hProxyN 关键技术

阳光男孩

http://www.65536.cn/work/2008/hProxyN/

hProxyN 关键技术

- C# 调用 Win32 API
- .Net 的 Reflection 机制
- Thread Pool 模式及 .Net 对 Thread Pool 的支持

C# 调用 Win32 API

托管的世界很美妙

- .Net 提供的“托管”环境是一个美妙的世界
  - 垃圾收集，不必担心内存泄漏
  - 强名称， DLL 不再是“一场恶梦”
  - JIT 即时编译，移植性更好

非托管的世界很弓虽大

- Windows 是用 C++ 语言编写的
  - Win32 API 是以 C++ 函数的形式提供的
  - Win32 API 可以操作 Windows 的几乎所有部分
- .Net 的功能有限
  - 许多 Win32 API，在 .Net 中没有对应

怎么办？

- interop—— 在 .Net 程序中调用非托管代码
- 三种调用方法
  - COM 组件，用 tlbimp 封装成 interop 类，就可以像 .Net 组件一样操作——可以操作 Outlook 类库；Win32 API 通常不是 COM 组件
  - P/Invoke，直接调用非托管代码
  - Visual C++.Net 包装 wrapper

P/Invoke

- hProxyN 使用 P/Invoke 方式调用 Win32 API
- hProxyN 需要调用 HTTP Server API 1.0
  - HTTP Server API 用于管理 http.sys 系统组件
  - http.sys 实现了 HTTP/1.1 协议
  - 相关资料： http://go.6.cn/chr8

P/Invoke 的工作原理

- 通过 P/Invoke 调用非托管代码时：
  - 寻找、加载非托管代码所在的 DLL
  - 查找函数在内存中的地址
  - 将参数从托管类型转换为非托管类型，压栈
  - 跳转到函数入口
  - 执行非托管代码直到返回
  - 将栈顶的返回值从非托管类型转换为托管类型
- 上述操作，由 CLR 自动完成，无需编写代码

开发者要做什么？

- 开发者需要告诉 CLR：
  - 要调用的是什么函数、位于哪个 DLL
  - 调用约定：参数的次序？谁负责清除栈？
  - 函数的参数和返回值是什么非托管类型，对应什么托管类型，如何进行转换——难点

要调用什么函数？

- C 语言的函数声明，来自 http.h
  - HTTPAPI_LINKAGE ULONG WINAPI
    HttpReceiveHttpRequest(
    IN HANDLE ReqQueueHandle,
    IN HTTP_REQUEST_ID RequestId,
    IN ULONG Flags,
    __out_bcount_part(RequestBufferLength, *pBytesReceived)
    OUT PHTTP_REQUEST pRequestBuffer,
    IN ULONG RequestBufferLength,
    __out_opt OUT PULONG pBytesReceived OPTIONAL,
    IN LPOVERLAPPED pOverlapped OPTIONAL
    );

要调用什么函数？

- 转换成对应的 C# 声明
  - 告诉 CLR 从 httpapi.dll 查找 HttpReceiveHttpRequest
  - [DllImport("httpapi.dll", EntryPoint = "HttpReceiveHttpRequest")]
    public static extern uint HttpReceiveHttpRequest(
     System.IntPtr ReqQueueHandle,
     ulong RequestId,
     uint Flags,
     System.IntPtr pRequestBuffer,
     uint RequestBufferLength,
     System.IntPtr pBytesReceived,
     OVERLAPPED pOverlapped
    );

类型的对应

- 简单类型， CLR 会自动对应
  - char <=> System.Byte
  - unsigned long <=> System.UInt32
  - int <=> System.Int16
- C 语言头文件中广泛使用 typedef
  - typedef ULONGLONG HTTP_OPAQUE_ID, *PHTTP_OPAQUE_ID;
  - typedef HTTP_OPAQUE_ID HTTP_REQUEST_ID, *PHTTP_REQUEST_ID;
  - 因此， HTTP_REQUEST_ID <=> System.UInt64

结构体类型的对应

- C 语言的结构体
  - typedef struct _HTTP_COOKED_URL
    {
    USHORT FullUrlLength;
    USHORT HostLength;
    USHORT AbsPathLength;
    USHORT QueryStringLength;
    PCWSTR pFullUrl;
    PCWSTR pHost;
    PCWSTR pAbsPath;
    PCWSTR pQueryString;
    } HTTP_COOKED_URL, *PHTTP_COOKED_URL;

结构体类型的对应

- 转换成对应的 C# 结构体
  - [StructLayout(LayoutKind.Sequential)]
    public struct HTTP_COOKED_URL
    {
    public ushort FullUrlLength;
    public ushort HostLength;
    public ushort AbsPathLength;
    public ushort QueryStringLength;
    [MarshalAs(UnmanagedType.LPWStr)]
    public string pFullUrl;
    [MarshalAs(UnmanagedType.LPWStr)]
    public string pHost;
    [MarshalAs(UnmanagedType.LPWStr)]
    public string pAbsPath;
    [MarshalAs(UnmanagedType.LPWStr)]
    public string pQueryString;
    }

结构体类型的对应

- 结构体中每个成员的偏移量要与 C 版本一致
  - 我的调试方法：用 C 和 C# 语言创建同一个结构，给某个成员赋相同的值，打印出结构体的指针值，然后用调试工具查看内存中相应位置的结构体是否有区别
  - 必要时，使用 [StructLayout(LayoutKind.Explicit)] 和 [FieldOffset(偏移值)]，显式指定每个成员的偏移量
- 定长数组、字符串、 union 等都有相应的规则（略）

指针类型的对应

- 指针 <=> System.IntPtr
  - 申请一块非托管内存： Marshal.AllocHGlobal
  - 将对象复制到非托管内存： Marshal.StructureToPtr
  - 这块内存的地址就是指针的值
  - 从非托管内存复制对象： Marshal.PtrToStructure
  - 释放非托管内存： Marshal.FreeHGlobal
- 如要避免对象的复制，可以使用 GCHandle 类，在托管内存上获取指针

好麻烦……

- PInvoke Interop Assistant
  - 从头文件生成 P/Invoke 声明的工具
  - C 语言非常复杂，此工具只能支持 C 语言的子集
  - http://www.CodePlex.com/clrinterop

but...

- .Net 已经封装了 HTTP Server API 1.0
  - 10 月 8 日我偶然看到 System.Net.HttpListener 类
  - 经试验， HttpListener 类就是 HTTP Server API 1.0 的托管封装
  - 参考资料： http://go.6.cn/n2fy
  - 因此， hProxyN 不再需要使用 P/Invoke
- 回到托管的世界～

Reflection

Reflection

- 通过 Reflection(反射)， .Net 程序可以了解自身或其他类的结构、函数等元数据
  - //Without reflection
    Foo foo = new Foo();
    foo.Hello();
  - //With reflection
    Type t = Assembly.GetCallingAssembly().GetType("FooNamespace.Foo");
    t.InvokeMember("Hello", BindingFlags.InvokeMethod, null, Activator.CreateInstance(t), null);
  - 上述举例来自： http://go.6.cn/740b

hProxyN 的插件支持

- 通过 Reflection 加载插件
  - 所有插件均继承于一个基类
  - 给出插件的类名，找到这个插件类的信息
  - 找到插件类的构造函数，调用构造函数，转换为插件基类的引用
  - 调用插件基类的 virtual 函数，以执行插件的函数

hProxyN 的插件支持

- 插件的运行时机
  - 借鉴 Apache HTTP Server
    - 请求 - 响应过程中，提供数十个 HOOK
    - 插件初始化时，在所需 HOOK 上注册函数指针
  - hProxyN 的插件
    - 请求 - 转发 - 响应过程中，产生若干个事件
    - 插件初始化或其他时候，监听所需的事件
    - 事件发生时，插件有机会运行
    - 插件可以干预当前的请求 - 响应过程，也可以控制 hProxyN 核心及其他插件

hProxyN 的插件支持

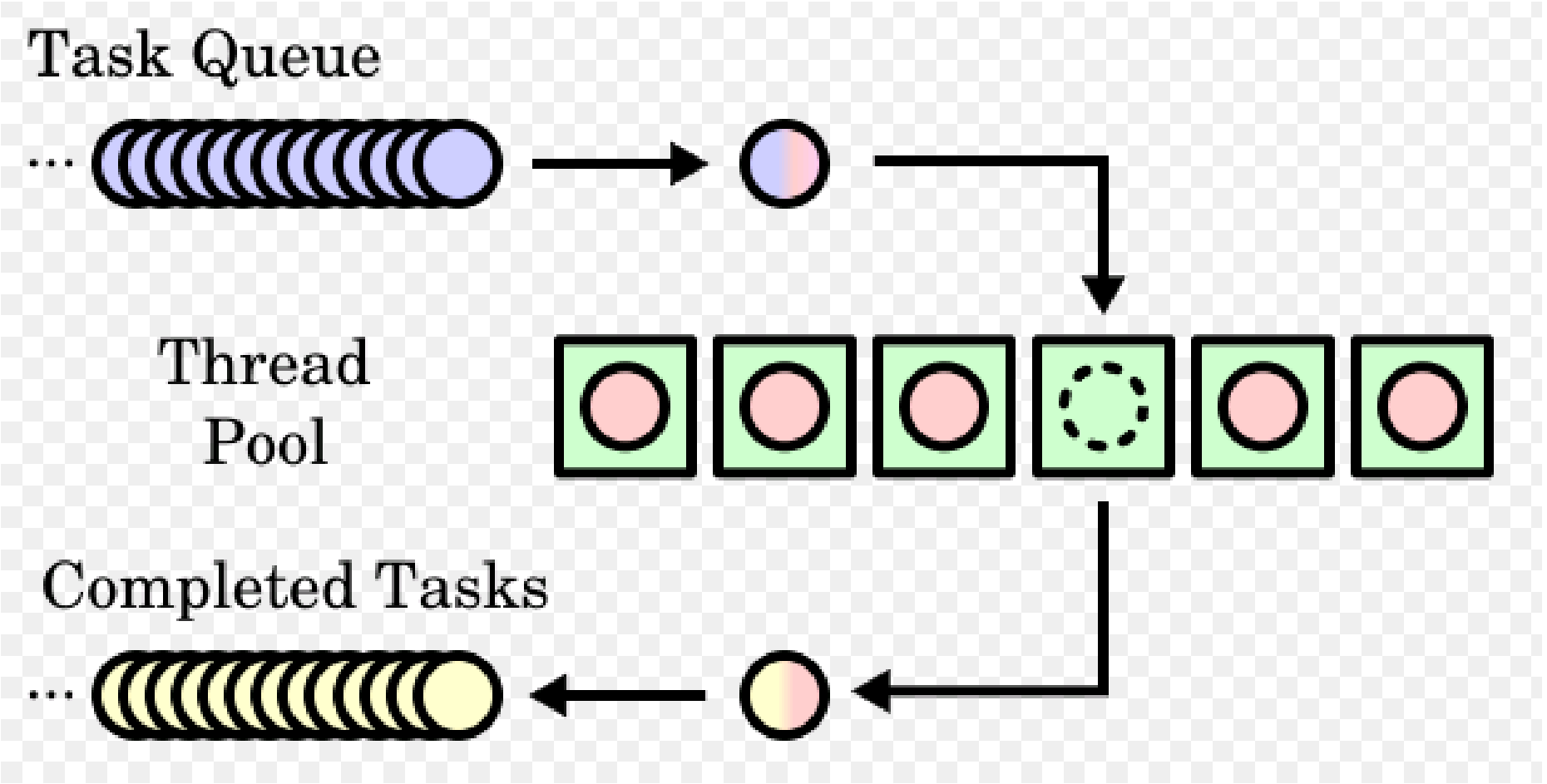
- 插件可以干什么？
  - 大部分功能都可以通过插件完成！
- 核心必须干什么？
  - 作为 Windows 服务运行
  - 接受 HTTP 请求，发送 HTTP 响应
  - 加载插件
  - 为插件提供必要的服务

ThreadPool

Thread Pool

- In the thread pool pattern in programming, a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available.

Thread Pool



- http://en.wikipedia.org/wiki/Thread_pool_pattern

ThreadPool 管理算法

- 需要一个算法来管理线程池
  - 在任务增多时，创建更多线程
  - 在空闲线程较多时，销毁一部分线程
- 线程池管理算法对性能的影响
  - 创建太多线程：浪费内存资源
  - 销毁太多线程：稍后重新创建时消耗时间
  - 创建线程太慢：等待时间长、用户体验差
  - 销毁线程太慢：造成其他进程的饥饿

System.Threading.ThreadPool

- .Net 通过 ThreadPool 类提供线程池支持
  - 每个进程只能有一个线程池
  - 第一次调用 ThreadPool 类时， .Net 创建线程池
  - 线程池由 .Net 自动管理，开发者可以设定参数
- ThreadPool.QueueUserWorkItem
  - 提交一项任务
  - 可以带一个 Object 参数

感谢大家对 hProxyN 项目的支持

- 下次汇报： 2008-10-23
  - 产品设计
- 项目主页
  - http://www.65536.cn/work/2008/hProxyN/