协议树:在网络模型的每一层实际上是把上一层的数据加个头部数据，那么在解析的时候，其实是以树形结构一层一层的解析的，每个解析器解析了自己的头部后，丢给下一个解析器，由于每一层的协议很多，对应也那么多个解析器，解析器其实也可以构成一个和协议一样的树形结构，我们想要添加一个自定义的解析器，也需要向这个树里添加一个节点。

WireShark插件（dll）编写步骤

wireshark启动时会加载所有$installdir/plugins/$buildversion/\*.dll，并且加载每个dll的以下两个导出函数和版本号。

\_\_declspec(dllexport) void plugin\_register(void)

\_\_declspec(dllexport) const gchar version[] = VERSION;

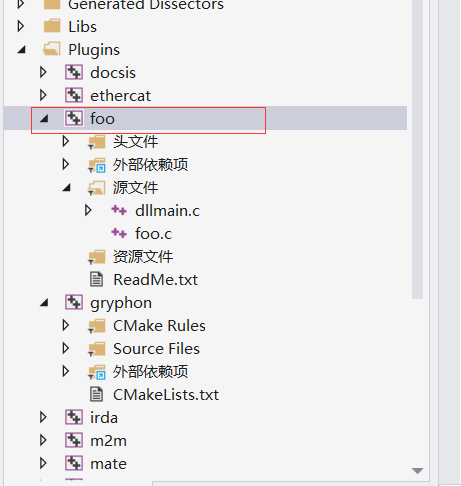
\_\_declspec(dllexport) void plugin\_reg\_handoff(void)

    插件在这两个函数里完成dissector的注册和初始化工作，其中plugin\_register注册插件，plugin\_reg\_handoff注册协议。

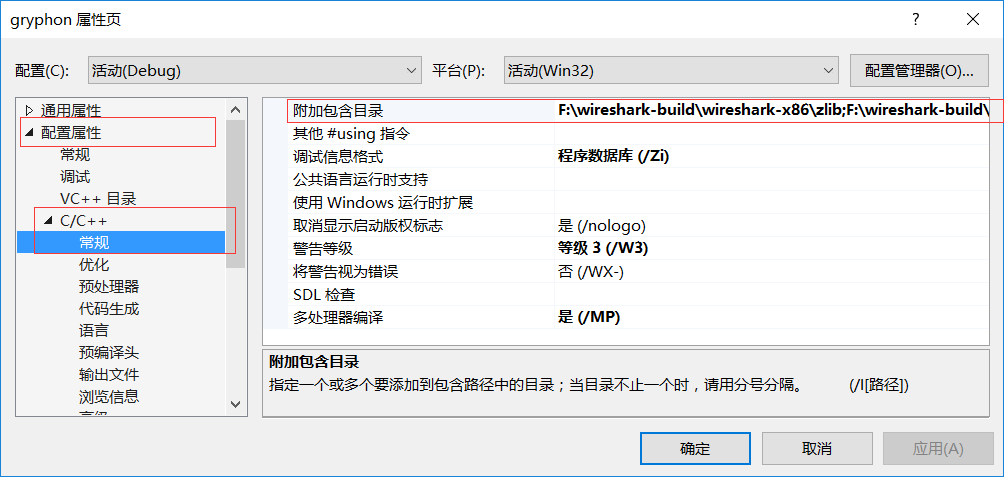
步骤1：

编译好WireShark工程之后。

1. 在编译好的工程的Plugings目录下新建一个dll工程foo



1. 在示例插件gryphon项目上右键---------属性---------配置属性-------C/C++------常规。将附加包含目录里的路径全部复制到我们foo项目的附加包含目录里。这一步解决了插件工程需要的大量外部依赖项。



1. 在示例插件gryphon项目上右键---------属性---------配置属性-------连接器------。将附加附加依赖项里的路径全部复制到我们foo项目的附加依赖项里。这一步解决链接时所需库文件等问题。
2. 编写我们自己的解析器：
3. void proto\_register\_foo(void)
4. {

//注册协议

1. proto\_foo = proto\_register\_protocol(
2. "FOO Protocol", /\* name \*/
3. "FOO", /\* short name \*/
4. "foo" /\* abbrev \*/
5. );
6. }

//此函数用于解析交给它的packets。

1. static int dissect\_foo(tvbuff\_t \*tvb, packet\_info \*pinfo, proto\_tree \*tree, void\* data)
2. {
3. col\_set\_str(pinfo->cinfo, COL\_PROTOCOL, "FOO");
4. /\* Clear out stuff in the info column \*/
5. col\_clear(pinfo->cinfo, COL\_INFO);
6. return 0;
7. }

//创建一个dissector handle，它和foo协议及执行实际解析工作的函数关联。接下来将此handle与UDP端口号关联，以便主程序在看到此端口上的UDP数据时调用我们的解析器。

1. void proto\_reg\_handoff\_foo(void)
2. {
3. static dissector\_handle\_t foo\_handle;
4. foo\_handle = create\_dissector\_handle(dissect\_foo, proto\_foo);
5. dissector\_add\_uint("udp.port", FOO\_PORT, foo\_handle);
6. }

//导出对应接口

\_\_declspec(dllexport) void plugin\_register(void)

{

proto\_register\_foo();

}

\_\_declspec(dllexport) const gchar version[] = VERSION;

\_\_declspec(dllexport) void plugin\_reg\_handoff(void)

{

proto\_reg\_handoff\_foo();

}

5.测试

把生成的dll放在Wiresharke/plugings/版本号/ 目录下。

写一个发送udp的小程序。往之前指定的端口号发数据包

