GEW宣讲稿

尊敬的各位老师、亲爱的同学们:

大家晚上好！

我们是第11小组，简称单身狗小组。

总所周知，要想搞一个大新闻的话，就必须要有一个亮瞎的标题。所以这个我们项目一就叫做Great Echo Wall，也就可以理解成一个比较高的回音壁，不知道比美国的华莱士高到哪里去了。

在这里，主要分成6个部分对项目的设计及实现过程以及遇到的一些问题进行讲解，之后会进行简短的演示。代码及各相关文档已经拷贝到 `\11-余泽江` 文件夹下，有兴趣的同学可以参考。

第一个部分，也就是项目的开发环境，首先，最大的环境是Mac下的一个Windows虚拟机环境。当然，实际上开发环境和运行环境是可以完全无关的，但由于没有生成相应的部署项目，也就是安装包，目前也就只能在Visual Studio 2013下运行，理论上生成的Debug及Release文件也能在具有相同版本的 .NET Framework 环境下运行，所以可能不能在机房里直接运行该项目。

这里简单介绍一下Visual Studio相关的基本概念，在连基本概念都完全不了解的情况下就讨论Visual C++ 6.0好不好的问题肯定是要吃亏的。Visual Studio 的第一个版本是 97 年发布的，简单地说就是整合了Visual C++ 5.0 和 Visual Basic 5.0 以及其他一些东东，所以要说VS的话最好说清楚版本，VS 97 比 6.0 还要早。当然，一直到Visual Studio 2010，你都可以单独下载VC、VB，所以说类似”VC自古以来就是VS不可分割的一部分”这种观点是明显错误的，既不是自古以来，也并非不可分割。Visual Studio 2002 是一个巨大的里程碑，从这里开始VS就已经包括了 .NET Framework，自然C#也就是从这里开始出现的。从08-13版的总体差别都不算太大，主要就是各个组件的版本更新，以及增加了对一些特定项目的内置支持。

Visual Studio 沿袭了微软一贯的风格，和Windows一样有各种版本。其实每个版本的功能都是不一样的，不过相信很多人也没真的知道旗舰版到底都多了哪些功能。这里也不做过多介绍，如果是开发非商业项目可以使用社区版，在功能上和专业版式一样的。当然，如果想要专业版的话，也可以通过DreamSpark网站认证下edu邮箱后就能免费下载，还有很多其他好玩的东西比如Windows Server、SQL Server也都是对高校师生免费的，有兴趣可以尝试。

现在到了答题时间，只要回答问题呢就有果冻，多答多得。第一个问题是“【翻译腔】VS2013好处都有啥？谁说对了就给他。”或者提示一点，为什么Python的包引入是from sth import sth而非import sth from sth，以及面向对象对程序员而非编译器都带来的最大便利是什么？

* 代码提示是一个在开发效率上十分有用的功能。还是先讨论一下面向对象的问题。对于编译器来说，我完全可以设计一个编译器把所有类的成员函数都变为全局函数，然后在参数列表里加一个指针变量就行，实际上C#的扩展方法机制就是这样实现的，也就是说可以存在一个编译器让编译后的代码里面向过程和面向对象的机器码是完全一样的，也就是说面向对象机制并不是对代码执行上的优化。那为什么还要用面向对象呢？因为面向对象在开发过程中会有极大的效率优化，代码提示就是其中的一部分。在面向过程的开发中，往往会用到各种全局函数和全局变量（在使用上宏等效于函数或变量），这样即便有代码提示，也会列出当前所有的全局函数和全局变量，没有实质的简化；而在面向对象时，由于已经写明了对象，这样就只需列出这个对象具有的成员变量和成员方法，然后直接Tab键就OK了。有的人可能会问，为什么需要代码提示呢？不应该直接记下来所有的函数么？简单的说，问这个问题可能是因为用到的库还不够多，不是说所有项目都能一个STL一个Socket就解决的。
* 语法检查和代码提示其实在VS里面都是intellisense，所以要说他是一个东西还是两个东西其实也都说得过去。用高版本VS以及任何当代的IDE开发的时候，基本都不会说编译的时候有不是0 error 0 warning的情况，为什么呢？因为你打错的时候别人已经告诉你了，不需要还等到编译的时候再提示。这里我的观点很简单，不要浪费时间在语法错误上，甚至可以说语法错误都根本不是错误，至于说由语法（比如说花括号或者++的位置）导致的逻辑错误，那基本也都是不作死就不会死的案例。
* 最后一点也就是关于Visual Studio为什么要叫Visual Studio不叫Visual Languages，虽然很多人看来Visual Studio就等于Visual C++ 加上Visual Basic加上Visual C# 加上Visual F# 加上A lot of languages，对吧？当然对，实际上也远不止微软自己的语言，包括JavaScript、Python、IronPython、IronRuby、IronScheme等等各种语言都可以在VS下开发，而且还提供了大量的模板。至于说模板的意义并不在于没有模板就不能开发相应的项目，MFC也就是一个库完全可以新建一个控制台项目然后自己引入。模板的两个重要功能（对我来说），一个是告诉你某个语言能做哪些东西，另一个是提供一种推荐的项目结构参考，比如分哪些文件夹、哪些公共类、相互的调用方式、甚至命名方式等等，虽然说推荐的并不一定是最好的，但也肯定不是最差的。

最后列一个为什么不推荐人工进行语法检查的原因，相信很多人在人人空间微博贴吧朋友圈都已经看过了。说白了就是，人类本身就不适合干这个，并且不是因为缺陷，而是因为优势。人类的模式识别机制本身就要比计算机强大很多，当然也可以说不精确很多，所以不管说你是换了发型换了眼镜甚至化了妆我都能认得出来，对吧？当然不一定，要看你化妆的水平了。总之意思就是说，如果你有很简单的语法错误你看了几遍都没发现，这是很正常的，和智商和视力都没有任何关系。也是因此，不推荐浪费时间在这个上面。

接下来讨论一下Visual Studio的项目层次。相信绝大部分（虽然我想说所有的，但出于严谨的角度）人，都是在新建一个项目的时候选择的是“创建解决方案”吧，那既然如此为什么还要分解决方案和项目呢？答案都可以根据这里的说明推出来，而且回答有果冻哦。

* 很好，一个很明显的地方，每个项目文件都带有语言的名称，也就是说项目是语言相关的。而解决方案呢？一个解决方案也就是一个产品，而产品本身是语言无关的，具体到每一个功能模块，可以根据实际情况选择具体的语言，然后只要模块间的调用接口设置好就可以了。
* 对的，一个项目（也就是一个模块）是可以多次重复利用的，相信很多同学都见过 .dll 文件，其实 .dll 和 .exe 是同一个格式（PE）不同的后缀，.exe 大家都知道是用来被我们运行的，那 .dll 呢？可以看成是用来被别的程序运行的，并且不一定要运行 Main 函数，只要是 public 的函数都可以被单独运行并返回结果，也就可以理解成是二进制形式的头文件。所以并不是所有的项目都要生成对应的 .exe 文件，有的可能只需要给其他的项目提供支持，这时候就可以生成为 .dll 文件。

说完了IDE，那就再来说说语言。语言的话这里用的是C#，嗯，首先，大家和我一起读三遍，See Sharp、See Sharp、See Sharp。很好，其实同学们应该也都看得出来，C#就是一个音名，即C调的升半调，和D flat，也就是D调的降半调是同一个调，都是277.18Hz，所以C#也叫Db。当然，这只是一个梗，万一别人不懂音乐找不到笑点也是很正常的。

C#有哪些特点呢？第一点就是它确实参考的C、C++的语法，至于这是优点还是缺点呢其实也有争议，不过鉴于国内大部分学校都是用C语言教学所以要入手C#也会比较容易。第二点就是dot net，.NET的核心组件就是CLI，即公共语言运行时，所有语言都被编译成与语言无关也与机器无关的代码，当然实际上也是机器码，每条指令有固定结构而且也有对应得汇编语言，只是目前还没有机器能直接运行而已。当然，重点在于 .NET 提供了一个足够丰富也非常庞大的类库，由于上面的机制所有 .NET 语言都能用这个类库，包括VB、C++，虽然类库本身就是用C#写的。最后一点也就是C#是完全面向对象的，语法上不存在全局变量和全局函数，一切都是对象（在CLR里类本身也是对象）。

作为 .NET 语言，最通常的运行方法是二次编译，即先由C#源代码编译成MSIL，然后在运行过程中，由JIT编译器根据需要把相应的MSIL编译成当前机器的机器码。看起来这样可能比较繁琐会拖慢速度，对吧？实际上来说，JIT编译器能根据实际的具体调用情况对生成的机器码进行特殊优化，粒度可以比直接编译要细很多，而且一次编译后会将机器码进行缓存，之后如果需要再用到这部分代码无需再次编译。总结就是，第一次会比较慢，多用几次就快了。另外，.NET也提供了一次性全部编译的支持，方式和Android的ART类似。另外强调一点，C#的编译器是C#写的，这点能理解吗？我印象中很多C语言的死忠粉都很难理解这一点，但要是说C语言编译器是C语言写的很多人就觉得是理所应当的。

虽然是 .NET Framework是在Win32 API和COM API之上的，但必要的时候也还是可以直接调用底层非 .NET代码的，后面会具体讲到。

C#的话其实也是经历了一个比较长的发展过程的，C#1的时候比Java都很难用，后面加的语法糖越来越多。另外，加入了 .NET 之后，实际上原有的语言都发生了版本分裂，比如对于VC++来说，高版本的VS里面都有两个子模板目录，Win32和CLR，里面都有一个控制台模板，即“Win32控制台”和“CLR控制台”，也就分别对应的C++和C++/CLI的控制台程序。也就是非托管C++和托管C++的区别，在托管C++中可以直接使用 .NET 类库。另外，托管C++中也并非所有代码都是托管代码，大部分仍然是非托管代码，暂时就绕到这里。

在很多人看起来可能 .NET 和Java很像，但实际上只是实现方式有点像，设计理念是完全不一样的，不过现在来说的话，.NET也有非Windows实现，JVM上也有很多其他语言，比如Scala等等。

界面库的话用的是微软的WPF，虽然很不情愿说，但是貌似在某些实现理念上和MFC还是比较像的，不过相比而言优点还是要多很多。首先WPF是基于XAML语言的，XAML也是一种声明式的语言，类似HTML，好处呢，就是我用键盘打的可以比你用鼠标拖的快，不知道大家有没有觉得要把控件拖下来还要去找一个个属性点进去来设置很慢（Low）？反正我是这么觉得的。而且需要特别注意，一切拖控件的操作基本都是减分项，要是在面试前端的时候说你自己用过DreamWeaver或者ASP.NET WebForm来制作网页的话，你懂的，注意，真的是连”用过“都不要说。

另外就是可以很方便的做出美观的界面和复杂的特效，相信大家也没见过哪个Windows应用程序是用的默认样式的吧？WPF的话可以直接像PPT那样通过时间线来做动画，虽然我个人来说也是讨厌做PPT的。另外，这里用到的图表库是WPF Toolkit，比如后面会说到的一个折线图每次刷新的时候都是带动画的，先把折线画出来然后所有样本点闪一下，比MFC的美观性是要总体上高很多的。

说完了环境，现在来简要介绍一下项目的开发过程。

可能忘了说，因为之前做得太水，然后上周看其他人都弄得太认真，所以就在上周里自己重新做了这个版本。然后项目的开发流程是先做客户端，客户端做完之后再做服务器。好了，问题来了，请问我为什么要先做客户端，再做服务器？那边的同学先别急，你们真的确定这就是一个问题吗，我觉得这完全是两个递进的问题，“我为什么不同时做服务器和客户端？”以及“我为什么不先做服务器再做客户端？”，对吧？我觉得也还不完全对，对于第一个问题我觉得还可以分成“我为什么能不同时做服务器和客户端？”以及“我为什么要不同时做服务器和客户端？”，现在开始真正的提问。

* 对于第一个问题，当然实际上希望大家都没有这个问题是最好的。我花了一些时间才想到这个问题，因为我一直觉得本来就是这样的嘛。因为实际上相互调试才是最不可行的方案，如果你做的是一个QQ客户端和一个QQ服务器，就算每边只有10万行，那你也能默默先写完10万行再开始调试，等你写完两个10万行在相互调试的时候你还记得你写过哪些东西么？等出了BUG还能知道去哪里找么？  
  好吧，就算没有10万行，一个几千行的程序（客户端和服务器），运行结果的基因型有4种可能，都正确、服务器有BUG、客户端有BUG、都有BUG，但是只有2种表现型，正确，有BUG。如果是有BUG的情况下，你要如何确定是哪边有BUG？如果两边都有BUG，你修正了一边之后，结果仍然有BUG，那你又怎么知道你的修正对不对呢？  
  就算程序再小一点，几百行，你就能确保你能调试了么？如果这个程序不是线性时不变的呢？比如涉及多线程相互作用，你单步一个线程可能整个程序逻辑就不正确了，可能有的BUG只在高速状态下发生你慢慢调一点事都没有呢？比如就是回音壁你用的计时函数精度不高导致两次计时的时间差为0然后又被你作为除数。  
  问题的本质在于，跨模块调试和单步调试都是工程上最不可行的方案，或者说最后没有其他选择时才会用到的方案，练习中采用当然没有问题，但是千万不要形成本应该的思维。那要如何验证程序的正确性呢？第一步当然就是写单元测试，既然叫单元测试，就是单独测试每个单元（包括每个类，每个函数在每种情形下）的正确性，Mock调无关部分。Mock，说的难听点就是Fake，伪造。举个例子，当我们需要测试客户端的登陆相关功能，我们并不需要用到真实的帐号数据库，伪造一个服务器并设定它返回登陆成功，看客户端的反应是否正确；再设定返回登陆失败，再看。至于真的服务器是用什么语言什么框架谁写的根本不重要，只要接口明确，各个部分的独立测试都正确，相互调用的结果自然也是正确的。当然，实际上也会有集成测试，并不是说单元测试没用，只是大部分单元测试的用例不足以完全覆盖现实中的各种情况（比如上面说到的非线性时变问题）。  
  在TDD开发理念下，更是应该先写测试，再写实现。
* 在上面一个答案中已经说明了为什么要独立开发各个模块，所以这里回答的是为什么不并行开发服务器和客户机。很简单，之前说过了这是我自己临时重写的版本，人类不适合多任务并行处理，一次专注一件事会更高效，当然也就包括一次专注一个人，专业人士另算。
* 最后一个问题，也就是为什么不是先开发服务器后开发客户端，而是先开发客户端后开发服务器呢？因为有现成的Mock服务器，在“控制面板”-“程序与功能“-”添加Windows功能“-”简单TCP/IP服务“，里面有一个自带的Echo服务，也就是熟知端口7（Both TCP and UDP）的作用。  
  所以我可以先用一个伪造的服务器，确定我客户端的功能是正确的；然后在知道我客户端正确性的基础上，再用我的客户端去确保我的服务器是正确的，从而避免上面说过的表现型与基因型问题。

接下来讲一下获取高精度时间，在 .NET中是没有自带高精度时间间隔的功能，虽然 .NET 的 DateTime 类型（是结构，不是类）的值可以精确到100ns，但是实际每16ms才会变化一次。所以必须到调用到底层的API。

众所周知Windows里最高精度的时间就是QueryPerformanceCounter和QueryPerformanceFrequency的组合了，时间精度由QueryPerformanceFrequency的值确定，每个机器可能不同，我自己机器上精度约为0.4微妙，作为本项目来说是足够的了。

这两个函数位于kernel32.dll文件中，而 .NET一个十分变态的特点就是可以直接调用大部分的非 .NET代码，只需要知道函数签名即可，就像这样声明一下，之后就可以把这两个函数当成普通的 .NET函数来用了。

顺带介绍一下，虽然C#中可以像C语言一样使用指针，但是出于可（bu）维（bei）护（da）性（si）的角度，应当尽可能的不使用指针，在C#中通常使用引用方式，使用ref或out关键字声明的变量即为引用方式传递，两者的区别为，前者必须是已初始化的，后者必须是未初始化的，用途应该就不必分别说明了。另外，希望大家不要搞混了引用类型和引用传递的概念，在面向对象的语言中，大部分非基本类型都是引用类型，但传递方式仍然为值传递，即可以看成形参中的变量和实参中的变量是两个不同的地址变量，只是他们指向同一个地址单元，但是直接对形参赋值的话是不会修改到实参的，引用传递则不同，两个是同一个实际变量，所以引用传递下可以直接实现Swap函数，而指针传递仍需要额外的取值、取地址过程。像Java这种又没有指针又不能引用传递的，要实现Swap函数的话就只能靠额外的封装类了。

最后再稍微介绍一个文件格式——JSON，全称是JavaScript Object Notation，一种冗余度低并且可读性高的内容格式，以JavaScript中的对象和数组为设计来源，具有高度的语言无关性，非常适合用作不同语言、不同平台间的数据传递，目前已经基本取代XML作为Web数据交互的主流格式。本项目中的日志记录就采用的是JSON格式，任何语言都可以无需额外协议直接读取和操作。

好了，废话有点多，现在开始介绍程序的功能。

第一点也就是使用了响应式布局，不管你怎么调窗体大小，所有控件都能够正确的显示。当然，窗体最小尺寸限制为300\*300，如果你是用240\*320的功能机刷成了Windows系统的话很可惜还是无法正确显示的，不过抱歉没打算修正这个BUG就是。而且控件本身也加了一定逻辑，文本框第一次点击时会自动全选所有内容以方便输入，再次点击即可正常选择。

第二点就是本着绝对不能相信用户的原则，对每一个输入控件，不管是文本框、选择框还是按钮都有验证，任何不正确的输入及其组合都会被直接提示，以及在任何控件不应被使用的情况下该控件都会被直接禁用，不会出现说因为用户输入了非法的内容就出现程序崩溃或者产生不正确结果的情况。这点应当是任何但凡要和用户直接交互的GUI以及CUI程序都应当要保证的，特别是CUI，图形界面至少还能保证一个地方输入的东西就是一个地方的，命令行的话往往是靠空格分离参数，常见的设计缺陷比如不验证参数个数甚至多输入的内容还能够作为下次的输入等等。

第三点就是整合了服务器和客户端，上周的时候也已经有人实现过了，这里就不再细说。

第四点是不管是服务器还是客户机，都可以TCP和UDP双栈同时运行。对于服务器来说，这点应当是必须的，不应该说每次客户机要用什么协议的时候服务器再来开成什么协议。当然，也可以说这个不是业务服务器，只是网络通信概念上的服务器巴拉巴拉的，这里就不深入纠结了。

第五点是客户端支持多拨，测试窗口采用独立窗口显示，可以同时开始多个测试，对不同的服务器进行通信。甚至还可以同时对同一个服务器同一个端口同时进行多个测试（服务器和客户端都是支持的），以便验证并行数目对通信延时的影响。

第六点是客户端测试是会同时开启一个TraceRoute程序获取本地主机到目的主机之间的跳数，以便后续分析延时与跳数之间的关系，并将TraceRoute的运行过程实时显示到程序界面中。于是问题来了，为什么一定要将TraceRoute的运行过程实时显示到程序中，而不只是在后台记录一个结果？

* 本题其实只是一个常识问题，如果没有该命令的使用经验可能无法直接猜到答案。traceroute（\*nix 为 tracert）程序由于本身的运行机制原因，一次从调用到最终到达目标节点可能要超过 5 分钟甚至 10 分钟。如果放在后台运行不利于用户获取当前的运行状况，甚至误以为测试已经结束或程序出现死循环从而一直占用 CPU 。在这里要强调的一个观点是，与用户的交互永远都是客户端最重要的部分之一。

第七点是测试过程中有实时的图表信息显示，上周也有人做过了，不做详述。

第八点是可以直接输入内容也可以直接输入长度生成内容。支持直接输入内容是因为既然叫回音壁，那么应该是我说什么都是可行的（政治敏感内容除外），都应该能够被正确返回；支持指定长度是为了测试消息大小对通信延时所产生的影响，而每次直接输入上千个字符显然是不现实的。

第九点是自带名称解析功能，既可以输入IP地址，也可以直接输入主机名称，比如localhost，程序会自动将主机名解析成IP地址（如果可能的话），当然也可以直接输入域名来对特定远程主机测试。（前提是别人有这个功能的话）

第十点是可以将数据一键导出为Excel，程序中调用了Office的COM API直接生成Excel文件（当然为了教学需要操纵Excel部分使用了单独的项目，以表示这个项目是可以被复用的，所以GEW实际上是一个解决方案下面的2个项目），当然实际上也可以直接生成图表，由于COM API太过复杂（我太懒）没有时间深入学习。

\*COM API全是动态类型没有代码提示的呀查文档累死了。

接着简述一下开发过程中遇到的问题以及解决方案。

上面的代码中，BeginInvoke是在后台线程需要调用UI资源时用到的方法。这里的第一个问题是：根据这段代码，在采用了非阻塞的后台线程处理的情况下（所有代码及逻辑均正确），为什么使用 localhost 测试时主线程依然完全阻塞？中途图表不会出现和更新而是等待所有测试完成后才会在最后shua一下显示？所以暗示均已在之前的内容中给出。

* 前面说过，这里的大环境是虚拟机，也就说整个Windows只是Mac上的一个程序，另外，这个GreatEchoWall也只是Windows上的一个程序，所以能够分到的只是一个时间片的一个时间片，再加上多线程的问题，每个线程实际的CPU资源是及其有限的。  
  图表更新的CPU消耗较高，前面也说过的带动画。  
  图表更新请求太过频繁，Localhost下的通信延时可能小于1毫秒。
* 结论为，后台线程调用UI线程的操作过于频繁，且每次获取的时间片不足以完成后台线程传递过来的更新图表任务，任务不断堆积导致UI线程无法响应其他操作。最后Socket通信全部完成后，UI线程还需要一定时间才能处理完堆积的任务，处理完全部任务后开始刷新图表，故仅显示出最后一次更新数据时的图表。
* 解决方案为，放弃每次Socket通信后更新图表，而是设定了一个定时器，在所有Socket通信结束前每隔1秒刷新一次图表。

上面的代码中，BeginAccept为 .NET类库的Socket中执行异步Accept的函数，其参数为一个回调函数，在接收到连接请求（或发生错误）时调用。在上面的代码中，像这样循环调用BeginAccept方法可能（这里是一定）会导致什么非预期后果？

* 内存泄漏。  
  BeginAccept为 非阻塞 代码，其回调函数并不会被立即执行，调用后会在后台建立一个 状态机 以实现状态转移。由于非阻塞，循环会一直执行下去不会停止，而每次调用后消耗的资源不会被释放（除非建立连接的频率比循环快），最终导致内存泄漏。（本例中内存使用量达到 1.6GB，随后出现 OutOfMemory 异常）
* 解决方案为，改为非阻塞版本的Accept，在接收连接后采用异步方法处理之后的逻辑，以保证可以同时处理多个请求。  
  另外，如果硬要采用BeginAccept，应该采用非循环方式单词调用，并在回调函数中再次调用。

在上面的代码中，不考虑未给出部分，仅考虑“在循环的每一轮中调用一次 Send 和一次 Receive 并记录延时”这个逻辑本身有何漏洞？

* 一次 recv 的数据不一定是一次 send 的数据：  
  虽然由于客户机和服务器均为一次 send 对一次 recv，但客户端的一次send可能服务器需要多次recv接收，由于服务器无法判断是否属于同一次发送只能逐次回送；另外服务器的每次send也可能需要客户端的多次recv来接收，故客户端一次 recv 的内容可能只是自己一次 send 的内容的一部分，并且下次 recv 时收到的可能是上次未收完的剩余数据，导致时间记录错乱。
* 测试临界安全值为TCP（1368），UDP（1472）。另外UDP既可以采用SendTo和RecvFrom也可以采用Send和Recv，在采用前者的情况下基本可以确保是一一对应的，后者和TCP一样，使用上可视为无边界的流传输。
* 解决方案暂时想到2个，第一种是根据发送数据长度while循环，直到你收到的长度等于发送的长度再进行发送和接收。第二种就是限制用户发送大数据。本项目采用第二种，大家不要笑，这确实是一种解决方案，既然明知道在什么情况下会出问题，那就避免掉这种情况当然是可以的，总比什么都不提示直接得出错误的数据好。

\*总而言之，一次send对应一次recv只是小数据低频率下的特例，千万不要认为是本应该。

另外，由于要进行UDP的多次重复延时测试，而UDP是不保证送达的，那万一一个UDP报文没有返回，怎么处理后面的测试？

* 如果直接放弃该数据点，接着后面的测试，则万一一定时间后该数据包 返回 ，则引起计时错乱。因为你根本不知道是不是真的丢了，万一被一个奇葩路由器缠住不放了然后最后警察蜀黍来了又给别人自由了呢？所以一定不能直接不管，暂时想到了4种解决方案，第一种是使用额外的 应用层协议 ，比如增加序号等；第二种是等待 一个足够长的时间后继续进行，但一定要足够长；第三种是切换到另一个 端口 后继续进行；第四种是直接放弃 后面的测试数据点，仅考虑该样本点之前的数据。本项目采用的依然是最后一种。

于是，问题来了，为什么在本项目的全部过程中通篇采用趋于低成本的解决方案呢？

* 这题不要想多，主要是人懒、时间少、发病率高，不想弄而已。

然后呢我们继续来讨论下测试方案。

先来讨论下近场通信的情况，当然这里指的不是真的近场通信的通信，只是在本地网络中的情况。设计的测试环境有：Localhost、LAN直连、WLAN直连、WLAN到LAN、WLAN-WLAN，最后两组因为涉及到无线网络可能会有干扰，所以同时进行了2.4GHz和5GHz下的测试，至少目前在18栋还没发现第二个5GHz的路由，所以应该是比较纯粹的通信延时。根据每种环境可以绘制出时序-延时图，用来判断第一次的额外延时以及TCP的慢启动之类的情况；另外可以绘制延时-频率图，了解各个延时的分布情况。

第二部分是远端的自动探测通信，前面已经说过很多主机的端口7就已经自带了回音壁服务器，所以方案是分段 随机 生成 IP 地址，（采用线程池和队列）尝试在 TCP 端口 7 建立连接，若连接成功（表明目标主机可作为回音壁服务器），则启动对目标主机的回音壁测试，同时启动 traceroute 程序获取到目标主机的距离。以获取延时与距离的关系。问题来了，若对每一次测试都启动一个回音壁线程和一个traceroute进程，有什么不和谐的地方？

* 速率不匹配。traceroute 的耗时远高于回音壁，若以回音壁测试为基础，则 traceroute进程会持续积压；若以 traceroute测试为基础，则测试过程过于缓慢。
* 为保证效率需要建立线程池及队列管理器，应该动态调整回音壁与 traceroute后台线程/进程比。

第三部分是压力测试，当然也是我最不擅长的部分，因为本身有精神分裂症所以基本上一测就会跑崩掉。还是继续谈项目，测试限制进程网络IO带宽条件下的Socket收发延时。实现方式很简单，目前主流流氓公司的杀毒软件和电脑管家均可。

另外强调一点，靠开视频播放器这样的方式测量CPU资源对延时的影响是理论不可行的。假设我的回音壁程序只需要5%的CPU时间片，你又开了一个需要70%CPU时间片的进程，就能够保证回音壁获得的CPU资源真的就少了吗？别说定量，定性都不能保证，调低优先级也是一样的。在进程的CPU资源变没变都还不知道的情况下，又何以得出”CPU资源的多少对回音壁延时基本没有影响“的结论呢？

不论这个结论的正确与否（其实就是因为是正确的才更麻烦，这样就更不会关注实验本身是否正确而是一旦符合标准答案就认为自己的过程也是正确的了），只要实验步骤本身的逻辑是不完备的，结论就不仅是无效的，而且是有害的，和传播谣言的本质是一样的，更过分的，还假借着科学实验的名义传播谣言。

最后到了数据结果及分析部分，由于各种各样的原因，上述所有测试均没有实际进行，不过可能遇到的问题已经经过思想实验大部分确认了，希望能够给诸位同学提供参考，谢谢！