像下面这样每次需要 404 的时候都包含文本：

```
ASender.Reply.SetReply(404, 'Resource Not Found'); 啊啊啊啊啊啊啊啊啊啊？？？
```

你可以这样写：

```
ASender.Reply.NumericCode := 404;
```

在 Indy 真的发送答复前，它将从 ReplyTexts 中对应的入口设置答复的 Text 属性。这允许全部答复文本保存在一起而且方便管理。

10.5 The Chicken or the Egg?

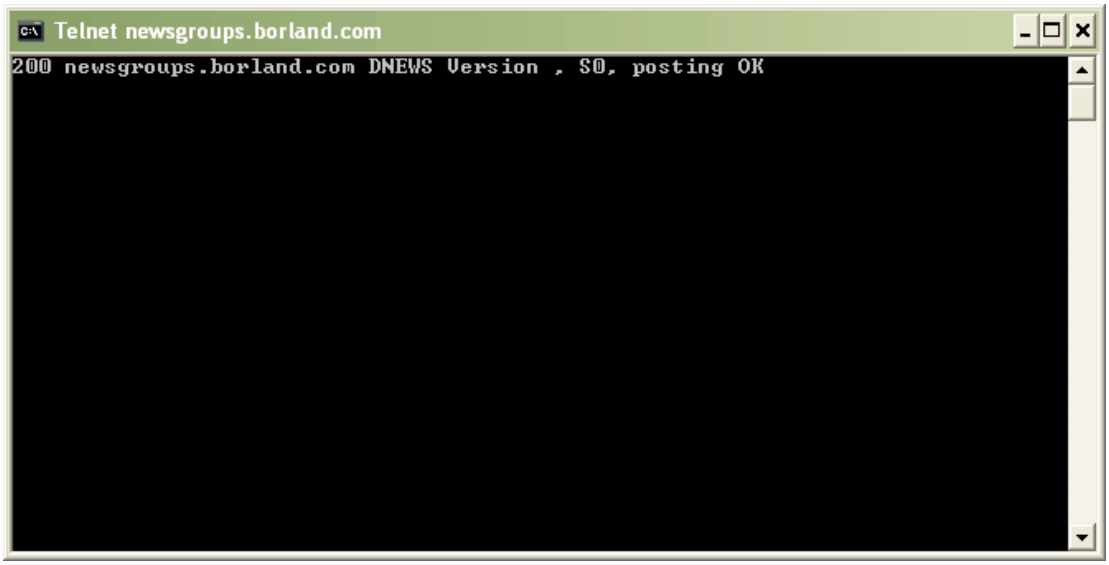
当建立个既要建立客户端又要建立服务器的系统时，你很可能会有以下问题。"我应该先建哪一个，客户端还是服务器?" 两个都需要另一个来相互测试。

答案比你想的简单的多。先建服务器会简单一些。为了测试客户端，你需要个服务器。为了测试个服务器，你需要个客户端。但是由于几乎所有协议都是文本基础的，客户端可以简单地用 telnet 应用来模拟。

为了测试，连接到已知服务器的端口。命令行中输入：

```
Telnet newsgroups.borland.com 119
```

然后输入 enter。当连接后，你应该看到类似于以下的屏幕。



Windows 95/98/ME, and NT 与 Windows 2000 或 Windows XP 可能看起来稍微差点，但是结果是一样的。其余版本的 Windows 以一个新应用来启动 telnet，有一个单独的窗口。上面的图像是 Windows XP 的 telnet 客户端，是一个线程应用。

命令"Telnet newsgroups.borland.com 119" 命令 telnet 客户端连接到在端口 119 的服务器 newsgroups.borland.com。端口 119 是 NNTP(新闻)的端口。刚刚一个 telnet 客户端被用来连接到 Borland 新闻服务器，telnet 客户端可以用在任何文本基础的协议的服务器。

为了从新闻服务器断连，输入"Quit"然后按回车。

10.6 Defining a Custom Protocol 定义个定制协议

网络开发者的任务不止是与已存的系统进行交互，还包括完全新的一个。这种情况下，需要创建个新的协议。

建立一个客户端或服务器的第一步是理解协议。对于标准协议，这只要读合适的 RFC 就行了。如果协议并不是标准的或者完全定义好的，那就必须先定义个。

当定义一个协议，以下决定应该先做：

- 文本还是二进制命令?除非有一个重载的需要，否则就用文本命令。文本命令更易于理解以及调试。
- TCP 还是 UDP?这真的是依赖于正在建立中的协议和需要。研究两者的特征并小心的决定。在大部分情况下 TCP 是正确的选择。
- 端口 — 每个服务器应用都需要个专用的端口用来监听。1024 以下的端口是预留的，并且除非是实现个赋值了 1024 以下的端口的协议，那就不应该被用。

一旦问题被决定了，命令，replies 和 response 需要被设计。

10.7 Peer Simulation

传统地，唯一的建立客户端和服务器的方式是先建立服务器，或者平行地建立他们。但是 Indy 有种你可以不需要另一个而单独建立服务器或客户端的方法。这使得任何一个都可以被先建

立。在一些情况下，其中一个可能需要在没有另一个的情况下被建立。在这种情况下可以用 peer simulation。在调试部分会讲 peer simulation。

10.8 Postal Code Protocol

这部分将更加深入的讲那个在邮政编码客户端部分讲的邮政编码协议。之后会展示服务器。

这个工程被设计的尽可能的简单。邮政编码(在美国叫做 Zip code)查询就是让一个客户端向服务器询问某邮编是哪一个城市或州。

服务器使用的样例数据是美国的邮政编码，zip codes。这协议也可以用于其他邮政编码，但是美国邮编在例子建好后就可以读了。

为不在美国的人，也就是不熟悉 zip code 的人简要介绍下。zip code 是美国的邮政编码，每个对应了一个邮寄区域。Zip codes 是五位数字。Zip codes 还可以跟着附加的 4 位数字，比如 16412-0312。这叫做 Zip+4。这四个附加数字说明了当地的邮寄信息，并不是用来定位具体城市用的。

对于邮政编码协议，先确定以下这些：

- Text commands 命令是文本的
- TCP 使用TCP
- Port: 6000. Port 6000 is a port commonly used in Indy for demos. It has no significance. 端口使用 6000。6000 是 Indy 的例子默认使用的端口，没有具体意义。

邮编编码协议将支持以下命令：

- Help
- Lookup <Post Code 1> <Post Code 2> ...
- Quit

当设计一个协议的时候，最好先熟悉最基本的协议，比如 NNTP，SMTP 和 HTTP 并用他们作为模板。别管 POP3 和 IMAP4。他们两是设计协议的很好的反面教材。

由于 NNTP 协议支持在一个协议中发送和接收信息，为了你好，这篇文档中将会常常提到它。

10.8.1 Help 帮助

Help 是一个很常见的被实现的命令，它对于自动的客户端基本没有用。Help 对于那些正在测试或者手动和服务器交互的人很有用。几乎所有的服务器都实现了某种形式的基本帮助。

Help 是用来确定服务器支持那些命令以及可能的扩展命令。

这是 Borland 新闻服务器的 Help 命令的答复的示例：

```
help
100 Legal commands
  authinfo user Name|pass Password
  article [MessageID|Number]
  body [MessageID|Number]
  check MessageID
  date
  group newsgroup
```

```
  head [MessageID|Number]
  help
  ihave
  last
  list [active|active.times|newsgroups|subscriptions]
  listgroup newsgroup
  mode stream
  mode reader
  newgroups yymmdd hhmmss [GMT] [<distributions>]
  newnews newsgroups yymmdd hhmmss [GMT] [<distributions>]
  next
  post
  slave
  stat [MessageID|Number]
  takethis MessageID
  xgtitle [group_pattern]
  xhdr header [range|MessageID]
  xover [range]
  xpat header range|MessageID pat [morepat...]
.
```

对于邮政编码协议，服务器也应该答复 100，在加上合适的文本。

10.8.2 Lookup 查询

lookup 命令接收一或多个邮政编码用以查询，然后对应的城市。数据是以 RFC 答复格式返回的。找不到对应的信息的，就不会返回。对一个成功的查询的答复是"200 OK"。

例子：

```
lookup 37642 16412
200 Ok
37642: CHURCH HILL, TN
16412: EDINBORO, PA
.
```

即使一个对应的信息都没有找到，lookup 命令也会得到答复"200 OK"。

```
lookup 99999
200 Ok
.
```

这是一个设计时的决定。如果 lookup 命令只接受一个参数的话，那最好是当找到了对应的城市的时候返回 200，没找到的时候就返回 4XX。但是由于这时 lookup 可以返回一部分的有效数据，所以最好总是返回 200.

返回部分数据的例子：

```
lookup 37642 99999
200 Ok
37642: CHURCH HILL, TN
.
```

如果协议返回了错误的代码，有效数据就会被忽略了。这个设计决定使得服务器能够返回有效数据而不受到错误邮编的阻碍。

10.8.3 Quit 退出

quit 命令十分的直白。Quit 就是说客户端将要结束会话然后断连。

再看看 Borland 新闻服务器，它这样答复的：

```
quit
205 closing connection - goodbye!
```

邮编协议的答复很相似：

```
quit
201-Paka!
201 2 requests processed.
```

邮编协议答复多行。这不是由协议本身定义的。任何 RFC 答复都可以是一行或者多行的。Indy 自动转换两种类型。

11 Proxies 代理

常常有人问：“我怎么用 Indy 来写代理呢？”。尽管大家都想要有个很简单的解决方案。但并没有这种方案，因为代理的种类是在太多了。一些协议甚至还有他们自己的谈判代理方法。后面会讲解最常见的代理类型。

代理和防火墙的角色很相似但是目的却不同。因为他们角色类似，他们可以相互表现另一方的功能。常常他们作为防火墙代理直接结合在一起。

代理可以可以被分为两类：

- 透明
- 非透明

11.1 Transparent Proxies 透明代理

透明代理就是那些不需要改变使用代理的协议的代理。透明代理常常在开发者或者用户不知情的情况下存在。

尽管透明代理不会影响客户端，他们却会影响服务器。在大部分情况下，使用这种代理的服务器对外界是隐藏的。为了让代理的另一端能够够得到服务器，端口要从外部映射到内部。

11.1.1 IP Masquerading / Network Address Translation (NAT) IP 伪装/网络地址转换

IP 伪装或者网络地址转换(NAT)代理允许所有向外的连接透明地发生，并且对客户端没有任何影响。客户端的行为很正常并且不再需要特别去设置啥。

微软网络连接分享(Microsoft Internet Connection Sharing)就是用这个方法实现的。

11.1.2 Mapped Ports / Tunnels 映射端口/隧道

映射端口或者隧道代理的工作方式就像在个阻塞的路上建个隧道。路被堵住的原因可能是网络设置，可能是网络间的桥连，也可能是故意阻塞的用以保护或作为内网的防火墙。

内网的定义是本地网络存在的那一边网络，外网的定义是内网的目标网络。

由于道路阻塞，所有访问都必须直接穿过映射端口。端口被赋值给一个可以访问外部世界的本地服务器。这个服务器于是在外网和内网间来回传递数据。映射端口的缺点是每个端口都被映射到个固定的异地服务器和端口。对于像邮件协议和新闻协议这类可以预先知道服务器的协议，它工作地很好。然而对于像 HTTP 这样的协议，这个方法就无法采用了，因为无法预先知道异地服务器的位置。

映射端口也可以用来从外网映射端口到内网中。映射端口也常常和 NAT 代理联合使用以使得服务器暴露给外网。

11.1.3 FTP User@Site Proxy

有多种方法实现 FTP 代理。最常见的 FTP 代理类型叫做 User@site。

使用 User@site 方法，所有 FTP 会话被连接到一个本地代理服务器。代理伪装成一个 FTP 服务器。代理服务器拦截并解释 FTP 请求。当代理要求用户名而不是仅仅发送用户名(?)，用户名以及想要的 FTP 服务器被以 用户名@ftp 站点 的形式被发送。代理然后连接到想要的 FTP 站点并解释传输命令。

对于每个传输命令，代理都动态地映射一个本地端口以用于数据传输，以及修改返回给客户端的传输命令。FTP 客户端和代理对话而不是直接访问真实的 FTP 服务器。由于转换，FTP 客户端并不知道经过了哪些代理。

比如给你个 FTP 站点 ftp.atozedsoftware.com, 用户名 joe 密码 smith ，一个正常的 FTP 会话会被设置成这样：

```
Host: ftp.atozedsoftware.com
User: joe
Password: smith
```

如果存在 User@Site 代理，代理的主机名是 corpproxy，FTP 会话会设置成这样：

```
Host: corpproxy
User: joe@ftp.atozedsoftware.com
Password: smith
```

11.2 Non Transparent Proxies 非透明代理

在使用非透明代理的时候需要改变协议。许多协议为特定的非透明代理做了准备。

11.2.1 SOCKS 防火墙安全会话转换协议

SOCKS 代理是个不需要改变更高层次协议的代理，它工作在 TCP 层。为了让协议使用 SOCKS 代理，软件必须在 TCP 层实现它。

如果软件不明确地支持 SOCKS 代理，它就没法使用 SOCKS 防火墙。许多主流软件比如 browsers 和 ICQ 都支持 SOCKS，但是大部分软件不支持。因此，SOCKS 的部署必须联合内部服务器，映射端口，其他代理，或者一些混合。

为了软件能够支持 SOCKS，软件并不直接连接到目标服务器而是先连接到 SOCKS 代理。然后它传递一个包含目标服务器及可能有的身份验证信息的数据记录给代理。SOCKS 服务器然后动态的为其开辟个隧道以连接到目标服务器。

因为 SOCKS 协议是通过连接后传递给它的记录动态地运作的，SOCKS 是高度可配置的，使用十分的灵活。

11.2.2 HTTP (CERN)

HTTP 代理，有时指代 CERN 代理，是一个只代理浏览器通信的特别代理。除了 HTTP 外，如果 FTP 是基于 HTTP 的，它也可以代理 FTP 通信。HTTP 代理也可以提供缓存，它常常单单用于这个用途。

许多企业环境有保护他们内部网络的完整的防火墙，只允许通过 HTTP 和邮件访问外部世界。邮件使用内部邮件服务器来提供，HTTP 使用个 HTTP 代理来提供。因此 HTTP 已经成为了"与防火墙最友好"的协议，这也是为什么那么多较新的协议都使用 HTTP 来传输。SOAP 和 Web services 就是例子。

12 IOHandlers

在不需要直接修改 Indy 源代码的情况下，Indy 就可以以许多方式实现定制和扩展。一个可扩展性的例子就是 IOHandlers。IOHandlers 使你能够用 Indy 使用任何 I/O 源。当你想要使用一个可选 I/O 机制或者创建一个新的传输机制的时候，你应该使用 IOHandlers。

IOHandlers 拥有 Indy 所有的 I/O(输入/输出)操作。Indy 在 IOHandler 之外不提供任何自己的 I/O 操作。
IOHandler 被用来为 Indy 组件发送和接收未加工的 TCP 数据。

IOHandlers 使得 Indy 的类可以被定义用它来处理输入输出。一般所有的 I/O 操作都是通过 socket 由默认的 IOHandler，TIdIOHandlerSocket，处理。

Indy 中每个 TCP 客户端都有个 IOHandler 属性，这属性每次连接服务器的时候都可以被赋值一个 IOHandler。如果没有指定一个 IOHandler，一个 TIdIOHandlerSocket 实例就会被隐式地自动创建然后被 TCP 客户端使用。TIdIOHandlerSocket 使用一个 TCP socket 来实现 I/O 操作。Indy 还有另外的 IOHanders：TIdIOHandlerStream 和 TIdSSLIOHandlerSocket。

可以创建其他的 IOHandler 以让 Indy 使用几乎任何你能想的到的 I/O 源。目前 Indy 对 I/O 源只支持 sockets，streams 和 SSL，但是 IOHandler 允许其他许多可能性。尽管还没有这样计划，IOHandlers 可以实现对 Tunneling，IPX/SPX，RS-232，USB 或 Firewire 的支持。Indy 不限制你的 I/O 选择，而且通过使用 IOHandlers，Indy 允许你选择你需要的任何 I/O 操作。

12.1 IOHandler Components　IOHandler 组件

12.1.1 TIdIOHandlerSocket

TIdIOHandlerSocket 是默认的 IOHandler，如果没有指定 IOHandler，就会隐式地创建个 TIdIOHandlerSocket 来使用。TIdIOHandlerSocket 处理和 TCP socket 有关的所有 I/O 操作。

通常，除非需要高级设置，TIdIOHandlerSocket 不会被专门创建来使用。

12.1.2 TIdIOHandlerStream

12.1.2 TIdIOHandlerStream 是用来调试和盒子测试(box testing)的。使用它的话，存在的 TCP 会话与服务器交互时可以被记录。之后，作为盒子的一部分，会话可以被"重播"。Indy 组件并不知道他们其实不是在和一个有真实连接的真正服务器会话。

这是除了 QA 测试工具外的另一个非常好用的调试工具。如果一个客户出了问题，可以发送个特别的创建，或者开启调试特性来记录会话。通过使用日志文件，你就可以在本地调试环境中重建客户的会话。

12.1.3 TIdSSLIOHandlerSocket(TIdSSLIOHandlerSocketOpenSSL)

TIdSSLIOHandlerSocket 被用来实现对 SSL 的支持。一般加密和压缩 handler 会通过 Intercepts 而不是 IOHandlers 来实现，但是，Indy 使用的 SSL 库(OpenSSL)接受个 socket 句柄并在库内完成 socket 通讯而不是通过翻译接受到的数据然后用 Indy 来发送。因此，它是作为一个 IOHandler 实现的，因为库的确做了 I/O 操作。TIdSSLIOHandlerSocket 是 TIdIOHandlerSocket 的子类。

12.2 Demo - Speed Debugger　例子-速度调试器

速度调试器展示了怎么模拟缓慢的连接。这对于调试和模拟缓慢的网络连接比如调制解调器以测试你的应用的行为十分有用。

速度调试器由一个主窗体和一个定制 IOHandler 组成。速度调试器使用一个映射的端口组件来为个指定的 web 服务器代理 HTTP 通信。然后浏览器连接到速度调试器，速度调试器从指定的 web 服务器获取个网页，但是发回给网页浏览器的时候会以一个指定的速度节流(减速)数据。

文本框(text box)被用来指定个 web 服务器。注意到它没有指定个 URL，因此没有包含 http:// 或个网页引用。它只指定了服务器主机名或 IP 地址。如果你有个运行中的本地 web 服务器，你可以设置主机 127.0.0.1 来使用本地 web 服务器。

组合框(combo box)被用来选择链接的限速，包含以下选项。模拟的限速写在圆括号中。

- Apache (Unlimited 无限)
- Dial Up (28.8k baud)
- IBM PC XT (9600 baud)
- Commodore 64 (2400 baud)
- Microsoft IIS on a PIII-750 & 1GB RAM (300 baud)

当 Test 按钮被按下，速度调试器将以 URL http://127.0.0.1:8081/. 启动默认浏览器。这将导致浏览器从速度调试器来发送它的请求。速度调试器在端口 8081 监听以避免与任何现存 web 服务器冲突。

速度调试器可以在 Indy Demo Playground 下载。

12.2.1 Custom IOHandler 定制 IOHandler(并没有 Recv 方法)

速度调试器的工作是由个定制 IOHandler 完成的。映射的端口组件有个 OnConnect 事件，这个事件是被用来使映射端口创建的每个向外客户端勾住我们的 IOHandler 的。这个事件看起来是这样的：

```
procedure TformMain.IdMappedPortTCP1Connect(AThread: TIdMappedPortThread);
var
  LClient: TIdTCPConnection;
  LDebugger: TMyDebugger;
begin
  LClient := AThread.OutboundClient;
  LDebugger := TMyDebugger.Create(LClient);
  LDebugger.BytesPerSecond := GSpeed;
  LClient.IOHandler := LDebugger;
end;
```

定制的 IOHandler 类被命名为 TMyDebugger，它是从默认的 TCP socket IOHandler-TIdIOHandlerSocket 继承的。因为 TIdIOHandlerSocket 已经实现了所有的现行 I/O 操作。TMyDebugger 只需要降低数据传输速率到指定的速度就行了。这是通过重写它的 .Recv 方法实现的。

在 .Recv 方法中，继承的 .Recv 被调用来接收数据。然后根据指定的速度限制，计算出了相应的延迟。如果算出的延迟比调用继承的 .Recv 所用的时间大，就调用 IndySleep 方法来弥补时差。这听起来好像很复杂，但是它真的很简单。 .Recv 方法是这样的：

```
function TMyDebugger.Recv(var ABuf; ALen: integer): integer;
var
  LWaitTime: Cardinal;
  LRecvTime: Cardinal;
begin
  if FBytesPerSecond > 0 then begin
    LRecvTime := IdGlobal.Ticks;
    Result := inherited Recv(ABuf, ALen);
    LRecvTime := GetTickDiff(LRecvTime, IdGlobal.Ticks);
    LWaitTime := (Result * 1000) div FBytesPerSecond;
    if LWaitTime > LRecvTime then begin
      IdGlobal.IndySleep(LWaitTime - LRecvTime);
    end;
  end else begin
    Result := inherited Recv(ABuf, ALen);
  end;
end;
```

13 Intercepts

一个 Intercept 在一个 IOHandler 的更高等级，它被用来独立于源或者目标来修改或者捕获数据。Intercepts 被用来写日志，调试，和加密。它还能够用于压缩或者进行数据分析。

Intercepts 在 Indy9.0 中变化极其的大。在 Indy8.0 中 Intercepts 可以提供一些 IOHandler 做不到的功能。Intercepts 可以变换已经接收到的数据或者在发送数据前修改它。因为新的 IOHandler 类可以处理所有的 I/O 操作，所以 Intercepts 已不再支持任何的 IOHandler 功能了。

Intercepts 仍然还可以进行数据转换并且是以一个比在 Indy9.0 中更加灵活的方式。Indy8.0 的转换能力有许多的限制，比如数据要保持同样的尺寸。这导致了压缩无法通过 Intercept 来实现，限制 Intercepts 用于记录日志和不改变数据大小的压缩。

Intercepts 的工作方式是在 IOHandler 接收到数据之后再修改数据，或者在数据发送给 IOHandler 前修改它。Intercepts 当前被用来实现写日志和调试组件。Intercepts 还可以被用来实现加密，压缩，数据收集，或者带宽限制。

13.1 Intercepts

Intercepts 截取向内及向外传输的数据，以使得数据能被记录或者修改。Intercepts 能在对内数据被从网络读取后，但在返回给用户前修改它。Intercepts 还使得向外数据在被用户接受后，但还没传输前修改它。Intercepts 能被用来实现日志，加密和压缩。

客户端的 Intercepts 是以连接为基础的，每个连接分别拥有。服务器也可以通过赋值给独立的链接来使用。

请注意，Indy9 中的 Intercepts 和 Indy8 中的不同。在 Indy8 中，Intercepts 即扮演了 Intercepts 的角色又扮演了 IOHandler 的角色。这导致很难分离 Intercept 和 IOHandler 的功能。在 Indy8 中 Intercepts 还不嗯呢该够改变数据的大小，因此无法实现压缩。

13.2 Logging

Indy8 有一个可以用于不同目标的 log(日志)组件。Indy9 的 logging 组件目前基于一个新的公有的 logging 类，被定义为 logging 类型。这公用的 logging 类还加入了一些属性和功能，比如它能够在记录数据的同时记录对应时间。

所有的 logging 类都是基于 Intercepts 实现的。也就是说，他们在数据被输入流读取后捕获进入的数据，或者在数据被写入输出流前捕获输出的数据。

有这些 logging 类：

- TIdLogEvent - TIdLogEvent 在数据被发送、接收或者产生一条状态消息时触发事件。TIdLogEvent 不需要实现一个新的logging类就能够实现定制日志。
- TIdLogFile - 把日志信息记录到日志文件中。
- TIdLogDebug - 把日志信息在Windows中记录到调试窗口，在Linux中记录到命令行窗口。它还把数据分别标志为接收的数据，发送的数据和状态信息。TIdLogDebug对于简易的调试十分有用。
- TIdLogStream - 并不像其他logging类那样注释或者标记数据。它做的几乎只是把原始数据写到指定的流中。TIdLogStream有许多用途，但它尤其对于QA测试和异地调试特别有用。

你也可以自己定制log类。

14 Debugging 调试

比起服务器，客户端通常调试器来容易的多。客户端只处理单个连接，并且常可以用常见的调试技术来调试。在这个部分会测试一些对服务器和客户端都很有用的小贴士。

14.1 Logging 记日志

一个十分简单的不用追踪代码就能知道客户端发生了什么事情的方法是使用 TIdLogDebug 或者 TIdLogFile 。TIdLogDebug 将会把日志直接记到调试窗口，这可以很方便的实时查看客户端正在发送还是接收什么数据。如果你不需要实时的查看数据通信，那就使用 TIdLogFile。在客户端结束运行后，你可以查看文件中的内容以得知在会话中发生了什么。

14.2 Peer Simulation 对等模拟

可能有的时候你需要模拟一个还无法访问的客户端或者服务器。这可能是为了安全，带宽或者其他原因。这种情况下你可以使用 Peer Simulation。Peer Simulation 亦可以用来在建立服务器前建立客户端，或者创建测试脚本

Peer Simulations 的实现是通过使用个 TIdIOHandlerStream，然后赋值输出流为一个文本文件，并把输入流置空。这会使得 TIdIOHandlerStream 从文本文件中读取全部的数据来发给另一端，并忽略所有的另一端发回的数据。

如果你赋值了输入流，它就会记录另一端传来的数据。

14.3 Record and Replay 记录及答复

能够记录并回放会话是一个真的非常有用的特性。这对于回归测试(regression testing)和异地调试同样也是极其有用的。如果你有一个异地的客户，你可以发给他一个特别构建的版本或者在你软件上加个开关来记录会话。这样他们就可以发给你记录文件，然后你就可以模拟他们的客户端或服务器，而不用真的连接到他们的服务器。

为了完成这功能，使用个 TIdLogStream 来把接收的数据记录到个文件中。你还可以记录客户端发送的数据到另一个文件中，但你其实并不需要这样，除非你想要手动看它们。当你收到了文件，你可以附加个 TIdIOHandlerStream 控件到客户端上。

IOHandlers 还有另一个封装的用途。那用途就是 记录及答复。一个进行中的会话可以使用 log 控件来记录，然后之后使用流 IOHandlers 来回放。想象力的一个客户那出了些问题，但你无法重新产生他发生的问题又无法上门服务。你可以让他记录个完整的会话到文件中，然后使用流 IOHandlers 在你的开发用机器上完整的回放会话。Indy 团队正开始使用这方法作为他们 QA 程序和盒子测试的一部分。我打算在未来直接在安装时中实现这个。

15 Concurrency 并发性

在个多线程环境中，资源必须受到保护，以免因同时被多于一个线程访问而导致资源出问题。

并发性和多线程是相互纠缠的，决定应该先学哪一个十分的困难。这部分会先讲并发性，这也将有助于你对多线程的理解。

15.1 Terminology 术语

15.1.1 Concurrency　并发性

并发性指的是许多任务同时发生。当并发被实现的很好时，它会被认为很"和谐"。实现的不好时，"混乱"。

大部分情况下，一个任务就是个线程。但是任务也可以是程序(processes)或纤程(fibers)。

线除以二常常正好是一(The line dividing the two is often a fine one?),关键是使用合适的技术。

15.1.2 Contention

Contention 到底是什么？ Contention 就是当两个以上任务试图同时访问单个资源的时候。

如果你在一个大家庭中长大的话，那肯定能理解 Contention，我也就可以举个简洁明了的例子。想象下，有个有六个孩子的家庭，一天，妈妈放了个小比萨到桌子上作为晚餐，然后会发生什么？这就是 Contention。

无论何时多个并行任务需要读写访问同个数据，都需要保护好那个数据的完整性。如果不控制好访问，当一个任务想要读个变量的同时另一个却想要写它,两个或多个任务就可能会"崩溃"。如果一个任务在另一个任务读取的时候进行写操作，读操作的那个可能会取回写了一部分的数据，也就是无效数据。但是这不会引发任何异常，只会很直接的之后在程序中导致错误。

Contention 问题在轻量级实现中常常不会发生，因此常常在开发中不会发生。因此在开发中要使用合适的技术和进行负载测试(load testing)。否则就好像在玩俄罗斯轮盘似的，问题会在开发中随机的发生，但是部署后频繁的发生。

15.1.3 Resource Protection　资源保护

Resource protection 是种防止 contention 问题的方法。Resource protection 的功能是，在给定时间只允许一个任务访问一个特定资源。

15.2 Resolving Contention　解决 contention

无论何时多个并行任务需要读写访问同个数据，都需要保护好那个数据的完整性。这对于不熟悉多线程的程序员是个陷阱。然而，大部分服务器不需要全局数据。全局数据就是那些在程序启动时初始化后只需要读取的数据。只要没有写访问，多线程读取全局数据就不会有任何的副作用。

常见的解决 contention 的方法后面会讲。

15.2.1 Read Only 只读

最简单的方法就是只去读它。任何只读的简单类型(integers,strings,memory)都不需要任何保护。这也适用于一些更复杂的类型，比如 TLists。只读的类类型也是十分安全的，只要它们没使用任何可以读写的全局或域(field)变量。

还有，可以在不可能有读操作的时候去写资源。也就是说可以在启动时初始化资源，那时候那些需要访问它的任务都还没启动。

15.2.2 Atomic Operations 原子操作

有一种方法论说进行原子操作的时候资源不需要被保护。原子操作就是种因为太小而无法被电脑中央处理器分开的操作。由于它的小尺寸，它不会有 convention 问题，因为它会被单独执行而不会在执行中被任务切换。特别地，原子操作就是编译到单个汇编指令的几行源代码。

典型地，像那些对一个整型或者布尔型域的读写任务被认为是原子操作，由于它们被编译为单个移动指令(move instruction.)。但是我建议你不要依赖于原子操作，因为在一些情况下，甚至写一个整型或布尔型都可以牵扯不止单个操作，这取决于数据一开始是从哪里读来的。还有，这依赖于内部编译器知识，这些知识可能在你不知晓的情况下改变。依赖于在源代码等级的原子操作会产生那些有未来不确定性，和可能在不同处理器和不同操作系统表现很不同的代码。

有许多人为激烈地为原子操作辩护。然而有个显著的即将出现的事情支持了我的观点，那就是 .net。当你的源代码先被编译为 IL(一种中间语言)，然后再在不同的平台上分别编译为机器语言，老实说，你确定你的那行源代码最终还会是个原子操作吗？

这是你的选择，并且支持和反对原子操作的人都有很多。依赖原子操作大部分情况下只节省了几微妙和几字节的代码。我强烈地反对原子操作，因为他们就这么点好处却有巨大的代价。不要把任何操作当做原子操作来用。

15.2.3 Operating System Support 操作系统支持

许多操作系统支持最基本的线程安全(threadsafe)的操作。

Windows 支持叫做联锁功能(Interlocked functions)的功能集。这功能的用处很有限，由一些简单的整型操作如 increment, decrement, add,swap 和 swap-compare 组成。

功能的数量随 Windows 版本的变化而变化，在较老版本的 Windows 中可能会发生死锁(deadlocks) 。在许多应用中他们对性能的帮助很小。

综合考虑它有限的使用，变化的支持情况，和极小的性能帮助，我建议你还是使用 Indy 自带的线程安全功能吧。

Windows 还包含特别的对 IPC(interprocess communication 进程间通信)对象的支持，Delphi 已经封装了它。这些对象和 IPC 一样对多线程极其有用。

15.2.4 Explicit Protection 显式保护

显式保护会让每个任务都知道资源被保护了，在访问那个资源前要执行明确的步骤。一般，这种代码是在个由许多任务并发执行的单程序中的，或者被封装在个由许多不同地方调用的程序中并表现的像个线程安全封装。

显式保护一般需要使用个资源保护对象(resource protection object)。简而言之，一个资源保护对象会限制单个资源同时只能被一个任务访问。资源保护对象并不是真的限制了对资源的

访问。如果它要这么做，那它就必须详细知道每个资源的类型。实际上它就好像个交通信号灯，代码被改为遵守并提供输入数据到交通信号灯。每个资源保护对象使用不同的逻辑，不同的输入，不同的开销以实现不同类型的交通信号灯。这使得可以选择不同的资源保护对象以更好的配合不同类型的资源和情况。

资源保护对象存在于许多窗体中，后面会单独讲它。

15.2.4.1 Critical Sections 临界区

临界区可以被用来控制访问全局资源。临界区是轻量级的，在 VCL 中在 TCriticalSection 里实现。简而言之，临界区使得多线程应用中的单个线程能够临时的阻塞其他想要使用同个临界区的所有线程。临界区就像个交通信号灯，这信号灯只在前方的路上没有任何交通工具的时候变绿。临界区可以用来确保任何时候只有一个线程在执行某块代码。因此，被临界区保护的语句块应该尽可能的小，因为如果用的不好的话他们可能会严重的影响性能。因此，每个语句块应该分别使用自己的 TCriticalSection，而不是全部写在单个大的 TCriticalSection 中。

用 Enter 方法来进入临界区，Leave 方法来离开临界区。TCriticalSection 还有 Acquire 和 Release 方法分别和 Enter 和 Leave 方法做完全一样的事。

想象有个服务器需要记录客户端里的日志信息并把信息展现在主线程中。一个选择是使用同步。但是如果同时记录许多客户端的话，使用这个方法会对连接线程的性能产生负面影响。根据服务器的需求，更好的选择是记录日志信息然后使用一个 Timer 来让主线程读取信息。下面的代码是个利用临界区的这种技术例子。

```
var
  GLogCS: TCriticalSection;
  GUserLog: TStringList;

procedure TformMain.IdTCPServer1Connect(AContext: TIdContext);
var
  s: string;
begin
  // Username
  s := AContext.Connection.IOHandler.ReadLn;
  GLogCS.Enter; try
    GUserLog.Add('User logged in: ' + s);
  finally GLogCS.Leave; end;
end;

procedure TformMain.Timer1Timer(Sender: TObject);
begin
  GLogCS.Enter; try
    listbox1.Items.AddStrings(GUserLog);
    GUserLog.Clear;
  finally GLogCS.Leave; end;
end;

initialization
  GLogCS := TCriticalSection.Create;
  GUserLog := TStringList.Create;
finalization
  FreeAndNil(GUserLog);
  FreeAndNil(GLogCS);
end.
```

在 Connect 事件中，username 在进入临界区前被读入临时变量。这是为了避免缓慢的客户

端链接阻塞临界区内的代码。这使得在进入临界区前先进行网络通讯。为保证最佳性能，临界区内的代码被保持绝对小。

Timer1Timer 事件由个在主线程中主窗体上的 Timer 触发。时间间隔可以缩短以提供个更频繁的更新，但是会默默地减速链接的接收。如果日志功能被扩展服务器的其他部分而不只是记录连接的用户，瓶颈的可能性就更大了。时间间隔越长，更新用户界面花费的时间越短。但是许多客户端根本没有用户界面，就算有用户界面的那些，用户界面也是次要的，比服务客户端的重要性低的多，所以长时间间隔是个可接受的权衡之计。

15.2.4.1.1 Note to Delphi 4 Standard Users 提示 Delphi4 标准用户

TCriticalSection 在 SyncObjs 单元中。Delphi4 标准版本中并不包含 SyncObjs 单元。如果你在使用 Delphi4 标准版，你可以在 Indy 网站上找到个没有完全实现 Borland 的 SyncObjs.pas 的 SyncObjs.pas 文件，但它实现了 TCriticalSection 类。

15.2.4.2 TMultiReadExclusiveWriteSynchronizer (TMREWS)

在之前的例子中，TCriticalSection 被用来保护对全局数据的存储。在那些情况下，全局数据总能被更新。但是，如果全局数据有时被作为只读访问，使用 TMultiReadExclusive-WriteSynchronizer 可能会更加有效率。TMultiReadExclusiveWriteSynchronizer 是个很长并很难读的类。因此，他将简单的被叫做 TMREWS。

使用 TMREWS 的优势是它允许多线程同步的读取，尽管表现的像个临界区并且在读的过程只允许一个线程访问。缺点是 TMREWS 的使用代价更大。

除了 Enter/Acquire 和 Leave/Release 方法。TMREWS 还有方法 BeginRead，EndRead，BeginWrite 和 EndWrite。

15.2.4.2.1 Special Notes on TMREWS 注意事项

在 Delphi6 之前 TMultiReadExclusiveWriteSynchronizer 有一个问题，那就是如果从读锁(read lock)升级到一个写锁(write lock)可能导致死锁(dead lock)。因此你决不能使用这个从读锁升级到写锁的特性，尽管文档上说可以这么做。

如果你需要这个功能，那有一个迂回的方法。这方法就是先释放读锁，然后获得写锁。但是一旦你获得了写锁，你必须再检查次你一开始想要写锁的前提条件。如果那条件没变，那就继续，否则就直接立刻释放写锁吧。

TMultiReadExclusiveWriteSynchronizer 在 Delphi6 中还有特别的使用条件。所有版本的 TMultiReadExclusiveWriteSynchronizer 包括 update pack 1 和 update pack2 中的都有个严重的会导致死锁的问题。目前还没有解决方法。Borland 意识到了这个问题并已经发布了非官方补丁并准备提交官方补丁。

15.2.4.2.2 TMREWS in Kylix

Kylix1 和 Kylix2 中 TMultiReadExclusiveWriteSynchronizer 的内部实现使用了个临界区，并且不会比使用临界区有任何优势。但是它还是存在，这样代码就可以同时给 Linux 和 Windows

用。在未来版本的 Kylix 中，TMultiReadExclusiveWriteSynchronizer 可能会跟在 Windows 中的表现一样。

15.2.4.3 Choosing Between Critical Sections and TMREWS

在临界区和 TMREWS 中选择

因为 TMREWS 有一些问题，我的建议是简单的不要使用它。如果你确是决定去使用它，你应该先确定这真的是更好的选择，并且你已经打了补丁来减少发生死锁现象。

大部分情况下，合理的使用 TCriticalSection 只会稍微比最快的结果慢一点，有时候甚至更快。学习去优化你需要使用 TCriticalSection 的部分，因为 TCriticalSection 用的不好的话会严重影响性能的。

资源保护的关键是使用多种资源控制器并保持锁住的部分小些。只要可以的话，就应该使用临界区而不是 TMREWS，因为临界区更轻量和更快。通常，总是使用临界区，除非你有充分的理由使用 TMREWS。

当满足以下条件时，TMREWS 类的工作效率更高：

1. 访问既有读又有写。
2. 主要是读访问。
3. 锁维持的时间被扩展，并且不能拆分成更小的部分。
4. TMREWS 类已经打了补丁，或者知道能良好的工作。

15.2.4.4 Performance Comparison 性能比较

就像前面说的，临界区更轻量，因此更快。临界区由操作系统实现。操作系统用很快和轻量的汇编指令来实现它们的。

TMREWS 类更加复杂，因此开销更大。它必须管理请求者列表来合理的管理双状态锁定机制(dual state locking mechanism)。

为了演示他们的不同，我建了个例子工程叫做 ConcurrencySpeed.dpr 。它测试了以下三项简单的性能：

1. TCriticalSection – Enter and Leave
2. TMREWS – BeginRead and EndRead
3. TMREWS – BeginWrite and EndWrite

进行这些测试的方式是在一个计数循环内运行一定的时间。为了测试的目的，默认为循环十万次。在我的测试里，得出的结果如下(单位毫秒)：

```
TCriticalSection: 20
TMREWS (Read Lock): 150
TMREWS (Write Lock): 401
```

当然咯，测试结果和机器配置有关。但是重要的是他们之间的差异，而不是具体的数字。可以看得出，TMREWS 中较快的读锁是临界区的 7.5 倍慢，而写锁甚至慢了 20 倍。

还应该注意，尽管临界区只进行了一个测试。TMREWS 在同时使用时性能还会下降。这里进行的测试只是简单的一个循环，TMREWS 并不需要处理其他的请求者或者已存在的锁。在真实情况下，TMREWS 会比这里显示的数字还慢。

15.2.4.5 Mutexes 互斥器

互斥器的功能和临界区几乎一样。区别就是互斥器是临界区的一个加强版，有更多特性，因此也要更多开销。

互斥器的附加功能包括可以命名，赋值安全属性，跨进程的访问。

互斥器可以被用来在线程间通信，但是这极少会用到。互斥器被设计，同时也是一般被用来在进程间通信。

15.2.4.6 Semaphores

semaphore 和互斥器很相似，但是不像互斥器只允许一个参加，semaphore 允许多个参加者(entrants)。可以参加的参加者的数量在 semaphore 被创建时指定。

想象下，互斥器就是个守卫银行存取款机(ATM 机)的保安。同时只能有一个人使用取款机，互斥器要维持同时想要使用取款机的一排人的秩序。

这种情况下，相对的 semaphore 就是安装了四台 ATM 机。这样，保安就能同时允许 4 个人进入使用 ATM 机，但不能多于 4 个。

15.2.4.7 Events 事件

Events 是可以被用来跨进程或线程通知某些事情已经发生的信号。Events 可以用来在某事完成或者需要某个行为时通知另一个任务。

15.2.5 Thread Safe Classes 线程安全类

线程安全类是特别设计来保护特定类型资源的类。线程安全类分别实现了个特定类型的资源，并且完全知道这资源的功能和结构等。

线程安全类可以像线程安全整型一样简单，也可以像线程安全数据库那么复杂。线程安全类使用线程安全对象在内部实现他们的功能。

15.2.6 Compartmentalization

Compartmentalization 是个隔离数据并把它赋值为仅被某个任务使用的进程。在服务器中，Compartmentalization 常是原生的，以使每个客户端由个专门的线程处理。

当 Compartmentalization 不是原生的，就应该评估有没有可能。Compartmentalization 常是有可能的，通过复制全局数据，使用数据，然后返回结果到全局区域。通过使用 Compartmentalization ，数据只在初始化时，任务完成后，或批量更新时被锁住。

16 Threads 线程

多线程编程唬住了许多程序员，它对多线程编程新手是一大障碍。多线程可以优雅地解决许多问题，一旦掌握了它，它就会成为你技能列表里一个强有力的技术。光多线程这个话题本身就可以轻轻松松写一本书。

16.1 What is a Thread? 什么是线程

线程是一个独立的优先执行路径。使用多线程使得不同的执行路径被同时执行。

想象你的电脑处理单独一条电话线路。因为只有一条电话线路，一个时刻只有一个人能使用它。但是，如果你安装多个电话线路，其他的人就可以同时打电话了，而不会因为有人在用了而用不了。多线程使得你的应用同时做多于一件事情。

即使你只有一个 CPU，也可以使用多线程技术。虽然这时实际上每时每刻只有一个线程在执行，但是操作系统会主动暂停线程然后切换到其他的线程。每次切换后，每个线程只会执行个非常短的时间。这使得每秒能运行成千上万的"碎片(slices)"。因为线程间切换是主动的和无法预测的，对于软件来说，多个线程看起来是被并行的执行的，于是软件必须对这采取预防措施。

在多 CPU 系统，线程们可以真正地平行地执行，但是每个 CPU 实际上仍然只能同时执行一个线程。

16.2 Threading Advantages 多线程的优势

使用多线程相较于非多线程设计有许多优势。

16.2.1 Prioritization 优化

独立线程的优先级(priorities)可以调整。这使得单独的服务器连接或者线程的客户端能够占用更多或更少的 CPU 时间。

如果你同时提高了程序的所有线程的优先级，效果不会很明显，因为它们仍然是同样的优先级。它们可能会得到其他进程的线程的时间。当你设置线程优先级的时候要注意不要把它们设的太高，否则可能会妨碍处理硬件输入输出的线程。

大部分情况下，调整线程优先级的时候，你会降低它而不是提高它。这使得没那么重要的任务把时间让给重要的那些。

线程优先级对服务器也很有用，有些情况下，你可能希望根据登陆信息来调整线程的优先级。如果一个管理者或者 CEO 登陆了服务器，你可能会想提升他们线程的优先级在其他用户之上。

16.2.2 Encapsulation 封装

使用多线程使得每个任务被安全地包含，也没那么可能介入其他任务甚至获得数据。

如果你曾经使用 Windows3.1 你应该还记得一个做得不好的应用可以多么容易的崩溃整个系统。多线程预防了这种情况。多线程不止做了这些事情，它直接影响了编程。如果没有多线程，全部的任务都将在同个代码路径下完成，这额外增加了复杂性。有了多线程后，每个任务可以被分到分别独立的部分，使得当多个任务必须同时执行的时候，你的代码更容易编写。

16.2.3 Security 安全性

每个线程都可以有它自己独立的安全特性，基于身份认证或者其他标准。这在服务器实现中特别地有用，在服务器中每个用户都有与他们的链接相关的特定线程。这使得操作系统为用户实现合理的安全保护，合理地限制他们对文件和其他系统对象的访问。如果没有这个特性，你将不得不重新实现安全性，很可能导致灾难性的安全漏洞。

16.2.4 Multiple Processors 多处理器

如果有多个处理器的话，多线程的使用会自动的利用它们。如果不用多线程，你的应用程序只会有一个线程，主线程。一个线程只能同时被一个 CPU 执行，因此你的应用的执行速度没办法很快。

其他进程就像操作系统一样会使用其他 CPU。你的程序发起的对操作系统的调用命令内部实现是多线程的，你的应用会从这获得一个小的加速。还有，因为其他的 CPU 可以处理其他的应用并减少了每个 CPU 的负荷，你的应用占用的 CPU 时间会更长。

最好的利用多 CPU 的优点的方式就是把你的应用弄成多线程的。这不止会让你的应用在 CPU 间被合理的分配，还能让你的应用因为包含了更多线程而获得更多的 CPU 时间。

16.2.5 No Serialization 无序列化

多线程提供真正的并行。没有多线程的话，全部请求必须被单个线程处理，因此每个要执行的任务必须被破碎成可以被快速地执行的小块。如果任何任务部分(task part)阻塞或占用了执行时间，全部其他任务部分将等到它执行完毕才能被执行。每当任务部分执行完了，就会执行下一个，以此往复。

有了多线程，每个任务都可以被编成一个完整的任务，操作系统会在任务间分配 CPU 时间。

16.3 Processes vs. Threads 进程 vs. 线程

进程和线程不一样，但他们两个经常混淆。进程是一个完整的新的应用的实例，这实例包含了运行个可执行程序需要的全部开销，包括操作系统管理开销和额外的内存。进程的优点是他们完全的相互隔离，而线程只是部分地相互隔离。如果一个进程崩溃了，其他进程不会受到影响。

16.4 Threads vs. Processes 线程 vs. 进程

线程和进程一样都是独立优先的可执行路径。可线程作为父进程的一部分执行。每个线程有它自己的栈，但是在一些进程中，几个线程分享了一个公有的堆。线程们也有更低的开销在操作系统的管理方面以及执行需要的内存量方面。

因为线程们并不完全的相互隔离，它们之间的通信也相对的简单些。

16.5 Thread Variables 线程变量

线程变量使用关键字 ThreadVar 来声明。

线程变量与全局变量很相似，并且以类似的方式声明。区别是全局变量对于全部线程都是可见的，而一个线程变量只对于特定的单个线程的全部代码可见。也就是每个线程定义了它自己的"全局空间(global space)"。

当难以在库或者独立部分的代码间传递对象引用时，线程变量十分有用。但是线程变量也有局限性。线程变量无法在 package 内使用和声明。只要有可能，线程类中就应该用成员变量而不是线程变量。成员变量的开销更小，并且在 package 中也可用。

16.6 Threadable and ThreadSafe

线程安全(threadsafe)这个词经常被滥用或者错误地应用。它经常即指可线程的(threadable)又指线程安全的，这导致了混淆和错误。在这段文本中，可线程的和线程安全的被严格的定义，指代不同的东西。

16.6.1 Threadable 可线程的

可线程的指的是一个项目可以在一个线程内被使用或者在使用资源保护合理地保护的情况下被多线程使用。被用 可线程的 形容的项目通常在大多数情况下并不知道线程。

如果一个项目是可线程的，就是说它每次可以被个单线程使用。大部分情况下，可以通过使他隶属于(local to)一个线程来实现，或者作为一个资源保护的全局项目。

常见的可线程项目比如整型，字符串，其他的序数类型，TList, TStringList, 还有大部分的不可见类。

可线程的项目无法无限制地使用全局变量或者直接访问 GUI 控制。无限制地使用全局变量是控件不可线程的最常见原因。

项目可能是个库，控件，过程，或其他东西。

16.6.2 Threadsafe 线程安全的

线程安全意味着这项目清楚的知道线程，并且提供它自己的资源保护。线程安全的项目可以被一个或多个线程使用而不用任何资源保护。

线程安全类的例子比如 VCL 的 TThreadList, 还有所有的 Indy 的 Threadsafe 类。某些操作系统

对象也是线程安全的。

16.7 Synchronization

Synchronization 是从附属线程传递信息到主线程的进程。VCL 通过使用 TThread 类的 Sychronize 方法支持它。

16.8 TThread

TThread 是线程类，它被包含在 VCL 中，在建立线程的东西之上提供了一个很好的实现。

为了实现线程，类从 TThread 类继承下来，然后重写 Execute 方法。

16.9 TThreadList

TThreadList 是 TList 的一个线程安全的实现。TList 可以被任意数量的线程使用，而且不需要对同时访问进行保护。

TThreadList 的工作方式和 TList 很相似，但是不完全一样。一些方法比如 Add, Clear, 和 Remove 是一样的。但是对其他操作，TThreadList 必须先用 LockList 方法锁住。LockList 方法是一个函数，返回对内部 TList 的直接访问。当它被锁住，全部其他线程都将被锁住。因此，应该尽快的解锁它。

操作 TThreadist 的例子：

```
with MyThreadList.LockList do try
  for i := 0 to Count - 1 do begin
    // Operate on list items
    Items[i] := Uppercase(Items[i]);
  end;
finally MyThreadList.UnlockList; end;
```

很重要的是当操作它的代码结束后 List 总是要被解锁的，因此它应该总是使用一个 try..finally 结构来加锁和解锁。如果一个 list 最后没有被解锁，当其他线程企图访问 list 的时候会造成个死锁。

16.10 Indy

Indy 包含许多附加的类作为 VCL 对多线程的支持的补充。这些类实际上独立于 Indy 核心库，并且在非 Indy 的应用里也很有用。他们在 Indy 中存在是因为 Indy 是以多线程为思想设计的。Indy 不止使用这些类来实现服务器，还把它们提供给开发者用。这个部分会给予个关于这些类的简短的综述。

16.11 TIdThread

TIdThread 是个 TThread 的子类，它除了提供些更适合在服务器中使用的特性外还添加了更多高级特性，它还提供了对线程池(thread pooling)和重用的支持。

如果你很熟悉 VCL 的 TThread，一定要注意的是 TIdThread 在一些关键区域和 TThread 极其

不同。在 TThread 中，Execute 方法被子类重写，但是在 TIdThread 中，是 Run 方法被重写。确保不要重写 TIdThread 的 Execute 方法，因为这会妨碍 TIdThread 的内部操作。

对所有的 TIdThread 的子类，Run 方法都要被重写。当线程被激活，Run 方法会被执行。Run 会被 TIdThread 不停的调用直到线程终结。这个的使用可能对大部分客户端都不是很明显，但是它在几乎全部服务器和一些客户端中特别有用。这也和 TThread 的 Execute 方法不同，因为 Execute 方法只执行一次，并且当它存在时不会重新调用 Execute。当 Execute 方法退出时，线程就结束了。

TIdThread 和 TThread 还有其他的不同，但是这些事最主要的不同，Run 和 Execute 是最大的一个不同。

16.12 TIdThreadComponent

TIdThreadComponent 是一个控件，它允许你通过简单地在设计时添加事件，来可视地建立新的线程。它本质上只是一个 TIdThread 的可视化封装，但是它使得建立新线程确实变简单了。

为了使用 TIdThreadComponent，添加一个到你的窗体上，定义 OnRun 事件然后设置 Active 属性。一个 TIdThreadComponent 的例子可以在 Indy Portal 的 TIdThreadComponent demo 上找到。

16.13 TIdSync

TThread 有一个 Synchronize 方法，但是它不支持传递参数给同步的方法的功能。TIdSync 允许通过传递参数给同步的(synchronized)方法来完成同步(Synchronizations)。TIdSync 还允许返回值被从主线程返回。

16.14 TIdNotify

当线程的数量很少时同步(Synchronizations)工作的很好。但是对于有许多线程的服务器或者应用，同步成为了个瓶颈，并且对性能有极其大的副作用。为了解决这个，可以使用通知(notifcations)来替代。Indy 的 TIdNotify 实现了通知。通知允许和首要的线程通信，但是不像同步，线程并不会阻塞到通知被执行。通知的功能和同步的功能十分相似，但是不会降低性能。

但是通知也有一些限制。一个限制是值无法从主线程返回到调用线程(calling thread)，因为通知不会暂停调用线程。

16.15 TIdThreadSafe

TIdThreadSafe 是个实现线程安全类的基础类。TIdThreadSafe 自身从来不会被使用，它只是被设计作为基础类。

Indy 包含以下可以直接使用的子类：TIdThreadSafeInteger，TIdThreadSafeCardinal，TIdThreadSafeString，TIdThreadSafeStringList，TIdThreadSafeList。这些类可以被用来创建线程安全版本的多个整型，字符串等。他们然后可以自由地被任何数量的线程访问，细节部分会

被自动处理。还有，为了扩展使用，他们支持直接上锁。

16.16 Common Problems 常见的问题

多线程中单个最大的问题是并发性。因为多线程平行的执行，在访问一般的数据时有并发性问题。当在应用中使用线程时，常常会遇到以下问题：
在线程中执行 Indy 客户端带来了以下并发性问题：

1. 从线程中更新用户界面
2. 从附属线程往主线程中通讯
3. 从附属线程中访问主线程中的数据。
4. 从一个线程中返回结果数据
5. 探测什么时候线程结束了

16.17 Bottlenecks 瓶颈

许多开发者创建了那些当并发线程数很少时工作的很好但是当线程数增加后很快地停顿的多线程应用程序。这通常是由于瓶颈效应(bottleneck effect)。瓶颈效应就是当一个特定部分的代码块阻塞了其他的线程的执行并且其他线程必须等待那块代码完成。不管其他代码允许的多快，最慢点就是瓶颈。换句话说就是，代码只可以执行的和最慢的那部分一样快。

许多开发者把时间花费在提速他们觉得"不够快"但实际上和存在的瓶颈比起来微不足道的部分，而不是去寻找瓶颈。

通常，排除一个瓶颈对于速度的净提升比做成千上万的其他优化提升的还多。因此，首先关注瓶颈。之后，只有在这之后，才应该找其他代码来进行有可能的优化。

后面会讲些常见的应该避免的瓶颈。

16.17.1 Critical Section Implementation 临界区实现

临界区是个用来控制资源访问的有效并且轻量的实现，所以同时只能有一个线程可以访问给定的资源。

经常一个临界区被用来保护多个资源。比如资源 A，B，和 C。用一个临界区来共同地保护它们，然而每个资源是独立的。于是就产生了个问题，当 B 被使用的时候，A 和 C 也一样被锁住了。临界区相当的轻量级，每个资源都应该专门的用一个临界区。

临界区有时候会锁住太多的代码。临界区中 Enter 和 Leave 方法间的代码数量应该被保证绝对的小，并在大部分情况下，如果有可能的话就用多个临界区来代替。

16.17.2 TMREWS

如果大部分访问都是读访问，只有小部分是写的话，相较于临界区，TMREWS 对性能有明显的提高。但是，TMREWS 类比临界区的系统开销大，获得锁需要更多的代码。对于很小块的代码，即使很少写访问，通常临界区的性能也比 TMREWS 的好。

16.17.3 Syncrhonizations 同步

同步常常用于进行用户界面更新。

同步带来的问题是调用线程会暂停到同步完成。因为主线程会处理所有的同步，同时只能处理一个同步，因此它们是一个一个处理的。这导致了全部全局的同步成为了一个巨大的瓶颈。

只要有可能，就用通知来替代同步。

16.17.4 User Interface Updates 用户界面更新

多线程应用常常有很多的用户界面更新。大部分你通过把应用程序做成多线程而得到的性能提升会由于用户界面更新导致的延迟而快速地丢失掉。许多情况下，这会导致一个多线程应用的运行速度还不如它的单线程版本。

实现服务器应用的时候应该特别考虑些东西。服务器是个服务者，它的主要功能是为它的客户端服务。用户界面是次要的。因此把用户界面作为一个单独的应用是完全能够接受的，这个应用本身就是个这服务器的特别的客户端。另一种解决方法是使用像通知或者批处理升级这样的技术。这两个都会导致你的用户界面轻微地延迟，但是使用一个延迟一秒的用户界面总比让 200 个客户端分别延迟一秒好。对于服务器，客户端就是上帝。

17 Servers 服务器

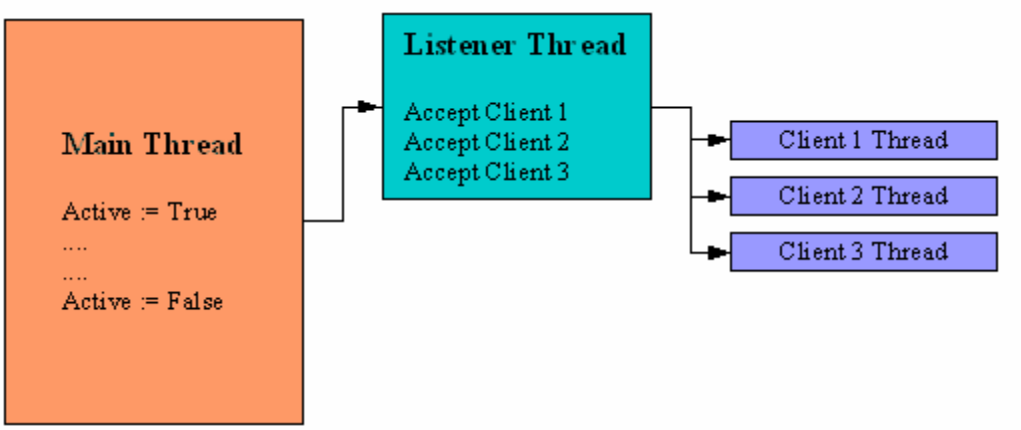
Indy 根据你的需要和使用的协议提供了多种多样的服务器模型。后面几部分会提供 Indy 服务器控件的综述。

17.1 Server Types 服务器类型

17.1.1 TIdTCPServer

最典型的 Indy 服务器是 TIdTCPServer。

TIdTCPServer 创建了个附属的监听线程，这线程独立于程序的主线程。监听线程监听来自客户端的请求。为每个它答复的客户端，它都会创建一个新线程来专门为那单个客户端链接服务。之后那线程里恰当的事件会被触发。



17.1.1.1 The Role of Threads 线程的角色

Indy 服务器是围绕着线程设计的，它的运作方式就好像 Unix 服务器似的。典型的，Unix 应用程序直接使用栈的接口，几乎没用或不用抽象层。Indy 通过高度的抽象和内部实现许多自动和透明的细节，使程序员避免了直接与栈交互。

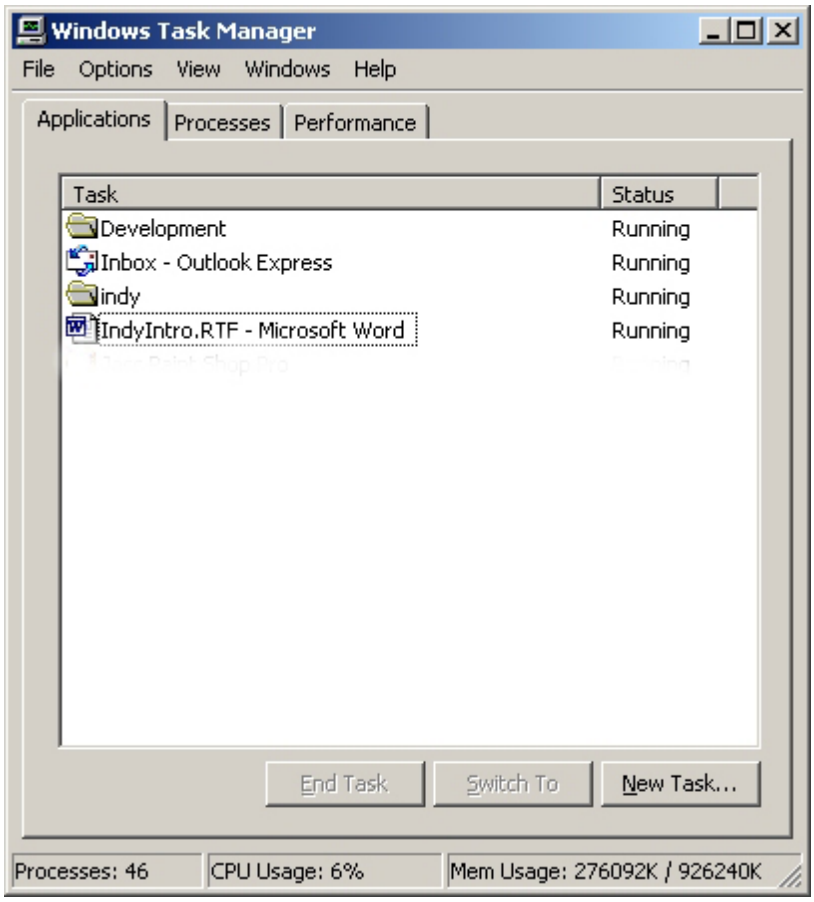
典型的，Unix 服务器有一个或多个监听进程，监听进程等待从客户端来的请求。服务器为每个监听进程接受到的客户端请求，服务器都分支(fork)出一个新的进程以处理每个客户端链接。用这种行为来处理许多客户端链接十分的简单，因为每个进程只处理一个客户端。每个进程还运行在它自己的安全环境中(security context),安全环境可以由监听线程或者进程本身基于身份认证，证书或其他东西设置。

Indy 服务器以相似的方式运作。Windows 的分支实现地不像 Unix 那么好，但是 Windows 的多线程实现的很好。Indy 服务器为每个客户端链接分配个线程而不是像 Unix 一样一个完全的进程。这提供了几乎进程的全部优势，又没有缺点。

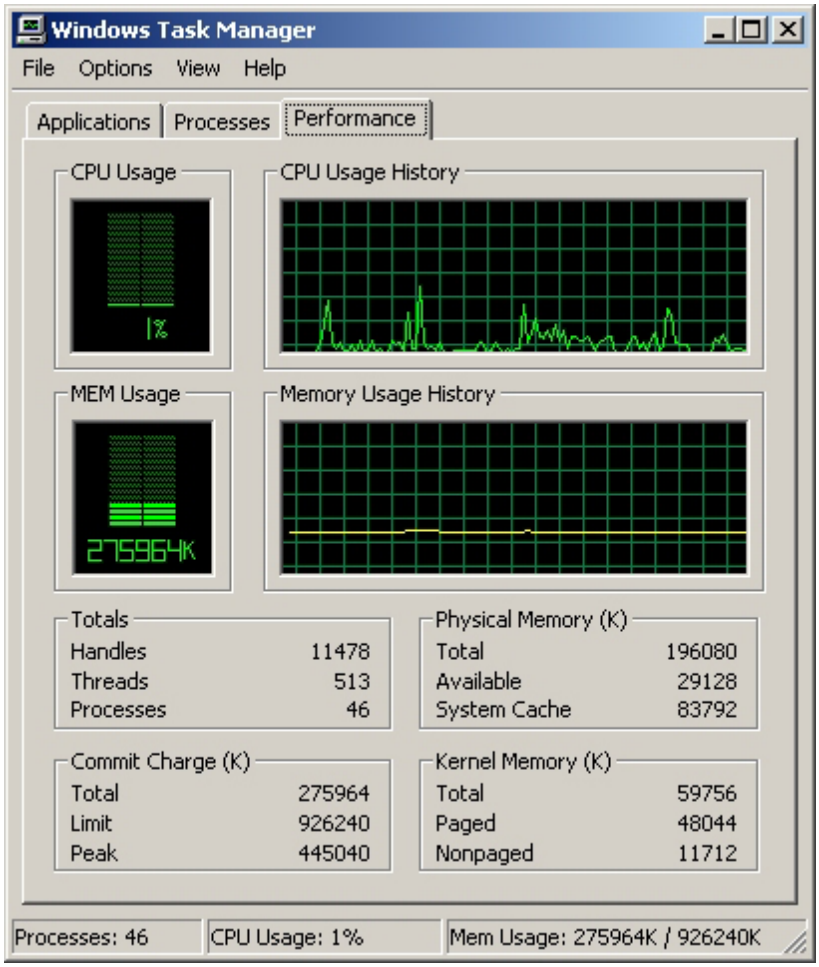
17.1.1.2 Hundreds of Threads 成千上万的线程

在一个忙碌的服务器里，经常需要成百上千的线程。有个常见的误解是成百上千的线程会让系统崩溃。这是个错误观念。

你系统当前在运行的线程数量可能会吓到你。仅仅启动了最少量的服务和运行了以下应用：



这是我系统的截屏，当前有 333 个线程：



即使有 513 个线程，你可以看到 CPU 的使用率才 1%。一个繁忙的 IIS(Microsoft Internet Information Server 微软网络信息服务器)会多创造成百上千的线程。这个测试会在 750MHZ 256MB 的 RAM 的 Pentium III 上进行。

在大部分服务器中，线程花费了大部分时间来等待数据。当等待阻塞调用的时候，线程是非激活的。因此在一个 500 线程的服务器中，同时可能只有 25 个是激活的。

在 socket 应用程序中，还有由最慢的组件控制的更多限制。在网络应用程序中，典型的，网卡是最慢的组件之一，因此没有东西可以超越网卡的性能。甚至一个很快的网卡在许多因素上都比 CPU 慢，因此大部分情况下即使被充分的利用也会成为瓶颈。

17.1.1.3 Realistic Thread Limits 真正的线程限制

实际上，平均水平的系统在单个进程创建了超过 1000 个线程之后开始会出现问题，这是由于内存问题。可以减少线程的栈的大小以增加线程的数量，但是这种时候可以研究一下另外一种选择。

大部分服务器最多只需要几百个线程。但是非常大容量的服务器，或者那些流量很低但是连接的服务器数量及其巨大的服务器，比如聊天服务器会需要以一个不同的方式来实现。这些服务器使用 Indy 会创建上千条线程。

一定要明白的是客户端的数量并不是一定要和同时存在的线程数量一样。尽管每个客户端都被分配在个单独的线程中，但线程只在客户端被连接的时候分配。许多服务器比如 HTTP 服务器服务生命周期很短的连接。HTTP 连接只需要不到一秒来请求网页，特别是用了代理或者缓存的时候。假设有 500 个线程，每个连接要 1 秒钟，这就能够每小时服务三万个客户端了。

Indy 10 除了服务器线程外，通过允许其他模型来满足这些需求。Indy 10 只受到用于分配 socket 的可用内存量的限制。

17.1.1.4 Server Models 服务器模型

有两个建立 TCP 服务器的方式，用 command handlers(命令处理器) 和 OnExecute 事件。Command handlers 使得构建服务器简单的多，但是并不总是适用。

Command handlers 适用于那些用文本形式交换命令的协议，但不适用于那些使用二进制命令结构或者无命令结构的协议。大部分协议是文本为基础的，可以使用 Command handlers。Command handlers 完全是可选的。如果不使用，Indy 服务器仍然支持旧的使用方法。Command handlers 会在 command handlers 部分更详细的讲解。

一些协议是二进制的或者无命令结构的所以不适用于 command handlers。对于这些服务器，应该使用 OnExecute 事件。只要链接还活着，OnExecute 就会重复的发生，并且传递链接作为其参数。一个很简单使用 OnExecute 实现的服务器长成这个样子：

```
procedure TMainForm.myTCPServerExecute(AContext: TIdContext);
var
  LCmd: string;
begin
  with AContext.Connection do begin
    LCmd := Trim(IOHandler.ReadLn);
    if SameText(LCmd, 'QUIT') then begin
      IOHandler.WriteLn('200 Good bye');
      Disconnect;
    end else if SameText(LCmd, 'DATE') then begin
      IOHandler.WriteLn('200 ' + DateToStr(Date));
    end else begin
      IOHandler.WriteLn('400 Unknown command');
    end;
  end;
end;
```

没必要验证链接的有效性，因为 Indy 已经自动做了那件事。也没有必要实现任何循环，因为 Indy 会帮你做那件事。它不断的重复调用事件直到链接结束了。链接的结束可以由于像下面展示的一样的精确的断连，由于个网络错误，或者由于客户端断连。实际上，不应该做任何的循环，因为这可能会妨碍 Indy 的断连探测。如果这事件内部必须进行循环，就应该做特别处理使得 Indy 能捕获这类异常，以处理它们。

17.1.1.5 Command Handlers

Indy9.0 包含了一个叫做 command handlers 的特性。command handlers 是一个新概念，用在 TIdTCPServer 中以使得服务器为你完成命令的转化和处理。command handlers 是一种"可视化服务器管理", 反应了 Indy 服务器的未来。

对于每个你想要服务器处理的命令，就创建一个 command handler。你可以把 command handler 当做服务器的 action list。command handler 含有属性，这些属性告诉它如何转化命令，包括怎么转化参数，命令本身，一些它可能要执行的行为，以及可能的自动答复。有些情况下，你只需要使用属性，一句代码都不用写，就可以完成所有的功能。每个 command handler 都有个独特的 OnCommand 事件。当调用事件的时候，不需要确定执行的是哪个命令，因为每个 command handler 的事件都是不同的。还有，command handler 已经可选地为你做了些准备，并为你转化好了参数。

这是个很基本的关于怎么使用 command handler 的例子。首先，我们必须定义两个命令：QUIT 和 DATE。为了做这件事，要再设计时像如下这样建立两个 command handler：

将 cmdhQuit 的 disconnect 属性设置为 true，cmdhDate 的 OnCommand 事件定义成以下这样：

```
procedure TForm1.IdTCPServer1cmdhDateCommand(ASender: TIdCommand);
begin
  ASender.Reply.Text.Text := DateTimeToStr(Date);
end;
```

这是 command handler 版本的全部代码。其他的细节都通过设置 command handler 的属性声明了。

在 command handler 部分会讲更多 command handler 的细节。

17.1.2 TIdUDPServer

因为 UDP 是非连接的(定义)，TIdUDPServer 与 TIdTCPServer 的运作方式不同。TIdUDPServer 和 TIdSimpleServer 没有任何相似的模式，但是由于 UDP 是无连接的，TIdUDPClient 确实只单独使用 listening 方法。

TIdUDPClient 在激活时创建了一个监听线程来监听进入的 UDP 数据包。为每个到达的 UDP 数据包，TIdUDPClient 都会在主线程中或者在监听线程内触发 OnUDPRead 事件，这取决于 ThreadedEvent 属性的值。

当 ThreadedEvent 的值是 false 时，OnUDPRead 事件会在主线程内被触发。当 ThreadedEvent 的值是 true, OnUDPRead 事件会在监听线程内触发。

不管 ThreadedEvent 属性是 true 还是 false，它的执行都会阻塞更多消息的接收。因此，OnUDPRead 事件的处理必须得很快。

17.1.3 TIdSimpleServer

TIdSimpleServer 是用来创建单独专用服务器的。TIdSimpleServer 设计为同时只服务单个链接。尽管它可以一个接一个服务请求，典型地，它是用来服务单独专用请求的。

TIdSimpleServer 不会产生任何监听线程或者附属链接线程。它的全部功能都在它自己使用的那个线程内发生。

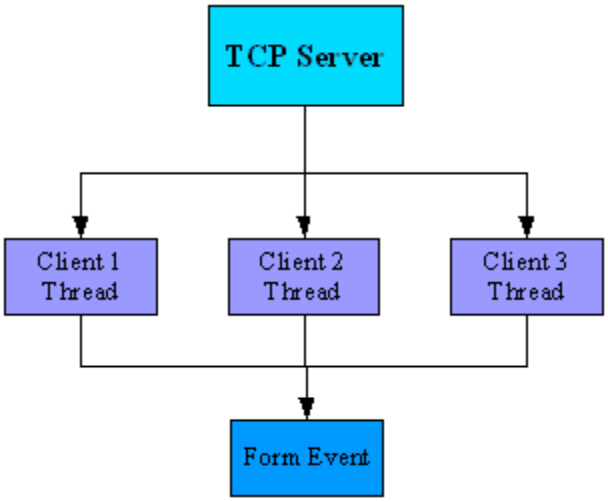
TIdFTP 客户端控件利用了 TIdSimpleServer。当 FTP 进行一个文件传输时，一个附属 TCP 链接会被打开以传输数据，并在数据传输完成后被关闭。这个链接被叫做"数据通道(data channel)"，每次文件传输都是独特的。

17.2 Threaded Events 线程性的事件

TIdTCPServer 的事件们是线程性的。换句话说就是尽管它们不是线程类的一部分，它们是在个线程内被执行的。这个细节十分重要。请确保你在编程前理解了这个细节。

可能这一开始有些令人困惑，事件为什么会看起来是窗体的一部分，结果却在个线程内执行。然而它内部被这样构建以便于事件能像其他事件一样在设计时被创建，而不用创建一个定制的类并重写方法。

如果你创建了子类控件，是可以重写方法的。但是对于建立应用程序来说，事件模型使用起来更加简单。



每个客户端都被赋值自己的线程。使用那些线程的时候，TCP 服务器的事件(当创建时是作为窗体或者数据模型的一部分)被从那些线程调用。这也就是说单个事件可能被从多个线程调用许多次。这些事件会收到一个 AThread(XE10 中 AContext)参数，这参数声明了正调用事件的线程。

线程性的事件比如服务器上的 OnConnect, OnExecute, 和 OnDisconnect。

17.3 TCP Server Models TCP 服务器模型

Indy 的 TCP 服务器支持两种构建服务器的模型。这些方法是 OnExecute 事件和 command handlers。Indy8 只支持 OnExecute。

17.3.1 OnExecute

OnExecute 指的是 TIdTCPServer 的 OnExecute 事件。当使用这个模型实现一个服务器的时候，必须定义 OnExecute 事件，或者必须重写 DoExecute 方法。

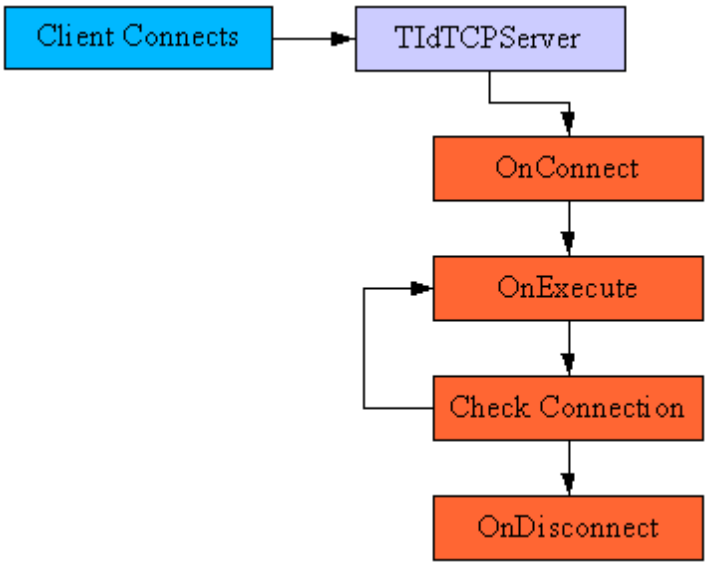
使用 OnExecute 模型使得开发者获得完全的控制，并使得可以实现包括二进制协议在内的所有类型的协议。

在个客户端连接到服务器后，OnExecute 事件会被触发。如果没有定义 OnExecute 事件，就

会引发异常。只要客户端连接着，OnExecute 事件就是在一个循环内被触发。这是个很重要的细节，因此开发者必须小心：

1. 记住事件在个循环内
2. 不要实现可能妨碍 Indy 的循环

这个图标展示了 Indy 内部构建的循环：



Check Connection(检测链接)步骤进行了如下检验：

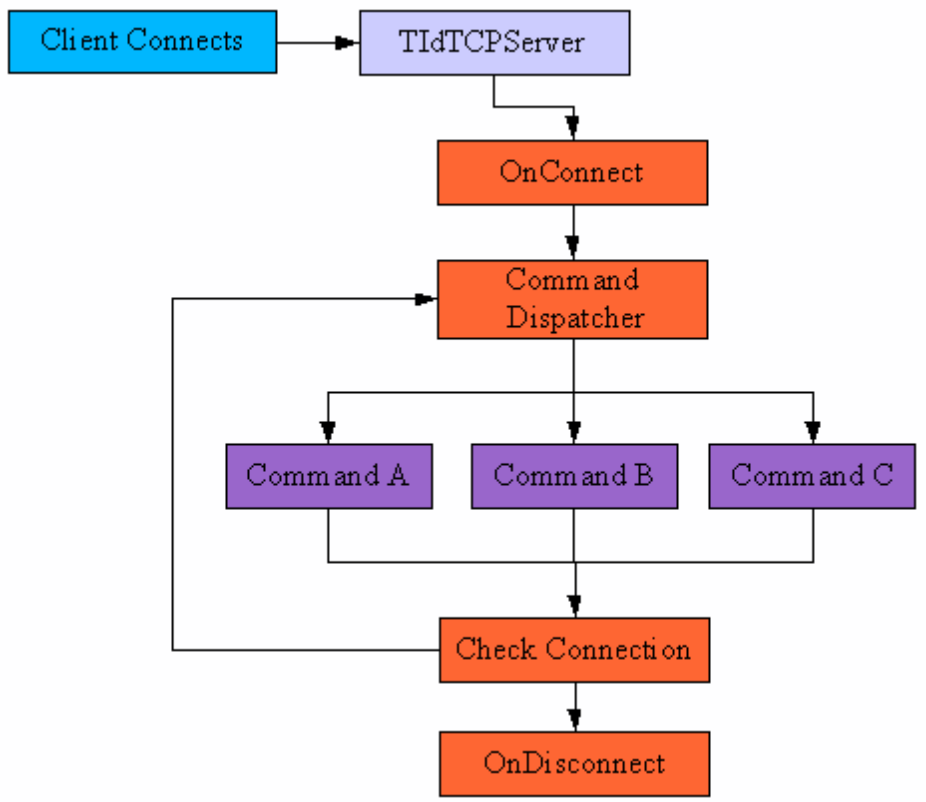
- 客户端还在连接中
- 在 OnExecute 中没有调用 Disconnect
- 没有致命错误
- 在 OnExecute 事件中没有无法处理的异常
- 服务器还激活着

如果全部这些条件和其他检验都为真，OnExecute 就被再次调用。因此，开发者不应该在 OnExecute 中构建他们自己的循环代码来重复执行，因为这会干扰 Indy。

17.3.2 Command Handlers

在设计时环境中使用 Command handlers 就像用 action lists 来搭建服务器，这是 RAD 的风格。Command handlers 只能被用于会话式文本协议，但是协议的数据部分是可以包含二进制数据的。

Command handlers 自动的读取和转换命令。然后特定的 OnCommand 事件被触发。



17.4 Command Handlers

在 Indy 中创建服务器一直以来都是十分直接的，但是在 Indy9 之后，由于 command handlers 引入了 Indy 的 TCP 服务器(TIdTCPcmdServer),这变得甚至更简单了。

command handlers 的方式就好像服务器的 action list。 command handlers 的运作方式要求你为每个命令都定义个 command handler，然后使用 command handler 为每个命令分别定义行为和答复。当从客户端接收到了一条命令，服务器会自动的转换它并把它传递给合适的 command handler。command handlers 不仅仅有定制它们的行为的属性，还有方法和事件。

command handlers 仅仅工作在以文本为基础的命令，并且答复(会话式的)TCP 协议。但是这已经覆盖了如今 95%的普通的服务器。尽管 command handlers 可以处理二进制数据，但它们只可以处理文本的命令。还是有使用二进制命令的协议的。对于那些使用二进制命令或者文本命令不是会话式的协议，command handlers 的实现也提供了向后兼容性，并允许服务器不使用它们而实现。

17.4.1 Implementation 实现

TCPServer 包含一个叫做 CommandHandlers 的属性，这属性是 command

handlers 的集合。Command handlers 常常是在设计时创建的，然而对于实现协议的子类，它们也可以在运行时创建。如果 command handlers 在运行时被创建，它们应该被重写过的 InitializeCommandHandlers 方法创建。这将确保它们仅仅在运行时被创建。如果他们在结构体内被创建，它们将在每次 TCPServer 进行流式处理(is streamed in)的时候被创建，在流处理结束后它们会被保留。不久，每个 command handler 的副本数量就会无限多。Initialize 方法

是在 TCPServer 被第一次激活后被调用。

TCPServer 包含许多与 command handlers 相关的属性和事件。属性 CommandHandlersEnabled 全局的启用和弃用 command handler 处理。事件 OnAfterCommandHandler 在每次 command handler 处理完成后被调用，事件 OnBeforeCommandhandler 在每次 command handler 开始处理前被调用。事件 OnNoCommandHandler 在没有发现对应的 command handler 时被调用。

如果 CommandHandlersEnabled 为 True 并且存在 command handlers，就会使用 command handlers 进行处理。否则，如果 OnExecute 事件被赋值了的话就会被调用。如果使用了 command handler 那么就不会调用 OnExecute 事件。

只要链接存在着，TCPServer 就会从连接中读取多行文本，并尝试将数据和 command handlers 匹配。任何空行都将被忽略。对于非空行，首先会调用 OnBeforeCommandHandler 事件。然后会搜索一个匹配的 command handler。如果发现了匹配的 command handler，并且它的 enabled 属性是 True，它的 OnCommand 事件就会被触发。否则 OnNoCommandHandler 事件会被触发。最终 OnAfterCommand handler 事件被触发。

17.4.2 Example Protocol 例子协议

为了给你个基础的 command handlers 实现的例子，我们定义个简单的协议。为了举例子，我们实现个包含三个命令的定制时间服务器。

- **Help** - 展现个支持的命令的清单，每个再附带些基本帮助。
- **DateTime <format>** - 使用指定的格式返回当前日期或/和时间。如果没有指定一个格式，就自动使用格式 yyyy-mm-dd hh:nn:ss。
- **Quit** - 终结会话和连接。

这是个非常基础的实现，但是作为一个例子来说十分的赞。你可能想要扩展它来多玩玩 command handlers。

17.4.3 Base Demo

首先我们建立个基础框架，之后我们在上面进行扩展。我假设你已经熟悉了 TIdTCPServer 并且知道以下几步做了什么事。为了建立例子的基本框架，做以下几步：

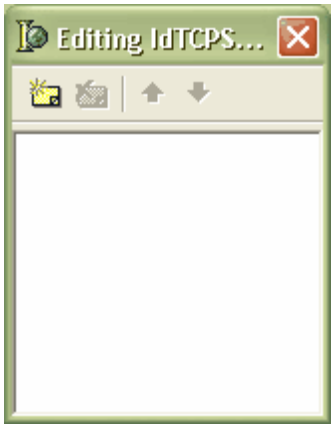
1. 创建一个新应用程序
2. 添加一个 TIdTCPServer 到窗体上(XE10 中应该是子类 TIdTCPCmdServer)
3. 设置 TIdTCPServer 的 Default 属性为 6000。6000 只是随便选的数字，任何闲置的端口都可以用。
4. 设置 TIdTCPServer.Active 为 True。这将在应用程序运行时激活服务器。

这将建立基本应用程序。因为没有建立事件或者 command handlers，它不会做任何事。

17.4.4 创建个 Command Handler

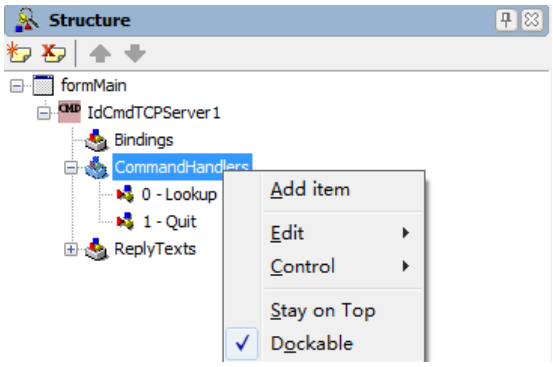
Command handlers 是通过编辑 TIdTCPServer 的 CommandHandlers 属性来定义的。CommandHandlers 属性是一个集合。Command handlers 可以在运行时和/或设计时被修改。为了在设计时编辑 command handlers，在对象查看器(Object Inspector)中选择

CommandHandlers 属性的椭圆按钮。应该会出现个长这样的对话框：



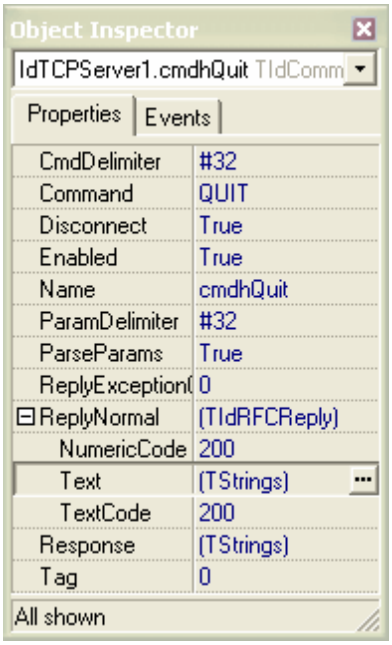
它是空的，因为还没有创建 command handler。为了创建 command handlers，要不然就右击空白区域然后选择 add，要不然工具栏上第一个按钮。做完这个之后，一个 command handlers 会被列在属性编辑器中，就像这样：

注：XE10 中添加 TIdCmdTCPServer 后直接可以在 Structure 中添加 command handler，像这样：



为了编辑 command handler，这个属性编辑器被用来选择 command handler，然后就用对象查看器(object inspector) 来编辑它的属性和事件。编辑 command handler 就像编辑 Dataset 的字段或者 DBGrid 的列。如果找不到对象查看器，按 F11 就出现了。

对象查看器长的差不多就像这样子。这幅图中已经对一些属性的默认值进行了修改以实现 QUIT 命令，这些内容之后会说：



这是修改属性以实现 QUIT 命令的每一步：

1. Command = Quit - 这是服务器用来配对配对客户端输入并且决定使用哪个 command handler 来处理命令的命令。命令是大小写不敏感的。
2. Disconnect = True - 这告知服务器在处理完命令后断连客户端。
3. Name = cmdhQuit - 这对 command handler 的行为没有任何影响，但是如果需要写代码的话会让它更好辨认。这步是可选的。
4. ReplyNormal.NumericCode = 200 - 典型地，答复命令时使用一个 3 位数字代码以及可选的文本。这告知 command handler 如果在处理命令的过程中没有错误发生的话，就使用 200 加上在 ReplyNormal.Text 中找到的文本来答复。
5. ReplyNormal.Text = Good Bye - 这段文本伴随 ReplyNormal.NumericCode 被发送。

现在建立了一个完整功能的 command handler 了。

17.4.5 Command Handler Support

现在已经建立了个初始的 command handler，然而还有一些和基于文本的服务器有关的全局选项，以及还要设置一些 command handlers。全部这里讨论的属性都是 TIdTCPServer 本身的属性，并不是独立的 command handlers。

Greeting

一个惯例是服务器在连接建立后，收到客户端的命令之前，提供一个答复给客户端。一个典型的说明服务器准备好了的答复是 200，并且把它设置为非零使得能够发送问候语。

设置 Greeting.NumericCode = 200 以及 Greeting.Text 为 "Hello"。

ReplyExceptionCode

如果在命令的处理过程中发生了个无法处理的异常，如果这个数字非零，这个数字就会被用来构建个给客户端的答复。500 是个内部未知错误的典型答复。而对于答复的文本部分，会发送异常文本(exception text)。

设置 ExceptionReply.Code 为 500。

ReplyUnknown

如果一个客户端提交的命令找不到匹配的 command handler，TIdTCPServer 的 ReplyUnknownCommand 属性就被用来返回一个错误给客户端。400 是个常见的内部错误的答复。

设置 ReplyUnknown.NumericCode 为 400，ReplyUnknown.Text 为 "Unknown Command"。

Other Properties 其他属性

TIdTCPServer 还有其他属性以及事件来实现和 command handlers 有关的附加行为，但这里列出来的是最起码要被实现的那些。

17.4.6 Testing the New Command 测试新命令

现在已经建立了初始的命令，由于 command handlers 是基于文本的，可以通过 telnet 很简单地测试它。为了测试新命令：

1. 运行应用程序。
2. 在开始菜单中的命令提示符中输入：telnet 127.0.0.1 6000 然后按回车。这指示 telnet 连接到电脑的 6000 端口，也就是示例程序监听的那个端口。
3. 服务器应该答复 200 Hello，也就是之前在 TIdTCPServer 的 Greeting 属性中定义的问候语。
4. Telnet 然后将显示一个脱字号。这是说服务器准备好了，并且等待着输入(比如一条命令)。
5. 敲出 HELP 然后按回车。服务器答复 "400 Unknown Command"。这是因为 HELP 命令还没有被实现呢，"400 Unknown Command" 就是被定义在 ReplyUnknown 属性中的东西。
6. 敲击 QUIT 并回车。服务器会答复 "200 Good Bye" 然后断连客户端。

恭喜！你刚已经使用 command handlers 创建了一个服务器。接下来的部分讲的是实现另外两个命令 HELP 和 DATETIME，它们同 QUIT 有着不同的行为和需要。

17.4.7 Implementing HELP 实现 HELP

除了这两个不同外，HELP 命令和 QUIT 命令的行为蛮相似的。

1. 它不会断开链接。
2. 除了状态 reply，它还提供帮助信息的文本答复。

为了实现 HELP 命令，做以下几步：

1. 创建新 command handler。
2. Command = Help。
3. Name = cmdhHelp。
4. ReplyNormal.NumericCode = 200。
5. ReplyNormal.Text = Help Follows。

这些步骤你应该都蛮熟悉的，因为它们和 QUIT 里实现的很相似。用来实现文本格式答复的另外的属性是 Response 属性，它是个 string list。如果答复包含文本，文本会在 ReplyNormal 被发送后发给客户端。为了实现 HELP 命令，使用 Response 属性的 string list 属性编辑器然后输入：

```
Help - Display a list of supported commands and basic help on each.
DateTime <format> - Return the current date and/or time using the specified
format.
```

```
If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
Quit - Terminate the session and disconnect.
```

现在，如果你连接到服务器并发送 HELP 命令，服务器会像这样答复：

```
200 Hello
help
200 Help Follows
Help - Display a list of supported commands and basic help on each.
DateTime <format> - Return the current date and/or time using the specified
format.
If no format is specified the format yyyy-mm-dd hh:nn:ss will be used.
Quit - Terminate the session and disconnect.
```

17.4.8 Implementing DATETIME 实现 DATETIME

DATETIME 是这个协议的实现中的最后一个命令。它和 QUIT 或者 HELP 都不同，它需要一些定制的功能，这功能不能单单使用属性创建。在 DATETIME 的实现中，要使用一个事件来实现这定制行为。

首先按照这些你熟悉的步骤建立基本的 command handler。

1. 创建一个新 command handler。
2. Command = DateTime。
3. Name = cmdhDateTime。
4. ReplyNormal.NumericCode = 200。

这次没有定义 ReplyNormal.Text，在事件中会为每个请求定制的生成它。为了定义事件，选定 DATETIME command handler 后使用对象浏览器。切换到 events 标签并创建个 OnCommand 事件，Delphi 将会创建这样的事件框架：

```
procedure TForm1.IdCmdTCPServer1CommandHandlers2Command(ASender: TIdCommand);
begin

end;
```

OnCommand 传入一个 ASender 参数，类型为 TIdCommand。这不是 command handler ，而是命令本身。Command Handlers 对于全部链接都是全局的，然而 commands 是特定于被当前 OnCommand 事件实例处理的链接的。这确保了事件可以为每个客户端提供特定的行为。

在事件被调用之前，Indy 将会创建一个 Command 的实例并基于 command handler 初始化它的属性。然后你可以在默认的属性之上改变 command 的属性，调用方法来指示 command 完成任务，或者访问它的 Connection 属性来直接和链接交互。

这个协议定义 DATETIME 为，接受一个可选的用来指定返回的日期和时间格式的参数。command 也已经支持了参数，参数在 Params 属性中，它是个 string list。当从客户端接收到了条命令，如果 command handler 的 ParseParams 属性是 True(True 是默认的)，Indy 将会使用 CmdDelimeter 属性(它默认是#32 或者空格)来转化命令为命令以及它的参数。

比如在这个协议中，客户端可能会发送：

```
DATETIME hhnnss
```

在这种情况下，ASender.Params 将包含字符串 "hhnnss"在 ASdender.Params[0]中。参数的数量可以通过读取 ASender.Params.Count 来确定。

OnCommand 事件可以使用这些属性像以下这样实现：

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end else begin
    LFormat := ASender.Params[0];
  end;
  ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

这个实现仅仅读取了参数然后使用 ASender.Reply.Text 来发送答复给客户端。没有必要设置 ASender.Reply.NumericCode,因为 Indy 已经按 command handler 的 ReplyNormal.NumericCode 属性把它初始化为 200 了。

注意 ASender.Reply.Text.Text 的使用。有两个 Text 是因为 command 的 Text 属性是一个 string list ， 我们访问的是 TStrings.Text。 因为他是一个 string list ， 其他方法或者属性，比如 Add,Delete,也可能被用到。这里使用 Text 是由于 ASender.Reply.Text 可能在某些情况下会被预初始化(pre-initialized)，使用 ASender.Reply.Text.Text 将会覆盖任何先前存在的文本。

如果再次使用 telnet 测试例子程序，答复差不多是这样的：

```
200 Hello
datetime
200 2002-08-26 18:48:06
```

有的时候参数无法使用。DATETIME 就有可能。想想看要是用户发送这条命令：

```
DATETIME mm dd yy
```

在这种情况下 Params.Count 会是 3， 事件将会失败并且只返回月份(mm)的值。为了那些参数有内含的分隔符的情况，应该使用 command 的 UnparsedParams 属性。可选地，ParseParams 属性可以被设为 False。UnparsedParams 将包含不管 ParseParams 的值的数据，但是把 ParseParams 设置为 false 将提高效率，因为这告诉 Indy 不需要转换参数到 Params 属性中。

使用 UnparsedParams 的事件代码被修改成这样：

```
procedure TForm1.IdTCPServer1TIdCommandHandler2Command(ASender: TIdCommand);
var
  LFormat: string;
begin
  if ASender.Params.Count = 0 then begin
    LFormat := 'yyyy-mm-dd hh:nn:ss';
  end else begin
    LFormat := ASender.UnparsedParams;
  end;
  ASender.Reply.Text.Text := FormatDateTime(LFormat, Now);
end;
```

17.4.9 Conclusion 结论

Command handlers 的使用很有弹性，并且包含的属性和方法不止这里讲的这些。这只是个 Command handlers 及它们能力的介绍。我希望这些足够引起你的兴趣并且足够帮助你起步了。

我们有计划在未来版本的 Indy 中让 command handlers 在设计时可见性更强。

17.5 Postal Code Server - OnExecute Implementation

OnExecute Implementation
Demo
Threads

17.6 Postal Code Server - Command Handlers

Command Handlers
Greeting
CommandHandlers
ReplyException
ReplyTexts
ReplyUnknownCommand
Demo

17.7 Thread Management 线程管理

在 Indy 中线程管理被抽象到了 thread managers 中。thread managers 允许不同的甚至是定制的线程管理的实现。

thread managers 的使用是可选地。对于那些支持线程管理的控件(如 TIdTCPServer)如果你不通过设定控件的 ThreadManager 属性指定个 thread manager。Indy 将会隐式地创建和摧毁一个默认的 thread manager 实例，TIdThreadMgrDefault。

17.7.1 TIdThreadMgrDefault

Indy 中默认的线程管理是很简单和基础的。每次需要个线程的时候，就会创建个。当线程不再需要的时候，它就被摧毁。对大部分服务器，这是可以接受的，除非你合理地需要使用线程池(thread pool)，否则你就应该使用默认的线程管理。大部分服务器中，两者的性能差异微不足道甚至没有差异。

这还有个额外的优点，每个线程都一定是"干净的"。线程常常分配内存或其他对象。当线程被销毁时，这些对象通常是自动地被清理的。因此使用默认的线程管理有助于确保所有的内存都被释放及清理。当使用线程池的时候，如果你存储了和线程有关的数据，你必须确保在重用线程前清理掉它。没这么做的话可能会给下一个用户提供上一个用户的敏感信息。而用默认的管理就不用做这种预防，因为每次线程和相关的数据都会被销毁。

17.7.2 Thread Pooling 线程池

一般地，默认线程管理是合适的。但是对于那些服务短生命周期的链接的服务器，服务每个请求的线程的创建和销毁过程的时间对比于服务请求的时间是无法忽略的。这像这种情况下，应该使用个线程池。

使用线程池，线程们会被回收利用和预分配。在一个线程池中，线程在需要它们之前就被创建好了并且在池中保持非激活状态。当需要线程的时候，就会从池子中取出一个并且激活它。如果需要比池子中创建好的线程还多的线程，就会创建更多的线程。当个线程不再被需要时，它不会被销毁，而是被使无效并被丢入池子中，于是它可以在之后被需要的时候重用。

线程的创建和销毁可能是资源集中的(resource intensive)。这对于服务短生命周期(short-lived)链接的服务器特别的明显。这样的服务器创建个线程后很简短的使用它然后就摧毁了它。这导致非常高频率的线程创建和销毁。这样的服务器的例子比如时间服务器甚至网页服务器。一个单请求被发送，然后返回一个简单的答复。当使用个浏览器去浏览个网页站点时，服务器上可能会发生成千上万的连接和断连。

线程池可以缓解这种状况。取代根据命令而创建和摧毁线程，线程被从个非激活但是已经创建的列表(池子)中借用过来。当个线程不再被需要，它被丢回了池子中而不是被销毁掉。当线程在池子中时，他们被标记为非激活的，因此也不消耗 CPU 周期。为了更加提升性能，池子的大小可以被动态地调整以满足系统当前的需要。

Indy 的确支持线程池。在 Indy 中，线程池可以通过使用 TIdThreadMgrPool 控件来获得。

18 SSL - Secure Sockets 安全 Sockets

SSL 代表安全套接层协议(Secure Socket Layer)，它是被接受的在网络上加密数据的加密方法。SSL 最常被使用在 HTTP(网页)流量上，这被叫做 secure HTTP(HTTPS)。但是 SSL 并不局限于 HTTP，它可以被使用在任何 TCP/IP 协议中。

在 Indy 中为了使用 SSL，你必须先安装合适的 SSL DLLs(动态链接库)。Indy 的 SSL 支持是由开源的方式完成的，但是目前唯一实现 SSL 的开源库是 OpenSSL。OpenSSL 库是一个 DLLs 的集合，你可以从 Indy 的主要分配网站分别地下载他们。

美国政府和其他政府以他们碉堡了的智慧和对技术的理解禁止了对特定加密方法比如 SSL 的输出。因此 SSL 技术不能被放在个网站上，除非采用了特别的方法来验证每个想要下载技术的客户端的位置。这不仅仅难以实现，还让网站负责人承担了好多法律责任。

限制仅仅应用于电子分发(electronic distributions)却不应用于任何形式的打印的源代码。而且只限制出口不限制进口。

所以简而言之，程序员可以把源代码打印在一些东西比如 T-shirt 上，穿越边境，然后再编辑和重编译代码。在这之后，那些没有和美国签署协议的国家就可以用任何形式自由地分发这种加密技术，而且因为没有任何进口限制，任何人都可以下载到已经编译好可以直接使用的加密技术。

许多这类问题这样就被解决了，一些政府可高兴了。当时仍然还是有许多的出口限制，并且每个国家都不同。因此，所有的 Indy SSL 工作都是在 Slovenia 完成的，所有的与 Indy 有关的 SSL 加密技术都从 Slvoenia 被分发。Slovenia 并不限制加密技术的出口。

除了加密技术的出口，一些国家还限制对加密技术的使用甚至拥有。在使用 SSL 之前你应该检查你国家的法律。一些国家比如中国(虽然作者这么说，当好像并没有这回事)，伊拉克共和国等有严重的刑罚甚至死刑对于甚至只拥有这种技术。

18.1 Secure HTTP / HTTPS

在 Indy 中实现 HTTPS 十分的简单。只需要传递一个安全 URL(secure URL)而不是标准 URL，然后 Indy HTTP 客户端(TIDHTTP)就会自动为你做所有的事情了。为了让个 URL 变安全，只要把 http:// 变为 https:// 就可以了。

注意，对于一个要建立的 HTTPS 会话，将要答复请求的那个 web 服务器必须支持 SSL 并且安装了加密证书。Indy HTTP 客户端并不自动验证服务器的证书，这是你的责任。

18.2 Other Clients

SSL 可以通过使用个 SSL IOHandler 来在任何 Indy TCP 客户端中被很简单地实现。一般来说，加密应该使用一个 Intercept 来实现而不是一个 IOHandler。SSL 在 Indy 中是通过一个 IOHandler 而不是 Intercept 来实现这回事因为 SSL 库实际上本身进行通讯。数据是以未加密的形式直接从 SSL 库中被返回和接受的。

为了让个 Indy TCP 客户端使用 SSL，只要从 Indy Misc tab 添加个 TIdSSLIOHandlerSocket 到你

的窗体上。然后设置你的 TCP 客户端的 IOHandler 属性为那个 TIdSSLIOHandlerSocket。只需要做这么多就可以支持基础的 SSL 了。TIdSSLIOHandlerSocket 确实有用来说明客户端这边证书的属性以及其他高级的 SSL 选项。

18.3 Server SSL

在服务器端实现 SSL 比在客户端实现 SSL 稍微复杂一些。典型地，对于客户端，全部要做的事情就是挂住个 TIdTCPClient 或者它的子类到个 TIdSSLIOHandlerSocket 实例。这是由于服务器进行大规模分享(larger share)的 SSL 通信。

为了在服务器中实现 SSL，要使用一个 TIdServerIOHandlerSSL。TIdTCPServer 的 intercept 属性被用来勾住个 TIdServerIOHandlerSSL 。 然而不像 TIdSSLIOHandlerSocket(客户端)，TIdServerIOHandlerSSL 还需要其他一些步骤。特别地，必须安装证书。这些证书必须以磁盘上文件的形式提供，并指定为 CertFile, KeyFile 和 RootCertFile，这些都是 SSLOptions 属性的子属性。

特别地，证书是从个可信任的证书当局(trusted certificate authority)获得的。你当然可以自己写自己的证书，自己做自己的证书当局，但是没有浏览器会信任你的证书并且当浏览器连接到你的服务器的时候会显示个警告。如果你想要部署到因特网上，你必须从证书部门获得个标准浏览器信赖的证书。Versign 是唯一被所有浏览器信任的证书当局。也可以使用 Thawte，但不是所有浏览器都默认的信任它。

如果你的客户端在你的控制之下，比如在个局域网或外联网(extranet)中，你可能会选择自己做证书当局。但是为了避免警告窗口，你必须在每个会连到你服务器的浏览器中安装你的证书。这使得浏览器会把你证书上的签名当做就和可信任当局得来的一样。

注意，这里指的都是 HTTP 服务器，但是 SSL 不止局限于 HTTP。如果你在实现客户端和服务器，你可以实现 SSL，并在可信任的连接上获得完全的控制。

18.4 转换证书为 PEM 格式

有可能你的证书并不是以 .pem 格式给你的。如果它们不是以 .pem 格式给你的，你必须转换它们以使用 Indy。

这个程序默认你已经从某个证书当局(像 Verisign 或者 Thawte)收到了密钥证书对，并且你已经在微软网络浏览器的个人证书中心安装了它们。

18.4.1 Exporting the Certificate 导出证书

选择那个证书并导出它为个 .pfx 文件(Personal Exchange Format 个人交换格式)，你可以用个密码保护它。

18.4.2 Convert .pfx to .pem

作为 SSL 下载的一部分，一个叫做 openssl.exe 的小工具也会一起下载下来。这个小工具将被用来转换你的 .pfx 文件。

为了使用 openssl.exe ，用这样的格式：

openssl.exe pkcs12 –in <your file>.pfx –out <your file>.pem

Openssl.exe 会提示你要个密码。如果你使用了密码就输入，如果你没有设置密码就留空。它还会问你要个新密码给 .pem 文件用。这是可选的，但是如果你使用了个密码来保护它，你要确保在 SSL intercept 中创建个 OnGetPassword 事件。

18.4.3 Splitting the .pem File 拆分 .pem 文件

如果你用个记事本来看那个新创建的.pem 文件，你会看到它由两个部分组成。这两个部分包含私钥部分和证书(公钥)部分。还有一些附加的信息。Indy 要求这些信息被拆分到不同的文件中。

18.4.4 Key.pem

用记事本创建 key.pem 然后复制并黏贴这两条语句间的所有东西：

-----BEGIN RSA PRIVATE KEY-----
-----END RSA PRIVATE KEY-----

18.4.5 Cert.pem

用记事本创建 Cert.pem 然后复制并黏贴这两条语句间的所有东西：

-----BEGIN CERTIFICATE-----
-----END CERTIFICATE-----

18.4.6 Root.pem

最后一个 Indy 需要的文件是证书当局的证书文件。你可以在网络浏览器的 Trusted Root Certificate Authority 对话框中获得它。选择给你证书的当局，导出它为 Base64(cer) 格式。这个格式也和 PEM 格式一样，所以导出完成后简单地重命名文件为 root.pem。

19 Indy 10 Overview 回顾

Indy 10 仍然在开发中。当前代码的连贯性可以被比作肮脏的汽车油。在几周内，它就会变成更加牢固和一致的好吃的蛋羹，因此，这里的任何信息都以最终的 Indy 10 发行版本为准。这里给出的信息是基于当前的代码，设计目标和项目方向的。

Indy 10 包含许多新特性，特别是和称为核的部分有关。为了使用起来更简单，Indy 10 的核已经进一步抽象了。Indy 10 内核还包含了许多新的能力，和极大的性能加强。

19.1 Changes

19.1.1 Separation of Packages　　Packages 的分离

Indy 10 被分为了两个 packages： Core and Protocols。

Core 包含了所有内核部分，基础客户端控件，和基础服务器控件。ore 并没有实现任何高等协议。

Protocols package 使用 core 并实现更高等协议比如 POP3，SMTP 和 HTTP。

这使得 Indy 的后勤维护人员更容易的定位特定部分。这对于那些正实现定制协议的用户也很有好处，他们可能不需要 Protocols package。

19.1.2 SSL Core　　SSL 内核

Indy 10 的 SSL 能力现在完全的可插入了。在 Indy 10 之前，对 SSL 的支持是在 TCP 层的可插入，然而像 HTTP 那样的为 HTTPS 使用 SSL 做的协议被固定的使用 Indy 的默认 SSL 实现，OpenSSL。

Indy 10 继续包含对 OpenSSL 的支持，然而为了其他协议的实现，Indy 的 SSL 能力在内核和协议层是完全地可插入的。

SSL 和其他加密方法你可以使用第三方控件 SecureBlackbox 和 StreamSec。

19.1.3 SSL Protocols　　SSL 协议

Indy 10 现在在以下客户端和服务器实现了隐式和显示的 TLS：

- POP3
- SMTP
- FTP
- NNTP

支持 SASL 的代码已经被重新设计以支持 POP3 和 IMAP4.Indy10 现在支持匿名 SASL，文本 SASL，OTP(一次性密码系统) SASL，外部 SASL，和 Auth Login。

19.1.4 FTP Client

FTP 客户端已经在以下方面进行扩展：

- 现在支持 MLST 和 MLSD 了。这提供一个被简单地转换的标准 FTP 目录列表格式。
- 已经添加了一个特别的组合命令以支持多文件上传。注意，这需要一个支持 COMB 命令的服务器，比如 GlobalScape Secure FTP 服务器或者 Indy 10 中的服务器控件。
- 已经增加了一个 XCRC 命令以获得文件的 CRC。注意，这需要一个支持 COMB 命令的服务器，比如 GlobalScape Secure FTP 服务器或者 Indy 10 中的服务器控件。
- 客户端现在支持 MDTM 命令以获得最后修改日期。
- OTP(One-Time-Only password 一次性密码)计算器已经组合进来并且现在 OTP 自动被检测。现在已经支持 FTPX 或者点对点文件传输(文件在两台服务器间传递)。注意，只有服务器支持它的时候 FTPX 传输才能工作(因为安全原因，现在许多管理者和开发者弃用了这

个功能)。

- 添加了 FTP 专用 IP6 扩展。

19.1.5 FTP Server

- FTP 服务器现在支持：
- MFMT 命令和 MFF。http://www.trevezel.com/downloads/draft-somers-ftp-mfxx-00.html
- XRC 和 COMB 命令以支持 Cute FTP Pro 的多文件更新。
- 对 MD5 和 MMD5 命令的支持(http://ietfreport.isoc.org/ids/draft-twine-ftpmd5-00.txt)。
- 支持一些影响路径是怎么给予的 Unix 的选项，这包含回归路径列表(-R)。
- 在 FTP 服务器上支持简单转换的列表格式。
- 可以使用 OTP 计算器用户管理。
- 可以使用个虚拟系统控件以使得 FTP 用起来更简单。
- FTP 专用 IP6 扩展

还有，FTPX 点对点文件传输功能现在可以禁用。这是为了安全原因，因为 Port 和 PASV 命令被滥用，这些地方说了这个问题：

- http://www.cert.org/tech_tips/ftp_port_attacks.html
- http://www.kb.cert.org/vuls/id/2558
- http://www.cert.org/advisories/CA-1997-27.html
- http://www.geocities.com/SiliconValley/1947/Ftpbounc.htm
- http://cr.yp.to/ftp/security.html

19.1.6 FTP List Parsing　　FTP 列表转换

为几乎每个现存 FTP 服务器类型，甚至一些不再需要的服务器，Indy10 在有给定的解析器的 FTP 列表解析系统内都包含了个塞子(plug)。

如果偶然遇到了一个不支持的系统，可以使用一个定制 handler。

FTP 服务器支持：

- Bull GCOS 7 or Bull DPS 7000
- Bull GCOS 8 or Bull DPS 9000/TA200
- Cisco IOS
- Distinct FTP Server
- EPLF (Easily Parsed List Format)
- HellSoft FTP Server for Novell Netware 3 and 4
- HP 3000 or MPE/iX including HP 3000 with Posix
- IBM AS/400, OS/400
- IBM MVS, OS/390, z/OS
- IBM OS/2
- IBM VM, z/VM
- IBM VSE
- KA9Q or NOS
- Microware OS-9
- Music (Multi-User System for Interactive Computing)
- NCSA FTP Server for MS-DOS (CUTCP)
- Novell Netware
- Novell Netware Print Services for UNIX

- TOPS20
- UniTree
- VMS or VMS (including Multinet, MadGoat, UCX)
- Wind River VxWorks
- WinQVT/Net 3.98.15
- Xecom MicroRTOS

19.1.7 Other 其他

Indy10 的其他明显的改变和改进包括但不止这些：

- 增加了 Server intercepts 使你能够记录 FTP 服务器的日志，它们的工作方式和 client intercepts 很像
- 添加了 Systat UDP 和 TCP 客户端和服务器
- 添加了一个 DNS 服务器控件
- 通过代理的 HTTP 链接已经被添加。
- 添加了 TIdIPAddrMon 以监视所有的 IP 地址和所有的网络适配器
- IP 支持
- 已经实现了一个一次性密码系统为一个客户端用的 OTP calculator 和一个 user manager 控件。 这支持 MD4, MD5, 和 SHA1 hashes。

19.2 Core Rebuild 内核重构

Indy10 的内核已经经历了巨大的结构性变化。这会使一些用户的内核代码失效，但是我们已经尽我们所能的去保持尽可能多的协议和应用等级的兼容性了。有时，可能看起来 Indy 的后勤维护人员并不关心这些改变对终端用户的影响。但并不是这样的。每个接口的修改都经过了评估，必定是利大于弊的。每个做的修改，都被设计为能够以最小的代价更新现存的代码。Indy 的后勤维护人员也使用 Indy 的私人版本和企业版本，所以我们的团队也能够感受到每个修改的变化。

Indy 团队相信，改进总是要付出代价的。通过接口上的一个小的修改，就能获得更大的回报，以及一个更好的 Indy。不进行这些修改的话，未来的道路就会被阻塞，包袱就会被遗留下来。就像 Indy8 的 Winshoes 在 Indy8 到 Indy9 的过程中增强了抽象，Indy10 也会。

Indy 的首要原则之一是易于编程，这是基于阻塞模型的。这使得开发起来很简单，用户的代码不需要序列化(serialization)。Indy 的设计目标是易于使用，速度快到足够满足 95%的终端用户的需要。

然而在大型安装(installations)中，这导致了 excessive context switching 和积累了过高的线程开销。这个限制只在大型安装中出现，在设计的很好的应用中出现在大约 2000 个并发线程时。在大多数应用中，用户代码中的限制早在 Indy 的限制出现前就出现了。

传统地，为了服务剩下的 5%的用户，就不得不抛弃写起来很简易的代码而使用复杂和难于维护的代码比如直接重叠 I/O(direct overlapped I/O),I/O 完成端口(I/O completion ports),或者高度分离的使用线程的工作单元也就是非阻塞 sockets。Indy10 使用阻塞模型以保持 Indy 的简便易用，同时在内部增强性能。为了达成目标，Indy10 使用高级网络接口并高效地把它转化为个用户友好型阻塞模型。Indy 10 因此打算服务 99.9%的社区需要，只剩极少数的人还需要为很特殊的情况定制代码。

Indy 10 以很多方式完成这个目标，但是纤程(fibers)是达成这个目标需要的最关键的技术。

纤程和线程十分像，但是更加灵活，如果使用合理的话占用的系统开销也比线程小。

19.2.1 IOHandler Restructuring　IOHandler 重构

为了增强性能，Indy10 中的 IOHandlers 已经被重构并被给予了个更加重要的角色。在以前，IOHandler 的角色只是做非常基础的 I/O 操作，包括仅仅以下功能：

- Open (连接)
- Close (断连)
- Read raw data 读原始数据
- Write raw data 写原始数据
- Check connection status 检查连接状态

这个角色使得可选的 IOHandlers 被创建，并从源，而不是个 socket 进行 I/O 操作。甚至默认的 socket I/O 都是通过使用个默认 IOHnadler 来实现的。

因为功能十分少，IOHandlers 的实现十分的直接。然而这经常导致 IOHandlers 用没那么有效的方法来实现他们的 I/O 操作。比如，一个 I/O handler 可能从一个本地文件接收数据来进行写操作，但是并没有办法知道这情况，因为它对所有数据都只有一个 write 方法。即使 I/O handler 有进行高效文件读取的性能，它也不能使用这性能。

Indy 10 的 IOHandlers 不止实现了最小的低级方法，还实现了高级方法。这种高级方法之前是由 TIdTCPConnection 实现的。

可以创建之前的那个只实现了低级方法的 IOHandlers。基础 IOHandler 包含了高级方法的默认实现，高级方法使用了低级方法。但是每个 IOHandler 都可以重写附加的高级方法以提供特定于 IOHandler 的更优化的实现。

19.2.2 Network Interfaces 网络接口

Indy9 只实现了一个网络接口。在 Windows 中，这个接口是 Winsock 而在 Linux 中是 stack。Indy 10 在实现了这些接口之外还在 Windows 下实现了一些更有效率的接口。目前在 Linux 下还没有实现其他有效率的接口，但是以后会有的。由于它的网络语义学(networking semantics),Linux 下对其他接口的需求并没有那么迫切。

一些附加(additional)接口在所有版本的 Windows 下都不可用，并且只应该在服务器的实现或者在客户端连接数十分巨大的系统中使用。在一般的客户端应用中根本不需要实现这些。

附加网络接口是：

- Overlapped I/O 重叠I/O
- I/O Completion Ports　I/O 完成端口

一般地，为了使用 overlapped I/O ，以及更加有效率的 I/O Completion Ports， 就必须写复杂的和极其难懂的代码。但是在 Indy10 中，Indy 在之前已经帮你处理好了这些细节，并给开发者提供了友好易用的接口。

19.2.3 Fibers 纤程

为了扩展我们的线程支持，Indy 10 还包含对纤程的支持。纤程是什么？简单来说，它也是一个“线程”，但是它是由代码控制的，而不是由操作系统控制的。实际上，可以认为线程是一个高级纤程。纤程和 Unix 用户线程(Unix user threads)很相似。

线程是操作系统用来分配时间的基本单元。一个线程包含它自己的栈(stack)，特定的寄存器(processor registers) ，以及一个线程上下文(thread context)。线程们自动地被操作系统调度时间。

通常来说，纤程在一个设计良好的多线程应用程序中并没有任何优势。然而当纤程配合一个可以提供相关信息的的智能调度器(intelligent scheduler)使用的时候就可以大大的提高效率从而提高性能。

多个纤程可以使用单个线程来运行。单个纤程也可以被多个线程运行，尽管同时只能有一个线程运行它。你可以在内部运行多个纤程。完成这项任务的代码可以十分复杂，但是 Indy 已经所有都帮你处理好了。所有的 Indy 控件，不管是客户端还是服务器，都支持纤程，大部分都以种透明的方式。

通过使用纤程，Indy 得以翻译复杂和不友好的更低层网络接口为对开发者友好的接口。

目前 Indy 只在 Windows 下实现了纤程。

19.2.4 Schedulers 调度器

Indy 的 fiber weaver 是个调度器，它调度纤程们到一个或多个线程中去。纤程储存工作项目到一个工作序列中然后等待。当纤程的工作项目被完成后，调度器把纤程放到一个可以被调度的纤程列表中。

操作系统时间调度器以一种智能的方式调度线程，但是由于在系统的所有任务间每个线程都是普通和泛型的(generic)，它们对线程的信息掌握十分有限。操作系统调度器只可以基于等待状态和线程的优先级来调度他们。

Indy 的 fiber weaver(线程调度器)使用特定于任务的高级信息来决定调度需求，优先级和等待状态。通过这个方法，Indy 可以大大的减少 context switches (上下文切换)的数量，上下文切换的发生和工作完成有关。这大大的增加了性能。

context switche 是当一个线程被挂起，另一个被调度时。为了做这件事，操作系统必须先中断处理器的执行路径并通过在内存中存储一些处理器寄存器(processor register)保存线程的上下文。它之后必须通过从另一个内存地址加载处理器寄存器来恢复另一个线程，并继续线程执行路径。

线程调度器必须平衡需求以减少上下文切换，同时还要保证每个线程都能得到足够的处理时间(processor time)。切换行为常常增加系统开销并降低性能。切换会不断导致不必要的内部线程等待依赖(wait dependencies)和反应迟钝，因为线程得不到足够的处理时间。

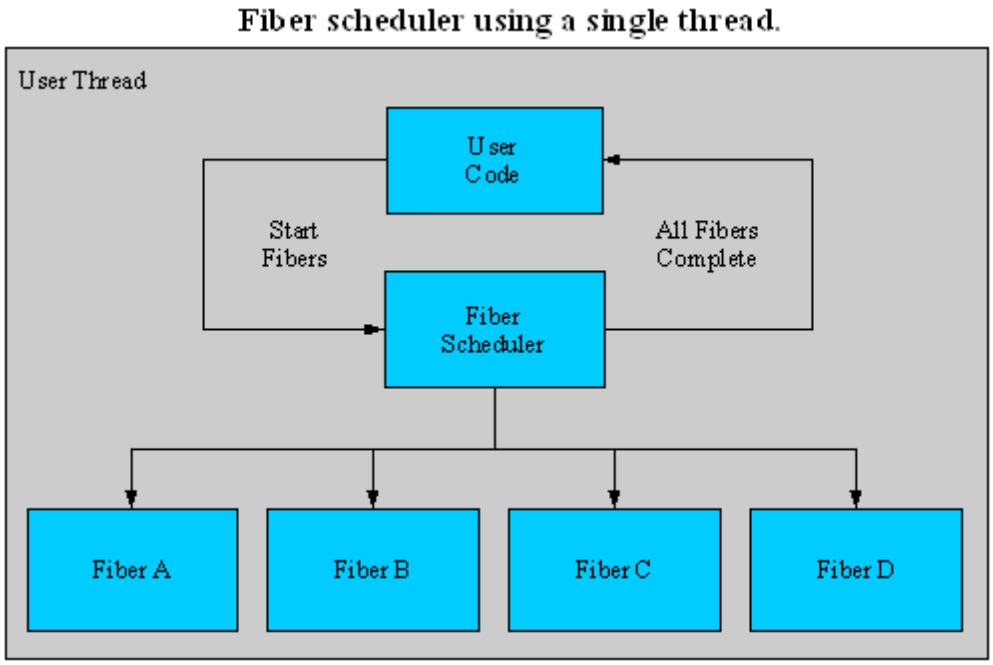
为了管理这个，操作系统定义了一个线程每次切换可以收到处理时间的一个时间片(quantum),或者个最大时间片(maximum time slice)。大部分情况下，当时间到达了，一个线程会进入等待状态以放弃优先级。等待状态显式地，或者更常是通过无法立刻完成的操作系

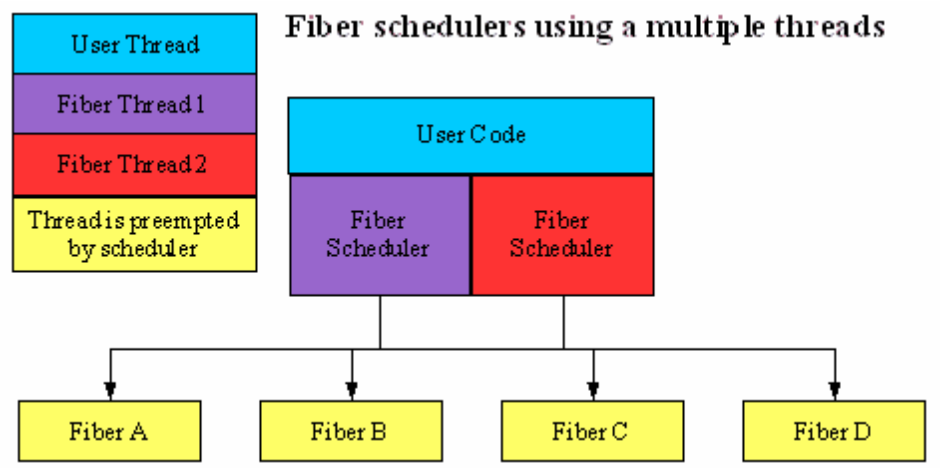
统调用(operating system call)或者 I/O 调用(I/O call)来隐式地发生。当发生这样的调用的时候，操作系统接受了对优先级的放弃并切换到另一个线程。如果线程正等待 I/O 或者其他一些阻塞操作，他会被放置在等待状态并不会被考虑调度时间，直到请求的操作发生了，或者时间超出了。

Indy 的 fiber weavers 以一种类似的风格工作，但是使用更多的信息以更高级的方式确定等待状态。纤程们可以被提前确定为是等待状态，而不需要上下文切换到它们中并等待它们进入等待状态。Indy 还分离了纤程和链引擎(chain engines)的工作，链引擎进行大部分低等级工作。

工作的分离使得可以使用更加有效率的网络接口，比如 I/O 完成端口。I/O 完成端口更加有效率，因为它们与硬件接口离的更近。Winsock 和其他与硬件接口层离的较远的调用必须与内核通讯以完成对硬件接口的实际调用。与内核通讯的调用必须经历上下文以实现功能。因此每个 Winsock 调用仅仅为了实现他的功能就常牵连许多不必要的上下文切换。

调度器可以使用单线程，多线程，需求线程(threads on demand)，甚至一个线程池。





19.2.5 Work Queues 工作队列

工作队列是先入先出队列，这个队列存储了纤程请求的工作项目。大部分这个功能是完全对开发者透明的，Indy 在内部使用这个功能。

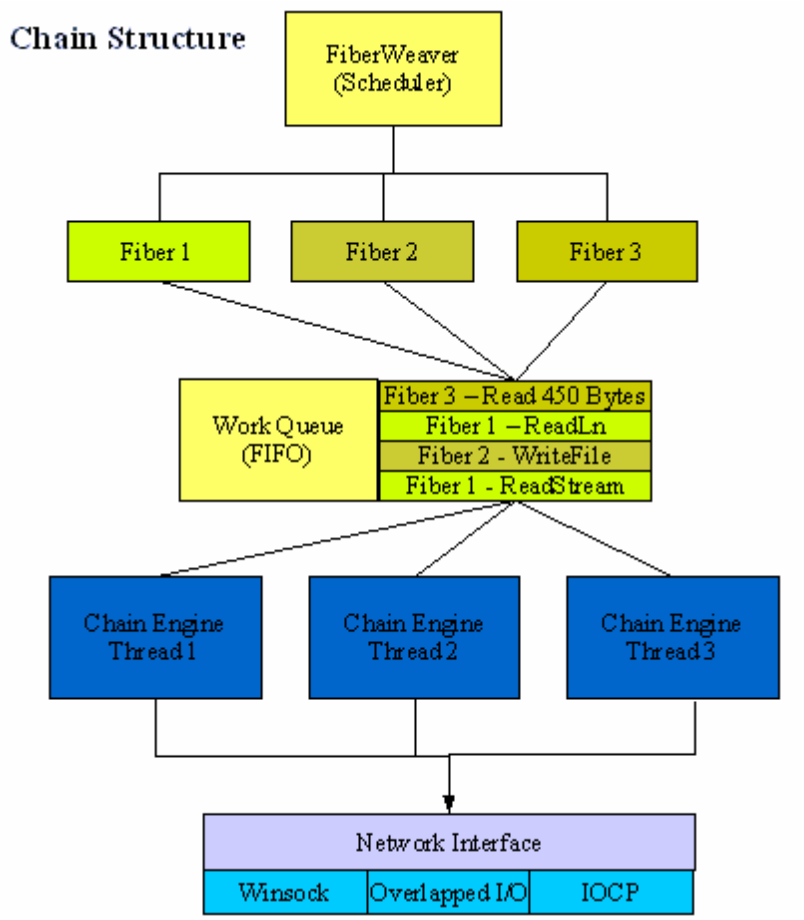
19.2.6 Chains 链

Indy 中的工作队列系统，调度器，和链引擎都被叫做链。尽管链被 Indy 使用，他们不止限于 Indy 的内部使用，并且也有终端用户应用程序。

当使用链的时候，一个基于链的 IOHandler 存储工作项目到有关的工作队列中。纤程然后被挂起直到一个工作单元被完成。这是因为纤程无法做任何事直到某工作的结果可以被处理。每个 IOHandler 方法都被简化为一个或多个工作任务。为了最佳性能，每个方法应该被分解为尽可能少的工作任务。

一个调度器然后在它们等待它们的工作项目被处理时管理纤程。

最终，一个链引擎处理工作队列中的请求并和调度器进行通讯。



19.2.7 Chain Engines 链引擎

链引擎是是链系统的最低等级。链引擎实现所有的真正的输入和输出。链引擎可能包含单个线程或者多个。

链引擎的工作是从工作队列中提取任务并完成任务。当每个任务完成时，链引擎通知调度器，然后调度器评估哪个纤程应该被考虑调度。链引擎然后继续工作队列中的下一个任务。

如果工作队列中没有项目，链引擎保持空闲状态。

有多个链引擎类型可以被用来实现 I/O 完成端口，Winsock,重叠 I/O，或者其他的。

19.2.8 Contexts

在 Indy9 的服务器中，链接特定(connection specific)的数据被作为线程类的一部分被存储。实现这个要不然通过使用 thread.data 属性要不然通过继承对应的 thread 类然后添加新的字段或者属性来存储。这能起作用，因为每个链接都有配套一个指定的线程。

Indy10 服务器的实现和 Indy9 的不一样。多线程并不和个单链接关联。实际上它可能根本没有使用多线程，而是使用了纤程。因此，之前的在线程内存储数据的方法不再使用了。

Indy10 使用一个叫做 contexts 的新概念来替代。Contexts 存储链接相关的数据，并由 Indy 来管理和追踪。