目 录

HttpClient 对象性能问题调查

作者：祁超超

前端时间测试一个系统的性能状况，其主要业务的 HTTP 请求内容在 Loadrunner 中的代码为：

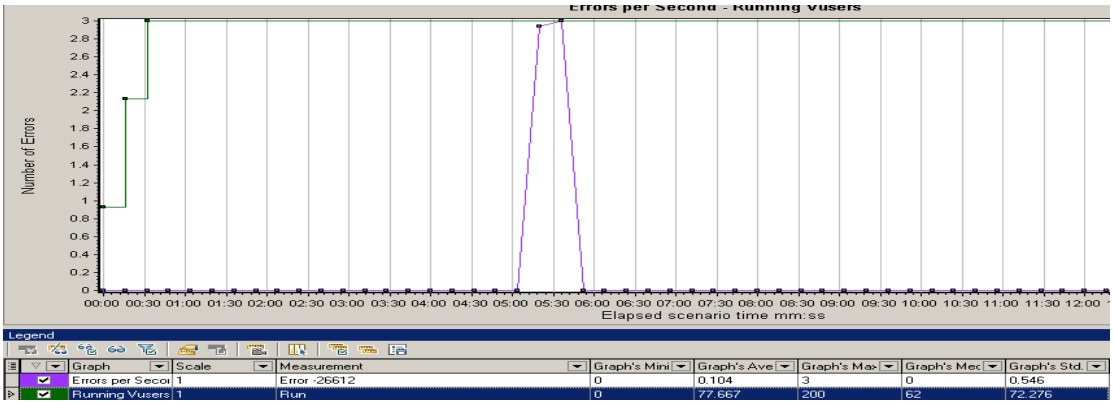
```
“URL=../imgs/icon_subscription.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/icon_resManage.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/icon_delete.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/toolbar_repeat.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/toolbar_left.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/toolbar_right.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=http：//192.168.102.43：9090/20131218/Thumbnail/docx_ec030663-c741-4067-9929-1bc5e8898d60/thumb.jpg"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/bg108.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=GetUserData?pageIndex=2&pageSize=32&info=from+loadScrnData&opkey=GetResListByUser"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
"URL=../imgs/icon_corner.png"， "Referer=http：//192.168.102.43：8003/Home/Index?token={token}"， ENDITEM，
LAST);
```

在 100 并发下持续略 5 分钟时，Loadrunner 出现以下错误提示，随后应用

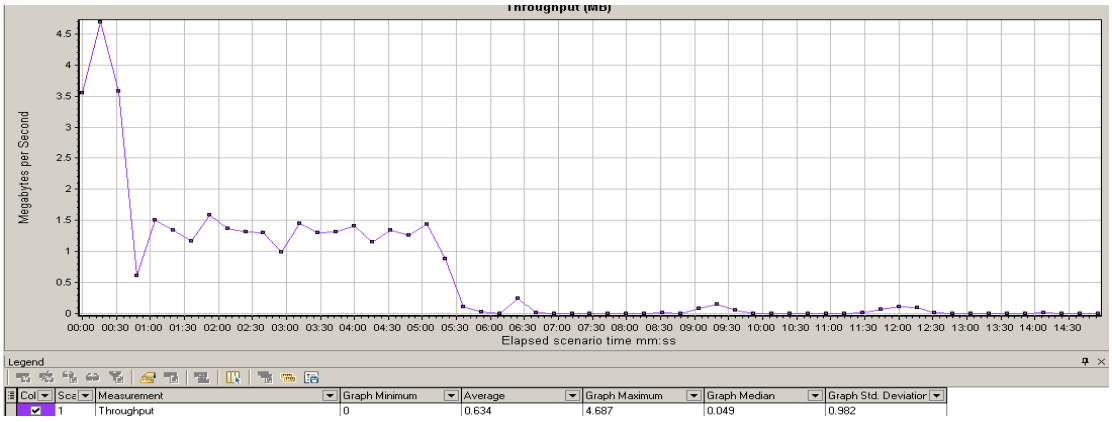
服务器崩溃：

Action2.c(11)： Error -26612： HTTP Status-Code=500 (Internal Server Error) for http：//192.168.102.43：8003/Home/Index?token=66e279bb2fc34c28ae119cdb26abab0b

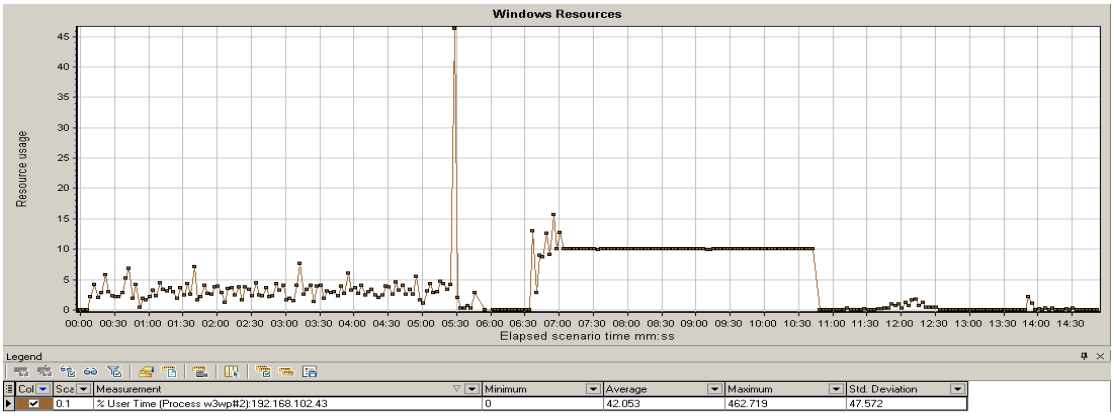
查看 Errors Per Second 可以看到在所有用户加载后，约 5 分钟出现了大量的错误。



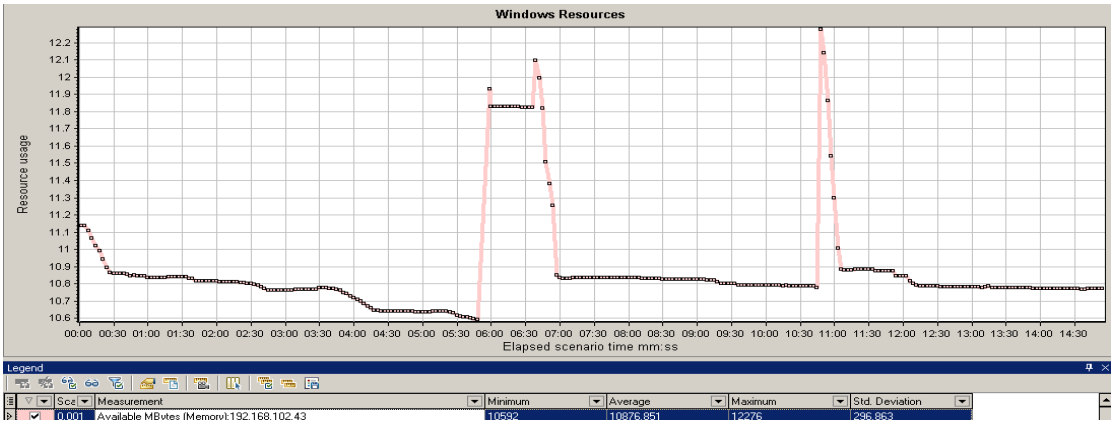
查看 Throughput 从 06 分钟后，几乎服务器不再处理请求了。



查看服务器的 CPU 使用资源情况在 05：30 的时候资源使用达到了一个峰值，随后降低了下来。



查看可用物理内存，压力开始后内存持续下降，崩溃后内存进行了回收，内存最小的时候仍有 10592MB，内存资源充足。

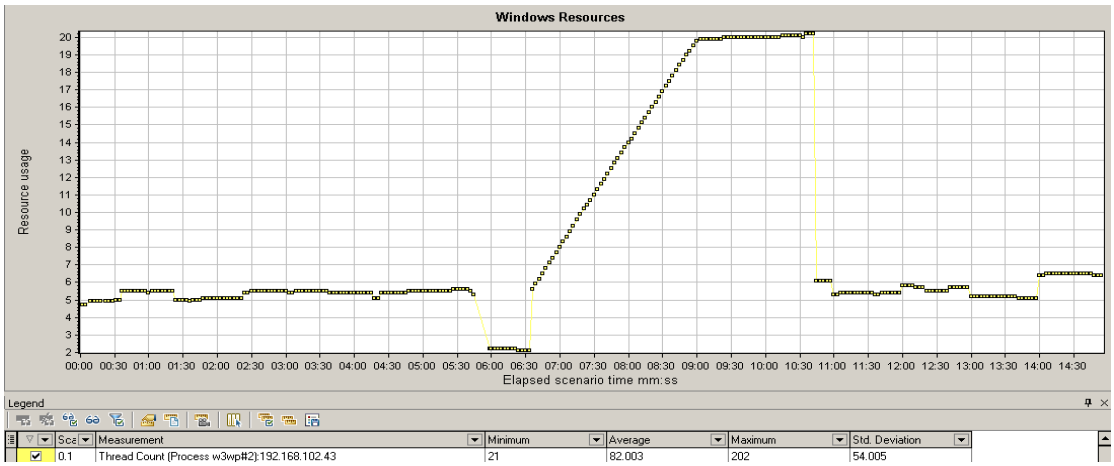


综上，怀疑是 CPU 方面的峰值时产生了连锁影响，导致出现了相关的问题。

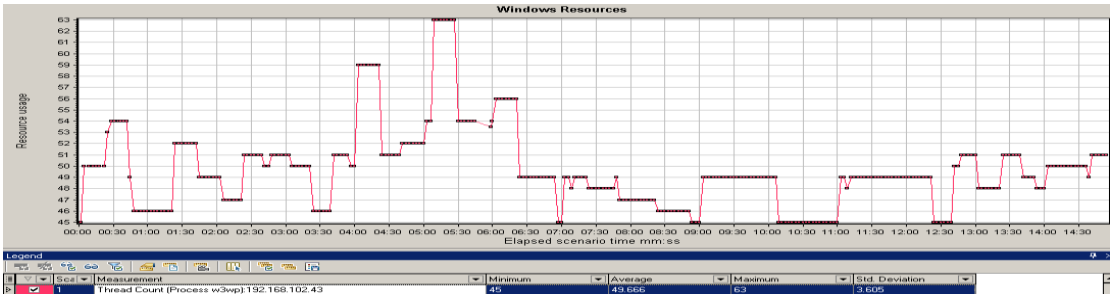
出现 CPU 问题时，首先考虑本程序是否为强算法型应用。本程序从业务实现的技术角度来看属于纯内存型应用，并且相关的数据操作都存在内存中，所以首先可以排除是 CPU 本身的问题。

其次考虑是否为内存不足引起。从上图的数据来看，内存是充足的，所以内存方面不是引起该问题的主要原因。

那么剩下的就只有可能是由于 IO 或者 Thread 方面引起的现象，从上面的介绍可知，本程序是内存型应用，所以 IO 可以排除，目前最有可能是 Thread 方面的原因引起，添加性能计数器，进行再次测试监控，发现 Thread 是如下发展：



从 05：30 至 07：30 这 2 分钟内，线程出现急降急升的状态，线程使用和回收不正常。通常来说，线程使用正常的情况下应该如下所示。



从上面的系列信息，我们目前所知的问题的初步线索为：

结论：100 并发访问业务请求 5 分钟后出现崩溃，系统的稳定性不足。

出错请求：http：//192.168.102.43：8003/Home/Index?token={token}。

出错原因：Thread 的使用和回收极有可能存在重大嫌疑。

至此，Loadrunner 能告诉我们的只有这么多，但是具体的原因倒底是什么呢？我们还能够继续么，做为测试人员我们被告诫需要尽最大的努力找到程序

出现错误的最短路径，所以让我们继续，查看原代码吧。一路追查，发现发出请求的主要是这个方法的这部分，但这个方法似乎很简单，里面只使用了一个方法 client.GetAsync(apiurl）。

```
publicvoid OnAuthorization(AuthorizationContext filterContext)
{
        //略
HttpClient client = newHttpClient();
string apiurl = ConfigHelper.MasterSiteUrl + "/api/User?token=" +
(!string.IsNullOrEmpty(token) ? token ： user.Token);
        client.GetAsync(apiurl);
}
```

查看 MSDN 中对于 HttpClient 相关方法的解释，见下：

Name	Description
GetAsync(String)	Send a GET request to the specified Uri as an asynchronous operation.
GetAsync(Uri)	Send a GET request to the specified Uri as an asynchronous operation.
GetAsync(String， HttpCompletionOption)	Send a GET request to the specified Uri with an HTTP completion option as an asynchronous operation.
GetAsync(String， CancellationToken)	Send a GET request to the specified Uri with a cancellation token as an asynchronous operation.
GetAsync(Uri， HttpCompletionOption)	Send a GET request to the specified Uri with an HTTP completion option as an asynchronous operation.
GetAsync(Uri， CancellationToken)	Send a GET request to the specified Uri with a cancellation token as an asynchronous operation.

GetAsync(String，HttpCompletionOption，CancellationToken)	Send a GET request to the specified Uri with an HTTP completion option and a cancellation token as an asynchronous operation.
GetAsync(Uri，HttpCompletionOption，CancellationToken)	Send a GET request to the specified Uri with an HTTP completion option and a cancellation token as an asynchronous operation.

Asynchronous 中文含义，异步。即 GetAsync(uri)方法是一个发送异步请求的方法，显示异步将会使用到 Thread 来进行，难道是这个方法本身出了问题么？

先不要着急，我们继续看 MSDN 对于 HttpClient 的介绍，我们可以发现这么一段示例代码：

```
staticasyncvoid Main()
{
// Create a New HttpClient object.
        HttpClient client = new HttpClient();
// Call asynchronous network methods in a try/catch block to handle exceptions
try
        {
            HttpResponseMessage response = await client.GetAsync("http：
//www.contoso.com/");
response.EnsureSuccessStatusCode();
string responseBody = await response.Content.ReadAsStringAsync();
// Above three lines can be replaced with new helper method below
// string responseBody = await client.GetStringAsync(uri);
            Console.WriteLine(responseBody);
        }
catch(HttpRequestException e)
        {
            Console.WriteLine("\nException Caught!");
            Console.WriteLine("Message ：{0} "，e.Message);
        }
// Need to call dispose on the HttpClient object
// when done using it，  so the app doesn't leak resources
        client.Dispose(true);
    }
```

哈哈，似乎问题找到了所在，我们的程序没有调用 Dispose 方法释放 HttpClient 对象。

// Need to call dispose on the HttpClient object　翻译：需要调用 disponse 方法释放 HttpClient 对象

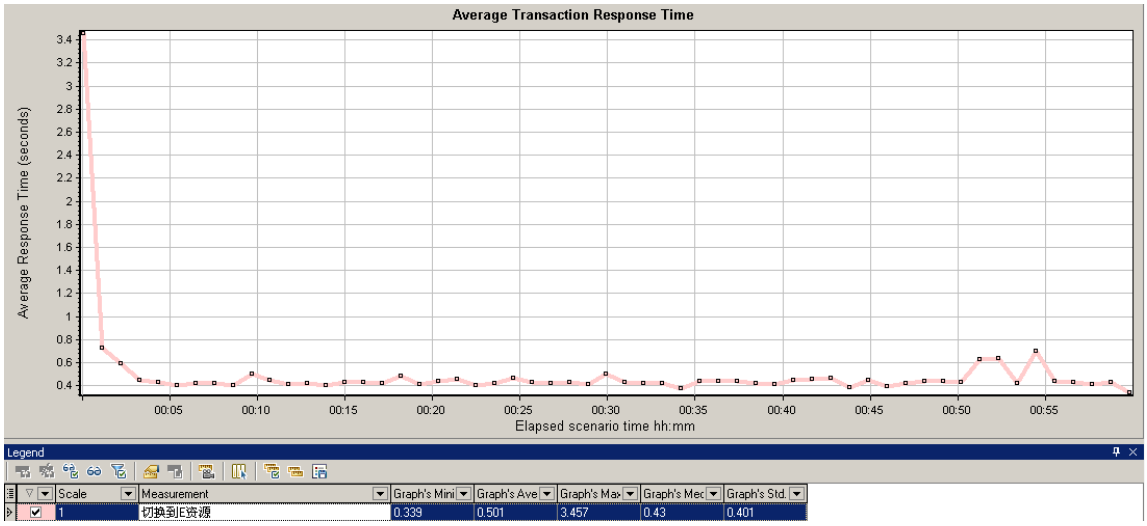
// when done using it，　so the app doesn't leak resources　翻译：不使用它，有一些应用不会释放资源

问题是.Net 不是存在拉圾回收机制么？client 对象按道理应该进行自动回收，不应该出现这个问题。

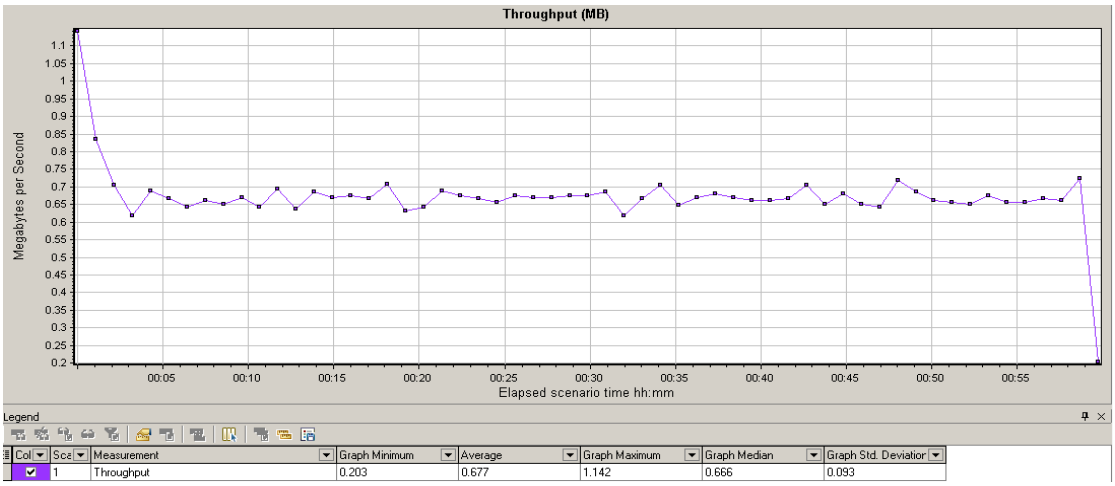
抱着这样的疑问，让开发修改代码为，并执行相同的测试。

```
publicvoid OnAuthorization(AuthorizationContext filterContext)
{
        //略
HttpClient client = newHttpClient();
string apiurl = ConfigHelper.MasterSiteUrl + "/api/User?token=" +
(!string.IsNullOrEmpty(token) ? token ：  user.Token);
        client.GetAsync(apiurl);
    client.Dispose();
        client = null;
}
```

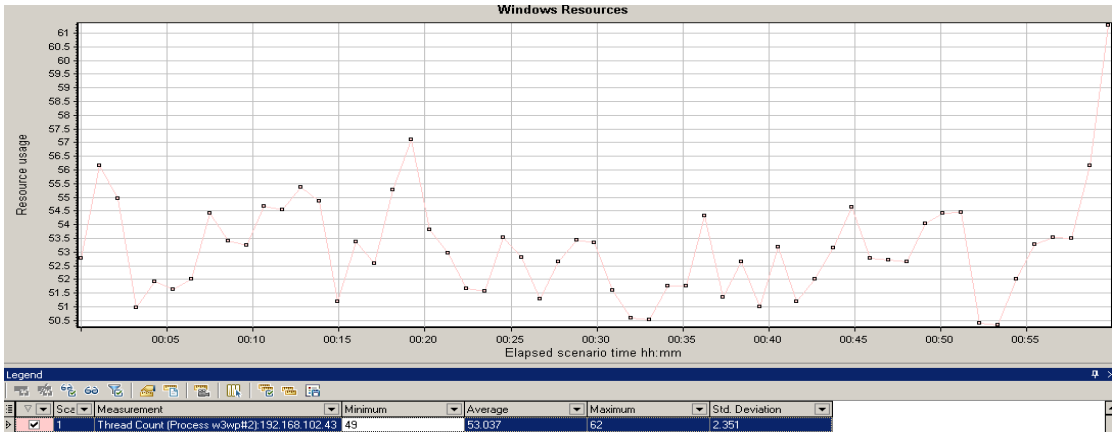
系统业务响应时间缓存加载后稳定，持续运行了 1 个小时。



流量也稳定，没有出现原有急速下降的现象。



查看 Thread 使用情况，呈波动发展，有使用也有回收，表现良好。



至此可以证明，性能问题已被解决。

那么回到为什么垃圾回收机制没有起作用呢？网上有较多文章进行了解释，其中下面的内容是主要原因：

首先，系统将托管堆内所有的对象视为可以回收的垃圾，然后系统从 GCRoot 开始遍历托管堆内所有的对象，将遍历到的对象标记为可达对象，在遍历完成之后，回收所有的非可达对象，完成一遍垃圾收集。

注意，托管堆的垃圾收集只会自己收集托管对象。

由于在执行完垃圾收集之后，托管堆中会产生很多的内存碎片，导致内存不再连续，因此在垃圾收集完成之后，系统会执行一次内存压缩，将不连续的内存重新排列整齐，变成连续的内存。（关于垃圾收集的详细信息，大家可以参考《CLR Via C#》

通过上面的简述，大家都知道什么样的对象不会被收集，即能从 GCRoot 开

始遍历到的对象。

最常见的 GCRoot 是线程的栈，线程的栈里面通常包含方法的参数、临时变量等。另外常见的 GCRoot 还有静态字段、CPU 寄存器以及 LOH 堆上 的大的集合。因此，如果想要让托管对象的内存顺利的释放，只需要断开与跟之间的联系即可。而对于非托管对象的内存，必须进行手动释放。

非托管对象无论在什么时候，都不会被垃圾收集所回收，必须手动释放。

在. net 内存泄露的原因当中，事件占据了非常大的一部分比例，事件是一种委托的实例，也就是与我们类中其他的字段一样，也是一个字段。

> By default， HttpWebRequest will be used to send requests to the server. This behavior can be modified by specifying a different channel in one of the constructor overloads taking a HttpMessageHandler instance as parameter. If features like authentication or caching are required， WebRequestHandler can be used to configure settings and the instance can be passed to the constructor. The returned handler can be passed to one of the constructor overloads taking a HttpMessageHandler parameter.

我们查看 msdn 可以了解到 HTTPClient 有这么一段解释：

至此也就解释了为什么大并发下会出现这个问题。

基于 VC 测试工具的二次开发

作者：冀方

实际的项目中，可以使用商业的测试工具，也可以使用开源的测试工具。如果这两种工具都满足不了测试的要求，只能自己从零开发或者在现有测试工具的基础上做二次开发。需要做二次开发的情形并不少见，比如，笔者最近亲历的某银行 OTP 设备项目。

先了解下 OTP 是什么？OTP 全称叫 One-timePassword，也称动态口令，是根据专门的算法每隔 60 秒生成的一个与时间相关的、不可预测的随机数字组合。动态口令可有效保护交易和登录的认证安全，而且无需定期更换密码。OTP 设备就是用来生成动态口令的终端。

首次介入 OTP 设备项目是在去年的 11 月初。正忙于手头测试项目的我，应邀参加了部门内部组织的 OTP 产品会议，重点讨论硬件测试、固件测试、软件测试等问题。说是讨论，实际上是分配测试工作。项目组领导安排我开发测试工具，评估 OTP 固件的可靠性。工具要求实现时间比对，动态口令验证，解锁验证，时间同步验证以及循环写种子密钥和时间同步的功能。我掐着手指头算了下，一共 5 个，还是倍感压力。压力不仅是要做的东西多，而且事先我对这个项目毫无了解。领导有板有眼的解释着我要做的东西，我还是稀里糊涂他说的一些名词；虽然肾上腺素在不停分泌，潦草的记下了他说的要点，但是对于内部实现的细节，还是一头雾水；虽然自己开发的底子还算扎实，但是对于短时间内就要出东西的项目，心里还是没底。

会议结束后，心急火燎的我，马上和测试组其他同事讨论测试工具的问题。我首先想到了 CardIDE。CardIDE 是公司自主开发的一个集成开发环境，可以开发基于 Delphi 语言的脚本，用于金融领域 KEY 类产品和读卡器类产品的测试。由于分工不同，我对这个工具并不熟悉。我的疑惑是通过 CardIDE 能否开发出满足 OTP 测试要求的脚本？这个想法很快被经常使用这个工具的同事否定了，理由是 Delphi 语言功能比较弱，开发难度大，几乎不能实现。同事的答复像一盆冷水泼在了身上，希望瞬间变失望。其中的一位同事还安慰我说，你要是做不了，赶紧报告领导，还能挽回点局面。我一时语塞，脸上微烫，突然发现自己，怎么有些不淡定了。一句无心的话，坚定了我做出来的决心。

摆在面前的第一个问题是如何用 VC 实现 PC 机和 OTP 设备的通讯？OTP 设备采用 USBHID 协议与上位机通讯。实现过程复杂，从头做的话，费时费力，有没有现成的工具呢？带着这个问题，我求助开发的同事。还真有，同事爽快地给了我一个之前用于生产的工具源码，工具基于 VC 实现了设备连接状态的自动检测和与 PC 机通讯的功能。同事的帮助，真如雪中送炭，顿时在我心中洒满了阳光。果真，上帝关上了一扇门，还会留下一扇窗。

俗话说，磨刀不误砍柴功。当前的首要任务还不是急于在现有工具的基础上做二次开发，着重点应放在需求文档、设计文档和对测试工具实现的梳理上。更何况我是项目中后期介入的，算是半路出家的和尚，只有做足了准备工作，才好开始下一步。

OTP 测试工具要实现的功能有 5 个，实际上也分别做了 5 个工具，来满足测试的要求。测试工具实现的方法大同小异，重点是弄清楚工具实现的业务流程。本文以循环写种子密钥和时间同步工具为例，详述基于 VC 测试工具二次开发的过程。

首先还是要明确目标测试工具的功能需求，包含基本功能需求和附加功能需求。基本功能需求：

- 自动检测设备的插拔（现有工具已实现）
- 随机产生 16 字节种子密钥 KeyData
- 获取 8 字节随机链路密钥 EKEy
- 用 EKEY 对根密钥 MKEY 做 ECB 模式下的 3DES 加密运算，运算结 KeyEndataTest
- 用 KeyEndataTest 对种子密钥 KeyData 做 CBC 模式下的 3DES 加密运算，运算结果 KeyEndata
- 用种子密钥 KeyData 对 8 字节 00 做 ECB 模式下的 3DES 加密运算，运算结果 KeyChkval
- 将序列号和 KeyEndata 和 KeyChkval 拼接后，写入到 OTP 设备中
- 发送时间同步指令，验证 OTP 和本地时间是否同步
- 擦除 COS
- 模拟 OTP 插拔的动作
- 程序暂停执行，等待生产工具将 COS 自动写入
- 模拟 OTP 插拔的动作
- 返回步骤 1 重新执行

附加功能需求：

- 添加配置文件，要求压力测试的次数可配置
- 实时显示测试完成次数
- 将中间结果数据写入日志，便于查看测试结果和辅助问题分析
- 功能点多，实现步骤繁琐，看起来有点“晕”。为了有助于我们理清思路，一个好的习惯是画出流程图。主流程图如图 1：

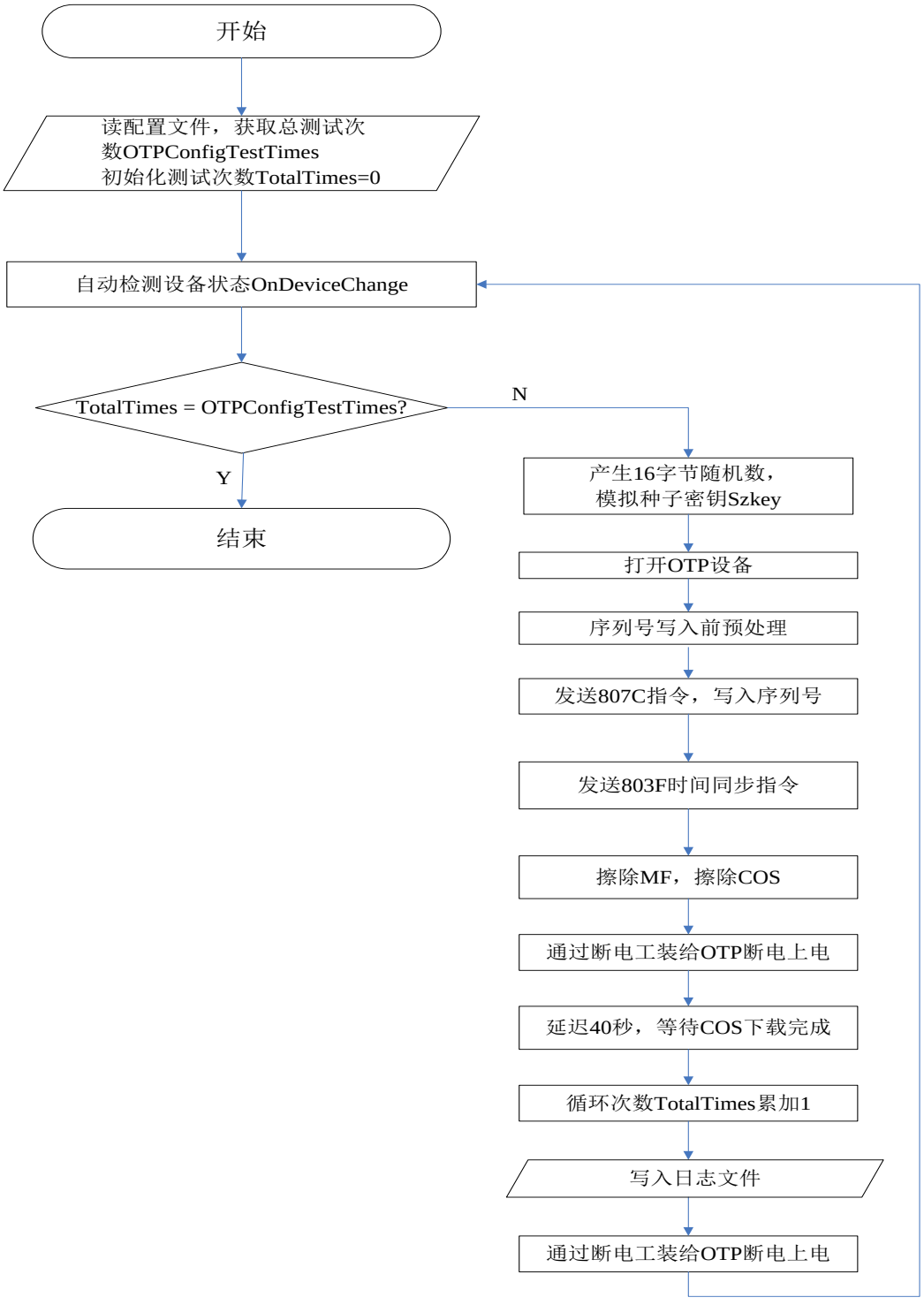


图 1

图 1 中有一步处理是：序列号写入前预处理。此步预处理流程可参考

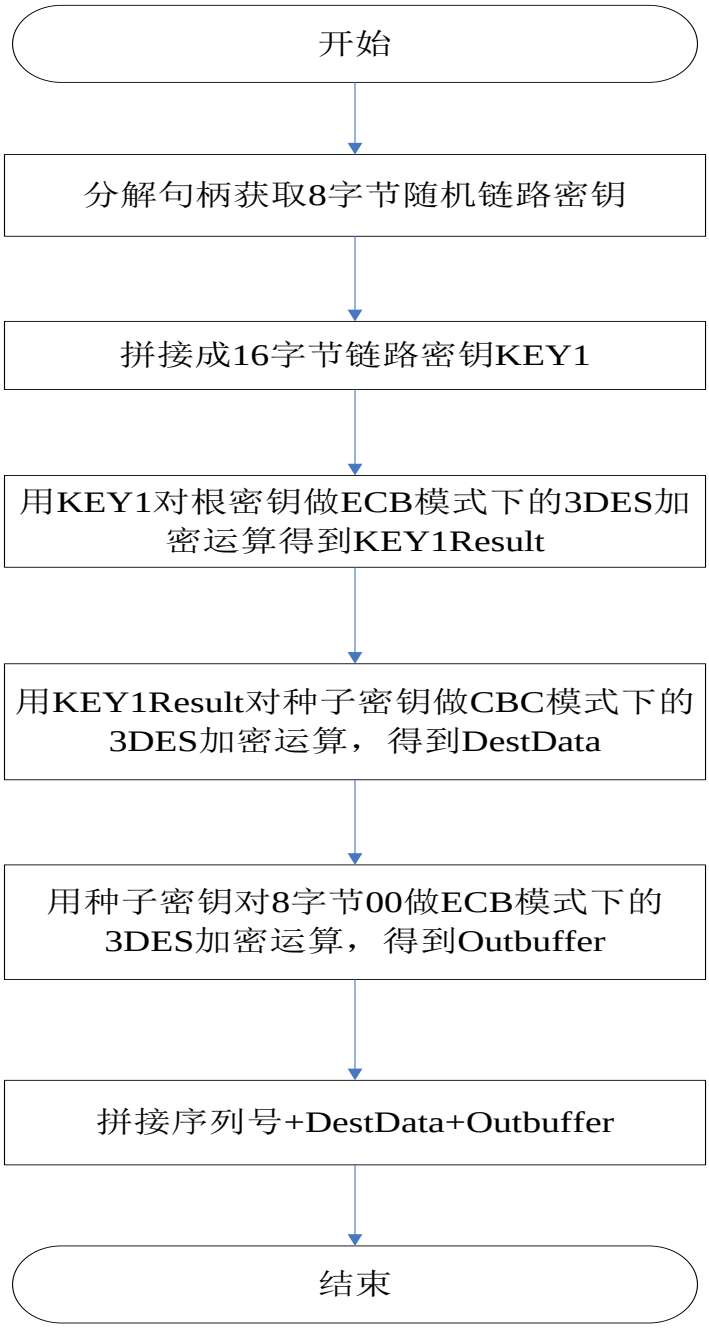


图 2

流程图展示了测试工具工作的过程。作为流程图的补充，也为了更清楚地叙述工具实现的过程，图 3 展示了设备连接框图。

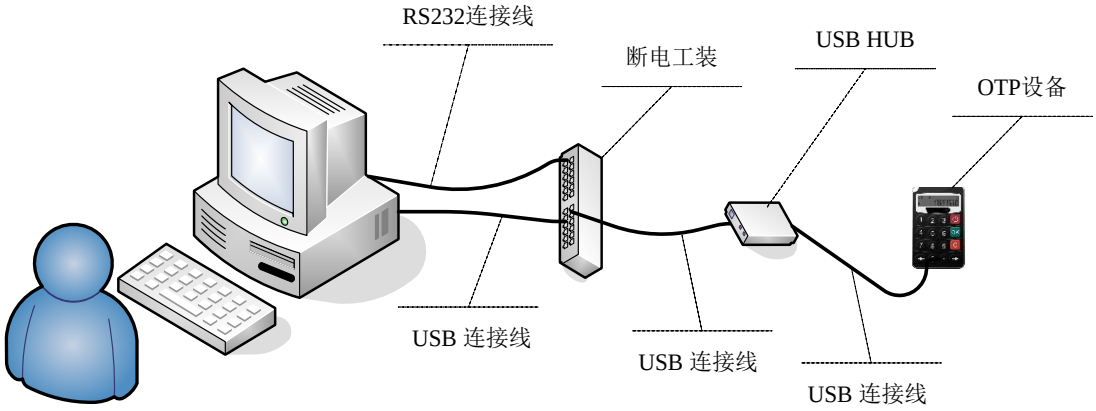


图 3

仔细观察流程图，工具开发的难点集中在三个方面：如何获取 8 字节随机链路密钥；如何实现 3DES 在 ECB 和 CBC 模式下的加密运算；如何控制断电工装给 OTP 断电上电。

先说第一个问题。随机链路密钥是由 OTP 内部产生的，但是 COS（OTP 内部操作系统）并不支持相关命令来获取随机链路密钥，这也是之前同事拒绝使用 CardIDE 的原因之一。经过调试，分析打开设备后返回的句柄值，我才明白，句柄是一个结构体类型的数据，而 8 字节的随机链路密钥则隐藏在此结构体的成员变量中。问题转化成如何获取句柄中成员变量的值，VC 中的具体操作如下：

```
typedefstructtag_screader
{
BYTEbReaderType;
HANDLEhDev;
DWORDdwShareMode;
BYTEnad;
BYTECardType;
DWORDdelay_time;
HANDLEg_hMutex;
charAdapterId[8];
charTargetID[8];
charnumAdapter;
intIndex;
BOOLbLineEncryption;
UCHAREncryptRandom[32]; //缓存 8 字节随机链路密钥
HANDLEhCDROM;
}SCRDRHANDLE，  *PSCRDRHANDLE;
```

在相关的头文件 WdKeyRW.h 中定义结构体，并用 typedef 把结构体类型重

命名为*PSCRDRHANDLE

在对应的文件 WdKeyRW. cpp 中，包含上述头文件，分解句柄，获取 8 字节随机链路密钥，并保存在字符数组 EKEyresp 中，编程如下：

```
#include "WdKeyRW.h"                    //包含头文件
unsignedcharEKEyresp[256] = {0};        //用于缓存 8 字节随机链路密钥
PSCRDRHANDLEpRdr = NULL;                //定义指向结构体类型的指针 pRdr
pRdr = (PSCRDRHANDLE)m_hDev; // m_hDev 是句柄，已在 WdKeyRW.h 定义过
for(i=0;i<8;i++)
{
EKEyresp[i] = pRdr->EncryptRandom[i];
}
```

第二个问题，3DES 加密运算的实现。短时间内，独立实现这个加密算法，显然不可能，也没有必要。通过上网搜索，实际上可以发现很多 C 语言实现的源码。问题是 C 语言实现的源码不能直接被 C++调用，因此，需要在 VC 中建立 C/C++混合编程的环境。建立方式如下：

首先，在 C 语言实现的 des. h 头文件中，将加密函数声明包含在下述语句内：

```
#ifdef __cplusplus
extern "C"
{
#endif
……
此处放函数声明
……..
#ifdef __cplusplus
extern "C"
{
#endif
```

实际的编程如下：

```
#ifdef __cplusplus
extern "C"
{
#endif
voiddes3Encrypt(DES_KEY *des3Key，
BYTE * inbuffer，
BYTE * outbuffer);

voiddes3Decrypt(DES_KEY * des3Key，
BYTE * inbuffer，
BYTE * outbuffer);
Intdes3InitKey(unsignedchar *userKey，
                IntuserKeyLength，
                DES3_KEY *des3Key);

#ifdef __cplusplus
}
#endif
#include "include/des.h"
BYTERandData1[16] = {0};      //缓存 16 字节链路密钥
BYTEoutbuffer1[255] = {0};    //缓存加密后的结果
UnsignedcharMkey[16];        //缓存根密钥
intlens = 16，j;

j = des3InitKey(RandData1，lens，&des3KeyNew);    //初始化密钥
des3Encrypt(&des3KeyNew，Mkey，outbuffer1);    //用链路密钥对根密钥做 3DES
加密运算
```

其次，需要将 des.h 包含在工具主文件 WdKeyRW.cpp 中，编程如下：

第三个问题，控制断电工装给 OTP 上下电。断电工装由公司自主开发，通过 PC 机的 RS-232 接口，发送控制指令。可以在 VC 中添加相应的代码，来实现 OTP 上下电的控制，此部分编程不具有通用性，不再赘述。

啃完了“硬骨头”，工具开发的难度自然也会降低，实现起来也相对容易。回到主流程图，工具初始化时，需要读取配置文件，实际的配置文件样式

如图 4：

由于工具初始化的操作是在 WdKeyRWICDlg. cpp 文件中完成，读取配置文件的操作也是在此文件中完成，而主文件 WdKeyRW. cpp 中要用到配置文件中的 OTPConfigTestTimes 变量，因此，此变量需要定义成全局变量，以便 CwdKeyRWICDlg 类和 CwdKeyRW 类共同使用。编程如下：

```
在主文件 WdKeyRW. cpp 中定义配置文件中总的测试次数：
#include ″WdKeyRW. h″
unsignedlongOTPConfigTestTimes；//定义全局变量
在头文件 WdKeyRW. h 中声明此变量
externunsignedlongOTPConfigTestTimes；//声明全局变量
并在 WdKeyRWICDlg. h 头文件中包含主文件 WdKeyRW. cpp 的头文件
#include ″WdKeyRW. h″
在 WdKeyRWICDlg. cpp 中包含相应头文件
#include ″WdKeyRWICDlg. h″
    OTPConfigTestTimes= GetPrivateProfileInt(″TestTimes″，″TotalTestTimes″，1，″C：
\\OTPConfigure. ini″)；//读取配置文件中总的循环次数
```

有一步操作是产生 16 字节随机数，模拟种子密钥。在主文件 WdKeyRW. cpp 中的具体实现如下：

```
unsignedcharszkey[16];    //缓存随机数，模拟种子密钥
srand((unsigned)time(NULL)); //利用系统时间改变系统的种子值
for(i=0;i<16;i++)
szkey[i] = rand() % (255-0+1) + 0;    //产生 0~255 之间的随机数，并赋值给 szkey 数组
```

每一轮测试完成后，需要写日志文件。这一步操作在主文件 WdKeyRW. cpp 中完成。编程如下：

```
#include<fstream. h>
charRnd16Data[256]={0};
charEKEyresplog[256]={0};
charDestData1log[256]={0};
of1. open("c: \\log. txt", ios:: app, filebuf:: openprot);//追加方式打开日志文件
of1<<Rnd16Data<<' \t' <<' \t' <<EKEyresplog<<' \t' <<' \t' <<DestData1log<<' \t' <<' \t' <<' \n';  //写日志文件
of1. close();//关闭日志文件
```

实际的日志文件样式如图 5：

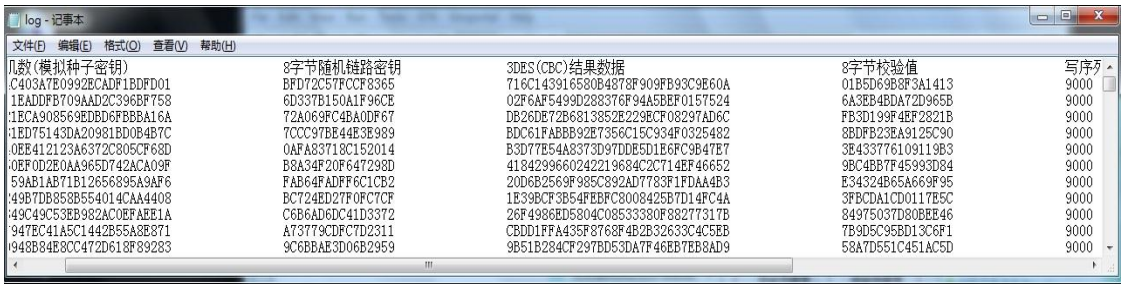


图 5

读者朋友可能注意到，每执行完一轮，就要打开关闭日志文件一次，是否过于频繁？为什么不，仅打开一次，等测试完成后，再关闭日志文件？两种方案都可行。如果日志文件始终处于打开的状态，在碰到突发事件，比如说停电或者测试中误关闭程序，测试日志就会丢失；实际中采用的方式损失了一些执行效率，但是可以确保任何情况下都能记录之前的情况。假如测试持续 1 周以上的时间，测试日志的及时保存就显得尤为重要了。

程序写到这儿，工具的框架基本上算是搭起来了。但是，在调试中，还是发现了问题。

预处理流程图中有一步是：用 KEY1Result 对种子密钥做 CBC 模式下的 3DES 加密运算。实际开发的时候，却误认为 3DES 运算是 ECB 模式，计算结果死活与开发人员提供的示例数据不一致。翻了翻设计文档，也没有提到这点。一开始以为是自己的程序计算错了，反复演算怎么也发现不了问题，耽搁了不少时间，和开发同事沟通后，才知道是 CBC 模式，立马傻眼。说明，不能想当然，不确定的因素，还是要积极和开发沟通。

调试中的另一个问题是没有办法使用类向导，也就不能修改程序界面。始终报 Parsingerror 错误，如下：

Parsingerror： Unrecognizedmacro.

InputLine： ″ON_WM_DEVICECHANGE()″

这实际上是之前工具遗留的问题。解决方法是将原来的 ON_WM_DEVICECHANGE()改为 ON_MESSAGE(WM_DEVICECHANGE，OnDeviceChange)即可。在此问题解决的基础上，优化了人机界面，如图 6：

图 6

程序写到这儿总算是可以用了，但仅是在 Debug 模式下可用，Release 模式下编译成功，但是链接失败。报错如下：

MSVCRT. lib(MSVCRT. dll)：

errorLNK2005： _strrchralreadydefinedinlibcmtd. lib(strrchr. obj)

造成这个问题的原因是主程序和 lib 所使用的运行库不一致。解决方法是 project->setting->Generation->UseMFCStaticLibrary，即将共享库改成静态库。再次在 Release 模式下编译，可以成功生成可执行的 exe 文件。

终于“大功告成”。怀着愉悦的心情在配置文件中设定了运行次数 10000 次，

第二天早晨赶到公司的时候，发现设备状态是，设备连接已断开！实际上也只运行了 600 多次，程序就停了。经过推算，测试要持续几天的时间，不可能那么早就结束。一种不祥的预感袭上心头。查看日志，至少这 600 多次的结果都是正确的。起先以为，是测试工具的问题，需要针对异常做处理。幸亏当时“现场”保留的比较好。在我插拔 OTP 设备的时候，发现一向灵敏的生产检测工具，对 OTP 没有一点反应。在测试机的设备管理器中也找不到 OTP 设备，这说明测试机操作系统已不识别 OTP。重起测试机后，系统重新识别 OTP。后来，就此问题咨询了开发人员。得到的建议是串入 USBHUB。这也就是图 3 设备连接框图中为什么要串入 USBHUB 的原因。后来经过实测，测试工具可以稳定运行。

测试工具的二次开发涉及到逆向思维和正向思维。逆向思维帮助理解原有工具实现的思路，正向思维是在原有工具的基础上开发新的功能，满足测试的需要。

有句话说，“站在巨人的肩膀上，可以看的更远”。在现有工具的基础上先继承一些东西，消化吸收后，再去开发一些自己的需要的东西。这样做的效率远比从无到有高的多。

本文对于入行不久的朋友或是需要做些开发工作辅助提高自己测试效率的朋友，可能会有一些帮助和启发。文中疏漏之处，也请同行包涵指正，愿与朋友们在测试工具的多彩世界中切磋探讨。

CICS 程序简单调优

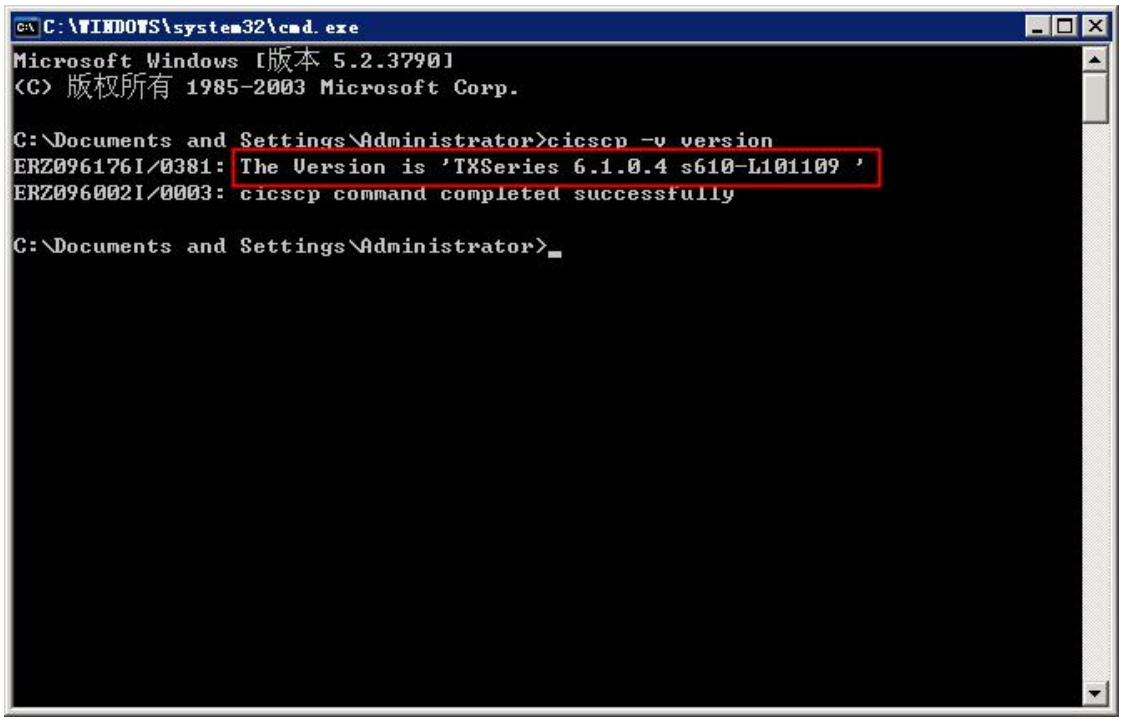
作者：杨明华

一、 TXSeries for Windows

TXSeries 是 IBM 的交易服务器、整合平台，它支持在多种操作系统上搭建分布式的 OLTP 服务，也就是我们通常所说的 CICS Server。目前 IBM TXSeries 官网上提供下载的最新版本为 7. 1，试用 90 天，由于实验环境有限，本文仅以 TXSeries 6. 1 PTF4 for Windows 为例，编写一个简单的 CICS 程序并对其中一些对性能影响显著的参数进行调优实验。

1. 安装 TXSeries 6.1 for Windows

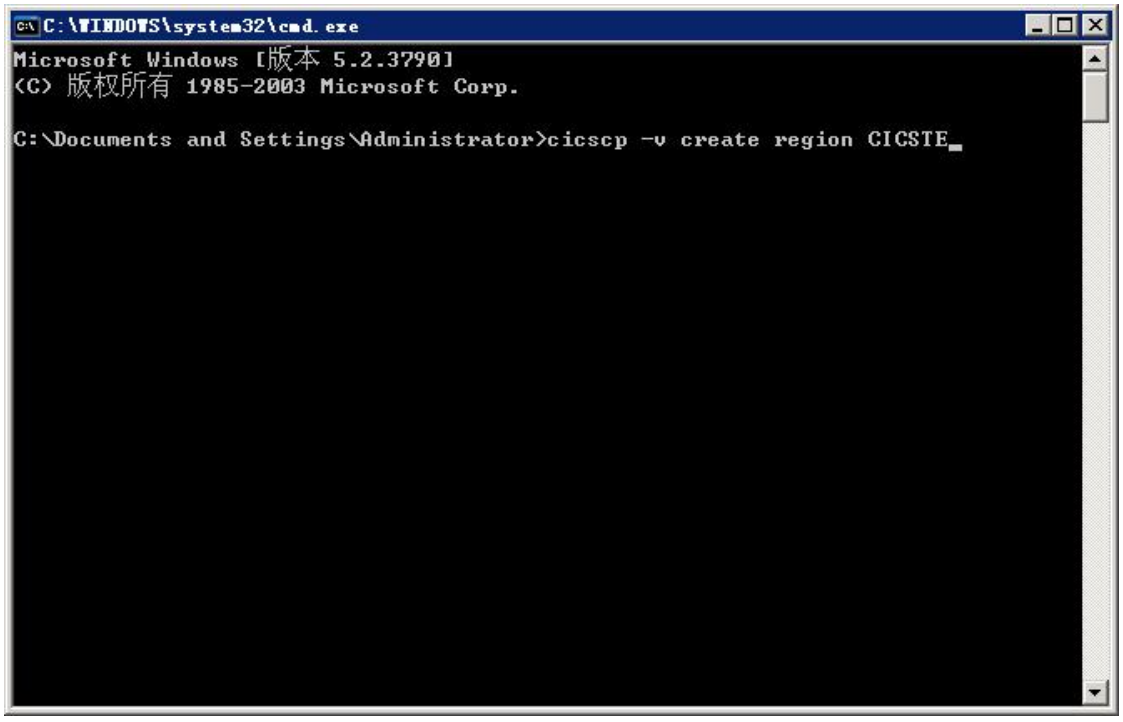
下载并在 Windows 2003 系统上安装 TXSeries 6. 1 for Windows，安装 PTF4 补丁，从命令行下确认版本信息：

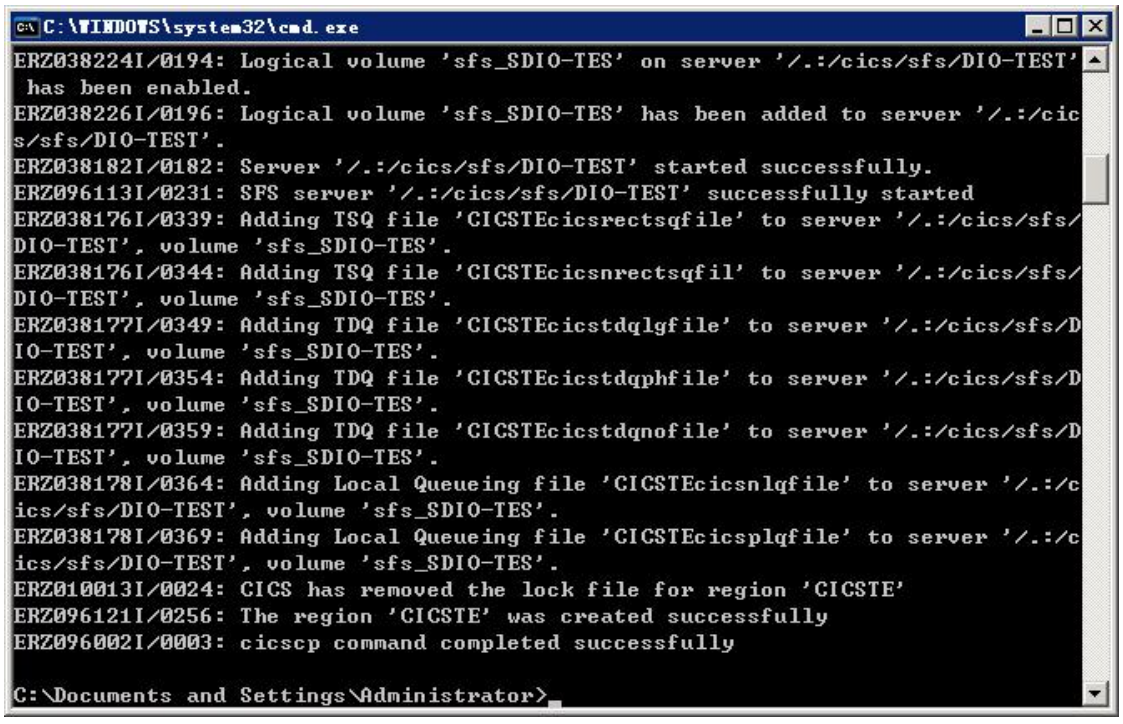


以下 Region 创建和参数修改等操作尽量基于命令行，以便和 UNIX 环境对应。

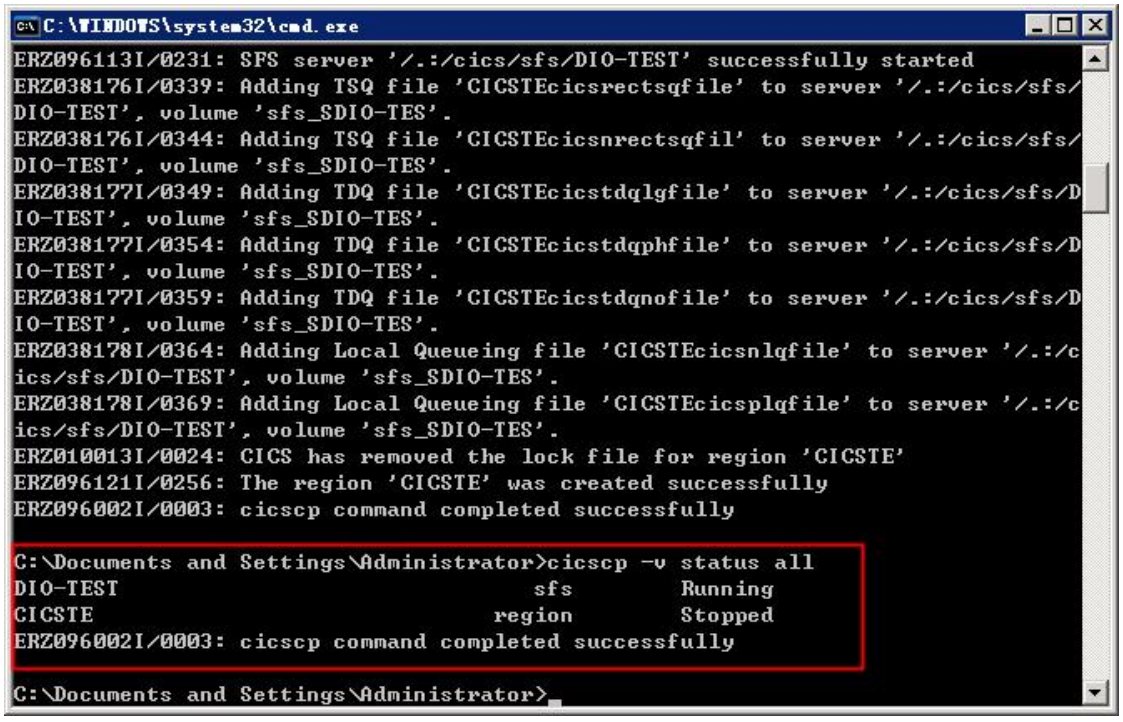
二、 创建 CICS Region

创建测试 Region “CICSTE”为例：cicscp –v create region CICSTE，默认会同时创建 SFS（IBM 的一种文件系统）：

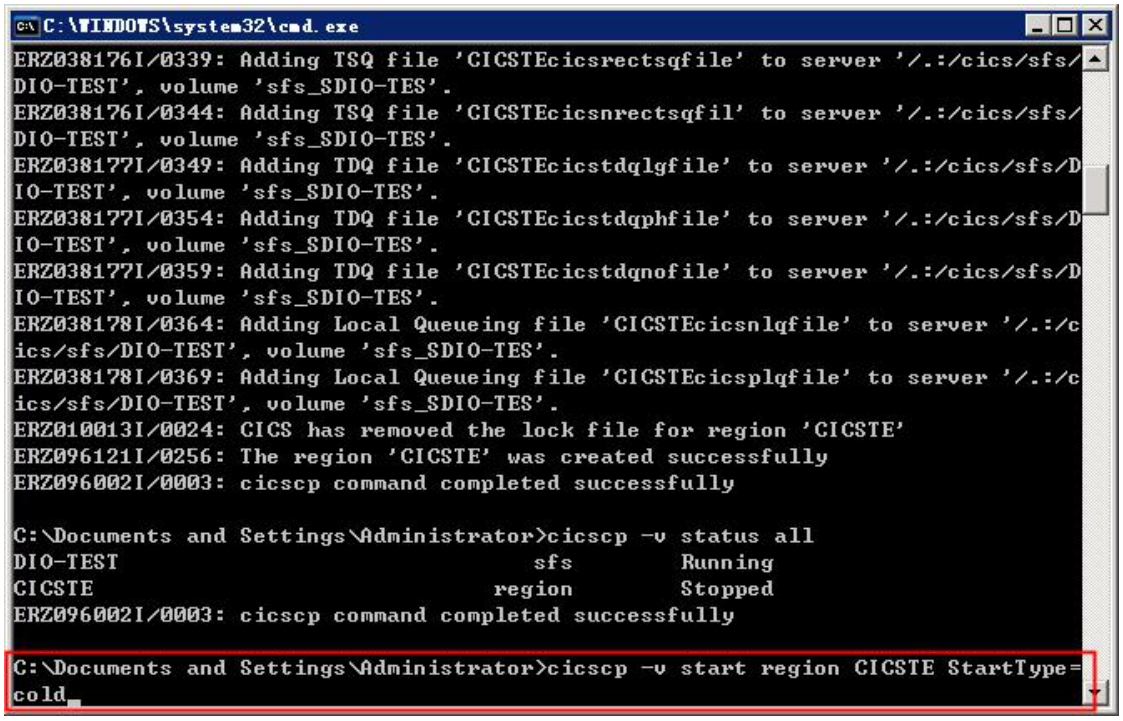


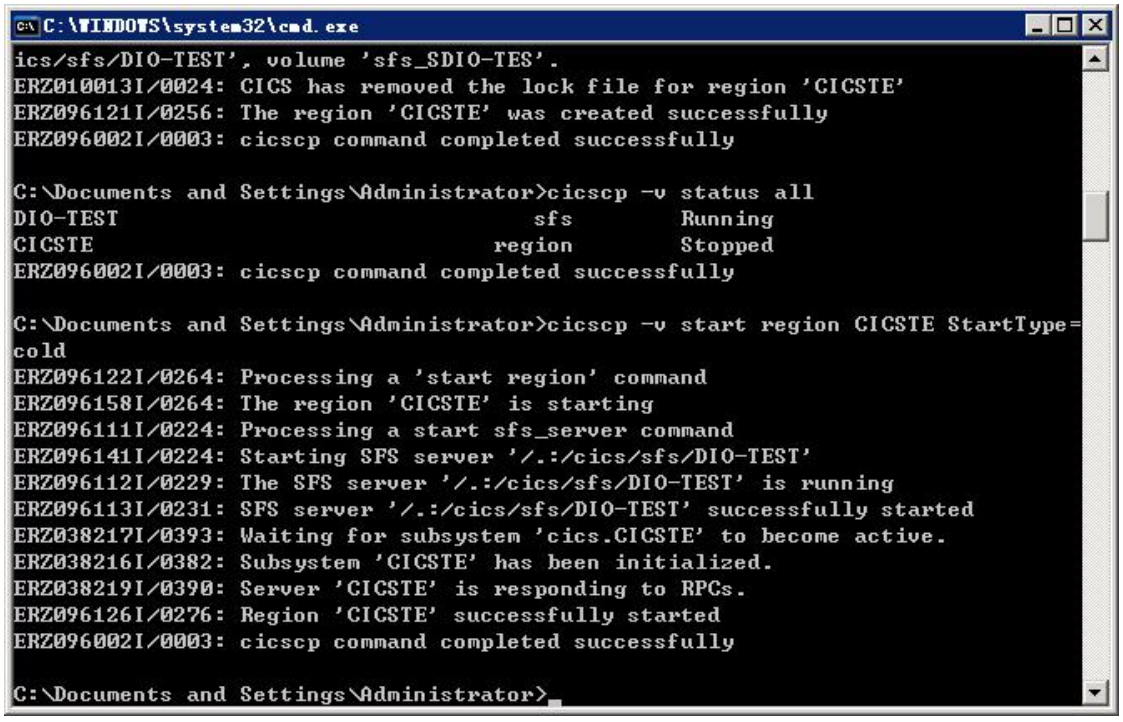


cicscp －v status all，检查状态，默认 SFS 已经运行，但 CICS Region 是 Stop 状态：

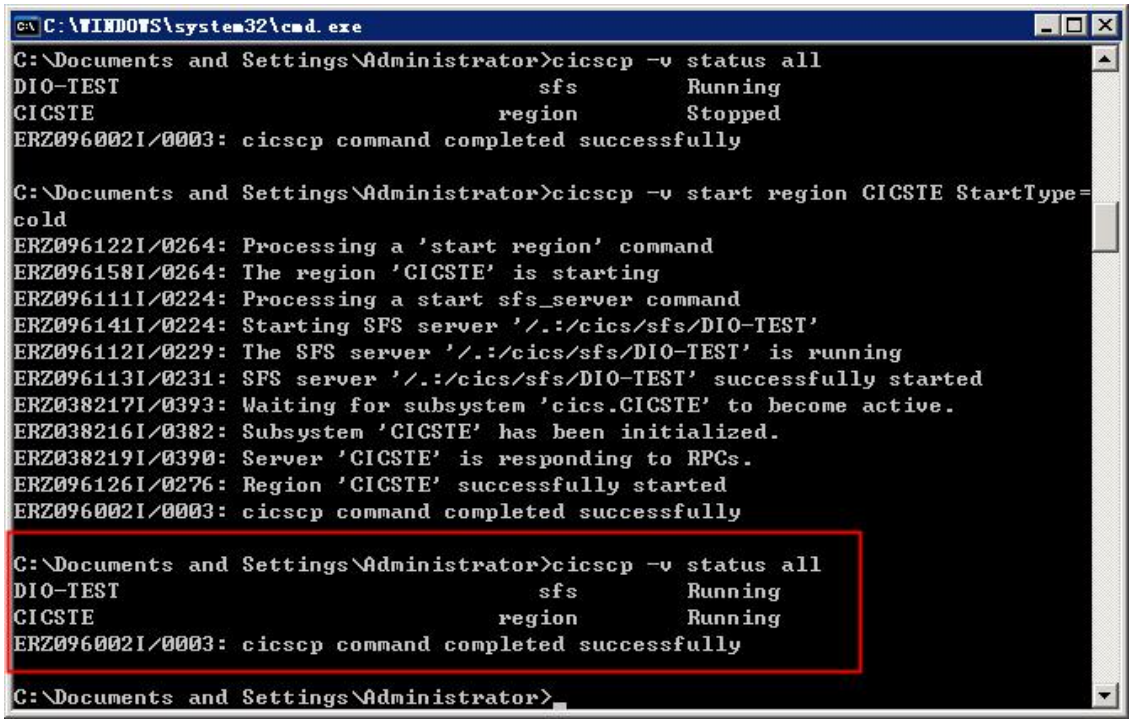


cicscp －v start region CICSTE StartType=cold，使用冷启的方式启动 Region，后面的 Region 配置修改后都使用冷启方式重启 Region：





再次检查状态，SFS 和 Region 都已经启动了：



三、 编写 CICS Server 程序

CICS 源程序包含 CICS 函数，因此必须经过预编译，再使用开发工具进行编译。本文以 C 语言为例，开发的 CICS 程序分为独立的两步：1、CICS 预编译；2、VS2008 编译。

在记事本中编写一个简单的 CICS 程序，程序中没有复杂的逻辑判断，收到固定字长的请求报文后返回根据指定位显示返回：

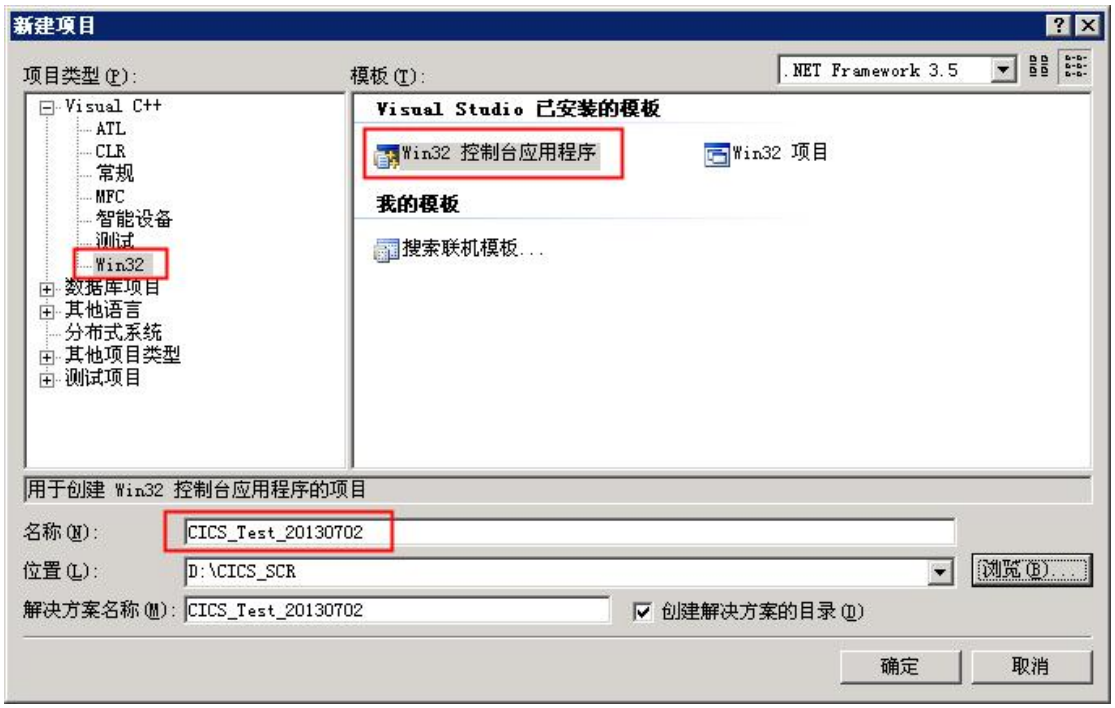
```
#include <string.h>
#include <stdio.h>
_declspec(dllexport) void main();

void main()
{
	unsigned long respCode;
	char *commArea;
	char commTitle[10 + 1];
	char commTeller[4 + 1];
	char commBank[4 + 1];
	char commTerm[4 + 1];
	char commContent[478 + 1];
	int nCalc，i，nCount;

	//memset(commArea，0x00，sizeof(commArea));
	memset(commTitle，0x00，sizeof(commTitle));
	memset(commTeller，0x00，sizeof(commTeller));
	memset(commBank，0x00，sizeof(commBank));
	memset(commTerm，0x00，sizeof(commTerm));
	memset(commContent，0x00，sizeof(commContent));

	EXEC CICS ADDRESS EIB(dfheiptr) RESP(respCode);

	EXEC CICS ADDRESS COMMAREA(commArea) RESP(respCode);

	if(respCode != DFHRESP(NORMAL))
	{
	strcpy(commArea，"error from CICS_Test_20130702.");
	EXEC CICS RETURN;
	}

	strncpy(commTitle，commArea，10);
	strncpy(commTeller，commArea+10，4);
	strncpy(commBank，commArea+14，4);
	strncpy(commTerm，commArea+18，4);
	strncpy(commContent，commArea+22，478);

	EXEC CICS SYNCPOINT;

	sprintf(commArea，"Teller： %s， Bank： %s， Term： %s， Title： %s，
Content： %s"，commTeller，commBank，commTerm，commTitle，commContent);

	EXEC CICS RETURN;
`
```

将文件保存为 CICS_Test_20130702. ccs，注意后缀名为 ccs。

在命令行中执行：cicstran –lC CICS_Test_20130702. ccs。cicstran 为 CICS 的预编译程序，–lC 表示使用 C 语言格式编译，默认在同级目录中会生成

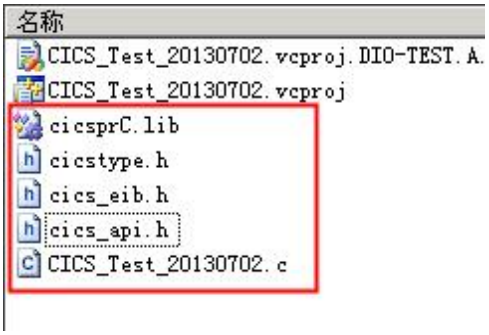
CICS_Test_20130702. c（后缀名为小写 c，如果是以-lCPP，C++格式编译，后缀名为大写 C）。

打开 VS2008，新建项目，选择 Visual C++ -> Win32 控制台应用程序或 Win32 项目都可以，输入项目名称“CICS_Test_20130702”：

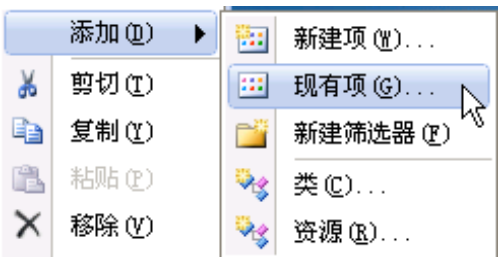


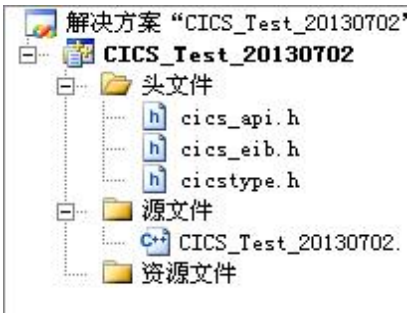
选择“DLL，空项目”：

将 c：\opt\cics\include 下的 cics_api. h， cics_eib. h， cicstype. h（一定是 opt 下的，不能是 CTG 下的）和 c：\opt\lib 下的 cicsprC. lib 和刚刚编译的 CICS_Test_20130702. c 五个文件拷贝到本工程目录下：



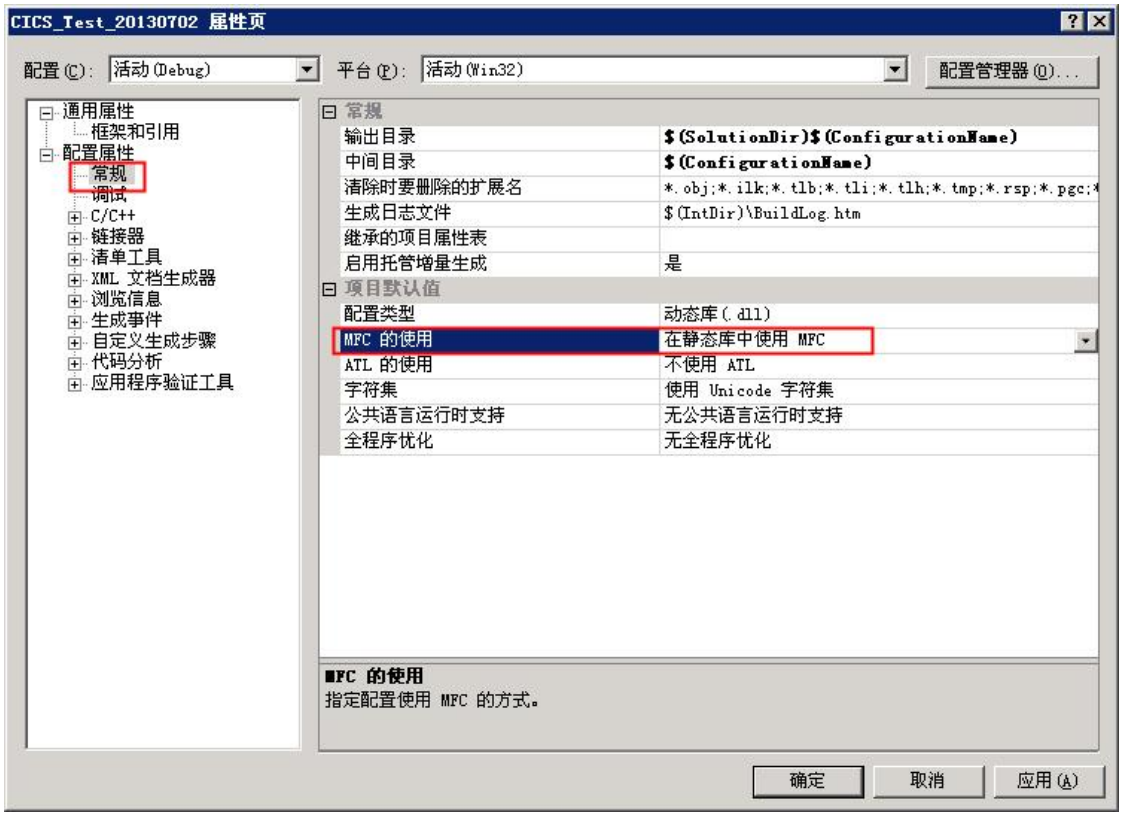
在“头文件”右键添加现有项，把 cics_api. h， cics_eib. h， cicstype. h 都添加进来，在“源文件”右键添加现有项，把 CICS_Test_20130702. c 添加进来：



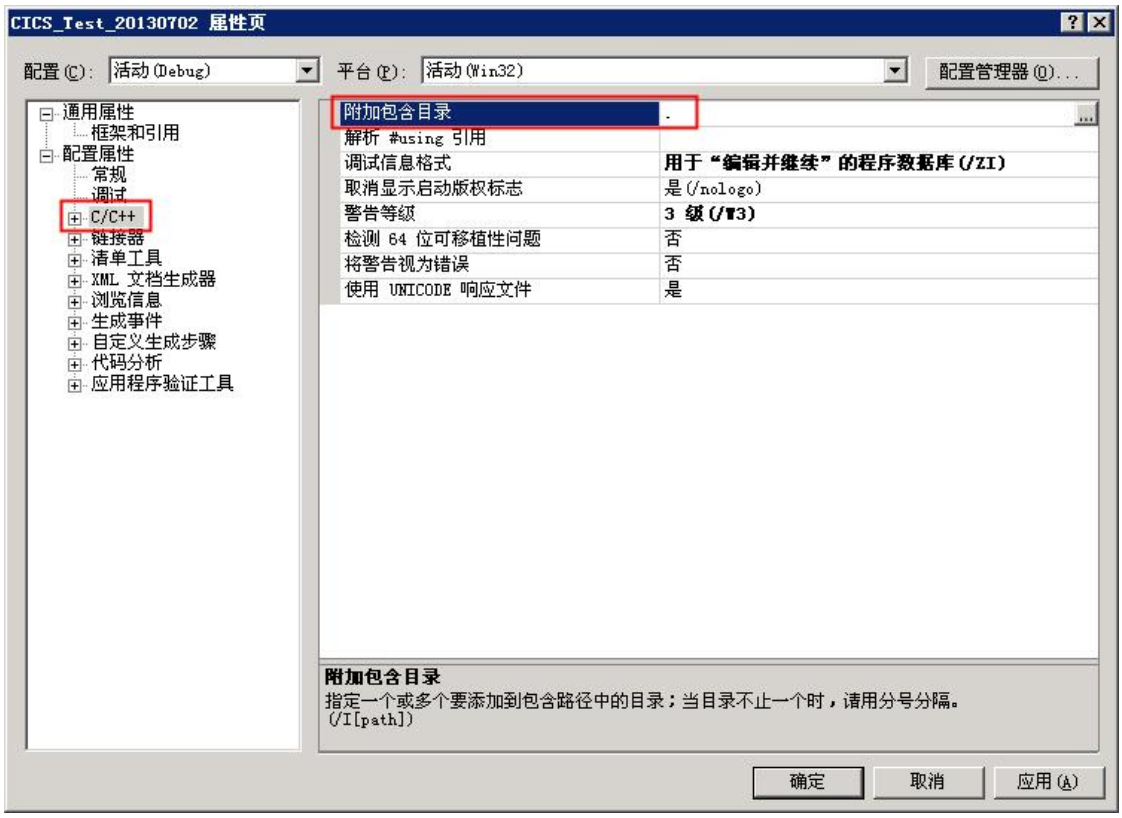


双击打开 CICS_Test_20130702.c 时可能会提示行尾标准化，选择使用 Windows 标准：

打开项目属性，选择“配置属性”->“常规”，“MFC 的使用”中选择“在静态库中使用 MFC”，确认每次修改后都按“应用”键，保持修改：

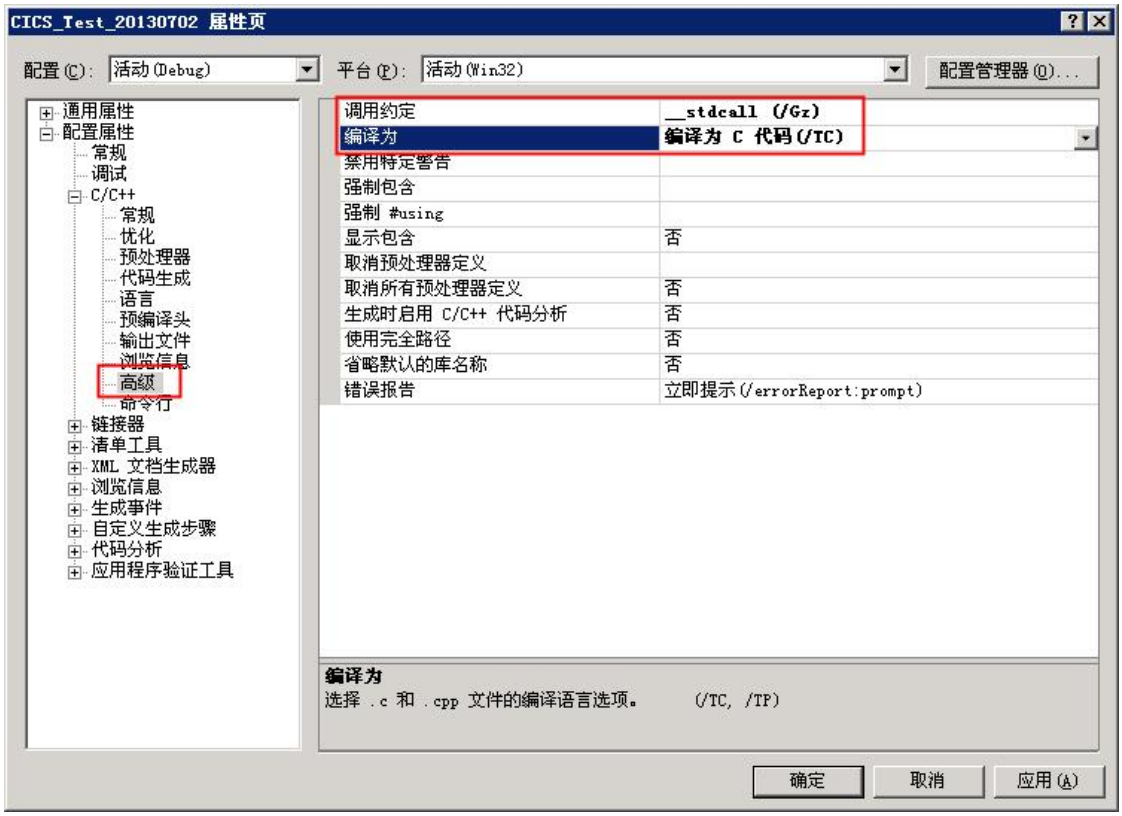


选择“C/C++”->“常规”，“附加包含目录”中包含当前目录“.”：

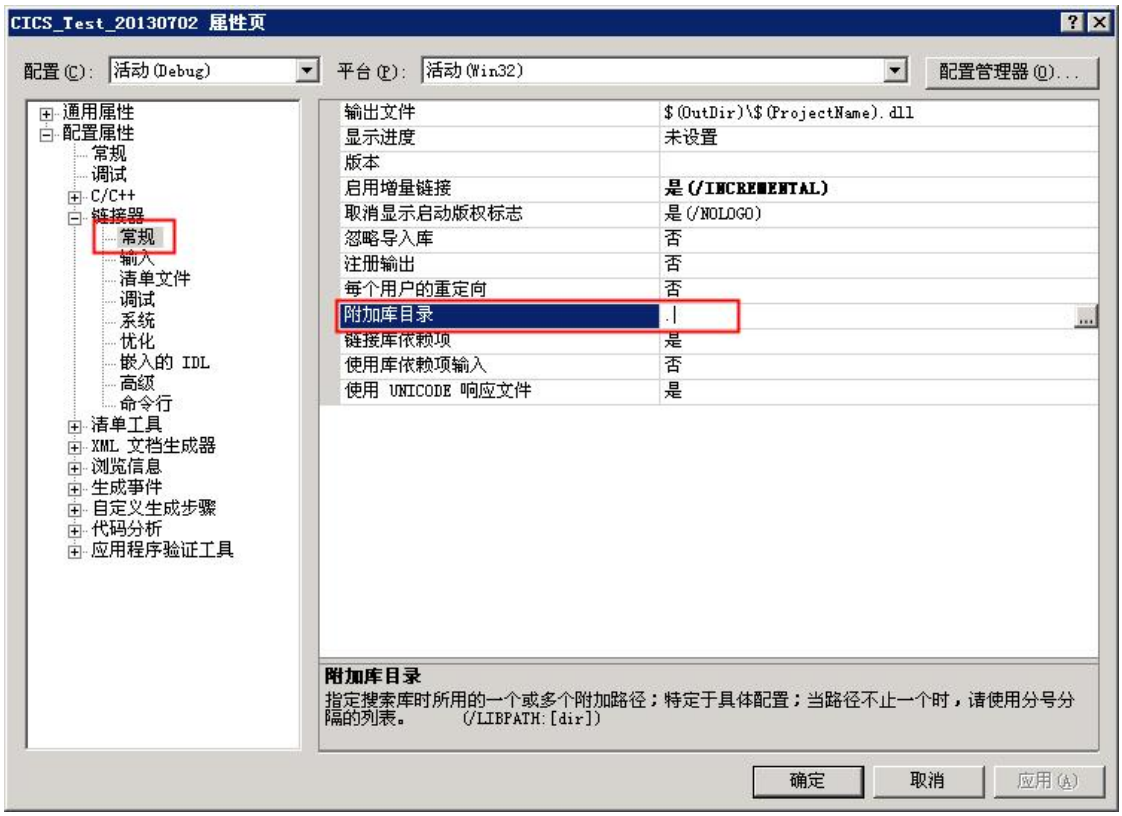


选择“C/C++”->“高级”，“调用约定”为“_stdcall（/Gz）”，编译为“编

译为 C 代码（/TC）”：

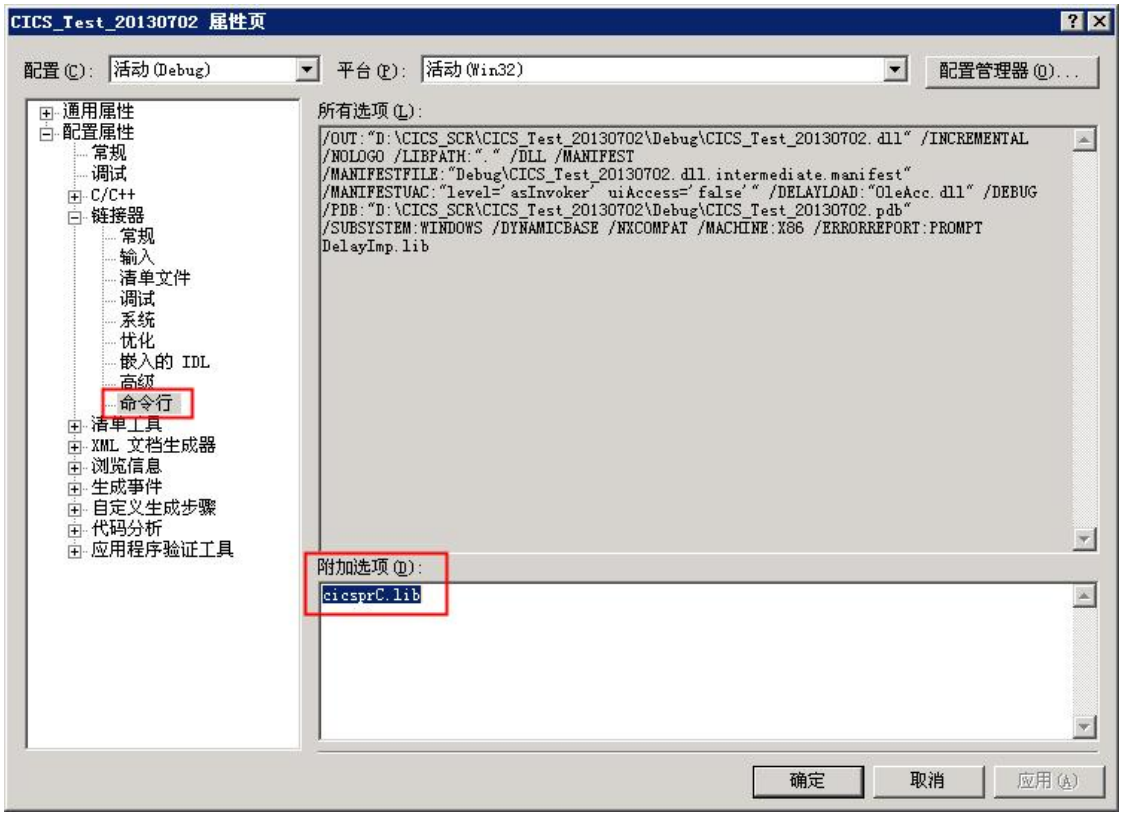


选择“链接器”->“常规”，“附加库目录”中包含当前目录“.”：

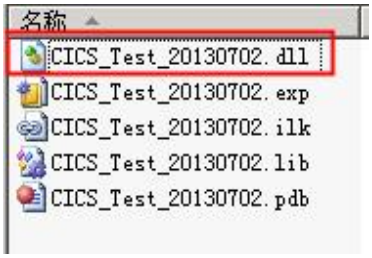


选择“链接器”->“命令行”，“附加选项”中输入 CICS for Windows 的

库文件 cicsprC. lib：

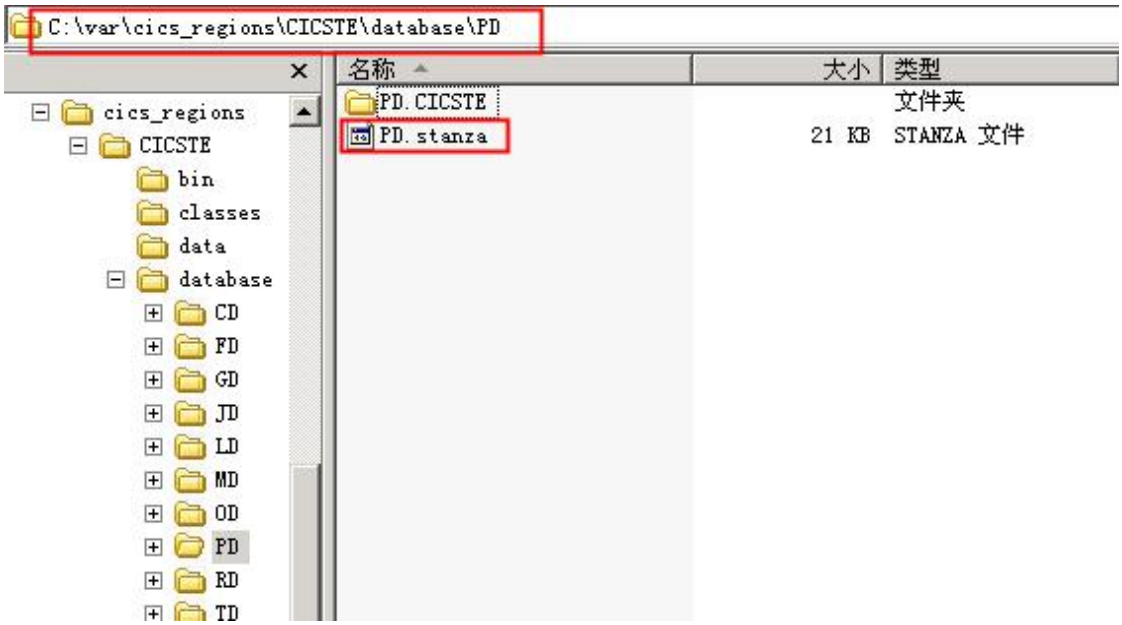


确定后选择工具栏“生成”->“生成解决方案”，即可在 Debug 中看到生成的 dll 文件：



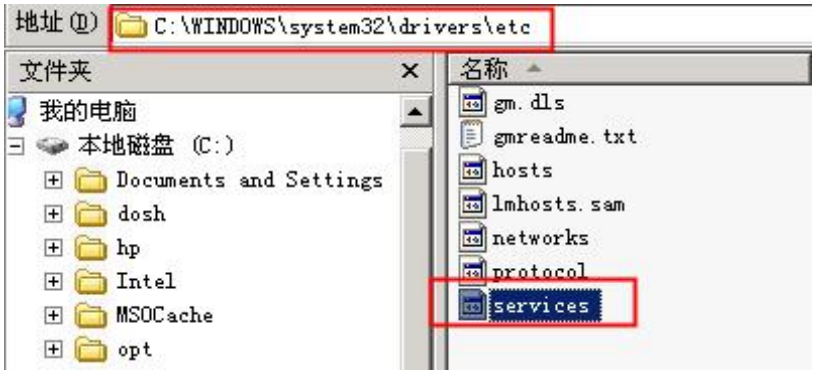
三、 配置 CICS Region

要使当前 CICS Region 正常运行，CICS 程序能够被调起必须对 Region 的一些配置进行设置。本文推荐使用直接编辑 CICS Region 的配置文件，CICS Region 的配置文件在%Dir%\var\cics_regions\%CICS Region%\database 下，各配置以文件夹形式分开，配置文件为文件夹下 stanza 后缀的文件：

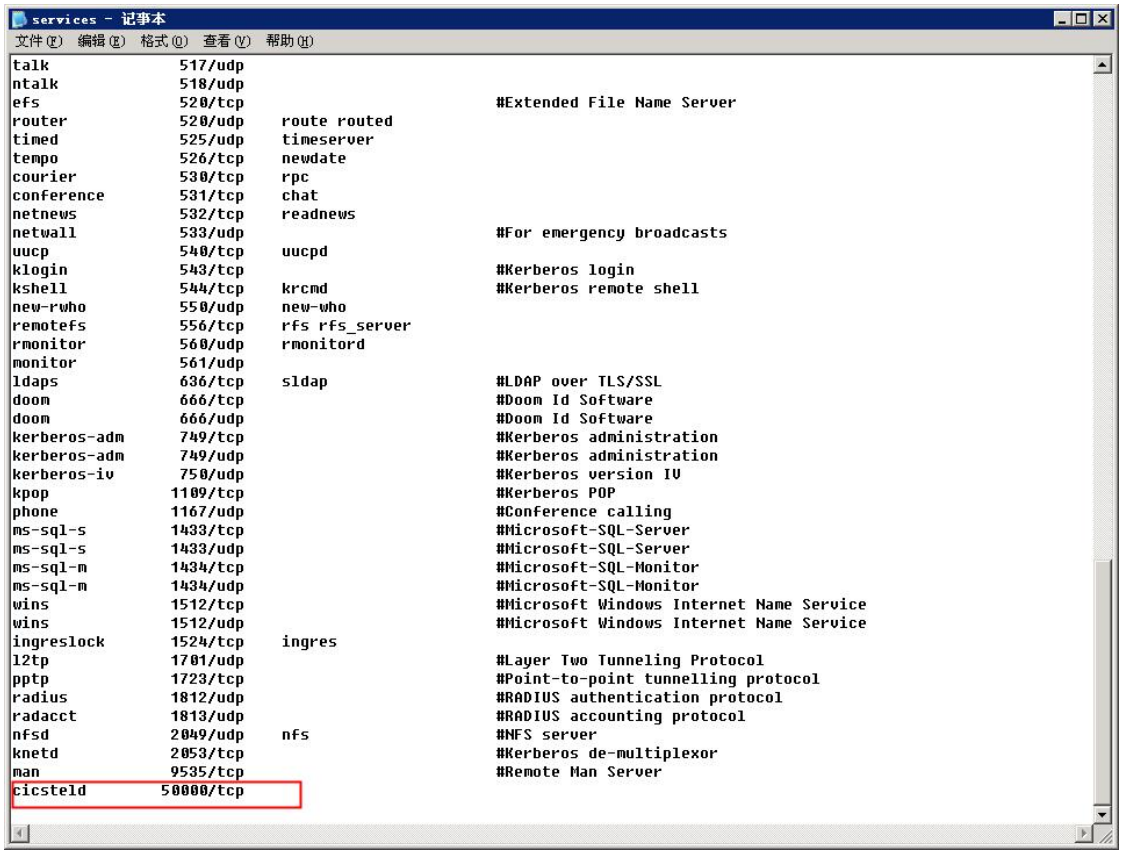


1. 配置 LD

每个 CICS Region 必须配置 LD 才能提供对外的服务端口。首先打开 C:\WINDOWS\system32\drivers\etc 下的 services 文件：

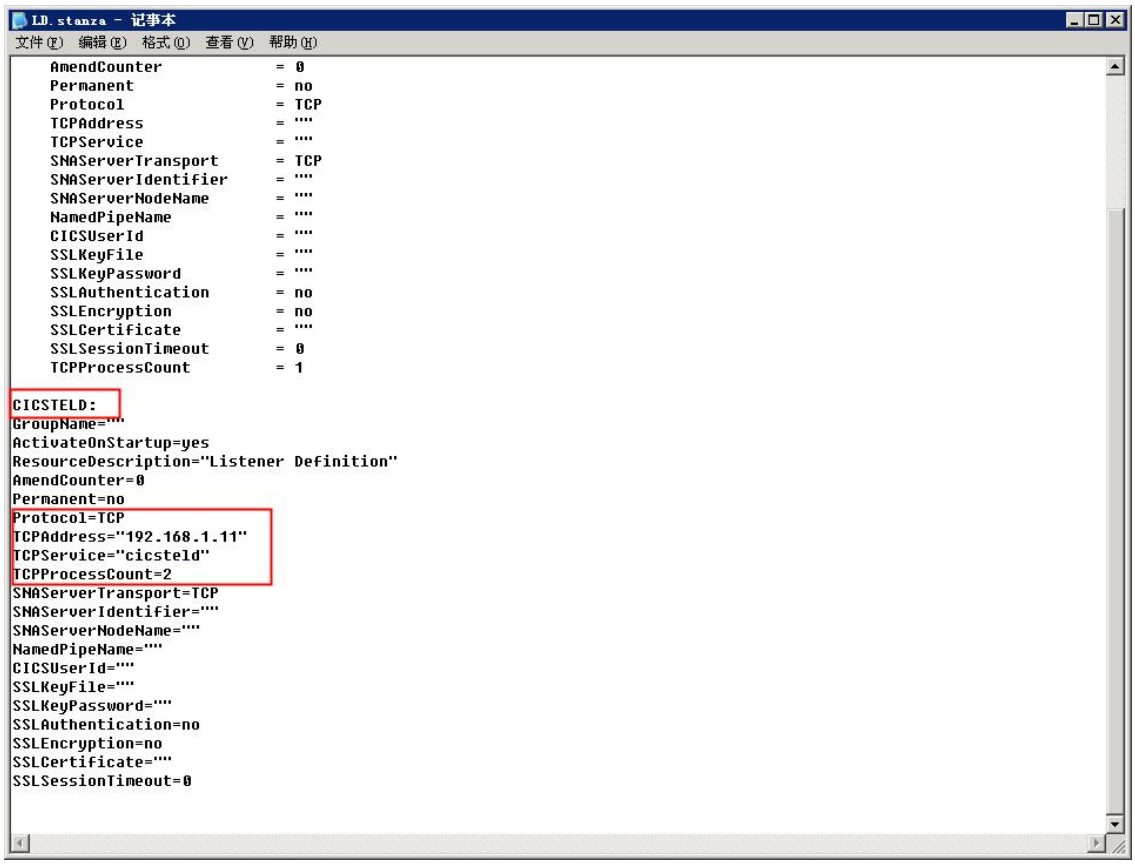


最后一行添加一个服务端口，cicsteld 50000/tcp #CICSTE LD（端口使用一个没有被其他服务占用的端口，services 文件最后留一个空行）：


services - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
talk 517/udp
ntalk 518/udp
efs 520/tcp #Extended File Name Server
router 520/udp route routed
timed 525/udp timeserver
tempo 526/tcp newdate
courier 530/tcp rpc
conference 531/tcp chat
netnews 532/tcp readnews
netwall 533/udp #For emergency broadcasts
uucp 540/tcp uucpd
klogin 543/tcp #Kerberos login
kshell 544/tcp krcmd #Kerberos remote shell
new-rwho 550/udp new-who
remotefs 556/tcp rfs rfs_server
rmonitor 560/udp rmonitord
monitor 561/udp
ldaps 636/tcp sldap #LDAP over TLS/SSL
doom 666/tcp #Doom Id Software
doom 666/udp #Doom Id Software
kerberos-adm 749/tcp #Kerberos administration
kerberos-adm 749/udp #Kerberos administration
kerberos-iv 750/udp #Kerberos version IV
kpop 1109/tcp #Kerberos POP
phone 1167/udp #Conference calling
ms-sql-s 1433/tcp #Microsoft-SQL-Server
ms-sql-s 1433/udp #Microsoft-SQL-Server
ms-sql-m 1434/tcp #Microsoft-SQL-Monitor
ms-sql-m 1434/udp #Microsoft-SQL-Monitor
wins 1512/tcp #Microsoft Windows Internet Name Service
wins 1512/udp #Microsoft Windows Internet Name Service
ingreslock 1524/tcp ingres
l2tp 1701/udp #Layer Two Tunneling Protocol
pptp 1723/tcp #Point-to-point tunnelling protocol
radius 1812/udp #RADIUS authentication protocol
radacct 1813/udp #RADIUS accounting protocol
nfsd 2049/udp nfs #NFS server
knetd 2053/tcp #Kerberos de-multiplexor
man 9535/tcp #Remote Man Server
cicsteld 50000/tcp

在 CICS Region database 的 LD 配置文件中新增一个 LD 定义，可以复制默认的定义，确认 Protocol 为 TCP，TCPAddress 为本机 ip 地址，TCPService 为刚才在 services 文件中添加的新服务名，TCPProcessCount 设置为 2，Region 将启动 2 个 cicsip 监听程序：

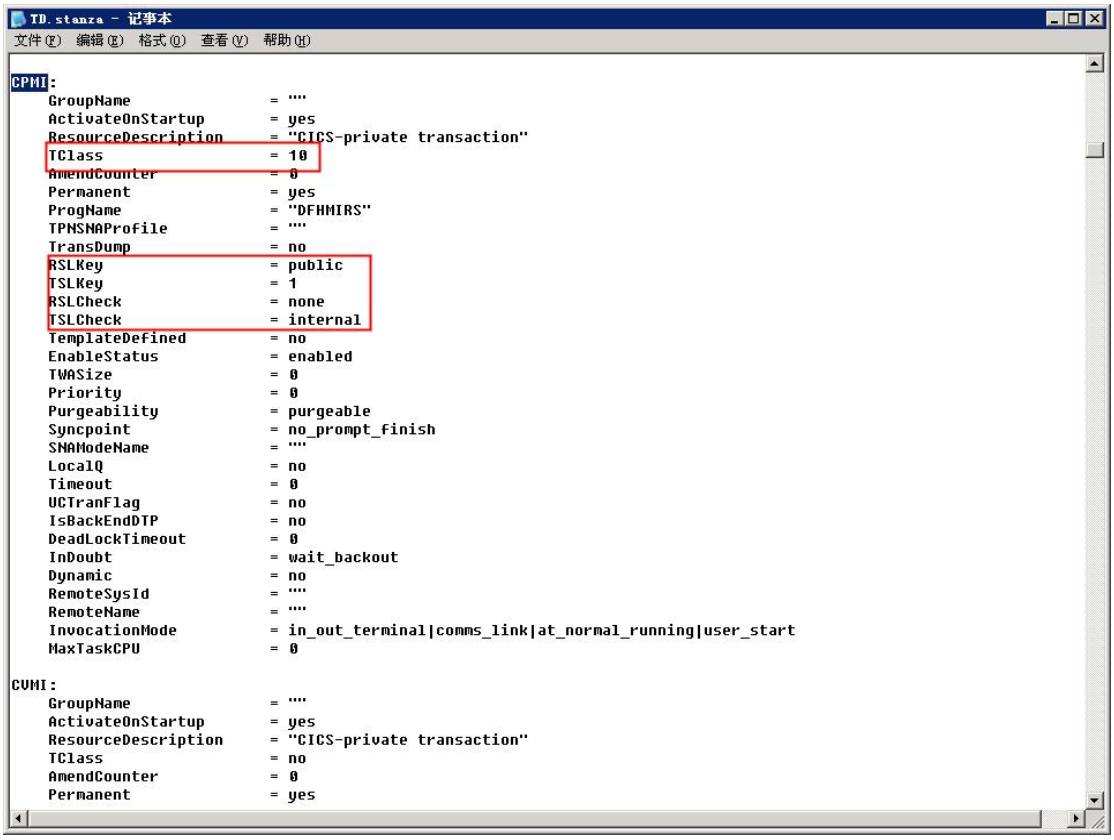
```
CICSTELD：
GroupName=""
ActivateOnStartup=yes
ResourceDescription="Listener Definition"
AmendCounter=0
Permanent=no
Protocol=TCP
TCPAddress="192.168.1.11"
TCPService="cicsteld"
TCPProcessCount=2
SNAServerTransport=TCP
SNAServerIdentifier=""
SNAServerNodeName=""
NamedPipeName=""
CICSUserId=""
SSLKeyFile=""
SSLKeyPassword=""
SSLAuthentication=no
SSLEncryption=no
SSLCertificate=""
SSLSessionTimeout=0
```


LD.stanza - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
AmendCounter = 0
Permanent = no
Protocol = TCP
TCPAddress = ""
TCPService = ""
SNAServerTransport = TCP
SNAServerIdentifier = ""
SNAServerNodeName = ""
NamedPipeName = ""
CICSUserId = ""
SSLKeyFile = ""
SSLKeyPassword = ""
SSLAuthentication = no
SSLEncryption = no
SSLCertificate = ""
SSLSessionTimeout = 0
TCPProcessCount = 1
CICSTELD:
GroupName=""
ActivateOnStartup=yes
ResourceDescription="Listener Definition"
AmendCounter=0
Permanent=no
Protocol=TCP
TCPAddress="192.168.1.11"
TCPService="cicsteld"
TCPProcessCount=2
SNAServerTransport=TCP
SNAServerIdentifier=""
SNAServerNodeName=""
NamedPipeName=""
CICSUserId=""
SSLKeyFile=""
SSLKeyPassword=""
SSLAuthentication=no
SSLEncryption=no
SSLCertificate=""
SSLSessionTimeout=0

2. 配置 TD

编辑 CICS Region database 的 TD 配置文件，修改 CPMI（我们的 CTG 都调用 CICS 的 CPMI 镜像交易），RSLKey 为 public，RSLCheck 为 none，确认 TSLKey 为 1，因为 CICS 默认的用户 CICSUSER 可以访问 TSLKey 为 1 的交易，修改 TClass=10：

```
CPMI:
        GroupName                = ""
        ActivateOnStartup        = yes
        ResourceDescription      = "CICS-private transaction"
        TClass                   = 10
        AmendCounter             = 0
        Permanent                = yes
```

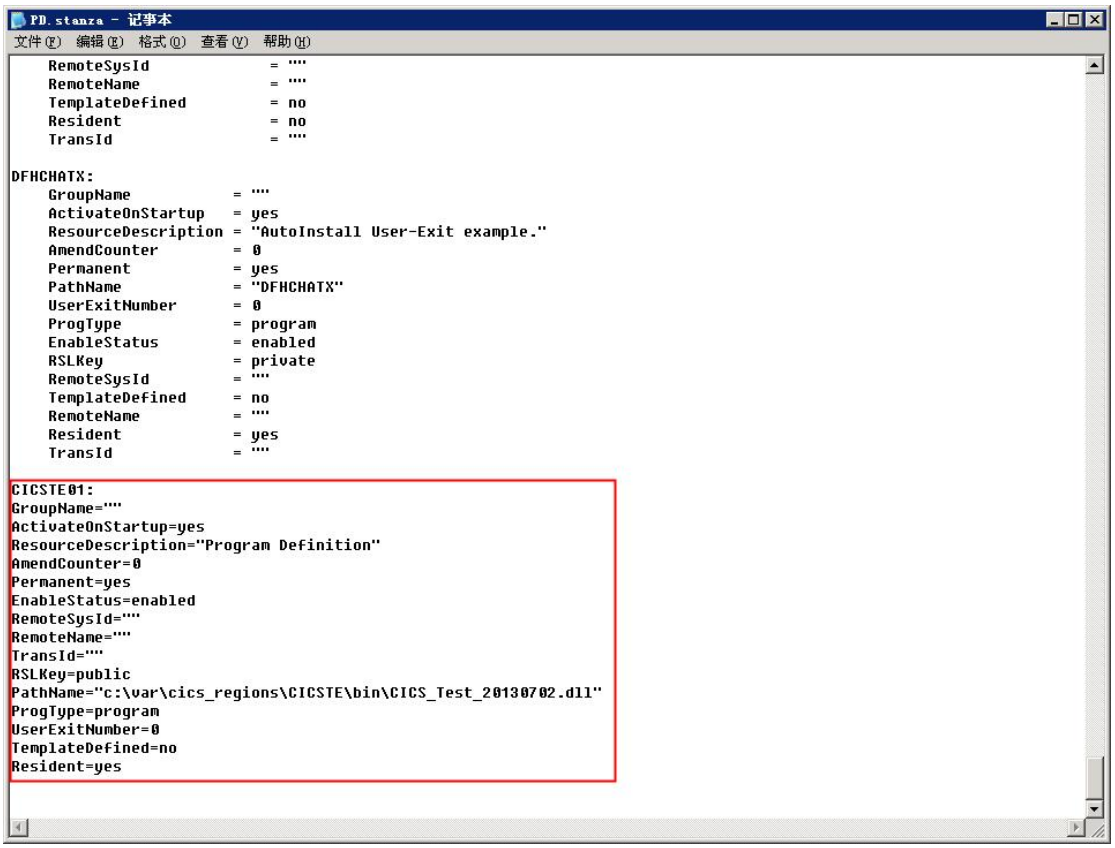


3. 配置 PD

将 CICS_Test_20130702. dll 程序复制到 %Dir%\var\cics_regions\CICSTE\bin\目录下，在 CICSTE 的 PD 中添加程序定义：

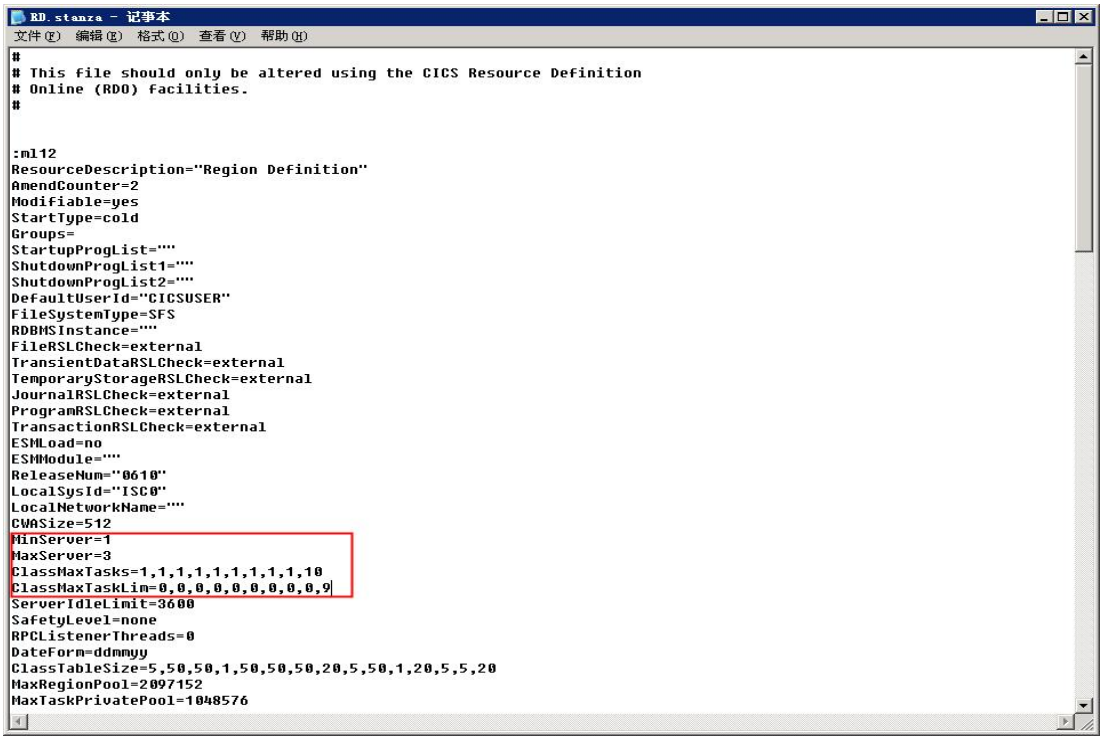
```
CICSTE01：
GroupName=""
ActivateOnStartup=yes
ResourceDescription="Program Definition"
AmendCounter=0
Permanent=yes
EnableStatus=enabled
RemoteSysId=""
RemoteName=""
TransId=""
RSLKey=public
PathName="c：\var\cics_regions\CICSTE\bin\CICS_Test_20130702.dll"
ProgType=program
UserExitNumber=0
TemplateDefined=no
Resident=yes
```

注意确认新程序 RSLKey 为 public，PathName 为程序存放的路径（推荐存放在 region 的 bin 目录下），路径中包括程序名：



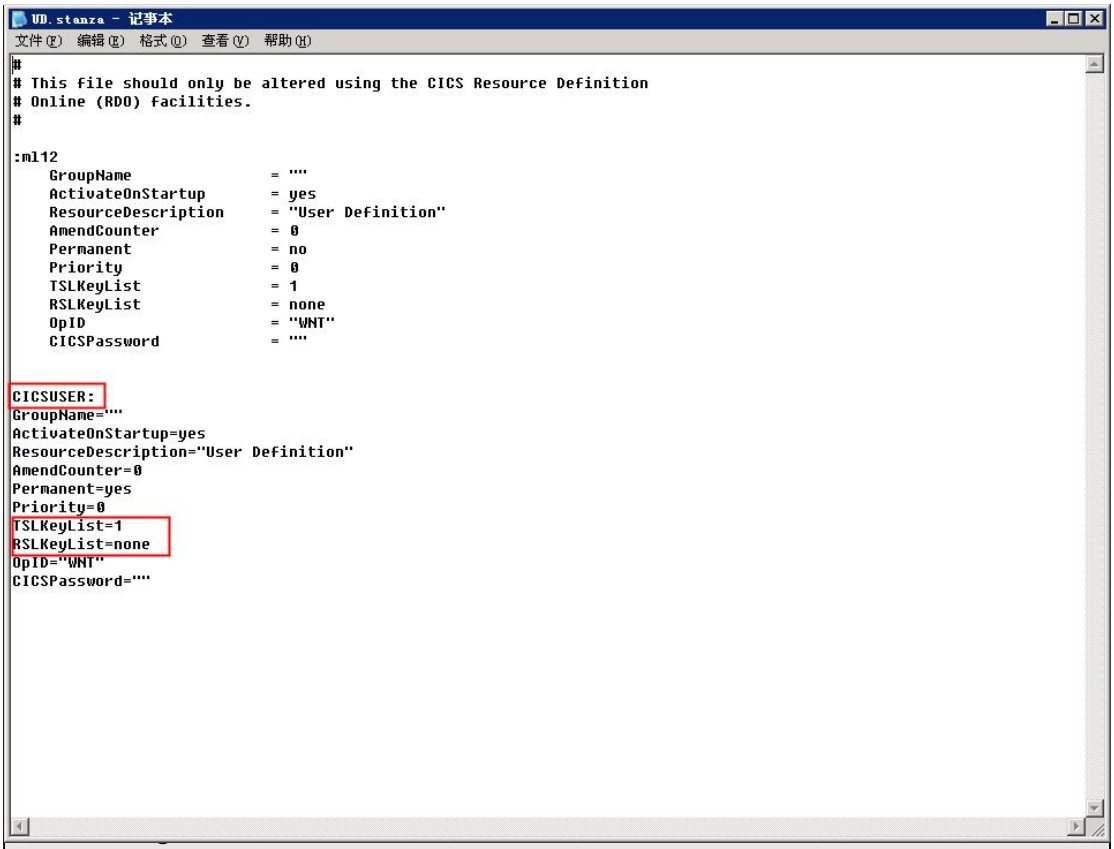
4. 配置 RD

编辑 CICS Region database 的 RD 文件，可以设置本 Region 的基本属性，CICS Region 的性能调节基本都在这里设置，保持 MinServer=1，MaxServer=3，修改 ClassMaxTasks 的最后一个值为 10，ClassMaxTaskLim 的最后一个值为 9，更多的配置在后续章节中实验：



：ml12
ResourceDescription="Region Definition"
AmendCounter=2
Modifiable=yes
StartType=cold
Groups=
StartupProgList=""
ShutdownProgList1=""
ShutdownProgList2=""
DefaultUserId="CICSUSER"
FileSystemType=SFS
RDBMSInstance=""
FileRSLCheck=external
TransientDataRSLCheck=external
TemporaryStorageRSLCheck=external
JournalRSLCheck=external
ProgramRSLCheck=external
TransactionRSLCheck=external
ESMLoad=no
ESMModule=""
ReleaseNum="0610"
LocalSysId="ISC0"

```
PurgeDelayPeriod=8
StatsRecord=yes
StatFile="statsfile"
SufficesSupported=yes
CheckpointInterval=1000
DefaultFileServer="/.：/cics/sfs/%H"
RecTSQFile="%Rcicsrectsqfile"
RecTSQIndex="cicsrectsqidx"
RecTSQVol="sfs_%S"
RecTSQPrePages=5
RecTSQMaxRecs=1000000
NonRecTSQFile="%Rcicsnrectsqfil"
NonRecTSQIndex="cicsnrectsqidx"
NonRecTSQVol="sfs_%S"
NonRecTSQPrePages=5
NonRecTSQMaxRecs=1000000
LogicalTDQFile="%Rcicstdqlgfile"
LogicalTDQIndex="cicstdqlgidx"
LogicalTDQVol="sfs_%S"
LogicalTDQPrePages=5
LogicalTDQMaxRecs=1000000
PhysicalTDQFile="%Rcicstdqphfile"
PhysicalTDQIndex="cicstdqphidx"
PhysicalTDQVol="sfs_%S"
PhysicalTDQPrePages=5
PhysicalTDQMaxRecs=1000000
NonRecTDQFile="%Rcicstdqnofile"
NonRecTDQIndex="cicstdqnoidx"
NonRecTDQVol="sfs_%S"
NonRecTDQPrePages=5
NonRecTDQMaxRecs=1000000
LocalQProtectFile="%Rcicsplqfile"
LocalQProtectIndex="cicsplqidx"
LocalQProtectVol="sfs_%S"
LocalQProtectPrePages=5
LocalQProtectMaxRecs=1000000
LocalQFile="%Rcicsnlqfile"
LocalQIndex="cicsnlqidx"
LocalQVol="sfs_%S"
LocalQPrePages=5
LocalQMaxRecs=1000000
TSQAgeLimit=20
ProgramCacheSize=0
CUBSDelayMinutes=5
CARPDelayHours=8
ProtectPurgeDelayPeriod=8
```



EAMLoad=no

EAMModule=""

ServerMemCheckInterval=0

5. 配置 UD

UD 中维护 CICS Server 上的用户，默认只有一个 CICSUSER，可通过命令：cicsadd -c ud -r %region name% -P %user name% CICSPassword=″%password%″来添加新用户，注意 password 部分在配置文件中为加密显示。

6. **cold 方式重启 Region CICSTE**

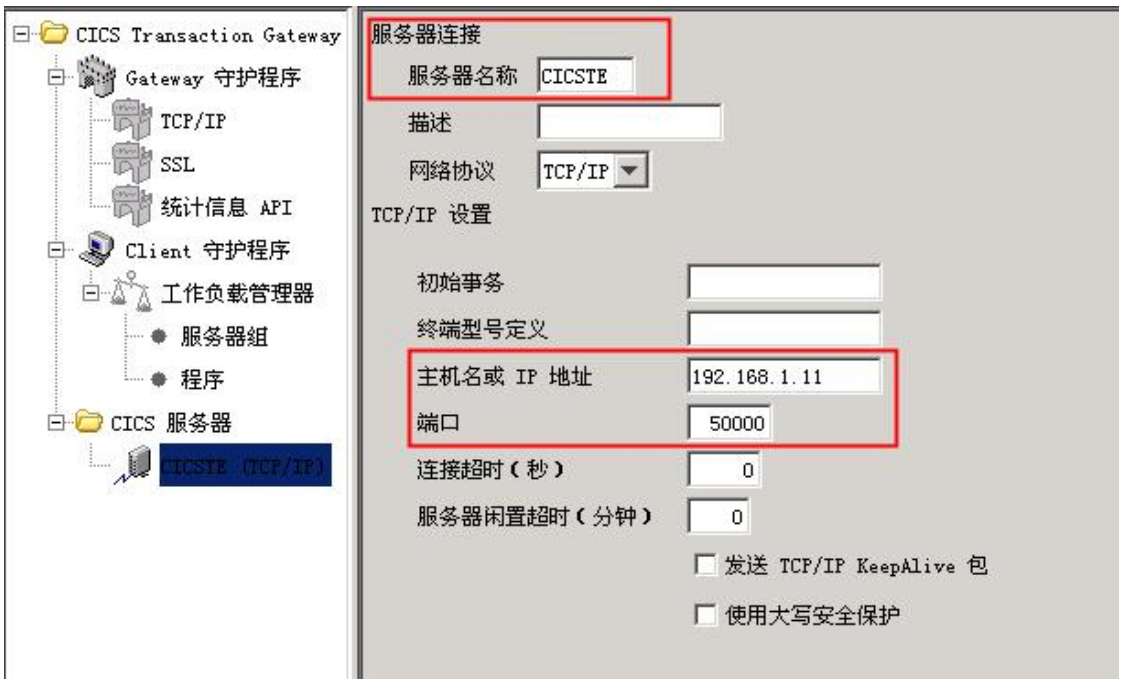
命令行中输入 cicscp –v stop region CICSTE

cicscp –v start region CICSTE StartType=cold

四、 编写 CICS 性能测试脚本

1. 使用 LoadRunner VUGen 调试 CICS 程序

在另一台机器上的 CTG 中配置好与 CICS Server Region 的连接：



在 LoadRunner 中新建一个 ECI_Call 的简单脚本（ECI_Call 脚本创建请参看笔者写的另一篇总结文档《使用 LoadRunner 调用 ECI 程序对 CICS Server 加压》）。

首先调用 CICS_Test_20130702 对应的 PD 程序 CICSTE01，执行后可以看到 replay log 中显示已经收到 CICS Server 上程序的返回信息：

```
Starting iteration 1.
Starting action Action.
Action.c(28): Notify: Parameter Substitution: parameter "Teller" = "T002"
Action.c(29): Notify: Next row for parameter Term = 2 [table = Term].
Action.c(29): Notify: Getting new value for parameter 'Term': table = 'Teller.dat' column = '2' row = '2'.
Action.c(29): Notify: Parameter Substitution: parameter "Term" = "Te02"
Action.c(30): Notify: Next row for parameter Bank = 2 [table = Bank].
Action.c(30): Notify: Getting new value for parameter 'Bank': table = 'Teller.dat' column = '1' row = '2'.
Action.c(30): Notify: Parameter Substitution: parameter "Bank" = "B001"
Action.c(34): Send commArea: 2013070101T002Te02B001ContentAAAAAAAAAABBBBBBBBBB
Action.c(36): Notify: Transaction "CICS_Test" started.
Action.c(42): Notify: Transaction "CICS_Test" ended with "Pass" status (Duration: 0.0292).
Action.c(44): the commArea is: Teller: T002, Bank: Te02, Term: B001, Title: 2013070101, Content: ContentAAAAAAAAAABBBBBBBBBB
Ending action Action.
Ending iteration 1.
```

2. 使用 CICS Terminal 监控

对 CICS Server 的监控可以通过 CTG 里的 CICS Terminal 实现，打开 CICS Terminal：

如果本地有多个 CICS Region 则选择要连接的 CICS：

在使用 CSTD 交易前，先对操作 CICSTERM 的功能键做一次简单的了解，在日常的操作中主要会使用到如下几个键位：

Ctrl：用于发送指令，刷新屏幕结果显示。

F3：用于返回到 CSTD 的主菜单。

PF5：停止自动更新（屏幕显示）。

PF6：开启自动更新（屏幕显示）。

PF7：显示当前信息的下一屏。

PF8：显示当前信息的前一屏。

在屏幕中输入 CSTD，按 Ctrl 键确认：

显示 CICS 统计信息的主菜单：

输入 7，按 Ctrl 键进入“程序统计”菜单，“显示非活动的程序”后输入 N，按 Ctrl 键确认，可以看到我们之前执行的 CICSTE01 程序执行了 13 次：

按 F3 返回主菜单后按 8，Ctrl 进入“事务统计”菜单，可以看到我们调用

的 CPMI 事务也执行了 13 次：

按 F3 返回主菜单后按 9，Ctrl 进入“类的最大任务统计”菜单，这里列出 CICS 中 10 个交易类别的最大活动、队列长度、当前、峰值等统计值。

TD 中每个事务都有一个 TClass，标记了这个事务的类别，默认 CPMI 的 TClass=10。RD 中的 ClassMaxTasks 标明了每个类别允许的最大并发数，超过的事务将放到队列里，这里我定义了 TClass 10 类别的事务最大并发数为 10，排队数量为 9-1=8，这里显示的队列长度为并发数+排队数=10+8=18：

```
ClassMaxTasks=1,1,1,1,1,1,1,1,1,10
ClassMaxTaskLim=0,0,0,0,0,0,0,0,0,9
```

按 F3 返回主菜单后按 11，Ctrl 进入“事务处理/程序速率”菜单，这里可以实时看到 CICS 程序的处理速度：

五、 使用 Controller 对 CICS 发压并监控

1. CICS Server 主要参数

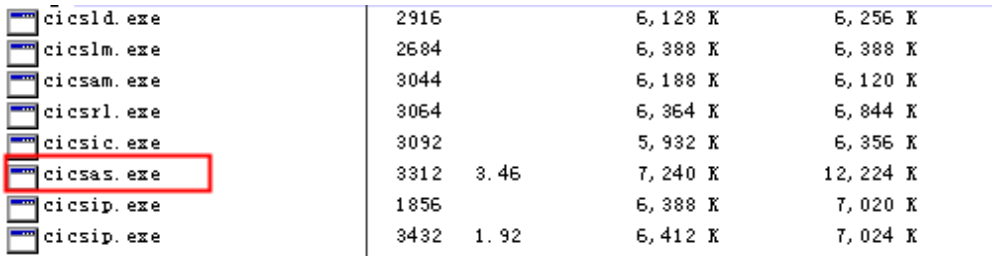
2. 基准测试

1） CICS Server 设置

在使用 LoadRunner 发压前确认 CICS Region 的部分参数：

- LD：TCPProcessCount=2，Region 会默认建立 2 个 cicsip 进程，用来并发监听多个 CTG 的请求，但 CTG 仅配置一个连接。
- RD：MinServer=1，MaxServer=1，Region 最少，最大均只创建 1 个 cicsas 进程用来处理请求。ClassMaxTasks=1，1，1，1，1，1，1，1，1，10，定义 TClass 为 10 的事务（CPMI）能并发处理的最大任务数。ClassMaxTaskLim=0，0，0，0，0，0，0，0，0，5，定义 TClass 为 10 的事务能最大排队的任务数。

此时查看系统进程可以看到 CICS Region 已经启动了 2 个 cicsip 进程和 1 个 cicsas 进程：



cicsld.exe	2916		6,128 K	6,256 K
cicslm.exe	2684		6,388 K	6,388 K
cicsam.exe	3044		6,188 K	6,120 K
cicsrl.exe	3064		6,364 K	6,844 K
cicsic.exe	3092		5,932 K	6,356 K
cicsas.exe	3312	3.46	7,240 K	12,224 K
cicsip.exe	1856		6,388 K	7,020 K
cicsip.exe	3432	1.92	6,412 K	7,024 K

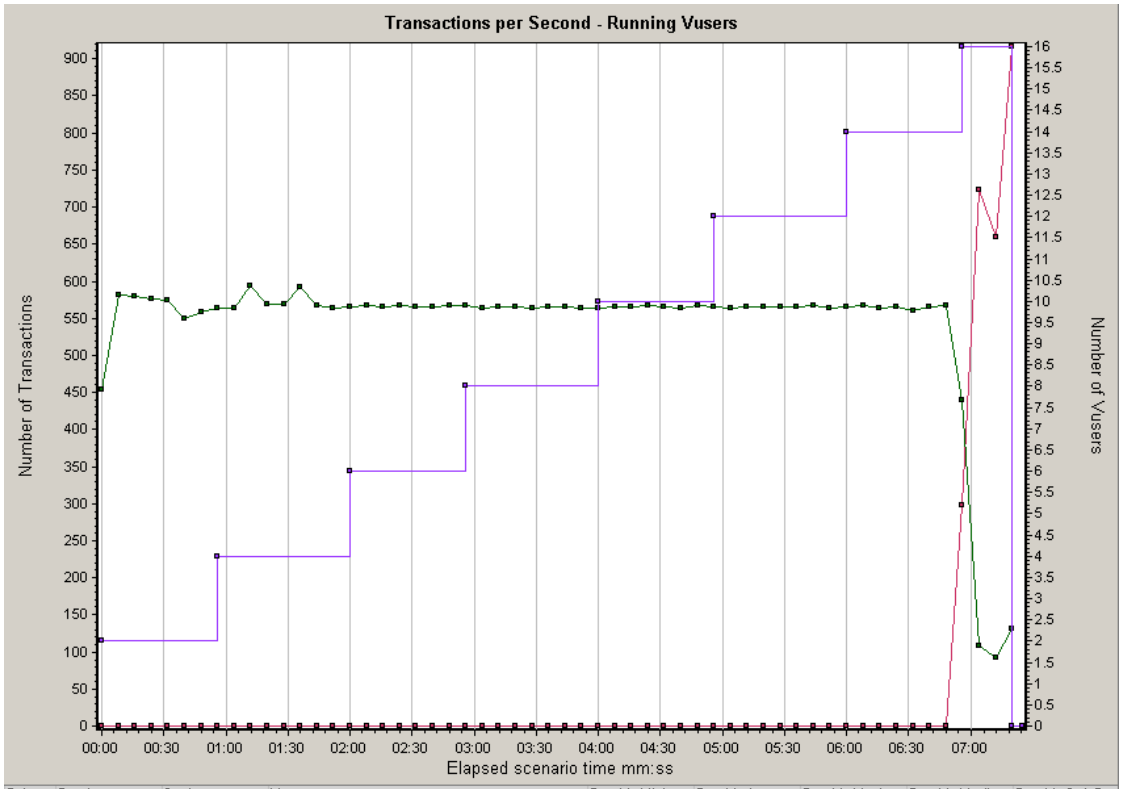
2) CTG 设置

在 CTG Configuration Tool 设置中默认只建立一个 Region.

3) 发压及 CICS 监控

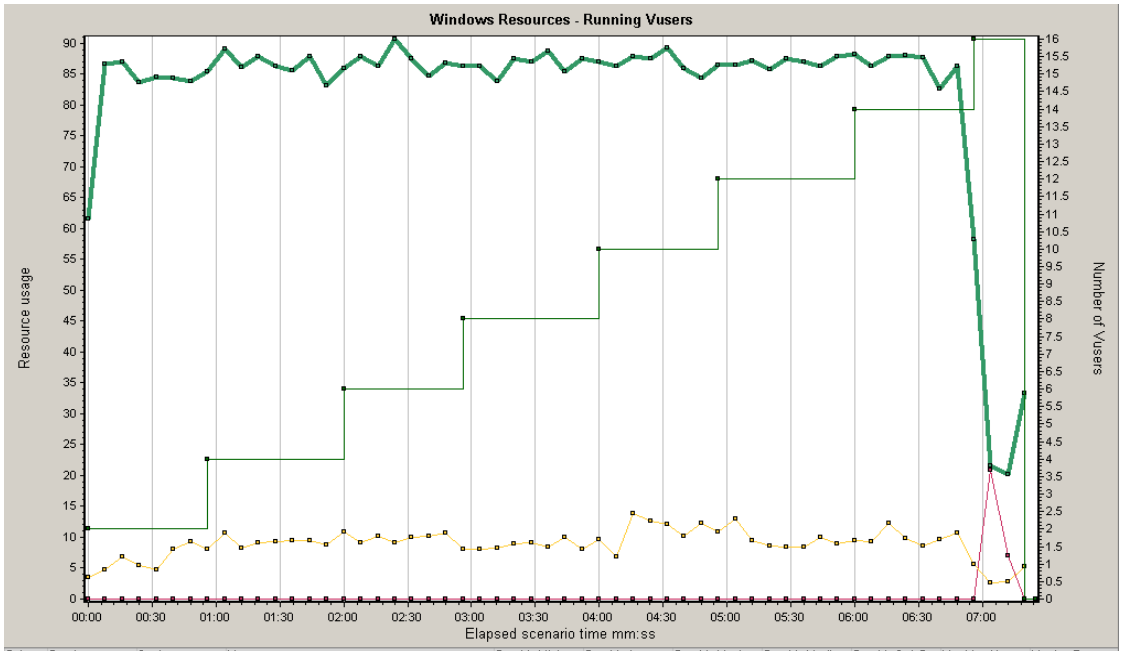
在 LoadRunner 脚本中对上送字段进行参数化，设置场景：每 1 分钟递增 2 个 VUser，最大 20 个 VUser。

从测试结果可以看出，场景初始状态 TPS 就基本稳定在 570 笔/秒左右，而 VUser 数量达到 14 时达到 RD 中 Task 队列的总数，Transaction 开始大量报错：



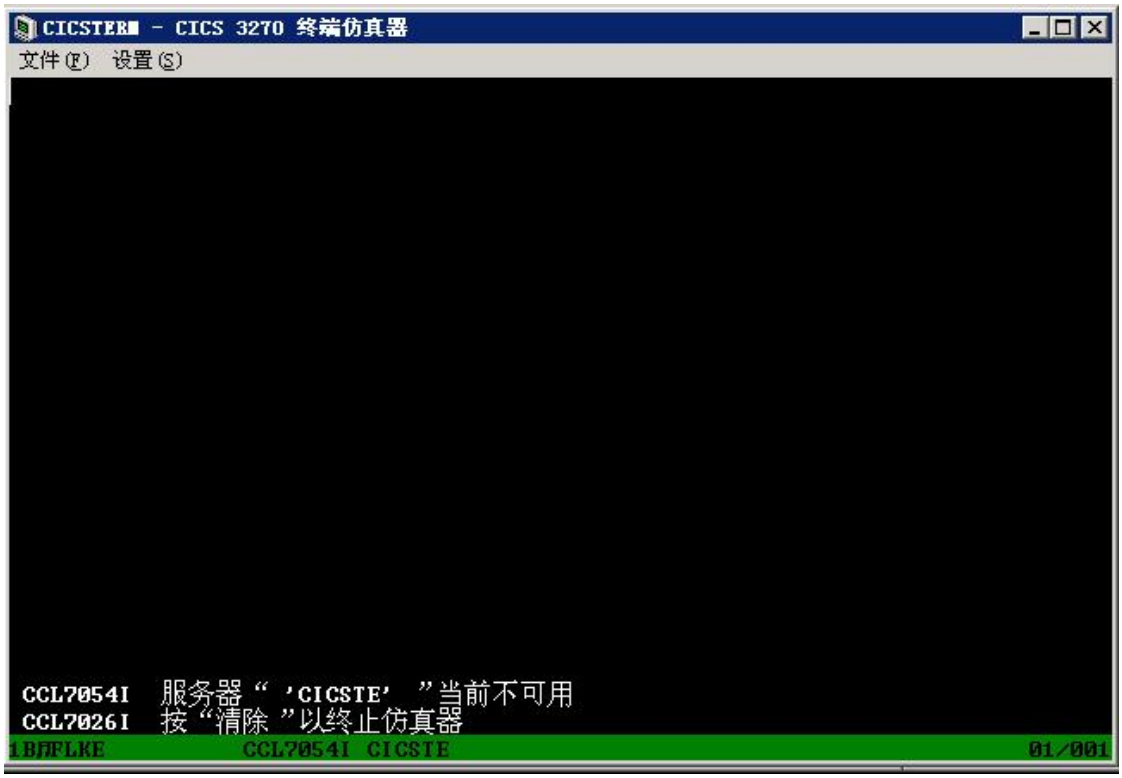
Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	CICS_Test:Fail	0	46.387	916.4	0	177.584
	Transactions per	1	CICS_Test:Pass	92.625	538.215	593.125	565.125	104.303
	Running Vusers	1	Run	0	7.2	16	8	5.455

结合系统资源图也可以看出，当 VUser 达到 14 时，cicsas 的 CPU 利用率也开始暴跌，此时只有一个 cicsip 的 CPU 利用率较高，另一个 cicsip 进程的 CPU 利用率几乎为零，Region CICSTE 已经停止服务了：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev	Machine Name	Monitor Type
	Windows Resour	1	% Processor Time (Process cicsas):192.168.1.11	20.139	82.271	90.625	86.285	14.638	192.168.1.11	
	Windows Resour	10	% Processor Time (Process cicsip#1):192.168.1.11	0	0.05	2.084	0	0.289	192.168.1.11	
	Windows Resour	1	% Processor Time (Process cicsip):192.168.1.11	2.604	8.861	13.802	9.202	2.372	192.168.1.11	
	Running Vusers	1	Run	0	7.2	16	8	5.455	N/A	N/A

CICSTERM 无法再显示：



3. 对比测试：MaxTasks 和 MaxTaskLim

1） CICS Server 设置

确认 CICS Region 的部分参数：

- LD：TCPProcessCount=2，Region 会默认建立 2 个 cicsip 进程，用来并发监听多个 CTG 的请求，但 CTG 仅配置一个连接。
- RD：MinServer=1，MaxServer=1，Region 最少，最大均只创建 1 个 cicsas 进程用来处理请求。ClassMaxTasks=1，1，1，1，1，1，1，1，1，20，ClassMaxTaskLim=0，0，0，0，0，0，0，0，0，10。增加并发 Task 数量和排队数量。

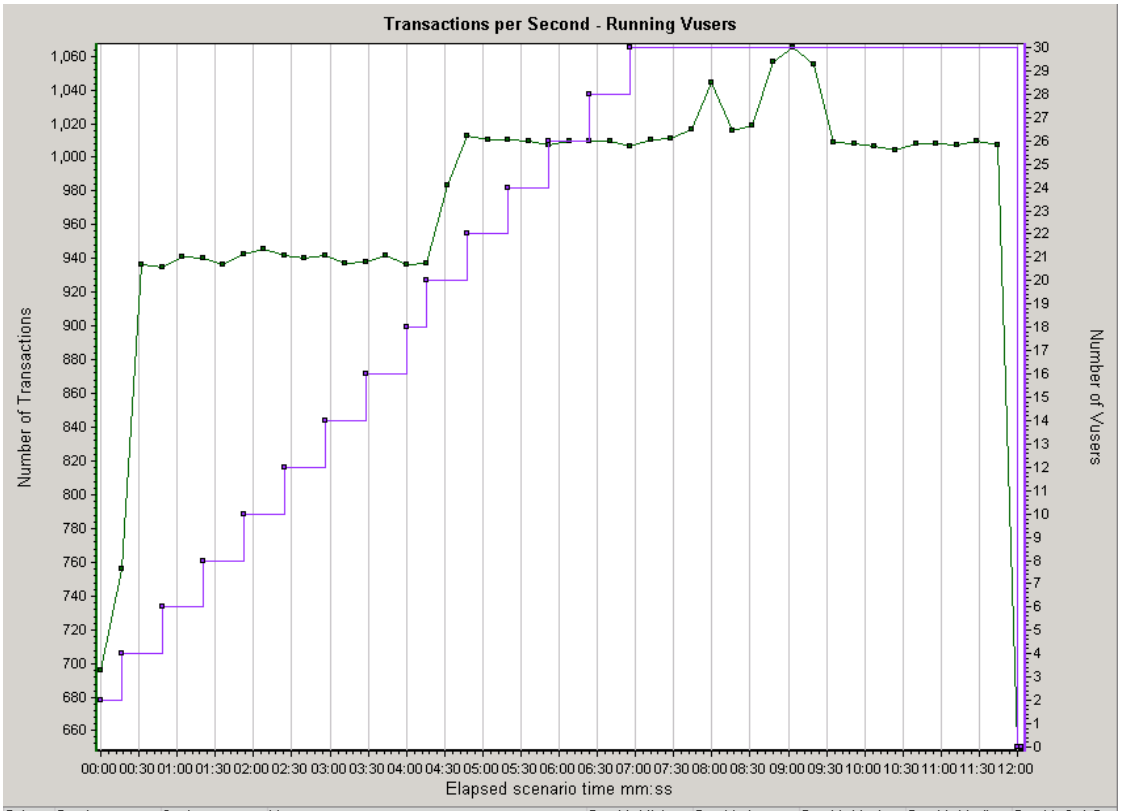
2） CTG 设置

CTG 保持不变。

3） 发压及 CICS 监控

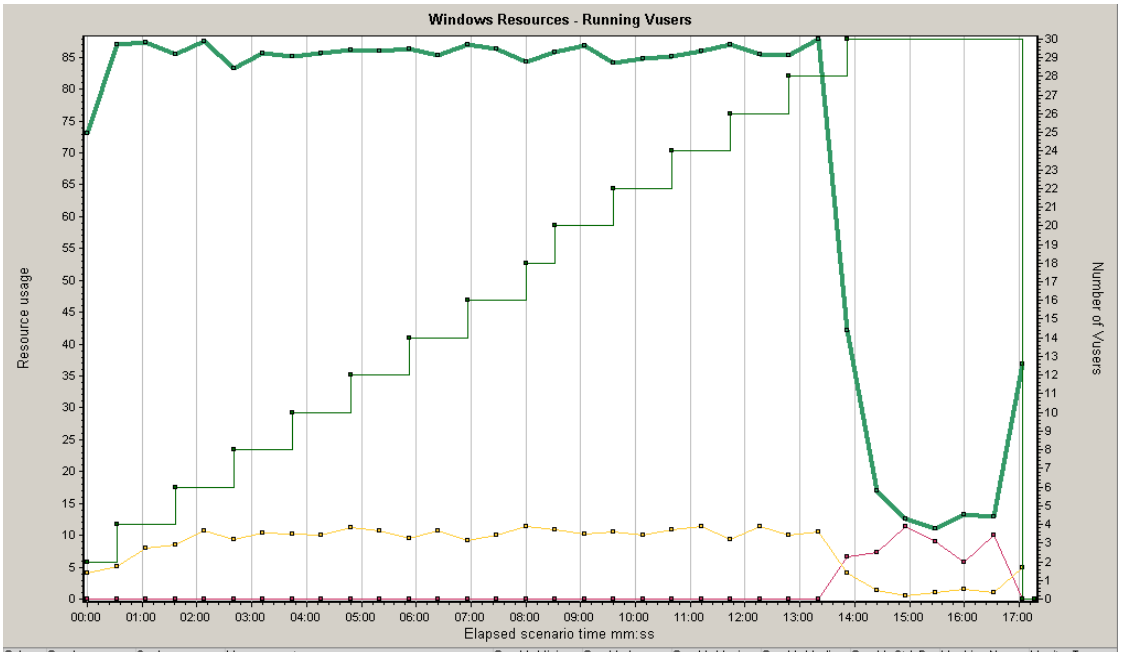
在 LoadRunner 脚本中对上送字段进行参数化，设置场景：每 1 分钟递增 2 个 VUser，最大 30 个 VUser。

从测试结果可以看出，整个场景运行期间 Transaction 没有报错，最高值甚至冲到了 1060 左右：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	CICS_Test:Pass	650.333	970.064	1065.125	1007.313	81.213
	Running Vusers	1	Run	0	14.118	30	14	9.615

结合系统资源图可以看出，当 VUser 达到 29 左右时，cicsas 的 CPU 利用率开始暴跌，Region CICSTE 已经停止服务了：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev	Machine Name	Monitor Type
	Windows Resour	1	% Processor Time (Process cicsas):192.168.1.11	10.985	71.666	87.832	85.469	27.073	192.168.1.11	
	Windows Resour	10	% Processor Time (Process cicsip#1):192.168.1.11	0	0.151	1.136	0	0.332	192.168.1.11	
	Windows Resour	1	% Processor Time (Process cicsip):192.168.1.11	0.568	8.132	11.364	9.991	3.577	192.168.1.11	
	Running Vusers	1	Run	0	14.118	30	14	9.615	N/A	N/A

CSTB 中看到 CICSTE 连接瞬时 TPS 为 622，和 Controller 中看到的总 TPS

基本吻合：

1） 对比测试：增加 cicsas

2） CICS Server 设置

确认 CICS Region 的部分参数：

- LD：TCPProcessCount=2，Region 会默认建立 2 个 cicsip 进程，用来并发监听多个 CTG 的请求。
- RD：MinServer=1，MaxServer=2，修改 Region 最大创建 2 个 cicsas 进程用来处理请求。ClassMaxTasks=1，1，1，1，1，1，1，1，1，10，ClassMaxTaskLim=0，0，0，0，0，0，0，0，0，5。将 Task 数量和排队数量改回基准测试时的配置。

此时查看系统进程可以看到 CICS Region 已经启动了 2 个 cicsip 进程和 2 个 cicsas 进程：

cicsld.exe	5824	6,136 K	6,252 K
cicslm.exe	5840	6,396 K	6,384 K
cicsam.exe	5584	6,192 K	6,148 K
cicsrl.exe	5852	6,524 K	6,960 K
cicsic.exe	4784	5,936 K	6,152 K
cicsas.exe	940	7,120 K	10,644 K
cicsas.exe	1360	6,912 K	8,716 K
cicsip.exe	3552	6,368 K	6,848 K
cicsip.exe	3920	6,356 K	6,592 K

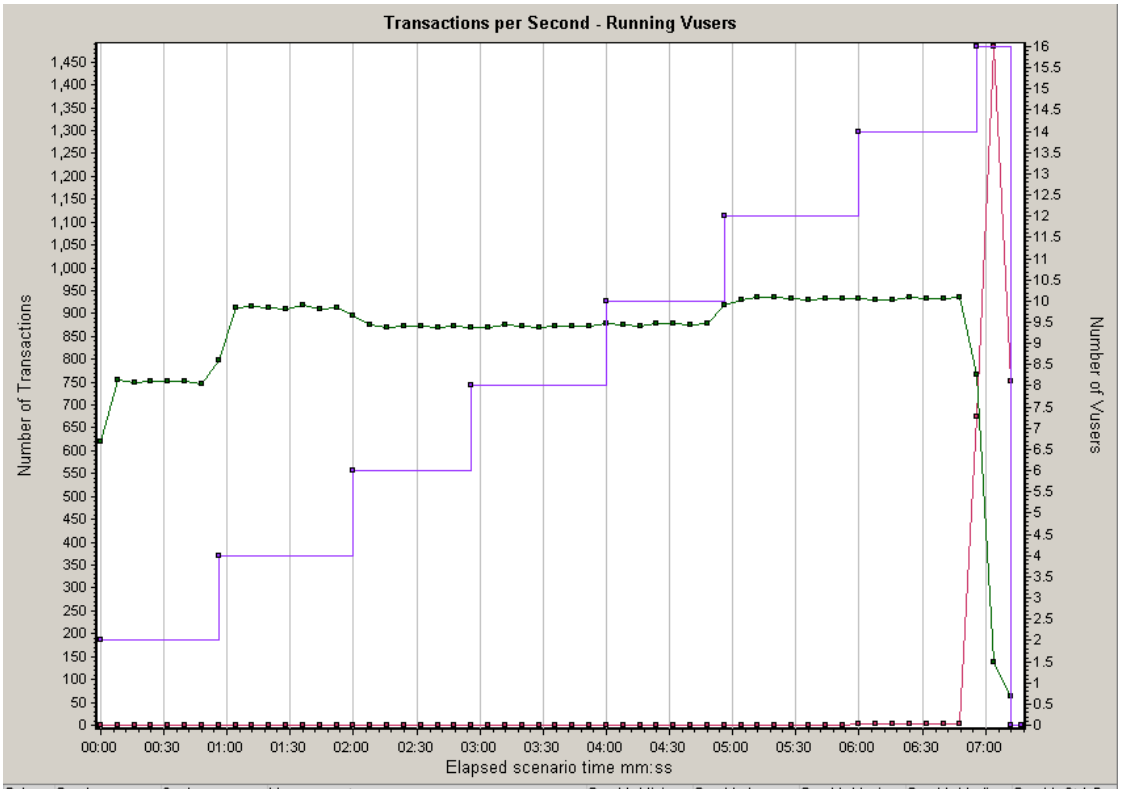
3） CTG 设置

CTG 保持不变。

4） 发压及 CICS 监控

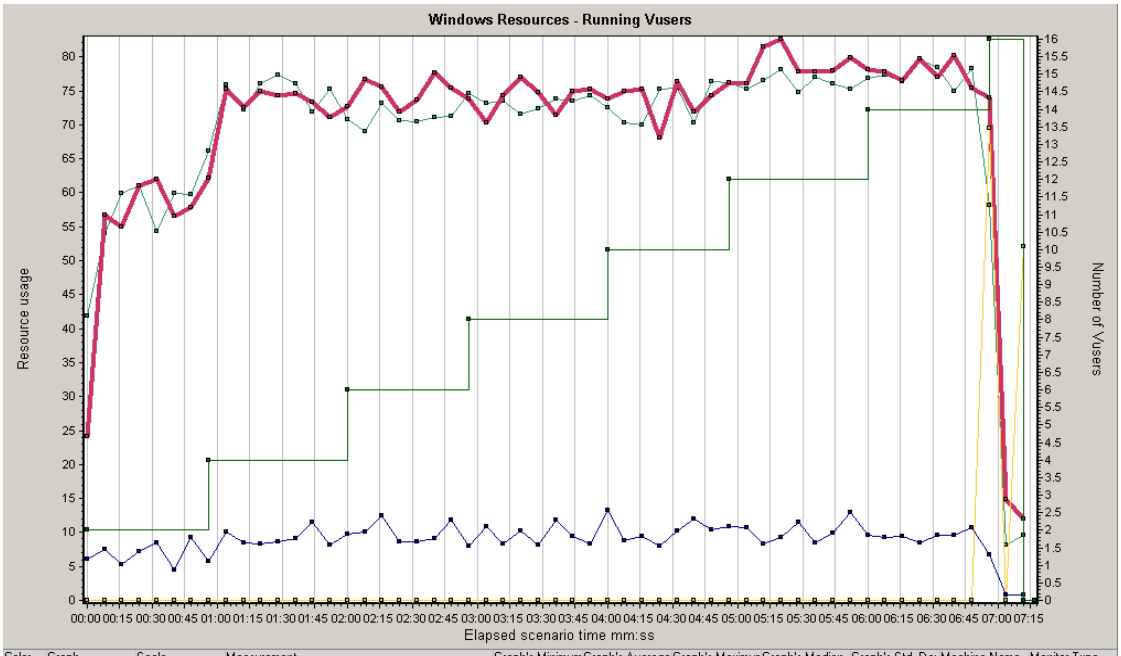
在 LoadRunner 脚本中对上送字段进行参数化，设置场景：每 1 分钟递增 2 个 VUser，最大 20 个 VUser。

从测试结果可以看出，由于 cicsas 进程增加了，因此总 TPS 较之前有明显增长，总 TPS 稳定在 900 笔/秒左右，由于 MaxTask 和 MaxTaskLim 调回基准测试时的数值，因此 VUser 数量达到 14 时 Transaction 开始大量报错：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	CICS_Test:Fail	0	53.281	1483.625	0	236.099
	Transactions per	1	CICS_Test:Pass	62.8	845.578	936.125	875.25	158.967
	Running Vusers	1	Run	0	7.2	16	8	5.455

结合系统资源图也可以看出，2 个 cicsas 进程 CPU 利用率都维持在 70~80%左右，当 VUser 达到 14 时，交易开始大量报错，2 个 cicsas 进程 CPU 利用率暴跌，Region CICSTE 停止服务了：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev	Machine Name	Monitor Type
	Windows Resour	1	% Processor Time (Process cicsas#1):192.168.1.11	8.073	69.126	79.514	73.437	13.748	192.168.1.11	
	Windows Resour	1	% Processor Time (Process cicsas):192.168.1.11	11.98	70.125	82.605	74.739	14.239	192.168.1.11	
	Windows Resour	100	% Processor Time (Process cicsip#1):192.168.1.11	0	0.022	0.695	0	0.115	192.168.1.11	
	Windows Resour	1	% Processor Time (Process cicsip):192.168.1.11	0.782	8.955	13.194	9.201	2.389	192.168.1.11	
	Running Vusers	1	Run	0	7.2	16	8	5.455	N/A	N/A

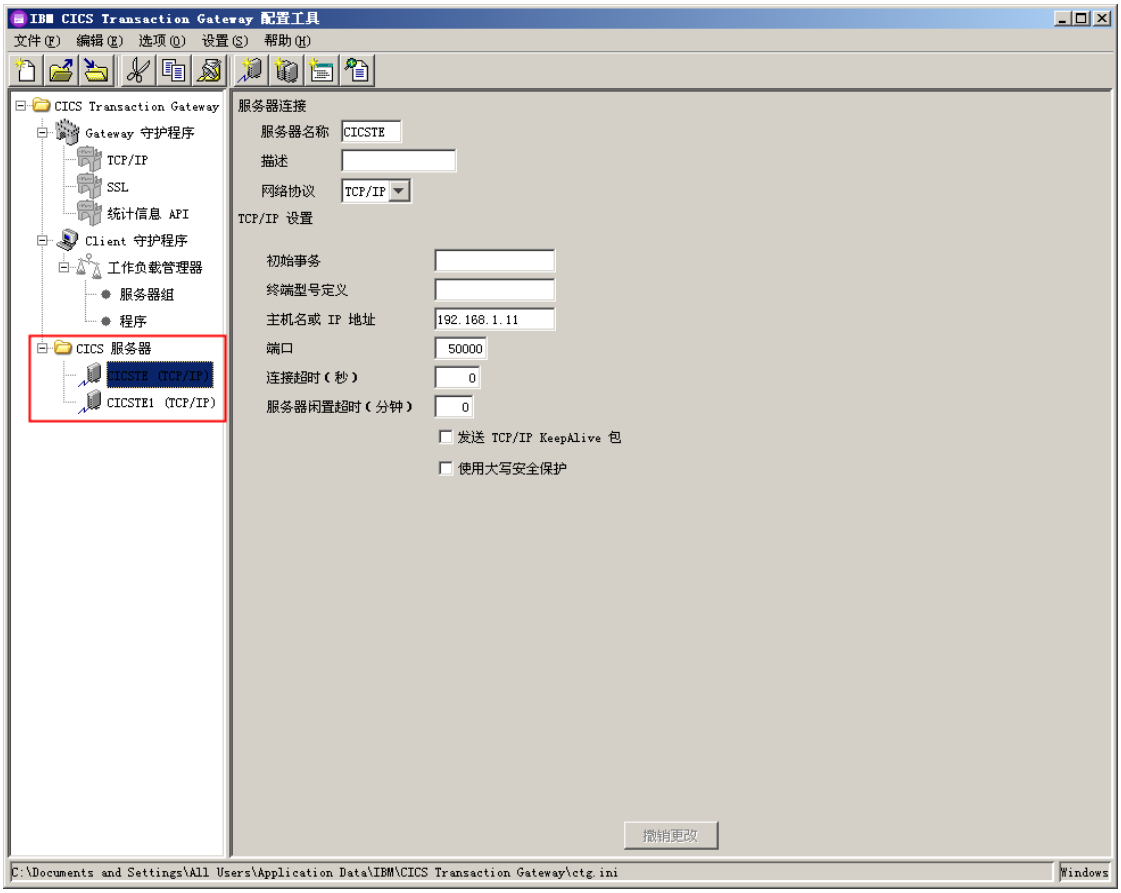
4. 对比测试：增加 CTG Region

1） CICS Server 设置

- LD：TCPProcessCount=2，Region 会默认建立 2 个 cicsip 进程，用来并发监听多个 CTG 的请求。
- RD：MinServer=1，MaxServer=2，Region 最大创建 2 个 cicsas 进程用来处理请求。ClassMaxTasks=1，1，1，1，1，1，1，1，1，10，ClassMaxTaskLim=0，0，0，0，0，0，0，0，0，5。

2） CTG 设置

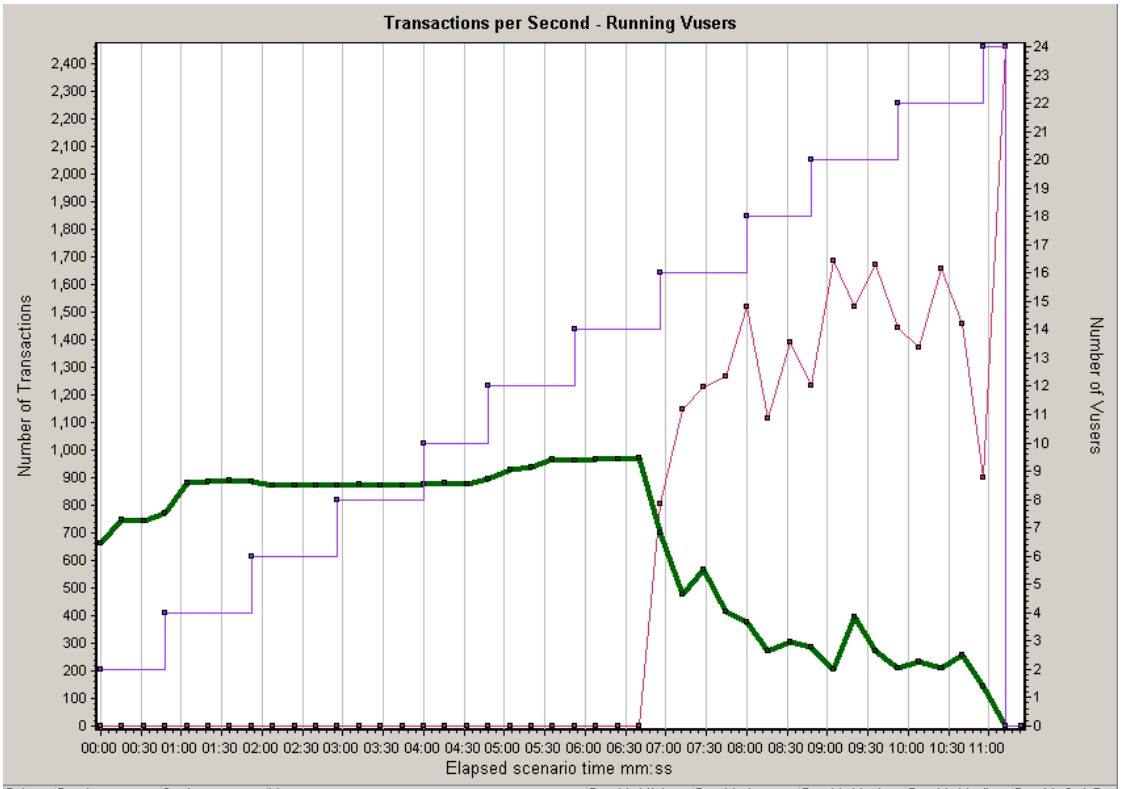
在 CTG Configuration Tool 设置中新建多个 Region（Region 名字可以与 CICS Server 端不一样，但 IP 和端口需一样），都指向 CICSTE：



3） 发压及 CICS 监控

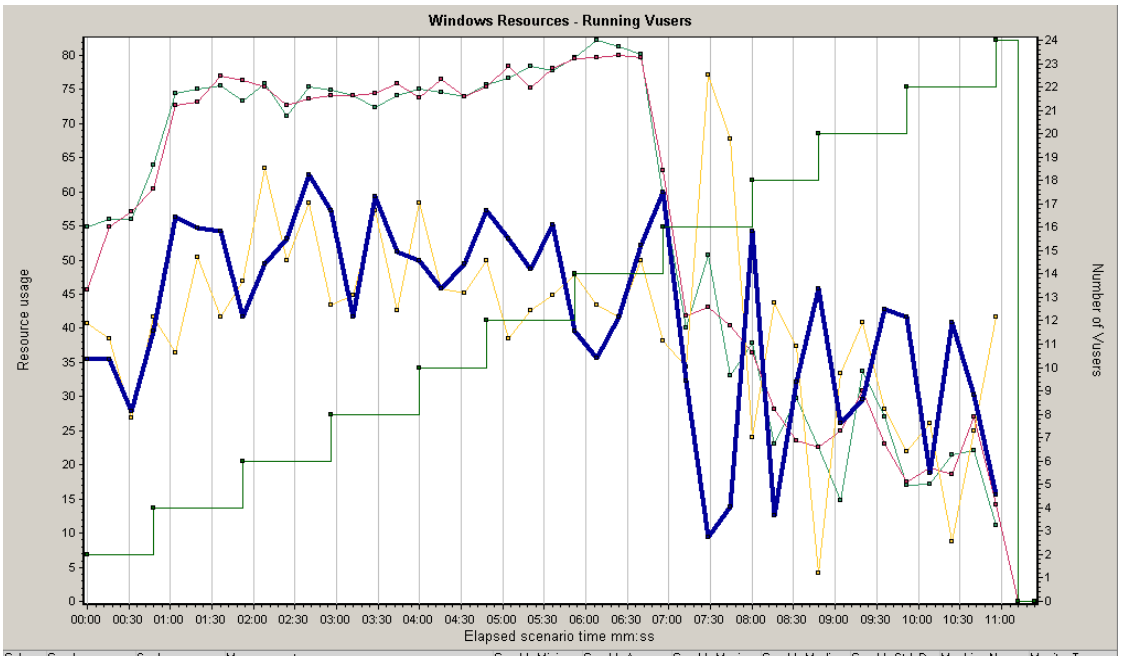
在 LoadRunner 脚本中对上送字段进行参数化，设置场景：每 1 分钟递增 2 个 VUser，最大 20 个 VUser。

从测试结果可以看出，和 2 个 cicsas 进程时一样，总 TPS 稳定在 900 笔/秒左右，由于 MaxTask 和 MaxTaskLim 保持基准测试时的数值，因此 VUser 数量达到 14 时 Transaction 开始大量报错，但 Region CICSTE 没有立即宕掉，仍坚持运行了一段时间：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev
	Transactions per	1	CICS_Test:Fail	0	554.779	2460.25	0	722.41
	Transactions per	1	CICS_Test:Pass	0	653.629	969.125	868.813	298.679
	Running Vusers	1	Run	0	11.143	24	12	7.845

结合系统资源图也可以看出，2 个 cicsip 进程 CPU 利用率都维持在 6~7%左右，Region CICSTE 服务持续运行了一段时间，没有立即宕掉：



Color	Graph	Scale	Measurement	Graph's Minimum	Graph's Average	Graph's Maximun	Graph's Median	Graph's Std. Dev	Machine Name	Monitor Type
	Windows Resou	1	% Processor Time (Process cicsas#1):192.168.1.11	11.111	56.246	82.205	72.292	23.629	192.168.1.11	
	Windows Resou	1	% Processor Time (Process cicsas):192.168.1.11	0	54.932	80	72.604	24.355	192.168.1.11	
	Windows Resou	10	% Processor Time (Process cicsip#1):192.168.1.11	0.417	4.15	7.709	4.254	1.375	192.168.1.11	
	Windows Resou	10	% Processor Time (Process cicsip):192.168.1.11	0.938	4.175	6.25	4.271	1.386	192.168.1.11	
	Running Vusers	1	Run	0	11.143	24	12	7.845	N/A	N/A

六、 总结

1. MaxServer

RD 中的 MaxServer 控制当前 Region 的 cicsas 进程数量，而 cicsas 起到一种负载均衡的作用，在一个 cicsas 进程处理能力达到极限的情况下，启动多个 cicsas 可以明显提升交易 TPS。

由于 cicsas 进程会占用一定内存空间，因此不建议设置过高的 MaxServer 数量（生产环境中一般不高于 50）。

2. ClassMaxTasks 和 ClassMaxTaskLim

RD 中的 ClassMaxTasks 和 ClassMaxTaskLim 控制当前 Region TD 中指定 TClass 事务的并发处理能力，对性能测试直接的影响就是 VUser 并发数量，如果并发数量大于 ClassMaxTasks+ClassMaxTaskLim−1 时交易会报错。

3. cicsip

LD 中的 TCPProcessCount 控制当前 Region 的 cicsip 进程数量，cicsip 负责监听当前 Region 的 TCP 请求，并将请求转发给 cicsas 进程处理，cicsip 进程对交易 TPS 影响不大，从上面的测试结果看只是增加了稳定性，TCPProcessCount 不需要设置很大（生产环境中一般设置为 2），IBM 推荐平均 500 个 Client 对 1 个 cicsip。

4. MaxRegionPool

MaxRegionPool 是 CICS 保持自己私有信息的内存大小，取决于当前环境中同时允许这多少个 Region，生产环境一般同时运行不到 10 个 Region，MaxRegionPool 设置为 50M 左右。

5. MaxTaskPrivatePool

MaxTaskPrivatePool 是 task 私有内存大小，对并发性能有一定影响，生产环境一般设置为 4M 左右。

6. MaxTSHPool

MaxTSHPool 是多 task 共享内存大小，包括：

- COMMAREAs
- Common Work Area（CWA）
- Terminal Input/Output Areas（TIA/TOA）
- TCT User Areas（TCTUAs）
- Main temporary storage
- Maps and tables
- Conversion templates
- Temporary storage for unprotected STARTs

本参数对并发性能影响不大，主要影响 CICS Region 共享内存、程序 ECI Call、Link 时允许传递的大小，生产环境一般设置为 20M 左右。

电信增值业务指标数据核对测试

叶子

摘要

本文主要描述电信行业某业务指标的数据验证；以中国移动的手机视频业务为例。

关键词： 本文关键词 【指标】

业务需求描述

根据集团提供的口径、底层接口数据、产品经理提供的核对数据，测试当前业务的指标数据是否正确，以手机视频客户端活跃用户 指标为例

测试步骤

个人测试此类业务指标数据时的步骤

- 获取该指标的口径
- 根据口径统计底层的数据
- 测试开发人员的统计口径算出来的数据是否与 2 一致
- 测试结果反馈

步骤一 获取口径

从维护人员那里获取集团提供的业务口径

手机视频客户端活跃用户口径：

统计周期内，获取【手机视频用户使用明细表】，过滤条件『“帐单类型”=’0’ AND “终端类型 ID”= ’T0000002’』，计算“手机号码”剔重汇总

注释：

“帐单类型”为’0’表示’使用帐单’；

“终端类型 ID”为’T0000002’表示客户端

步骤二 统计底层数据

```
select
  count(distinct t.number)
  from tb_video_use_list t
 where t.statis_date  ='20131201'
   and t.account_type = '0'
   and t.dm_type_id = 'T0000002'
```

	COUNT(DISTINCTT.SERV_NUMBER)
1	22000

根据步骤一的口径去统计底层的数据：

如上图：底层接口 12 月 1 日的活跃用户数是 22000

备注：底层数据 就是 集团/业务平台下发最原始的接口数据

步骤三 数据核对

查询开发人员汇总的客户端活跃用户数据，根据用户标识为 1 的查找

```
select
 count( distinct t.number)
  from tb_video_user_info_day t
 where t.statis_date ='20131201'
 and t.client_user_flag='1'
```

	COUNT(DISTINCTT.SERV_NUMBER)
1	21994

如上图：显示数据比底层的少了几个用户

指标结果分析

查找不存在的几个用户号码，然后查询本地用户画像信息；

1、把步骤二和步骤三 的用户号码剔重后存入临时表，然后找出差异的这几号码，因为数据量比较大，所以借用临时表汇总数据，分担数据库压力

```
create    table wxy_video_tmp as
select t. number，(case when t. number=t1.number then ‘1’ else ‘0’ end ) client_use_flag from
wxy_video1 t，   wxy_video2 t1
where t.use_number=t1.use_number
```

2、然后根据 2 张临时表关联结果，查找标识为 0 的号码，即为差异号码

```
Select * from wxy_video_tmp a where a.client_use_flag='0';
```

3、把差异号码，关联本地用户基本信息表

```
select /*+parallel(t，4)+*/   *   from
  tb_cust_info_day t
  where t.date_time='20131201'
and t. number in
('number1'， number2'， number3'， number4'， number5'， number6')
```

说明：执行结果没有当前 6 个号码的信息，因本地用户基本信息都是正常状态的数据，这 6 个号码不存在，也就是停机状态或离网状态。故本次的 客户端活跃用户数据是正常的；一般核对的数据关联本地客户基本信息后与集团提供的核对数据相差 1%均为正常。**因用户号码涉及个人隐私，故不做截图说明**

步骤四 测试结果

简单的测试报告输出：

编号	标题	功能描述	测试步骤	测试时间	验证结果
1	指标数据核对	1. 核对本地开发的活跃用户指标 数据正常	1、登录 plsql，执行相应的存储过程 20131201 的数据 2、查看对应表中活跃用户指标的结果为 21994 3、集团下发的数据为 22000，对比相差 6 个用户	2014-1-10	通过

测试反馈：本次的手机视频业务客户端活跃用户指标数据属于正常情况。

备注：本文仅个人工作验证步骤，如 有不对的地方请大家指正，谢谢！

用 Visual Studio 2010 测试 WEB 性能测试及其统计学的初步研究

作者：吴晨昊

摘要：VS2010 的性能测试功能非常强大，有许多可以值得探讨的内容，大体可以分 2 部分，第一部分是建立 Performance testing；第二部分是根据 performance testing 来建立 Load testing；由于在 Load testing 之后，我们可以对其进行具有统计学意义的分析，我将其归为第三部分。 因此这篇文章将从三个部分来和大家一起探讨和研究性能测试统计学意义的分析，并得出合理有效的解释。

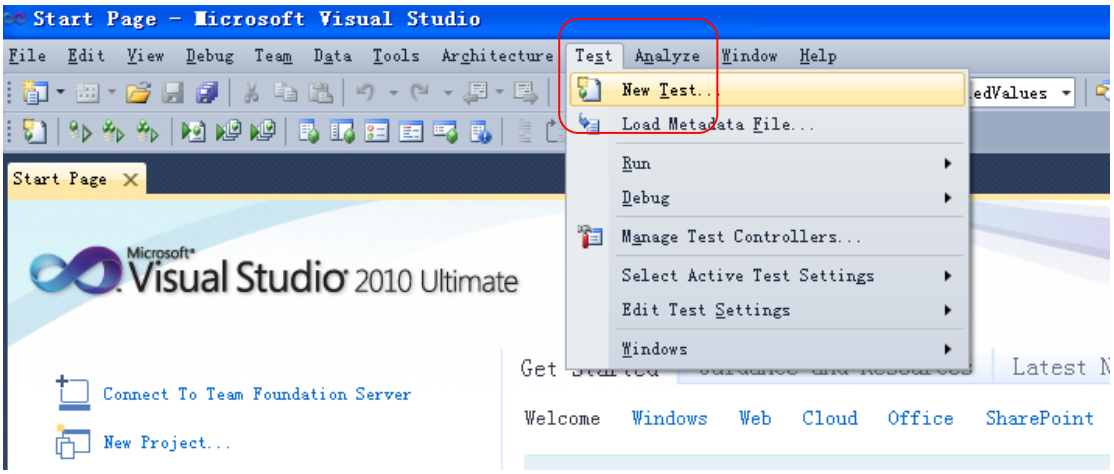
关键词： VS2010， performance test， load test， 统计

正文：

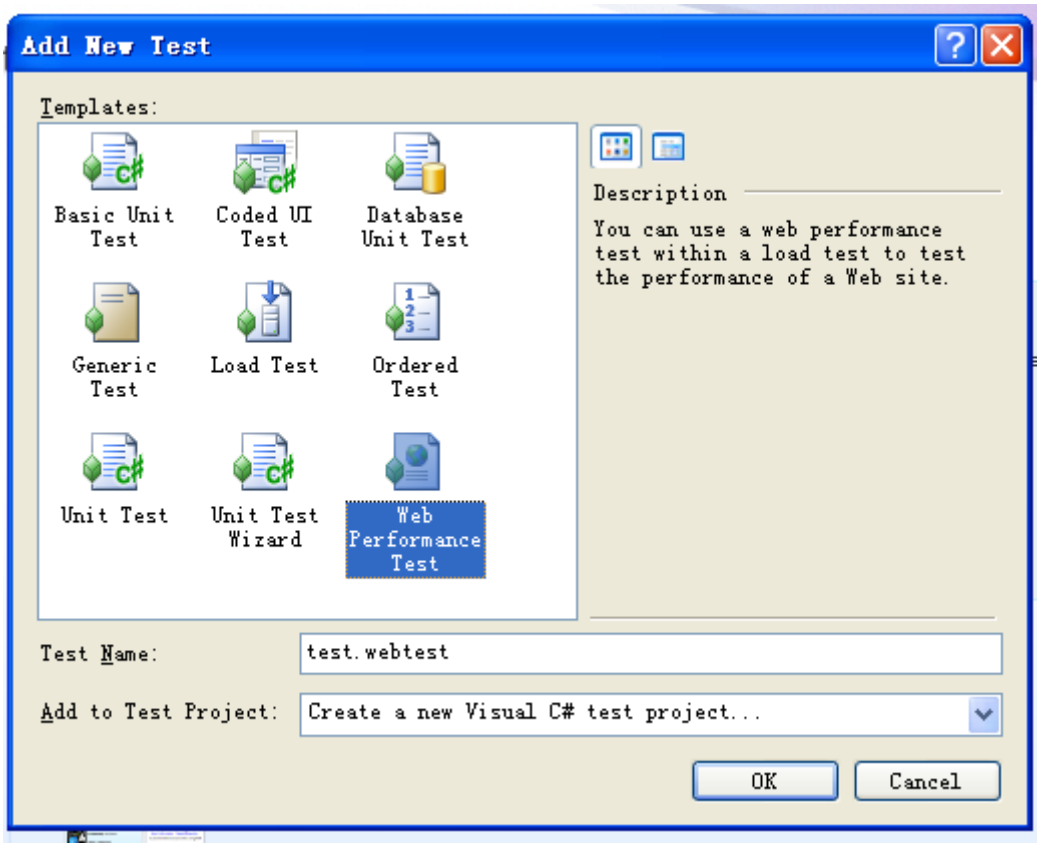
第一部分：Performance testing 的学习

1： 点击 “开始”-> 点击 “所有程序 ”-> Microsoft Visual Studio 2010 -> 点击 “Microsoft Visual Studio 2010” 就可以打开 Microsoft Visual Studio 2010 的软件。

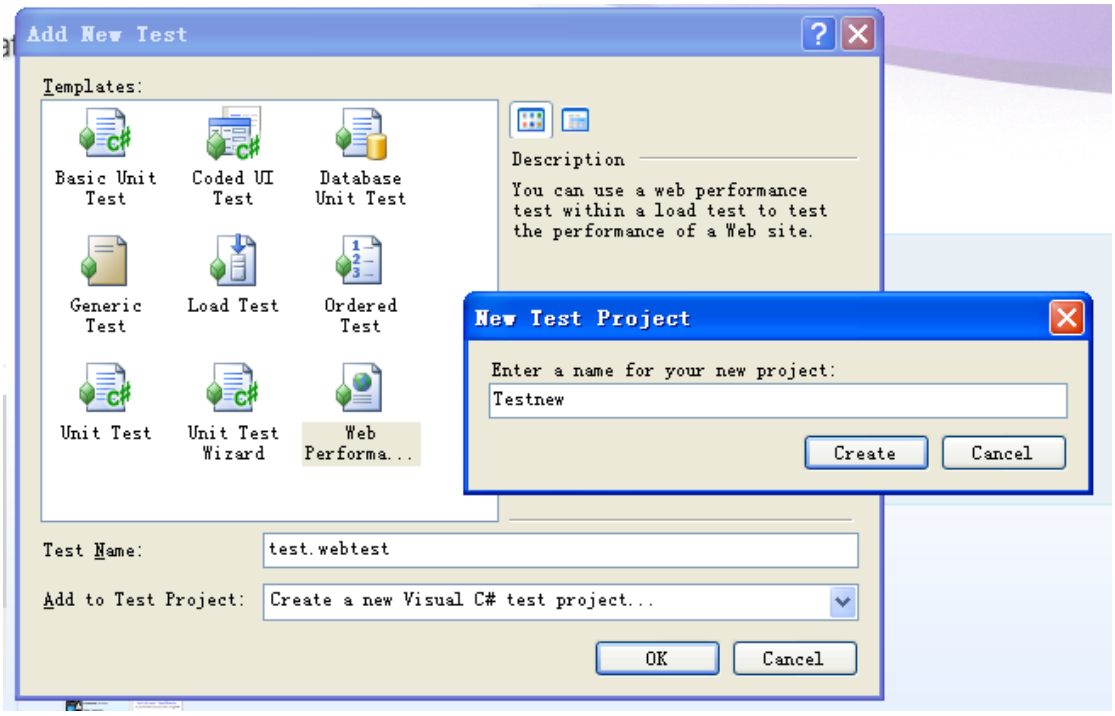
2：在打开的 VS 2010 主界面上， 点击导航条上的“Test -> New Test”， 这样就能打开“Add New Test”的对话框。



3：在“Add New Test”的对话框中，选择“Web Performance Test”， 更改 Test Name 为 XXX. webtest ，选择 Add to Test Project 为 “Create a new Visual C# test project…”，然后点击“OK”按钮。

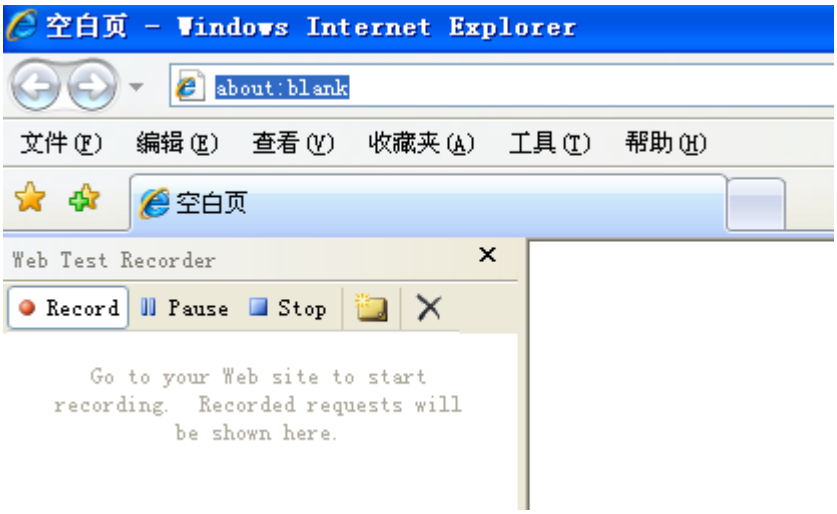


4：在弹出的“New Test Project”的对话框中，输入新的 project 的名称，最后点击“Create”按钮。

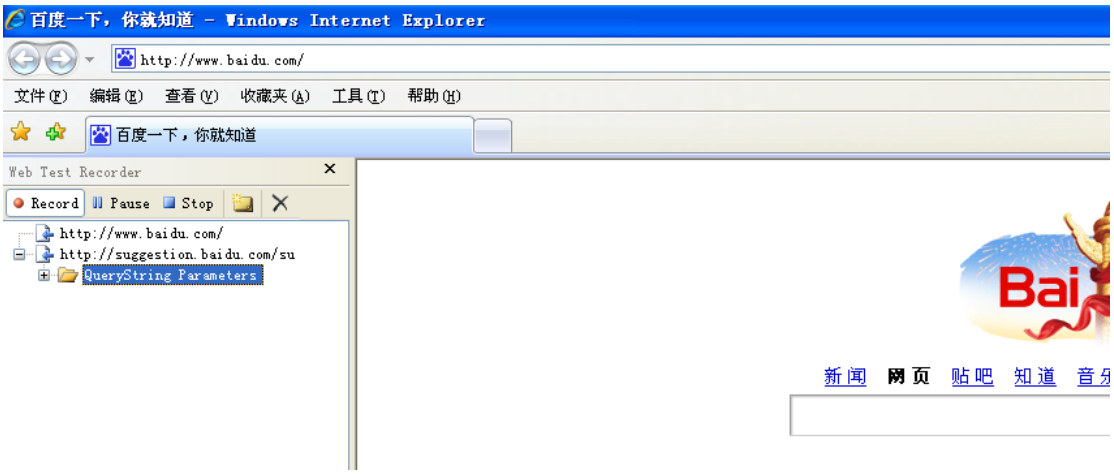


5：系统自动弹出一个 IE 浏览器，左边一栏，显示了“Record， Pause， Stop”

等按钮。

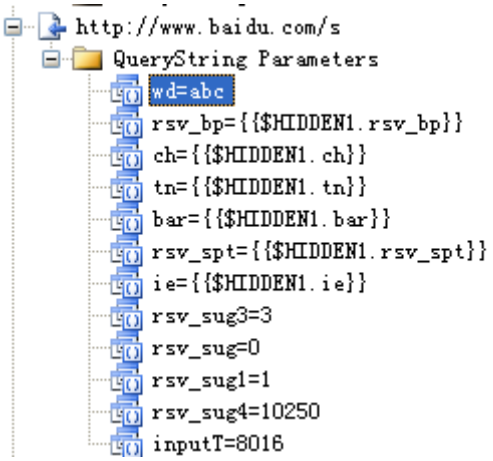


6：在 URL 的地址栏中，输入 www. baidu. com， 然后点击“回车”按钮。这样就弹出了一个百度的页面，同时我们可以看到左侧的信息，那就是 Record 所记载的信息。

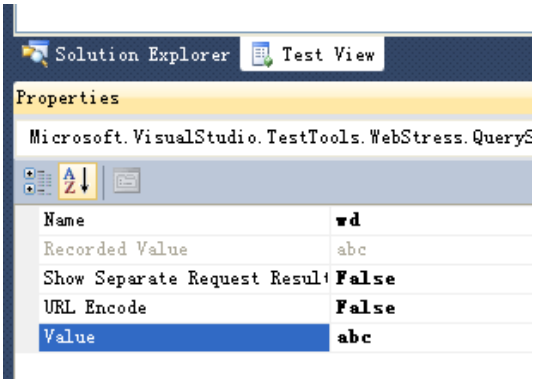


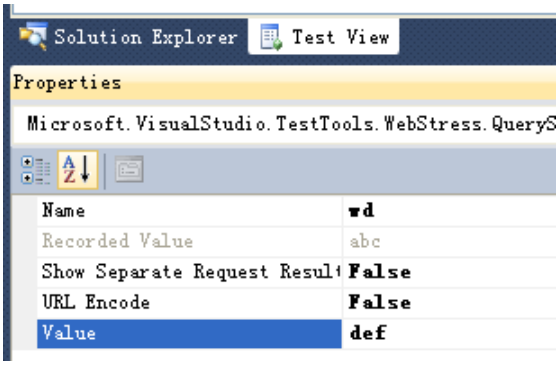
7： 在百度主页的输入框中，输入“abc”，然后点击按钮“百度一下”。点击左上方“Stop”按钮。IE 浏览器就会自动关闭，然后我们回到 VS 2010。

8： 在 VS 2010 中的 http：//www. baidu. com/s 下的 QueryString Parameters 下面的 wd=abc， 就代表我们输入的参数：abc 。

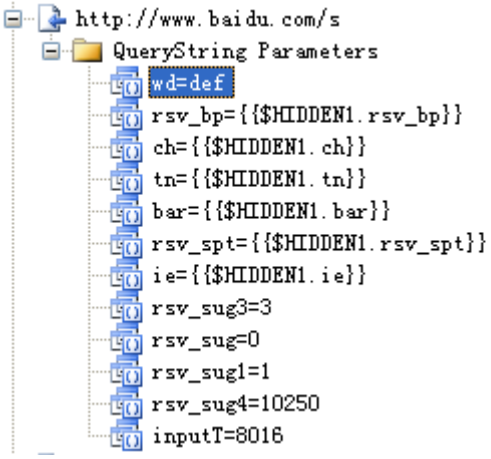


9：点击 wd=abc，同时，我们查看 VS 2010 右边 properties 中，有 Value 是 abc，在这里，我们就能够改变在百度主页输入的值了。 比如改为“def”。





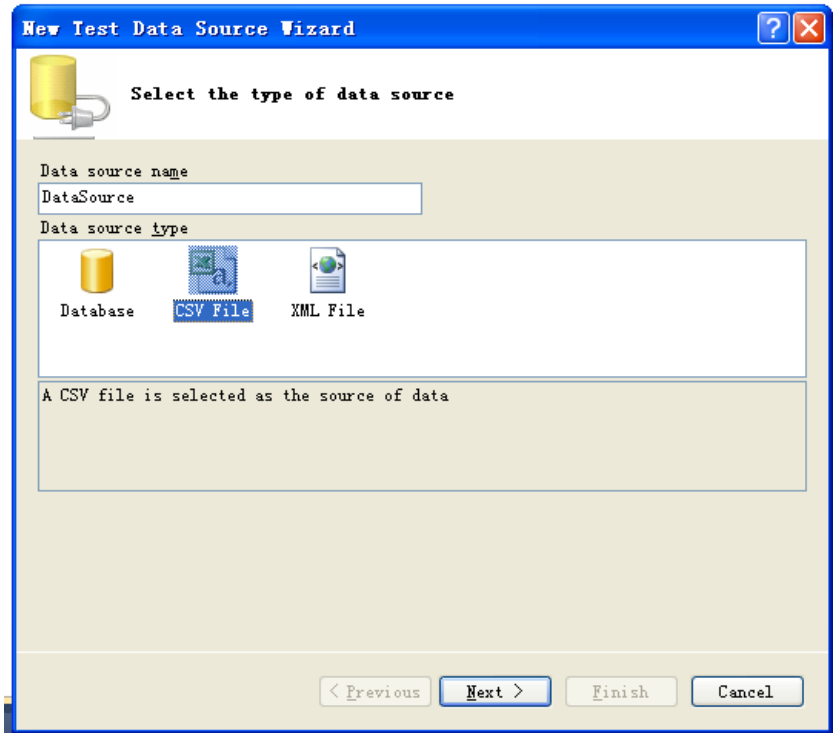
左侧的 wd 也变成了 def：



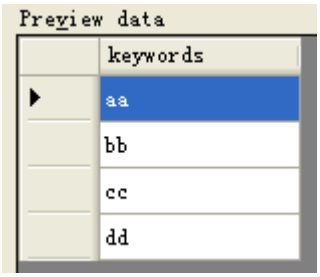
10：保存后，点击左上方的“Run Test”按钮，就可以进行运行，和查看性能情况。

11：同时我们也可以新建一个后缀名为“. CSV” 的 Excel 文件，里面的内容是所有你输入百度的关键词。比如关键词为：aa， bb， cc， dd， 保存的时候选择 CSV 文件。

12：在 VS 2010 的 webtest 界面的左上方，点击 “Add Data Source”  按钮。这时候，就可以弹出 “New Test Data Source Wizard”对话框。

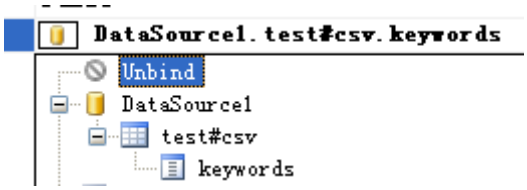


输入 Data source name，选择 CSV File，点击“Next”按钮。然后选择一个 CSV 的文件。这时候，我们可以 preview 这个 CSV 文件中的数据。

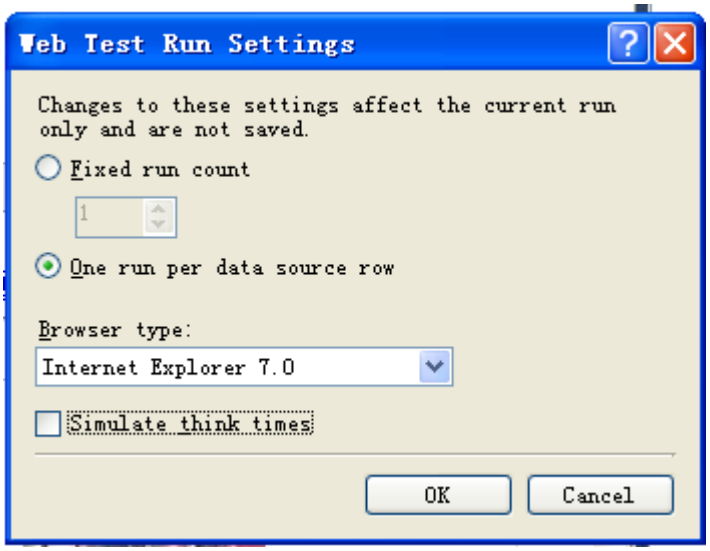


最后点击“Finish”按钮以完成创建数据源的操作。

13：这时候回到第 9 步，查看 wd 的属性，点击 value 右边下拉图标，点击所创建的 DataSource，展开每一层，直至展现参数的列名：keywords。点击 keywords，value 的值就设好了。最后进行保存。这样 wd 的值就应该是 keywords 这个属性下的 aa，bb，cc 和 dd。

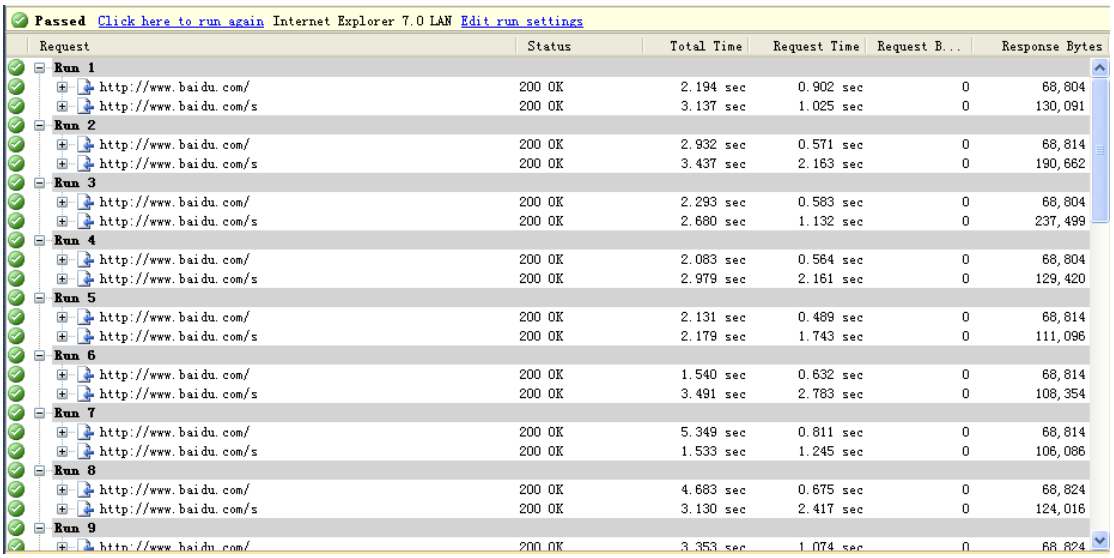


14：点击“Run Test”按钮，就会自动运行一次，我们看到有 passed 的信息，在它的右边，有 Edit run settings 的链接，点击这个链接，弹出“Web Test Run Settings”的对话框。选择 One run per data source row 的单选框， Browser type 可以任意选择， 最后点击“OK”按钮，关闭此对话框。



然后，点击 Passed 右边的“Click here to run again”的链接。这样就可以循环运行 CSV 文件中的各个参数了。

当运行都完成之后，我们可以看到运行情况。



Passed Click here to run again Internet Explorer 7.0 LAN Edit run settings

Request	Status	Total Time	Request Time	Request B...	Response Bytes
Run 1					
http://www.baidu.com/	200 OK	2.194 sec	0.902 sec	0	68,804
http://www.baidu.com/s	200 OK	3.137 sec	1.025 sec	0	130,091
Run 2					
http://www.baidu.com/	200 OK	2.932 sec	0.571 sec	0	68,814
http://www.baidu.com/s	200 OK	3.437 sec	2.163 sec	0	190,662
Run 3					
http://www.baidu.com/	200 OK	2.293 sec	0.583 sec	0	68,804
http://www.baidu.com/s	200 OK	2.680 sec	1.132 sec	0	237,499
Run 4					
http://www.baidu.com/	200 OK	2.083 sec	0.584 sec	0	68,804
http://www.baidu.com/s	200 OK	2.979 sec	2.161 sec	0	129,420
Run 5					
http://www.baidu.com/	200 OK	2.131 sec	0.489 sec	0	68,814
http://www.baidu.com/s	200 OK	2.179 sec	1.743 sec	0	111,096
Run 6					
http://www.baidu.com/	200 OK	1.540 sec	0.632 sec	0	68,814
http://www.baidu.com/s	200 OK	3.491 sec	2.783 sec	0	108,354
Run 7					
http://www.baidu.com/	200 OK	5.349 sec	0.811 sec	0	68,814
http://www.baidu.com/s	200 OK	1.533 sec	1.245 sec	0	106,086
Run 8					
http://www.baidu.com/	200 OK	4.683 sec	0.675 sec	0	68,824
http://www.baidu.com/s	200 OK	3.130 sec	2.417 sec	0	124,016
Run 9					
http://www.baidu.com/	200 OK	3.353 sec	1.074 sec	0	68,824

右击选择 copy all， 就可以将这些数据黏贴到 Excel 表中进行分析。

第二部分：Load test 的学习

1：前期网站架构搭建和数据库搭建

以上是如何进行 performance testing 的方法，performance test 就如同设置一个场景，若要分析此场景的运行情况究竟如何，它在这个系统中的响应情况是否令人满意，就要使用 Load test，同时，还必须使用到 SQL server，我这里使用了 SQL server 2008，它有 2 个作用：

第一是网站的 table 及其数据都可以动态的存储到 SQL server 2008 中，进行增，删改查等等的操作；

第二就是有关各方面性能的数据会在 Load 测试的过程中存入名为 LoadTest2010 的数据库，这个是用来进行 Loadtest 分析的数据库，而这个数据库并不需要我们建立，在目录 C：\Program Files\Microsoft Visual Studio 10.0\Common7\IDE 下，有一些 sql 文件： loadtestresultsrepository 就是 Sqlserver Query file，用于创建 LoadTest2010 数据库。打开这个文件，我们就可以看到里面有“CREATE DATABASE［LoadTest2010］”这样的语句，在 SQL server 2008 中进行运行这个 SQL 语句之后，就可以创建数据库 LoadTest2010 了。

下面我来演示一下我自己的 LoadTest 的使用：

首先，配置一个 IIS+ ASP 的网站，数据库是 SQLSERVER 2008，其中主要的 table 是用户表（users）和发表的文章内容（content）表。 架构是： IIS-ASP-SQLServer2008， 测试是在 localhost 中进行测试。

这个网站的代码，是参考了《ASP+SQL server 网络应用系统开发与实例》（人民邮电出版社）这本书。当然你可以按照自己的喜好自己编写和开发网站。

这个网站就是其中的 Discuss 论坛，可以注册用户，登陆系统，退出系统，

发表文章，浏览别人的文章等等。

其次，创建 LoadTest2010 数据库。

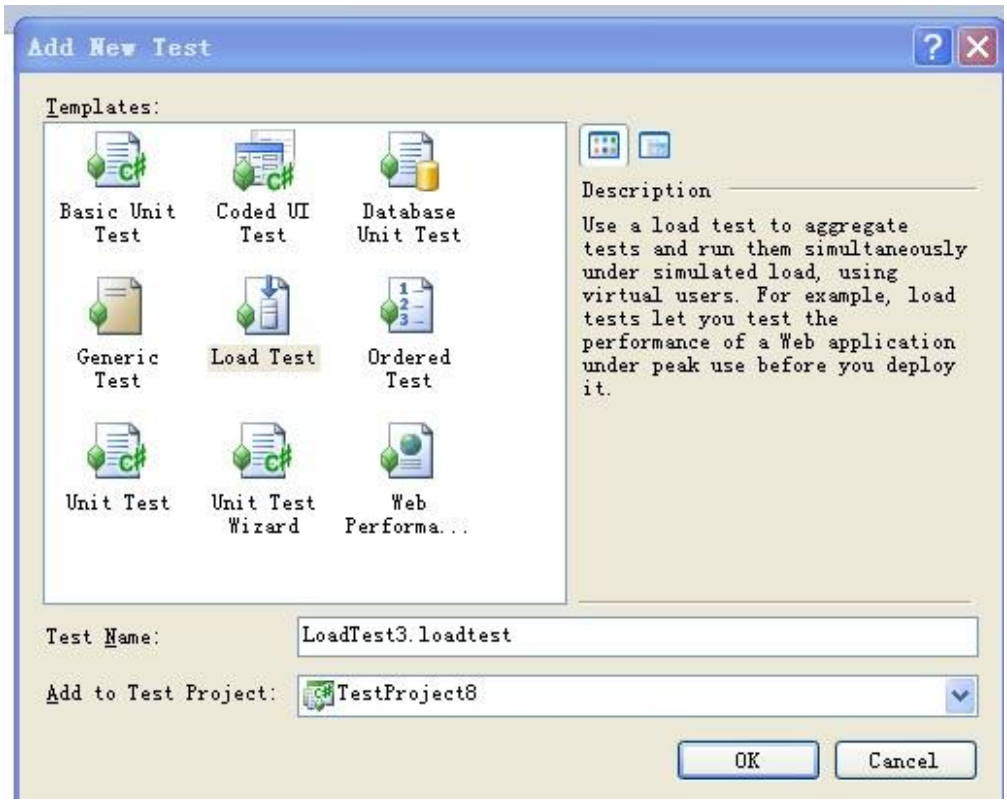
2：Load test 的创建和运行

以上准备工作完成之后就可以录制 Web performance test 了。录制的方法就和前面部分类似，只是我录制 2 次，分别模仿“修改密码”操作，和“浏览” 操作。

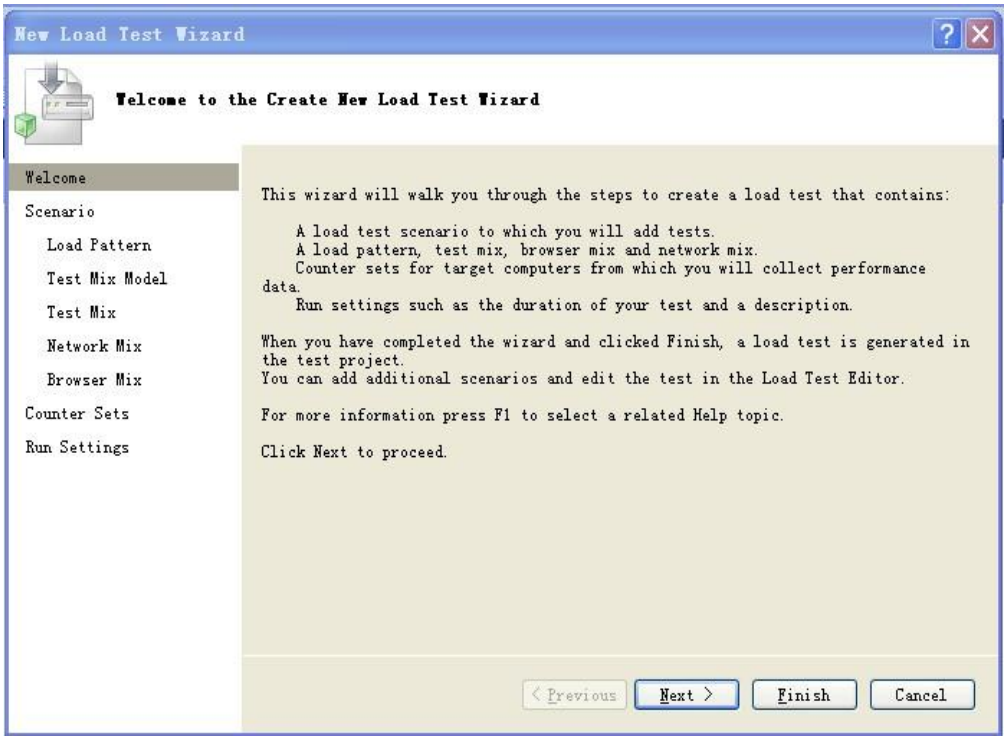
在 VS2010 右边的 solution explorer 将 2 个 Web performance test 的文件分别命名为：browse. webtest（代表浏览操作）和 changepw. webtest（代表修改密码操作）。

接下来就是正式建立 Load Test 的过程。

1：点击导航条上的“Test -> New Test”， 选择“Load Test”，点击“OK”.

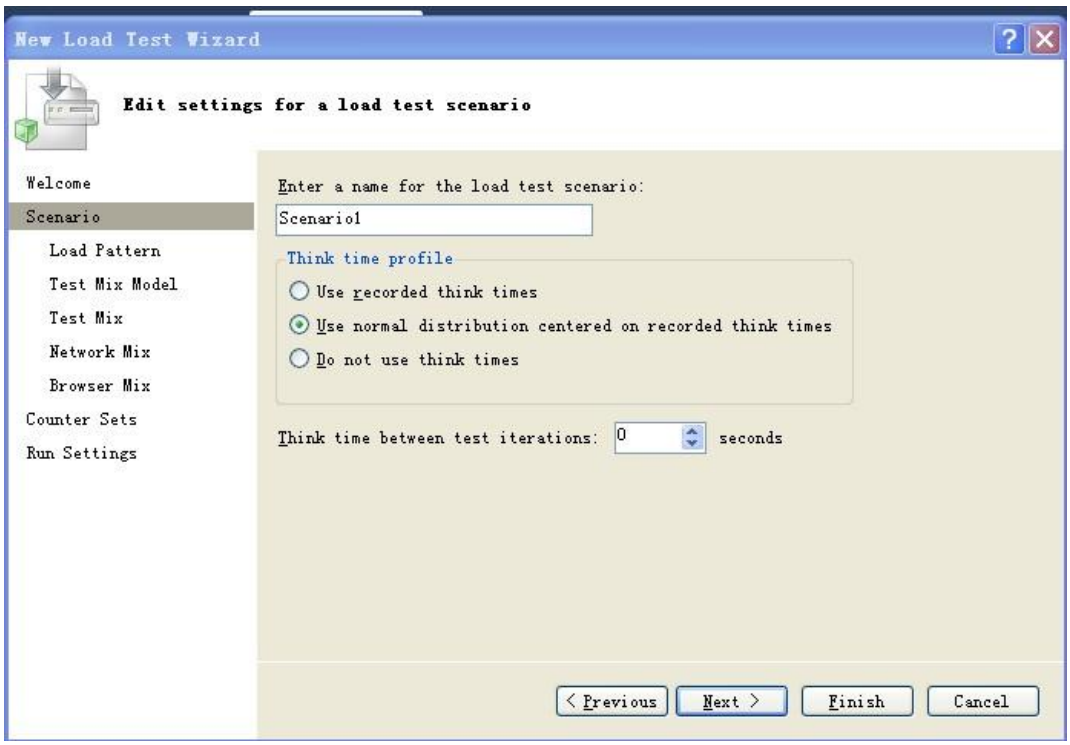


2： 我们可以看到，出现 New Load Test Wizard 的对话框，这是创建 Load Test 的向导。点击“Next”按钮进入下一页。

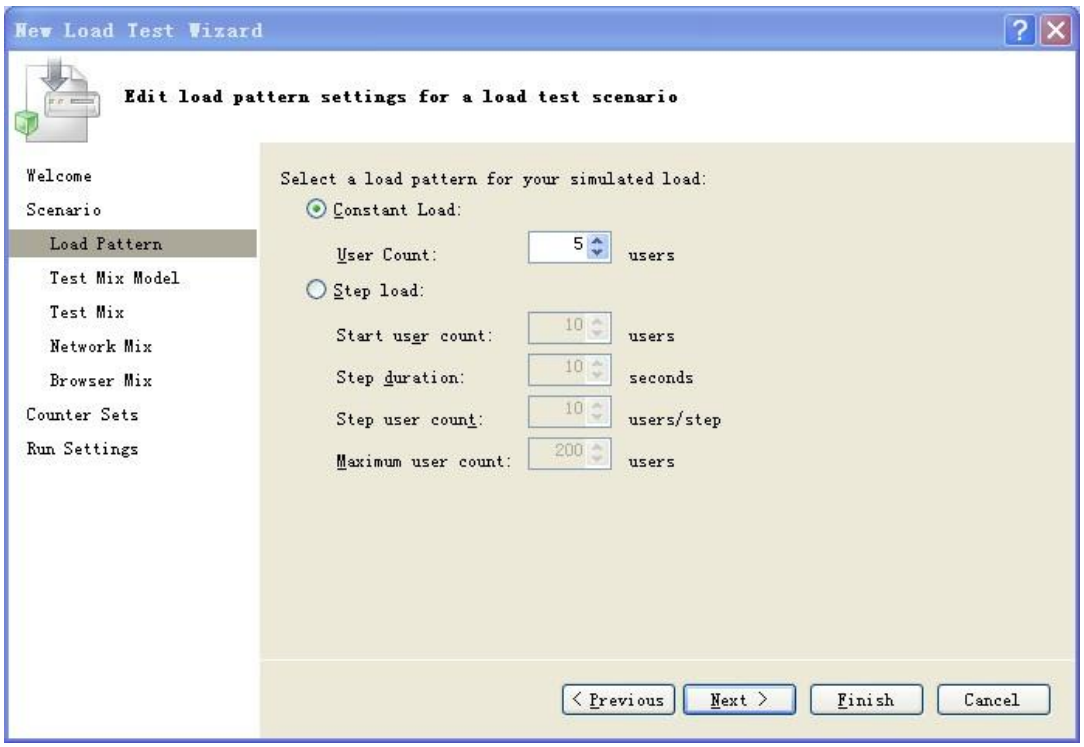


3：输入场景名称，是否需要 Think time，每一轮间隔的 Think time 是多少，

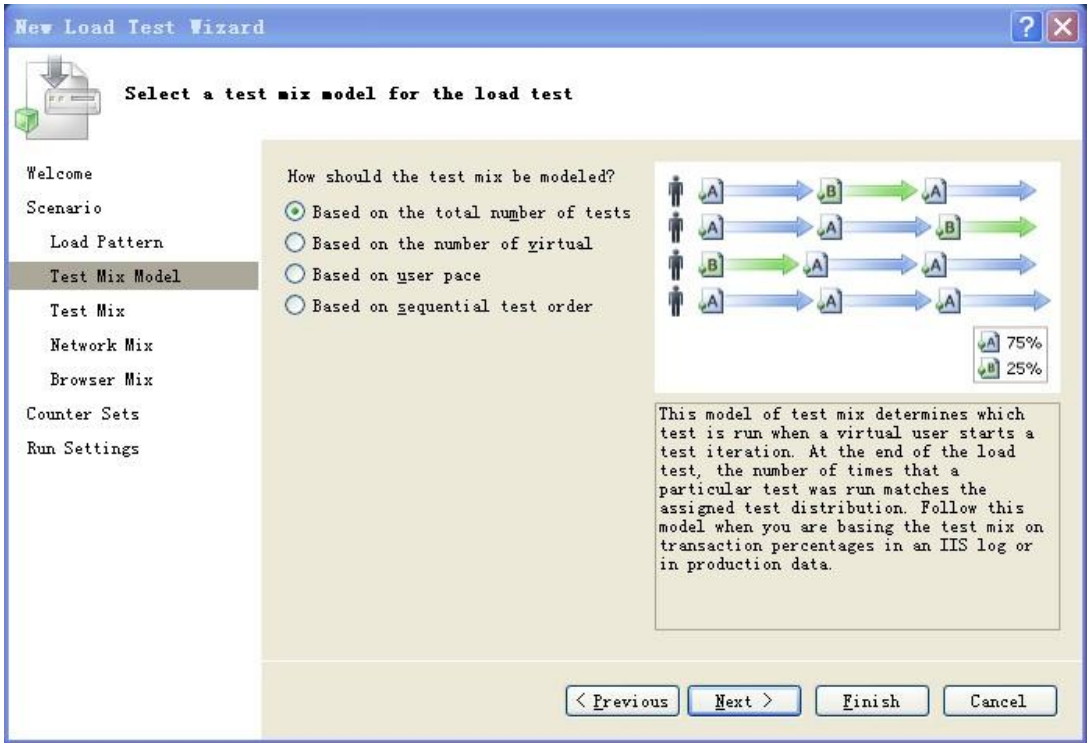
点击“Next”。



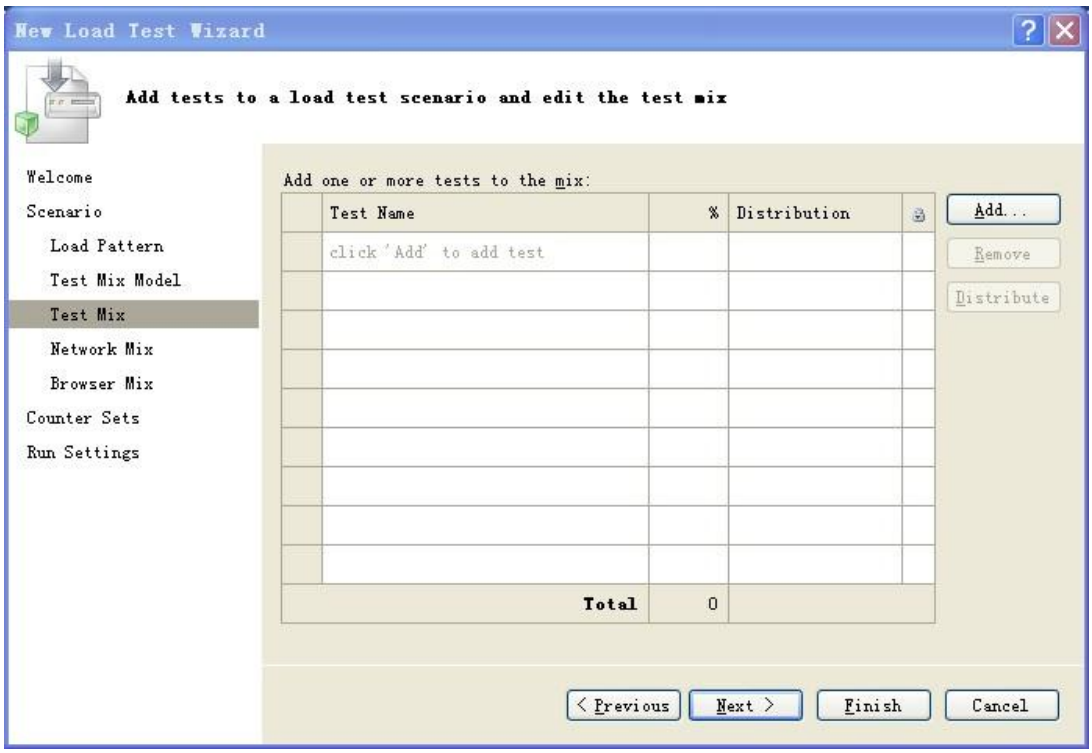
4： 输入保持不变的用户数量，或者用户逐级增加的用户数量设置。

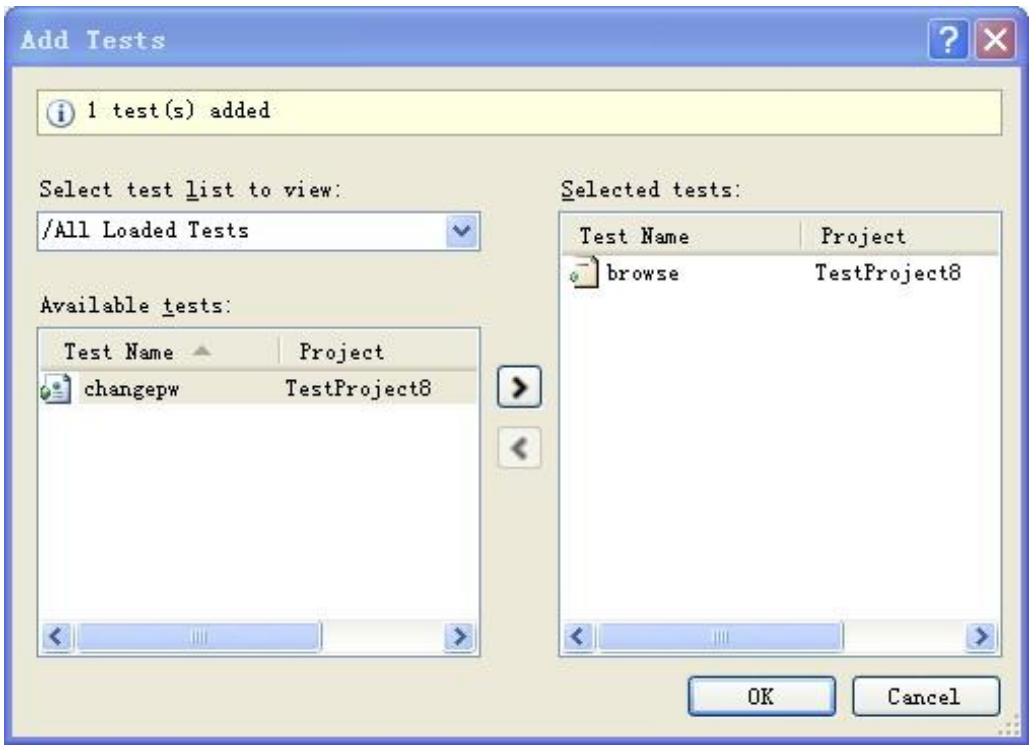


5：选择一个合适的测试模型：

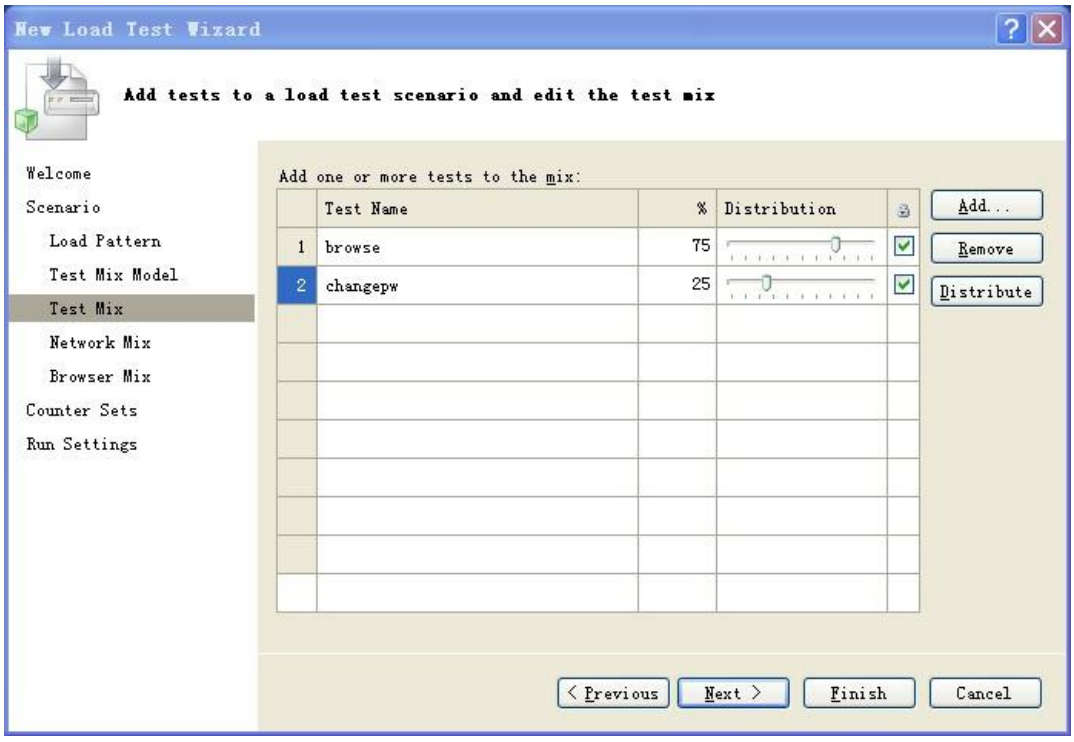


6：点击“Add”按钮，选择想要进行 Load Test 的 Performance Test

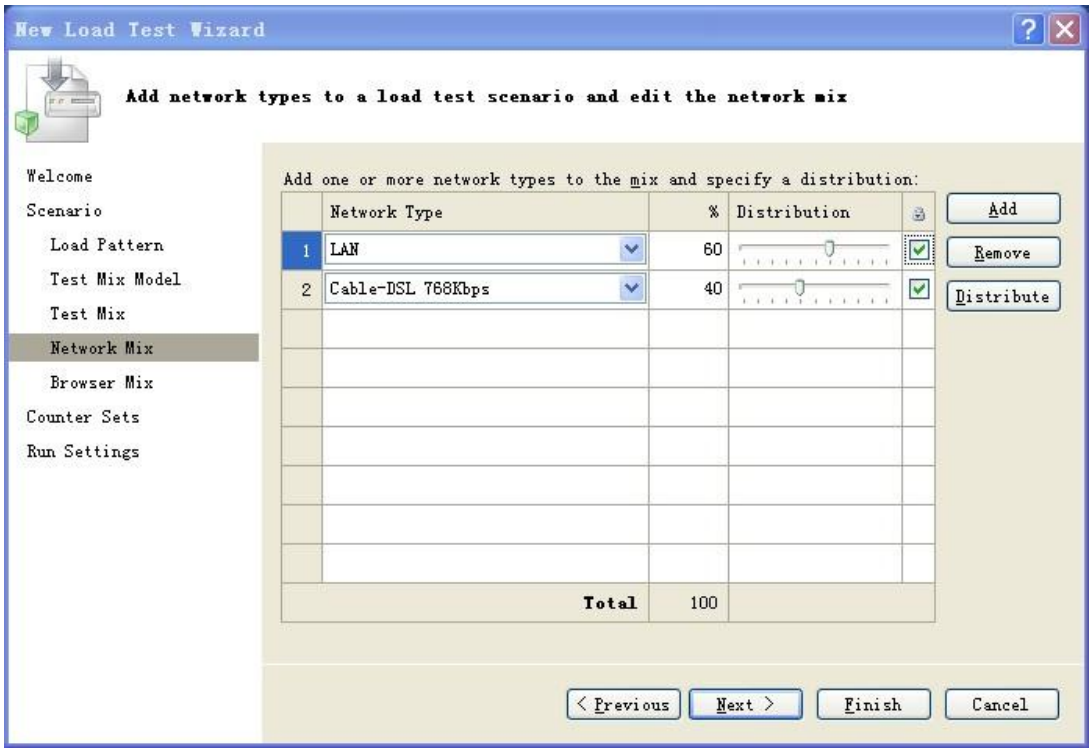




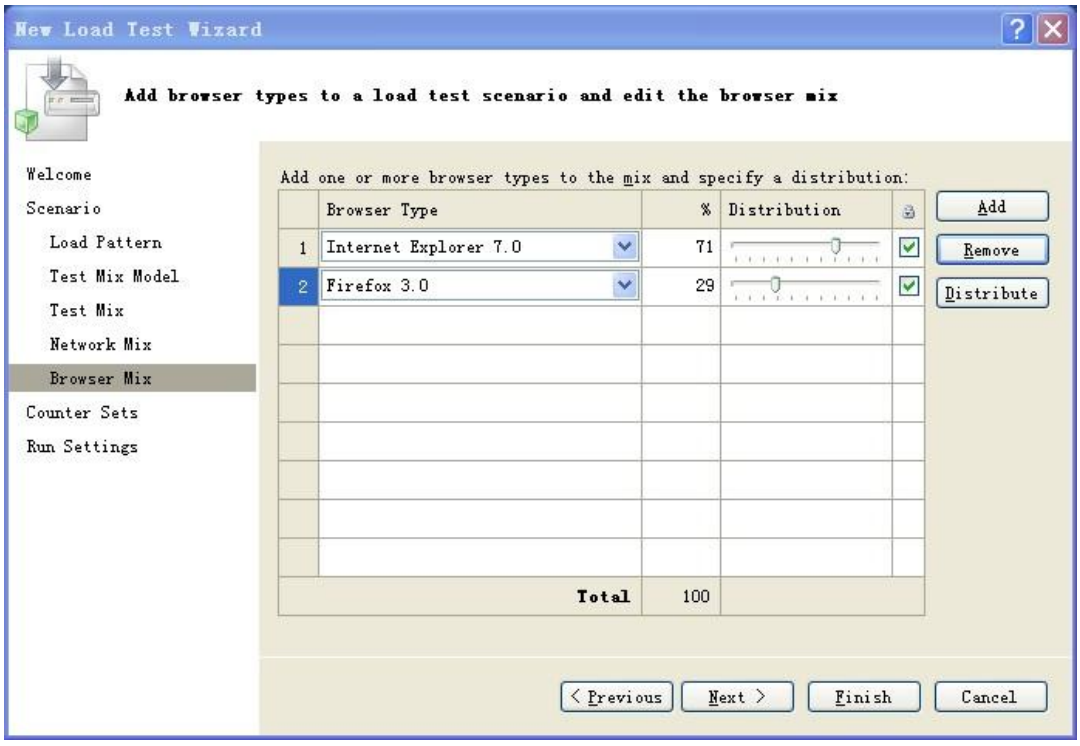
7：每种操作的分布，因为浏览的用户量大于修改密码的用户量，所以浏览操作我设置为占了百分之 75。



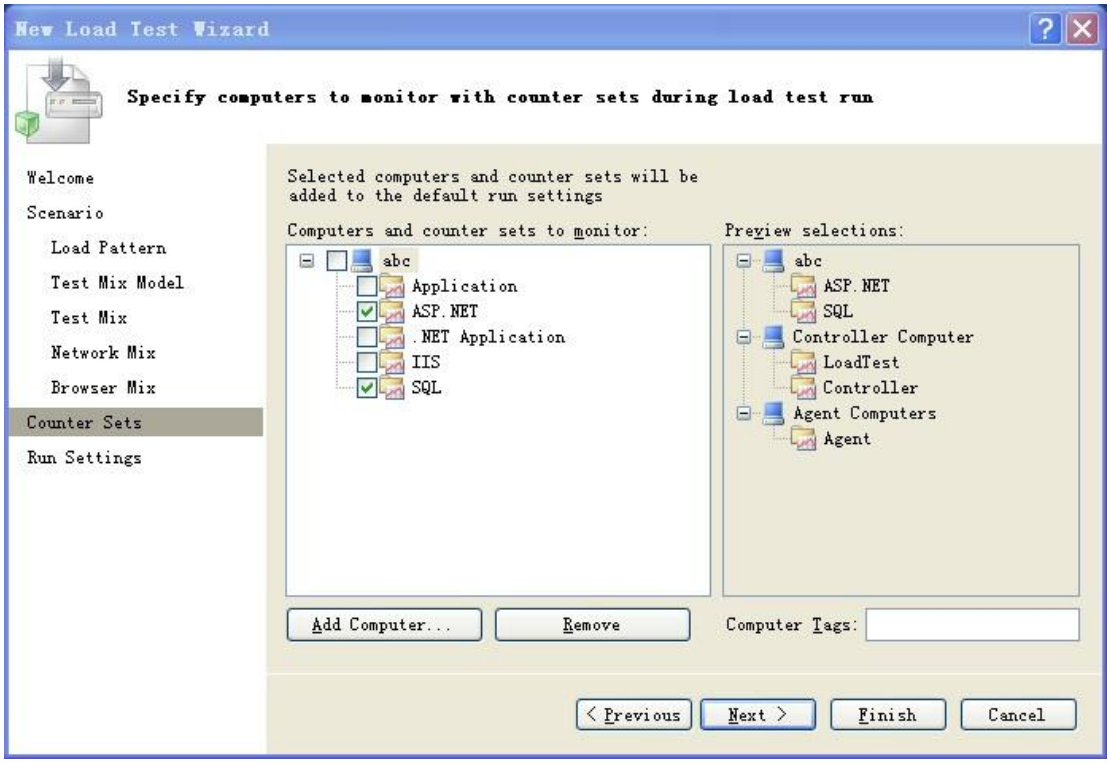
8：选择网络类型，以及所占百分比。



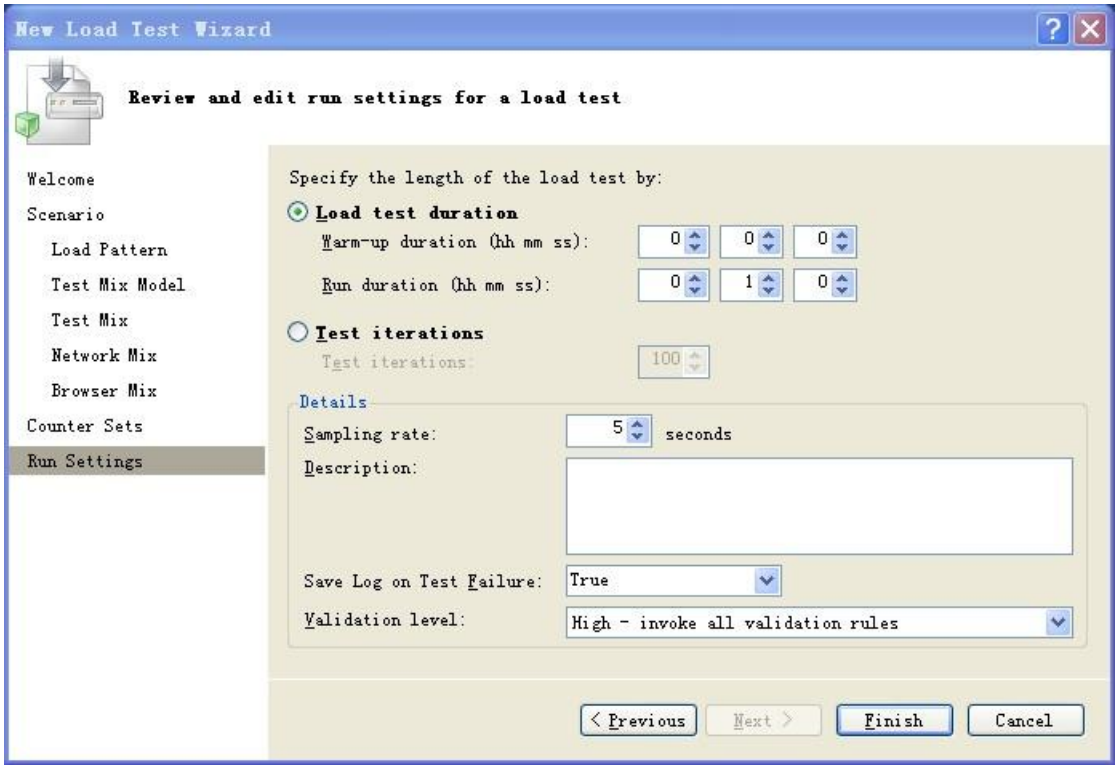
9： 选择浏览器类型，及其所占百分比。



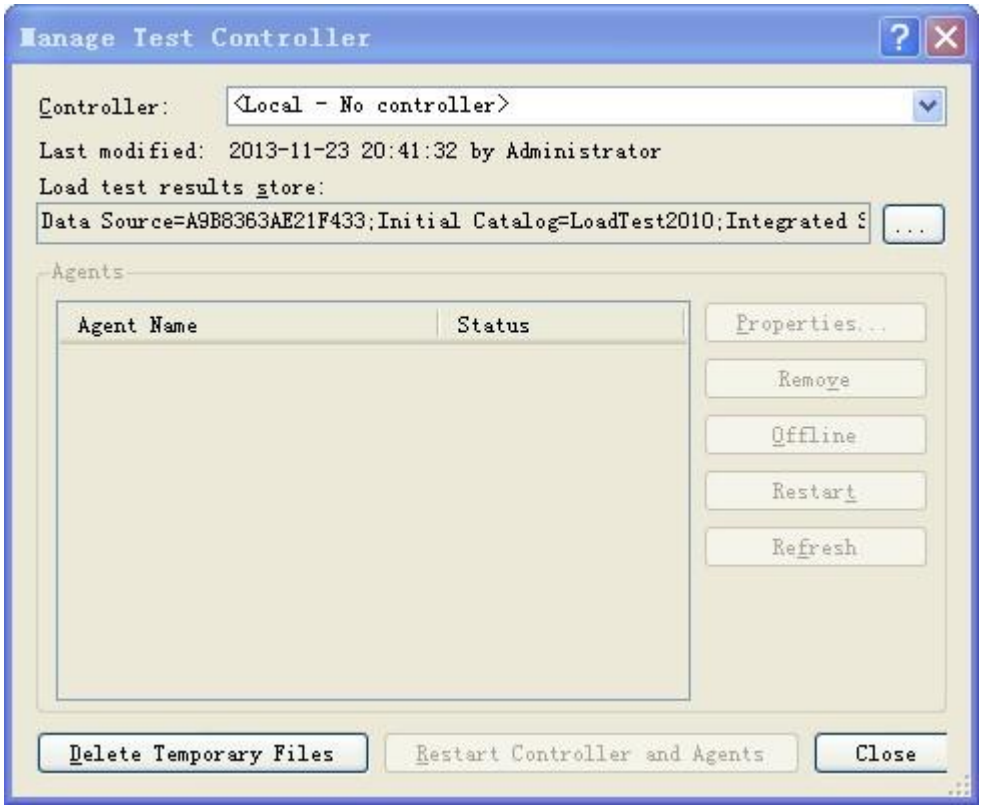
10：点击“Add computer” 输入计算机的名称，选择计数器类型。



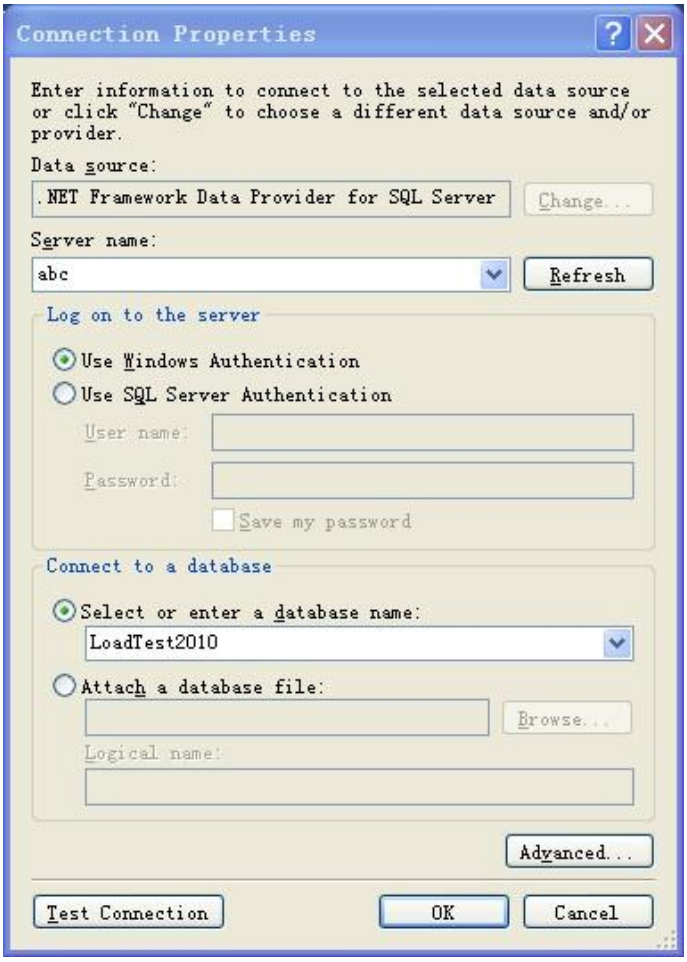
11：我们可以设置运行需要多少分钟，或者是迭代多少次运行。 我这里设置为运行 1 分钟，而且每 5 秒钟进行一次采样。最后点击“Finish” 按钮。



12： 点击 Test- Manage Test Controllers， 点击 “…”；



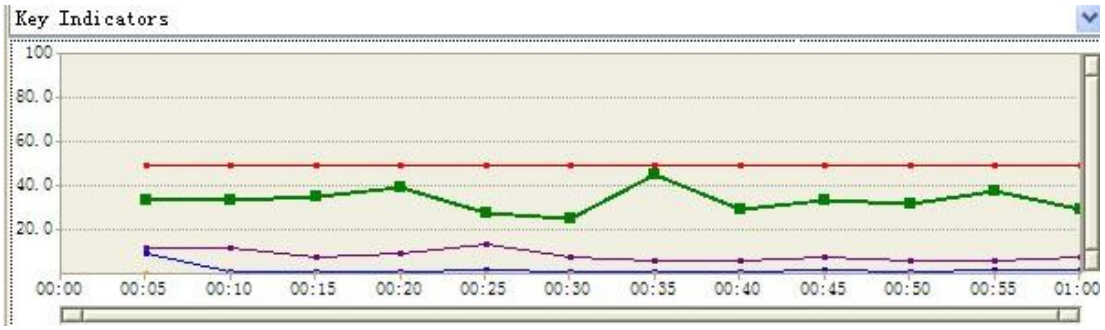
输入数据库服务器名称，选择数据库验证方式，选择数据库名称，就是LoadTest2010。可以测试一下链接数据库是否成功，最后点击“OK”完成。



13： 在 LoadTest 标签下，点击“Run Test”按钮  ，这样，Load Test 就开始运行了，总时间就是先前设定的 1 分钟，

14： 我们可以在 Graph 界面，看到 Load Test 的数据图。

比如说 pages/Sec 就是每秒多少页（绿色）



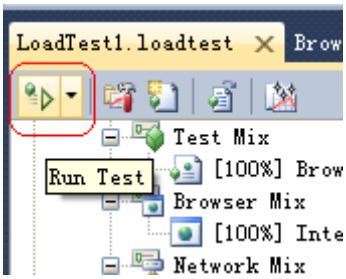
下方的 Key Indicators 则更为明了的显示出各种颜色的含义。

Counter	Instance	Category	Computer	Color	Range	Min	Max	Avg
Key Indicators								
User Load	_Total	LoadTest:Scen...	A9B8363AE21F433		10	5	5	5
Pages/Sec	_Total	LoadTest:Page	A9B8363AE21F433		10	2.60	4.60	3.40
Avg. Page Time	_Total	LoadTest:Page	A9B8363AE21F433		1	0.011	0.093	0.022
Errors/Sec	_Total	LoadTest:Errors	A9B8363AE21F433		10	0.60	1.40	0.87
Threshold Violations/Sec	_Total	LoadTest:Errors	A9B8363AE21F433		0	0	0	0

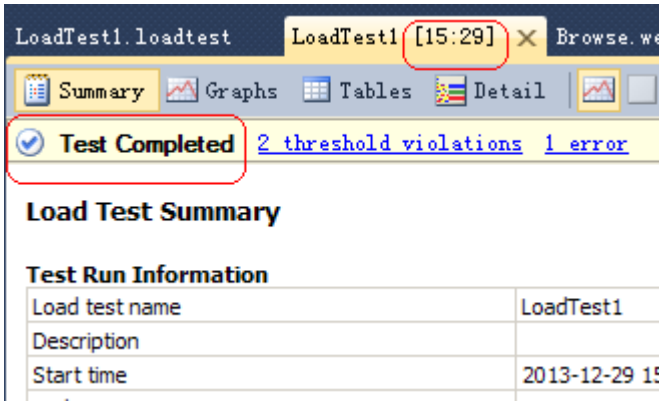
第三部分：对于这些数据进行统计学意义的分析

1：统计分析

点击”LoadTest1” 的 ”Run Test” 按钮，就可以再次运行此 Load Test。



而多次运行所形成的图表具有比较和分析的作用，通过多次运行，比较网站性能的稳定程度。每次运行 Load test 之后，就会显示出 “Test Completed”，而在 Load Test 的右边，则显示了运行开始的时刻，这有助于区分不同的运行结果。



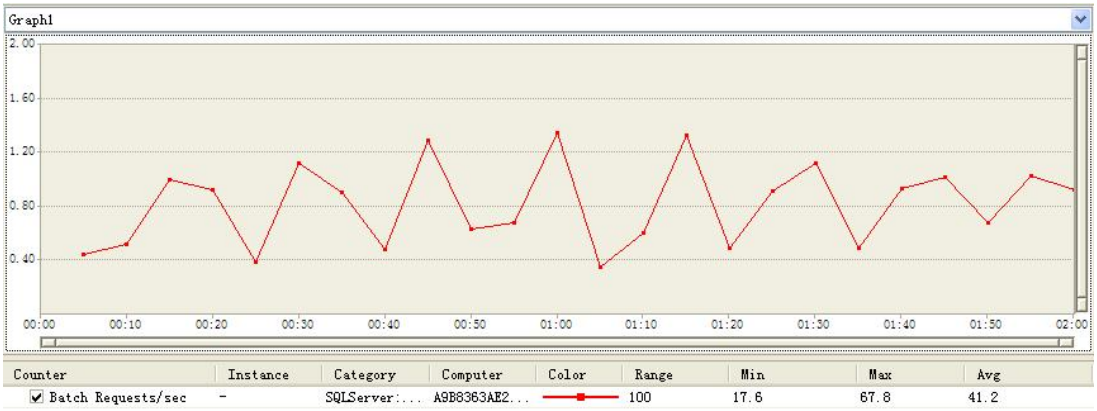
现在有了 2 个运行结果，它们是 LoadTest1［15：29］和 LoadTest1［12：11］

我可以对它们进行分析和比较。

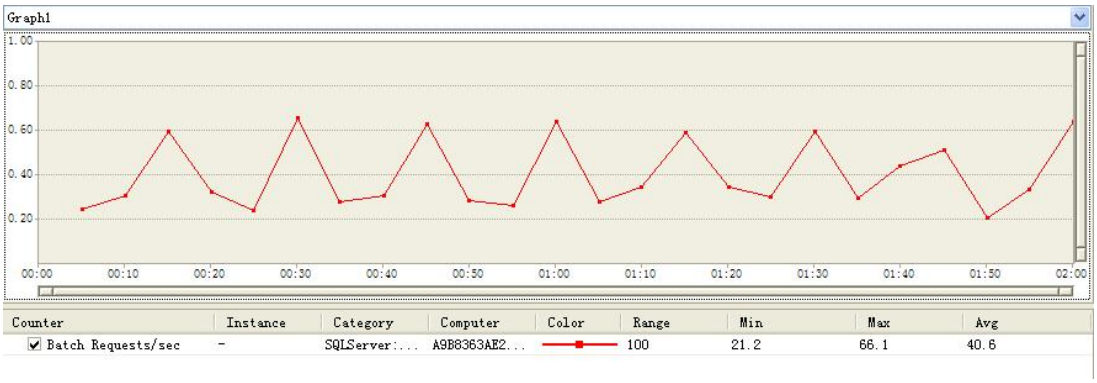
在统计学中，有一个字符就是表示某一组数据中数据的离散程度，就是σ，它叫做总体标准差，而σ^2叫做总体方差。

而我们现在有 2 个图表，分别是 LoadTest1［15：29］和 LoadTest1［12：11］的运行数据。

LoadTest1［15：29］：



LoadTest1 [12：11]：



这 2 张表所表示的数据库批处理的离散程度有和区别？

如果

$\sigma_1^2 = \sigma_2^2$，就说明，在 2 次的检验中，数据库使用的波动变化不大，否则就说明有一方的数据库的波动明显的增加了。

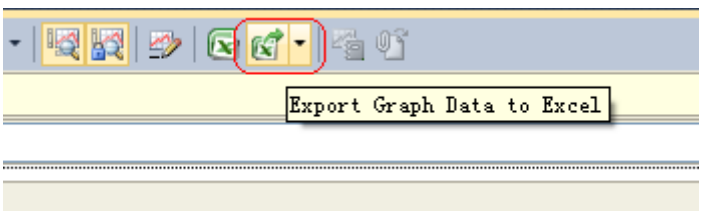
如何方便的计算它们的统计量？

水平：$\alpha = 0.05$，$H_0 : \sigma_1^2 = \sigma_2^2$（代表 2 次数据的波动一致）

$H_1 : \sigma_1^2 \neq \sigma_2^2$

（代表 2 次数据的波动不一致）

点击“Export Graph Data to Excel” 按钮。



这时候，一个 Excel 带着所有数据的值就这样出现了。

Elapsed TA9B8363AE21F433, SQLServer:SQL S	
0:00	
0:05	22.39103
0:10	26.39631
0:15	50.13951
0:20	46.29089
0:25	19.62696
0:30	56.22424
0:35	45.62265
0:40	24.14487
0:45	64.74222
0:50	31.65382
0:55	33.96549
1:00	67.83524
1:05	17.59086
1:10	30.19069
1:15	66.70529
1:20	24.65878
1:25	46.04256
1:30	56.25263
1:35	24.85381
1:40	46.96763
1:45	51.26444
1:50	34.06772
1:55	51.52633
2:00	46.47145

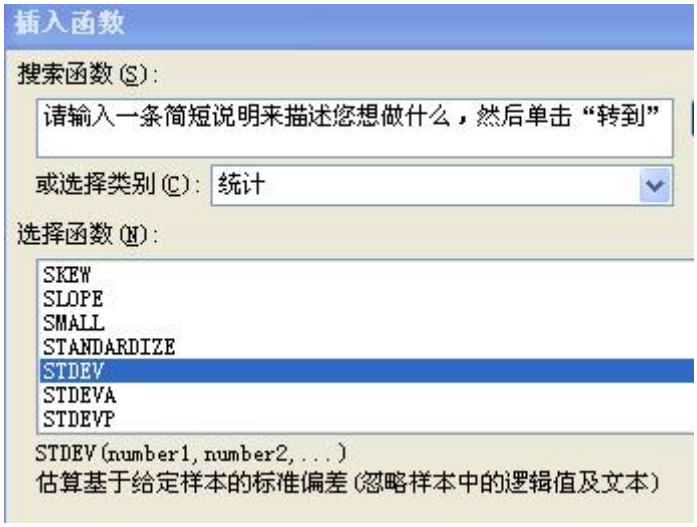
这样就可以计算出他们的无偏估计量 $S^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$

对于 LoadTest1 [15：29]

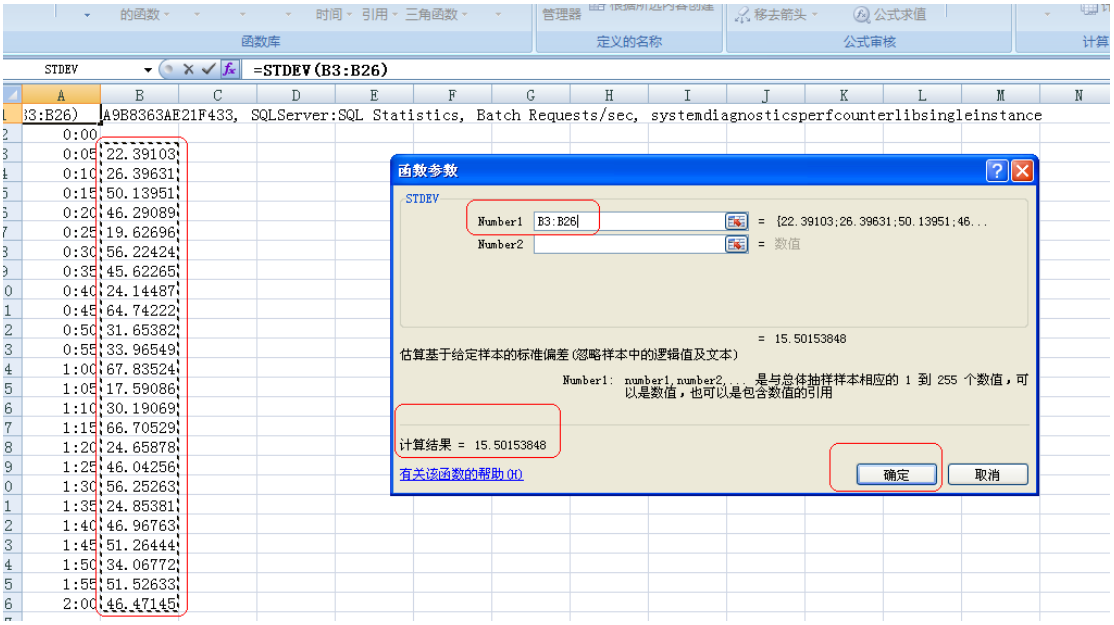
点击 Excel 中的”插入函数”按钮

弹出插入函数的对话框，

选择 STDEV



选取选用的样本区域，然后点击确定。



得出了 15.50154 这个值，就是 S，

所以 S^2 就是 15.50154*15.50154=240.2976952469207104

而对于另一个 LoadTest1［12：11］，我们用相同的方法得到了 S^2=248.3444322961563361

F=

$$\frac{S_1^2}{S_2^2}=\frac{240.2976952469207104}{248.3444322961563361}$$

=0.96759848016387292991025812687359

$\lambda_2 = f_{0.975}(23,23)$

=FINV(0.025，23，23)= 2.31

$\lambda_1 = \frac{1}{f_{0.975}(23,23)}$

约为 0.433

0.96 介于这 λ_1 和 λ_2 之间

于是我们可以得出结论，2 次数据的波动一致

所以这对于数据库的影响是一样的。所以这个网站是稳定的。

如果这个值大于了 2.31，那么我们可以得知，LoadTest1［15：29］的波动大于 LoadTest1 ［12：11］的波动，如果这个值小于 0.433，那么我们可以得知，LoadTest1［15：29］的波动小于 LoadTest1［12：11］。

2：三种检验及其必要性分析

1：以上方法可用来计算 2 次数据库波动情况是否存在差异。

2：如果对于第一种解释中，计算 2 次数据库波动情况是没有显著差异的，就可以继续检查第二种情况----波动平均值的检验。

由于只是算出 2 次数据库波动情况差异是不够的，因为也有可能一个 LoadTest 对于数据库的波动都是在比较高的水平之下，另一个 LoadTest 数据库的波动都是在比较低的水平之下，而它们的波动范围变化情况却可能是一致的。所以这时还需增加波动平均值的检验，来确保它们的波动都是在同一个平均值为中心的附近进行。使用 t 分布或者 Z 分布来检验，可以参考数理统计有关书籍。

3：分别检查 2 批数据波动的总体标准差 σ 的值是否都是在指定标准差的范围之内。

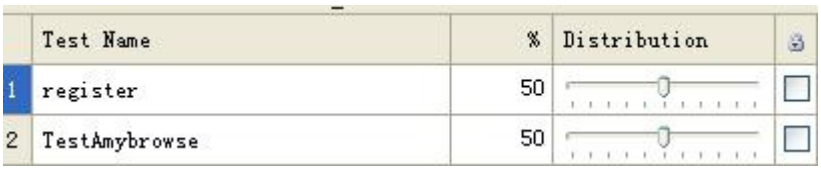
第三种检查需要对于 2 批波动的值分别进行检验，因为即使前 2 次检验都符合要求，第三次检验也可能因为，波动的总体标准差 σ 非常大而被认为不符合要求。因此分别检验总体标准差 σ 的值也是非常有必要的。

如果以上三种检验都符合要求，那么我们可以得出三个结论。

这 2 批数据以相同的波动幅度（1）在相同的区域内（2），同时也在合理的波动幅度上（3）进行波动。

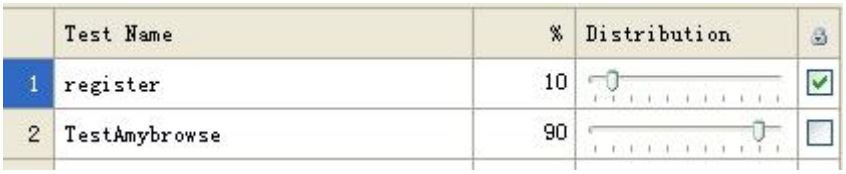
3：适用范围举例

此方法可以用在当统计相同场景的 performance test 之下，某 2 种不同的 LoadTest 之间的设置对于数据库到底是否会产生数据库性能波动方面的显著影响，比如：第一种 LoadTest，注册的人数和浏览的人数相等：



	Test Name	%	Distribution
1	register	50	
2	TestAmybrowse	50	

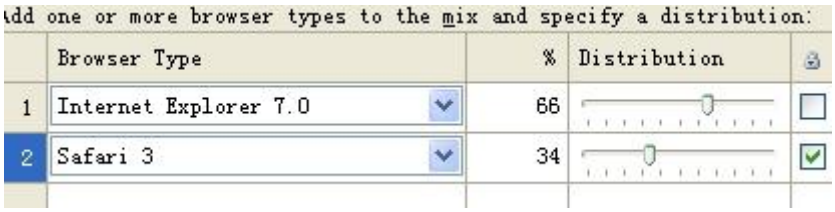
第二种，LoadTest，注册占百分之 10，浏览占百分之 90。



	Test Name	%	Distribution
1	register	10	
2	TestAmybrowse	90	

如果多次试验下，这 2 个 LoadTest 数据库批处理波动情况明显不同，一个明显小于另一个，就可以说明，不同操作的分布会影响到数据库的处理的。

同理，也可以统计不同浏览器的类型对于数据库波动情况的分析。等等。



dd one or more browser types to the mix and specify a distribution:

	Browser Type	%	Distribution
1	Internet Explorer 7.0	66	
2	Safari 3	34	

小贴士：有关运行 Load test 时报 403—Forbidden 的解决方法

由于默认设置下对于此 ASP 系统进行 Load test 的运行过程中，会产生大量“403—Forbidden”的错误，因此我在网上找了一个方法可以显著减少 Loadtest 时报 403—Forbidden 的次数，但是无法彻底避免。

1. Open load test， under Run Settings->Run Settings1[Active]， set the WebTest Connection Model to ″ConnectionPool″
2. Set the WebTest Connection Pool Size to 50
3. Save and run the load lest again

参考文献：

[1]唐国兴．高等数学（二）．武汉大学出版社，1999 年 10 月第二版．

[2] Explore Virtual Labs．http：//msdn. microsoft. com/en-us/aa570323. aspx.

由遗漏 bug 引发的思考

作者：爱菱

来淘宝已经五年多了，做多大大小小的项目日常已经自己也数不过来了。虽然在项目日常中，我总是竭尽全力去保证质量，但是无可避免，也会遗漏 bug，想对以前遗漏的 bug 和自己知道的其他人遗漏的 bug 做一个总结，反思总结才能让自己进步，不要范相同的错误。

（1） 不一致问题。
（2） 紧急情况。
（3） 性能问题。
（4） 兼容性问题。
（5） 并发问题。
（6） 交互问题。
（7） 沟通问题。
（8） 特殊场景。
（9） 未通知回归。
（10） 需求遗漏。
（11） 浏览器测试不全面。
（12） 关联的问题。
（13） 用户体验问题。
（14） 应用发布顺序有问题。
（15） 需求“搭车”发布导致问题。
（16） 优化代码导致错误。

一、需求阶段

1. 沟通问题。双方理解不一致，最后导致实现的需求不一致。这个问题需要有一个牵头人，组织相关人员碰头，进行评审，把需求明确，并把会议纪要以邮件的形式发给相关人员。实例：一个需求需要两个团队合作完成，需求评审时，没有沟通清楚，大家都按照自己理解的需求来开发，最后的判断原则却两边相反的，多么可怕。

2. 需求遗漏。测试要往前走，参加需求评审和技术方案、详细设计的评审，而不能只停留在 UC 评审后。需求能找出问题，是代价最低的。实例：底层应用改动，需要评估上层修改和回归，大家重点都放在大应用上，结果小应用的主流程都被影响，无法通过。

二、设计阶段

1. 兼容性问题。新系统上线，没有兼容老数据，新老方案不兼容，会引起严重问题，大批量用户数据不能使用。实例：增加一个关键字段，而该字段是不能为空，那么老数据就必须要进行初始化该字段为一个合理的值。还有数据结构变更，老数据迁移到数据结构，结果导致数据错乱。

2. 并发问题。尤其是对于多线程的功能，需要特别注意。实例：两个线程都在处理数据，并发锁时间设置过短，导致锁被提前释放（任务未执行完），任务状态未被置为成功，导致任务被重复执行。又或者任务在第一天没有执行完，第二天任务又开始，会重复处理。

3. 交互问题。两个系统交互，调用超时没有处理得当。实例：被调方超时，调用方一直没有得到响应，如果不设置合理的 timeout 时间，系统一直得不到对方的应答，最后系统无法执行。

三、开发阶段

1. 紧急需求。时间紧，考虑也不够全面，有些模块修改了，另外模板没有修改；有些应用修改，另外的应用没有修改。就会出现问题。紧急需求更需要进行代码 review 并提交给测试同学测试，并通知回归。（紧急需求，流程可以特殊，流程短，测试能尽快响应测试）。实例：特殊节日，要做活动，有 deadline，开发在没有评审、，没有代码 review、甚至没有测试的情况下紧急上线。

2. 优化代码。影响到非优化的模块，而又未回归，导致出错。首先优化代码不能私自优化，需要走正常流程，测试通过并全网回归通过后才能发布。实例：优化重构，开发修改了代码，把其他非优化的模块的代码修改错了，但是测试没有回归被修改的模块，全网回归也没有覆盖该功能点的脚本，就导致 bug 遗漏了。

3. 未通知关联应用回归。关联紧密的应用，其中一个应用修改功能，若不通知其他应用修改、回归，会导致关联应用都出现问题。比如底层应用修改，上游的应用都需要修改或者回归测试，这种情况就不必须要通知，邮件通知是一种形式，但是必须要确认大家是否都测试完成。如果发生这种情况就属于责任心的问题。实例：应用出现问题，排查了很久才知道底层应用升级了，但是完全没是收到通知。

四、测试阶段

1. 性能问题。存在模块需要处理大数据量，必须专业的评估是否存在性能问题，而不是开发测试想当然觉得性能没问题，很有可能导致重要模块功能异常。实例：项目中有一个接口是用于删除数据，开发和测试认为删除逻辑简单，性能

应该没问题，结果超时导致操作失败，因为删除数据需要进行递归，出现性能问题。

2. 特殊场景。有些特殊场景功能和接口测试无法模拟，需要采取新的技术手段，在测试阶段就应该模拟，如果无法实现，可以让开发帮忙。实例：消息堆积导致消息无法发出去，但是消息是否收到对功能是至关重要的影响，

3. 浏览器测试不全面。对于自己的应用，需要和相关人员确定都要 测哪些浏览器，也要关注有没新的浏览器出现和版本的问题。实例：新出现的浏览器，大家认为用户量会比较小，bug 就 later 了，但是随着时间的推移，新浏览器的用户量也增大，就变成了较大问题。

4. 用户体验问题。这个很容易被忽略，测试同学总是在关心功能是否正常，忘记用户体验也是非常重要的一个问题。用户体验如果不好，很容易丢失用户，让用户对产品失去信心，进而不想用。实例：后台应用，面对的是内部小二，开发的工具没怎么考虑用户体验，觉得到时候培训一次就可以了，但是用户会更新，操作流程也只能通过口口相传才可以，到最后很少有人会用功能，有些好功能都荒废了

5. 关联的问题。是测试同学容易遗漏的，这需要对功能十分了解，需要知道架构，如何交互，如何调用。需要提高能力和透彻的分析。实例：应用中有些数据是通过接口从其它应用读取的，如果其它应用返回的是空或者其他应用挂了没有返回，自身应用没有对这些异常进行处理，导致自身功能也异常。

五、发布阶段

1. 不一致问题，日常环境和线上环境不一致，beta 机器上和线上代码不一致，比如配置、数据存储方式，都有可能导致问题，测试同学在日常环境和预发环境测试通过，但是发到线上就有故障。这个在发布前需要有 checklist 进行确认，尽量 beta 测试。实例：daily 的数据是数据库，而线上是走缓存，结果测试通过后发布却出现问题。

2. 应用发布顺序有问题。应用之间有些是有依赖关系，如果发布顺序有问题，也会引起线上问题。这个是需要发布之前制定方案，然后项目组全体同学评审才能发布。实例：两个应用，一个是底层应用，一个是上层服务，上层服务需要调用底层服务，新加一个字段，上层服务先发，底层应用后发导致数据写不进去

3. 需求“搭车”发布导致问题。项目发布时，有日常要和项目一起“搭车”发布，两者技术方案或者代码有冲突，导致未测试通过的代码被发上线，引起事故。实例：项目测试完成后自动化回归通过，在不知情的情况下开发把一个需求的代码也合并到已经测试通过的代码中，此时没有经过自动化回归就发布了，发

布后就出问题了。

以上的总结是我今后做项目和日常要借鉴的，希望能对大家有帮助。

用 Python 来做单元测试

译者：于芳

Python 有许多资源可以用来做单元测试。

单元测试是软件开发中一个必不可少的部分。通过单元测试，我们可以评估每个代码组件，查看软件性能如何，然后决定它如何对合法或非法输入做出反应。一个单元测试的回归套也是一种发现由重组代码或者写入新代码引起的意料之外的变化的非常好的方式。

在本文中，我检查了 Python 中的单元测试机制，从 unittest 模块与其关键类开始。单独地检查了测试，也集合式地检查测试，讨论怎样简便他们的构建和使用。读者应该对 Python 有一些应用知识。例用的测试代码需要 Python2. 5 或者更高的版本。

Unittest 模块

Unittest 模块开始以第三方模块 PyUnit 应用。PyUnit 是 JUnit 的一个 Python 端口，Java 的单元测试框架。由史蒂夫. 玻塞尔设计，PyUnit 从版本 2. 5 开始成为正式的 Python 模块。

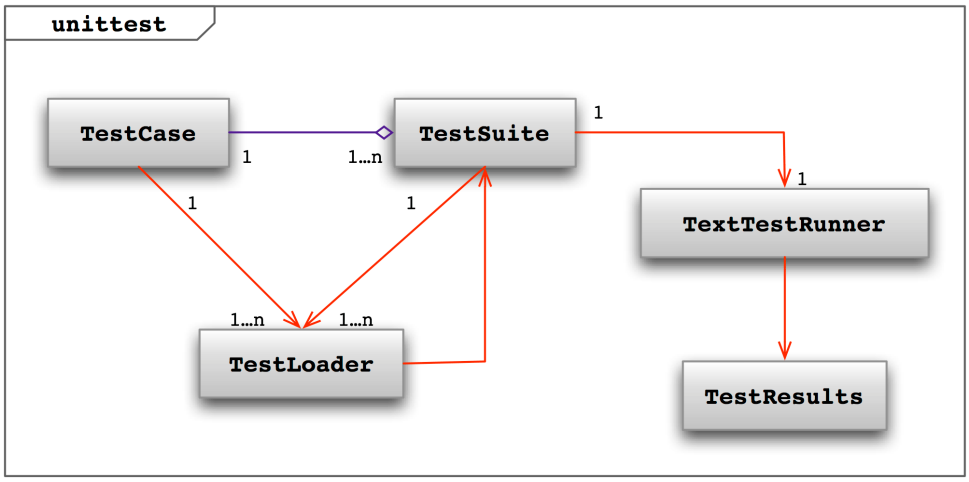


图 1：unittest 的核心类

如图 1 所示，unittest 模块有 5 个关键类。TestCase 类长官测试路径，为随后的每个路径和赚大钱提供吸引力。TestSuite 类用作收集容器。它可以容纳多个 TestCase 对象和多个 TestSuite 对象。

TestLoader 类装载本地定义的或者从外部文件导入进来的测试用例和用例套，发射出一个容纳这些用例和套的 TestSuite 对象。TextTestRunner 类负责提供跑用例的标准平台。TestResults 类提供装载测试结果的标准容器。

在这 5 个类中，只有 TestCase 必须子类化。其他 4 个类也可以子类化，但是通常它们只用基本类。

准备一个测试用例

图 2 显示了 TestCase 类的架构。在 TestCase 类里，是 3 个方法集，最常用来设计测试用例。在一个方法集中，是测试前钩和测试后钩。setUp()方法在每次测试工作前开工，tearDown()方法在每次测试工作后开工。创建定制的测试用例时，就重写这些方法。

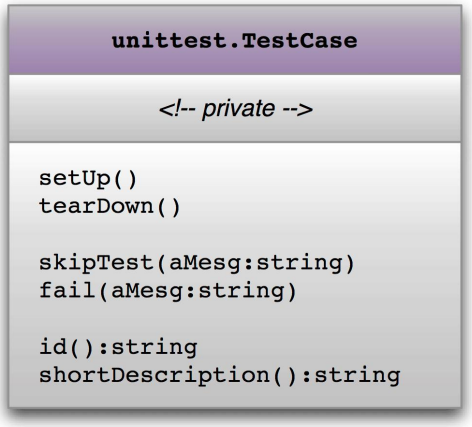


图 2：一个 TestCase 类的结构

第二对方法控制测试的执行。两个方法都以信息字符串作输入，而且都会中止进行中的测试。但是 skipTest()方法取消当前测试，而 fail()方法直接放弃掉当前测试。

第三组方法帮助识别测试。id()方法返回一个包含有 TestCase 名字对象与其测试例行工作名字的字符串。而 shortDescription()方法在每个测试工作开端返回 docstr 标注。如果测试工作没有这样的标准，shortDescription()方法就不返回值。

列表一显示样例-测试用例 FooTest 的基本要素。FooTest 有两个测试程序：testA()和 testB()。两个程序从自身得到需要的变量。两者对第一行代码都有 docstr 标注。

列表一：显示单元测试执行顺序的代码

```
1       #!/usr/bin/python
2
3       import unittest
4
5       class FooTest(unittest. TestCase):
6               """Sample test case"""
7
8               # preparing to test
9               def setUp(self):
10                      """ Setting up for the test """
11                      print "FooTest: setUp_: begin"
12                      ## do something. . .
13                      print "FooTest: setUp_: end"
14
15              # ending the test
16              def tearDown(self):
17                      """Cleaning up after the test"""
18                      print "FooTest: tearDown_: begin"
19                      ## do something. . .
20                      print "FooTest: tearDown_: end"
21
22              # test routine A
23              def testA(self):
24                      """Test routine A"""
25                      print "FooTest: testA"
26
27              # test routine B
28              def testB(self):
29                      """Test routine B"""
30                      print "FooTest: testB"
```

图 3 展示了 FooTest 在程序运行时的行为。

注意相同的 setUp()和 tearDown()方法在每个测试程序之前和之后运行。因此你怎样使 setUp()和 tearDown()方法知道是哪个程序在运行？你必须首先通过调用 shortDescription()或 id()方法（见列表二）来识别程序。他们使用 if-else 代码块来指向合适的代码。在样例块中，FooTest 调用 shortDescription()方法来获取程序的 docstr 标注，然后为那个程序运行准备代码和清除代码。

列表二：使用测试描述。

```
1   import unittest
2
```

```
3   class FooTest(unittest.TestCase):
4           """Sample test case"""
5
6           # preparing to test
7           def setUp(self):
8                   """ Setting up for the test """
9                   print "FooTest: setUp_: begin"
10
11                  testName = self.shortDescription()
12                  if (testName == "Test routine A"):
13                          print "setting up for test A"
14
15                  elif (testName == "Test routine B"):
16                          print "setting up for test B"
17
18                  else:
19                          print "UNKNOWN TEST ROUTINE"
20
21                  print "FooTest: setUp_: end"
22
23          # ending the test
24          def tearDown(self):
25                  """Cleaning up after the test"""
26                  print "FooTest: tearDown_: begin"
27
28                  testName = self.shortDescription()
29                  if (testName == "Test routine A"):
30                          print "cleaning up after test A"
31
32                  elif (testName == "Test routine B"):
33                          print "cleaning up after test B"
34
35                  else:
36                          print "UNKNOWN TEST ROUTINE"
37
38                  print "FooTest: tearDown_: end"
39
40          # see Listing One...
```

设计一个测试程序

每个测试程序在其名字中必须含有一个“test”前缀。没有那个前缀，程序就不会运行。为执行一个测试，测试程序应当使用断言方法。断言方法获取一

个或多个测试变量和一个可选的断言消息。当一个测试失败时，断言中止该程序，发送错误消息给 stdout。

有三个断言方法集。在第一个集（表 1）中，是基本的布尔型断言，会在遇到 True 或者 False 值时启动。

Assert	**Complement Assert**	**Operation**
assertTrue(a， M)	assertFalse(a， M)	a = True; a = False
assertEqual(a， b， M)	assertNotEqual(a， b， M)	a = b; a ≠ b
assertIs(a， b， M)	assertIsNot(a， b， M)	a is b; a is not b
assertIsNone(a， M)	assertIsNotNone(a， M)	a = nil; a ≠ nil
AssertIsInstance(a， b， M)	AssertIsNotInstance(a， b， M)	isinstance(a，b); not isinstance(a，b)

表 1：unittest 中的基本的断言

为了测验一下是 True 还是 False 值，使用 assertTrue()或者 assertFalse()方法，如列表三：

列表三：测验一个值是 True 还是 False

```
1       self. assertTrue(argState， ″foobar() gave back a False″)
2       # -- fires when the instance method foobar() returns a True
3
4       self. assertFalse(argState)
5       # -- fires when foobar() returns a False
6       # Notice this one does not supply an assert message
```

测验两个变量值是否相同，使用 assertEqual()和 assertNotEqual()方法，见列表四。这后两个断言测验变量的值和他们的数据类型。

列表四：测验变量值。

```
1     argFoo = ″narf″
2     argBar = ″zort″
```

```
3
4       self. assertEqual(argFoo， argBar， ″These are not the same″)
5       # -- this assert will fail
6
7       self. assertNotEqual(argFoo， argBar， ″These are the same″)
8       # -- this assert will succeed
9
10      argFoo = 123
11      argBar = ″123″
12
13      self. assertEqual(argFoo， argBar， ″These are not the same″)
14      # -- this assert will fail
```

测验变量是否属于相同的对象，使用 assertIs()和 assertIsNot()方法。像 assertEqual()和 assertNotEqual()方法，这两个断言检查变量的值以及他们的数据类型。检查一个变量是否是一个特定类的实例，使用列表五种的 assertIsInstance()和 assertIsNotInstance()方法。

列表五：检查一个变量是否是一个特定类的实例

```
1       argFoo = Bar()
2
3       # checking against class Bar
4       self. assertIsInstance(argFoo， Bar， ″The object is not an instance of class Bar″)
5       # -- this assert will succeed
6
7       # checking against class Foo
8       self. assertIsNotInstance(argFoo， Foo， ″The object is an instance of class Foo″)
9       # -- this assert will fail
```

两个断言都获取一个类命名为一个次要的变量。两者对库函数 isInstance()有着类似的行为。最后，检查一个值是否为 nil，使用 assertIsNone()和 assertIsNotNone()断言.

第二组断言是用于比较的。（见表 2）

Assert	**Complement Assert**	**Operation**
assertGreater(a， b， M)	assertLess(a， b， M)	a ＞ b; a ＜ b
assertGreaterEqual(a， b， M)	assertLessEqual(a， b， M)	a ≥ b; a ≤ b

assertAlmostEqual(a， b， n， M)	assertNotAlmostEqual(a， b， n， M)	round(a-b， n) = 0 round)a-b， n) ≠ 0
assertItemsEqual(a， b， M)	*none*	sort(a) = sort(b); sort(a) ≠ sort(b)

表 2：比较断言

检查一个变量是否比另一个变量大或者小，使用 assertGreater()和 assertLess()，见列表六。

列表六：大或者小。

```
argFoo = 123
argBar = 452

self.assertGreater(argFoo， argBar， "Foo is less than Bar")
# -- this assert will fail

self.assertLess(argFoo， argBar， "Foo is greater than Bar")
# -- this assert will succeed
```

测验一个变量比另一个变量大，小或者相等，使用 assertGreaterEqual()和 assertLessEqual()。这四个断言中的变量可以是原始值型（整数型，浮点型，字符型），一个序列或者一个集合。但是，两个变量必须数据类型相同。

要做模糊检查，使用 assertAlmostEqual()和 assertAlmostNotEqual()。这两个断言按固定的十进制小数点位置取整数值之后再比较数值大小。默认是取小数点后 7 位。要换成其他的话，把位置标签改成一个新的数值。

要比较两个序列，使用 assertItemsEqual()。它的两个变量必须是同一种序列类型（列表，数组，集合等等）。注意这个断言在比较前会先给序列里的项目排序。

第三个断言集（表 3）与像字典，列表，集和数组这样的集合对象一起工作。

Assert	**Complement Assert**	**Operation**
assertIn(a， b， M)	assertNotIn(a， b， M)	a in b; a not in b
assertDictContainsSubset(a， b， M)	none	a has b
assertDictEqual(a， b， M)	none	a = b

assertListEqual(a， b， M)	none	a = b
assertSetEqual(a， b， M)	none	a = b
assertSequenceEqual(a， b， M)	none	a = b
assertTupleEqual(a， b， M)	none	a = b
assertMultilineEqual(a， b， M)	none	a = b

表 3：集合断言

变量必须是一个集合类型。有些断言不需要使用具有相同类型的变量。这组断言中只有一个断言没有补充变量。

要检验一个字典对象是否像另外一个拥有某些键/值对，使用 assertDictContainsSubset()，见列表七。

列表七

```
argBar = {'narf': 456, 'poink': 789}

self.assertDictContainsSubset(argFoo, argBar, "Foo does not have Bar")
# -- this assert will succeed

self.assertDictContainsSubset(argBar, argFoo, "Foo does not have Bar")
# -- this assert will fail

argBar = {'narf': 456, 'egad': 789}
self.assertDictContainsSubset(argFoo, argBar, "Foo does not have Bar")
# -- this assert will also fail
```

第一个变量用作引用；第二个掌控提到的字典对象对。要检查两个字典对象是否有相同的键/值对，使用 assertDictEqual()。每对必须有同样的键标签和数据类型。这些字典对怎样排列的不相关。

要检查两个序列对象，使用 assertSequenceEqual()。序列类型包括列表，集，数组，甚至字符串。对于相同的序列对象，他们必须有同样的数据项目。项目必须有相同的值，而且必须是相同的排列方式。序列类型必须也是相同的。

要检查两个列表对象是否相同，使用 assertListEqual()。两个对象必须有同样数量的项目。这些项目必须有同样的值和同样的顺序。要检查两个集对象，使用 assertSetEqual()。对于列表，两个集对象必须有相同数目的项目和项目值。但是项目顺序没关系，因为集对象会内部排列自己的项目。

最后，要检查两个数组是否相同，使用 assertTuplesEqual()。要检查两个字符串是否相同，使用 assertMultilineEqual()。而要查看一个字符串是否包含在另

这第三个断言集有一个有趣的特征。如果集合对象不相等，断言会报出两者之间的差别。也会将这个不同结果添加到断言消息里，如果有断言消息可用的话。

准备一个测试套

通常情况下，一些测试用例对你需要测试的单个类来说足够了。但是如果你手头上有数十个或者更多的测试用例呢，有一些是自己写的，有一些是别人写的呢？如果你只想用一个测试程序的子集跑同样的测试用例呢？如果你想重构测试程序，使用更简单的类别划分和发布呢？对于这些情况，你可能需要一个测试套。

图 4 展示了 TestSuite 类的基本结构。

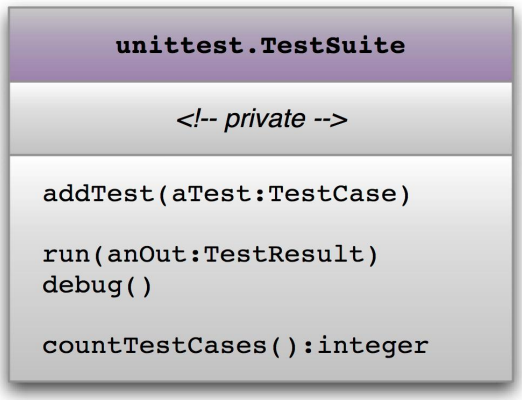


图 4：TestSuite 类

有三个实例方法集。第一集让我们可以将测试用例添加到套种。要添加单个测试用例，使用 addTest()方法，见列表八。

列表八

```
class FooTest(unittest.TestCase):
        def testA(self):
                """Test routine A"""
                print "Running test A"

# creating a new test suite
newSuite = unittest.TestSuite()
```

```
# adding a test case
newSuite.addTest(unittest.makeSuite(FooTest))
```

使用便利函数 makeSuite()将测试用例（这里是 FooTest）传给实例方法。

```
newSuite = unittest. makeSuite(FooTest)
```

你也可以使用 makeSuite()来将测试用例‘转化’进一个测试套。

要添加一个特定的测试程序，通过同样的 addTest()方法将测试用例对象传递给测试套。然后将测试程序的名字传递给测试用例构造器。注意程序名称是作为

```
newSuite. addTest(FooTest("testA"))
```

你也可以使用相同的方法将两个或者多个测试程序添加到同一个套：

```
newSuite. addTest(FooTest("testA"))
newSuite. addTest(FooTest("testB"))
#. . .
```

列表九

```
testList = [FooTest， BarTest]
testLoad = unittest.TestLoader()
caseList = []
for testCase in testList：
        testSuite = testLoad.loadTestsFromTestCase(testCase)
        caseList.append(testSuite)

newSuite = unittest.TestSuite(caseList)
```

用一个 for 循环检索该列表，然后使用 TestLoader 对象(testLoad)读取每个用例。添加读取过的用例到第二个列表中（caseList）。然后创建 TestSuite 对象（newSuite），传递给类构造器列表对象 caseList。

假设你想添加另一个测试套到该套，只要将另一个套传给 addTest()方法---不需要重用 makeSuite()函数来准备添加套。

```
fooSuite =
unittest. TestSuite()
fooSuite. addTest(unittest. ma
keSuite(FooTest))
#. . .
barSuite =
unittest. TestSuite()
```

套中的测试用例。run()方法以一个 TestResult 对象作输入，而 debug()方法与此不同。但是 debug()方法的确使一个外部调试器监控进行中的测试。

最后，最后一个集包括 countTestCases()方法。这个方法返回该套中掌控的测试用例的数目。

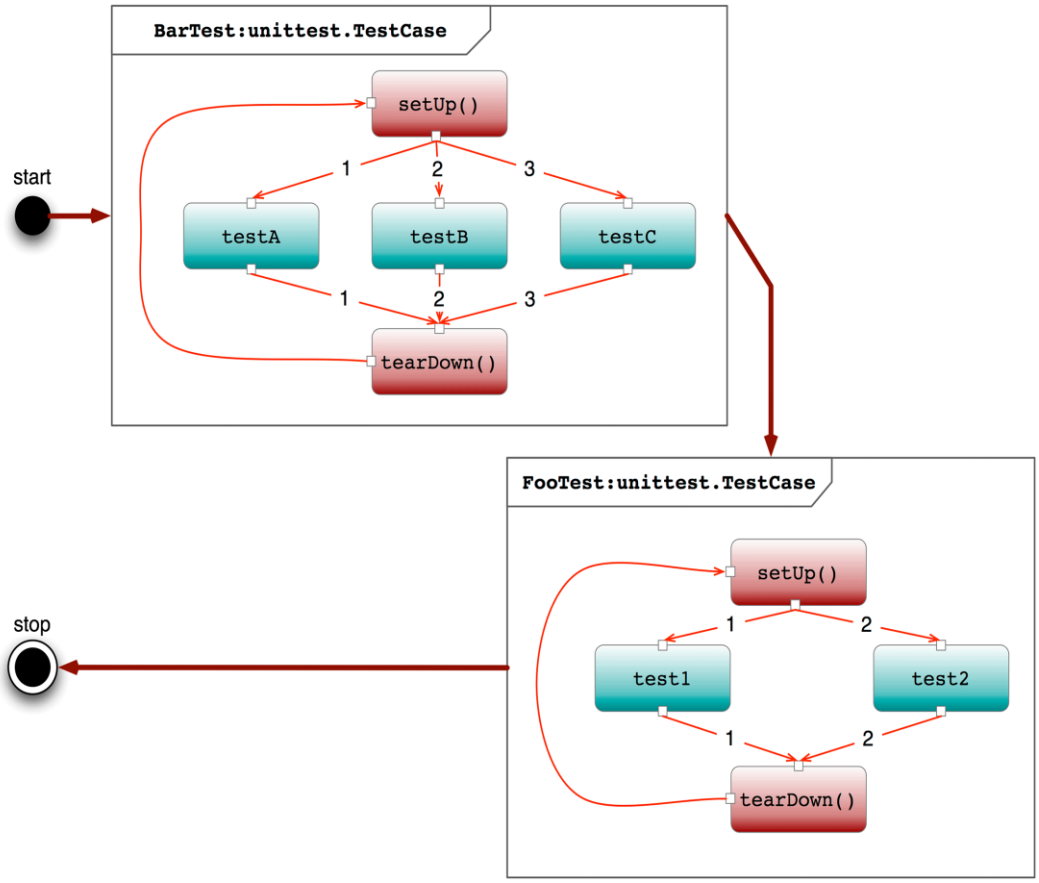
```
testCount = fooSuite. countTestCases()
```

运行测试

有两种方式来运行单元测试。如果测试脚本是一个有一个或多个测试用例的单独文件，添加下面代码到最后一个测试用例。

```
if __name__ ==
″__main__″:
unittest. main()
```

文件是怎样被操作的。如果这个文件被导入进另一
持不变。如果这个文件是直接执行的，或者通过文
者通过命令行，__name__ macro 会演变成
n()会被调用。这反过来激活了每个由脚本文件定义或者导入的测试用例的 run()方法。



如果这个脚本文件定义了一个测试套，首先创建一个 TextTestRunner 实例。然后将测试套对象传给运行器的 run()方法。

```
fooRunner =
unittest. TextTestRunner(
)
fooRunner. run(fooSuite)
```

列极其程序是字母混合数字的顺序运行的。BarTest
est 在 Test123 之前运行，test_12()在 test_A()之前
试脚本。在那个脚本里，你有两个测试用例：
t 有三个测试程序，FooTest 有两个。

图 5 显示了这两个测试用例怎样运行的。BarTest 先运行，FooTest 其次。

图 5：BarTest 和 FooTest

BarTest 中的测试程序以从 A 到 C 的顺序运行。而在 FooTest 中的那些以从 1 到 2 的顺序运行。但是 BarTest 程序有自己的 setUp()和 tearDown()，对 FooTest 程序也同样。

最周，你可以选择跳过或者放弃掉测试程序的某些测试用例。要无条件地跳过一个程序，使用类方法 unittest. skip()，如列表十所示。

列表十：跳过一个程序

```
@unittest.skip("Skip over the entire test routine")
def testB():
        """Test routine B"""
        # the following code will not run
        fooA = narf()
        self.assertNotEqual(fooA， 123)
```

这个方法得到一个变量：一个描述跳过的原因的日志消息。将这个方法的调用放在测试程序前，并确保用一个@标记给该调用标前缀。或者，也可以使用实例方法 skipTest()，可以将方法调用放在测试程序里，见列表十一。

列表十一

```
        """Test routine B"""
        self.skipTest("Skip over the rest of the routine")

        # the following code will not run
        fooA = narf()
        self.assertNotEqual(fooA， 123)
```

要有条件地跳过一个测试程序，使用类方法 unittest. skipIf()和 unittest. skipUnless()，如列表十二：

列表十二

```
1       @unittest.skipIf(self.fooC > 456， "Skip over this routine")
2       def testB():
3         """Test routine B"""
4         # the following code will run only when fooC is less
5         # than or equal to 456
6         fooA = narf()
7         self.assertNotEqual(fooA， 123)
8
9       @unittest.skipUnless(self.fooC > 456， "Skip over this routine")
10      def testC():
11        """Test routine C"""
12        # the following code will run only when fooC is
13        # greater than 456
14        fooA = zort()
15        self.assertNotEqual(fooA， 123)
```

第一个方法在一个条件满足的时候调动了一次跳过，第二个方法调动跳过在一个条件没有满足的情况。两个方法获取到两个变量：条件和跳过的原因。两个必须在测试程序前布置。跳过的原因可能涉及到一个类的属性或者另一个类的方法。

要失败一个测试用例，使用实例方法 fail()，见列表十三：

列表十三

```
1         def testB():
2            """Test routine B"""
3            self.fail("Force this routine to fail.")
4
5            # the following code will not run
6            fooA = narf()
7            self.assertNotEqual(fooA， 123)
8
9         def testC():
10           """Test routine C"""
11           print "This routine still runs after testB."
```

这个方法用一个日志消息解释失败的原因。像 skipTest()， the fail()方法进入到测试程序里面。放在 the fail()方法后的代码不会运行，但是在失败程序后的测试程序仍将继续运行。

查看测试结果

从 TextTestRunner 里，有两种可能的输出形式：控制台文本或者一个 TestResult 对象。首先，让我们看一下控制台输出，它显示每个测试的结果。你可以通过传递三个可选的变量给类构造器来控制其输出。

unittest. TextTestRunner（stream=sys. stderr， descriptions=True，verbosity=1）

第一个变量（标签为流）设置输出目的地，默认值为 sys. stderr，如果没有特定值的话。下一个变量（标签是描述）控制错误和失败的报告方式。传递一个 True 值（这是默认值）告诉运行者对那些出错，失败或者跳过的程序命名。传递一个 False 值则告诉不要命名。

最后一个构造器变量（标签为冗余）设置细节水平。有三个水平。对冗余值为 0 的情况，测试结果只显示执行的测试数目和那些测试的最终输出结果。

对冗余值为 1 的情况，测试结果会用一个点来标记运行成功的测试，用 F 标记运行失败的用例，用 s 标记跳过的测试用例，用 E 标记有错的用例。对冗余值为 2 的情况，测试结果列出每个测试用例及其测试程序，加上每个程序的输出结果。

为演示清楚，让我们运行列表十四种的测试脚本：

列表十四：测试脚本

```
1    import sys
2    import unittest
3
4    class FooTest(unittest2. TestCase):
5            """Sample test case"""
6
7            # preparing to test
8            def setUp(self):
9                    """ Setting up for the test """
10                   print "FooTest: setUp_"
11
12           # ending the test
13           def tearDown(self):
14                   """Cleaning up after the test"""
15                   print "FooTest: tearDown_"
16
17           # test routine A
18           #@unittest2. skip("FooTest: test_A: skipped")
19           def test_A(self):
20                   """Test routine A"""
21                   self. skipTest("FooTest: test_A: skipped")
22                   print "FooTest: test_A"
23
24           # test routine B
25           def test_B(self):
26                   """Test routine B"""
27                   fooA = 123
28                   fooB = 234
29                   self. assertEqual(fooA, fooB, "A is not equal to B")
30                   print "FooTest: test_B"
31
32           # test routine C
33           def test_C(self):
34                   """Test routine C"""
```

```
35                          fooA = 123
36                          self. assertEqual(fooA， fooB， ″A is not equal to B″)
37                          print ″FooTest：test_C″
38
39              # test routine D
40              def test_D(self)：
41                          ″″″Test routine D″″″
42            self.fail("FooTest：test_D：fail_")
43          print "FooTest：test_D"
44
45    # Run the test case
46    if __name__ == '__main__'：
47       fooSuite = unittest.TestLoader().loadTestsFromTestCase(FooTest)
```

如果你创建了如下的测试运行器：

```
fooRunner =
unittest. TextTestRunner(desc
ription=True)
fooRunner. run(fooSuite)
```

容：

```
======================================================================
ERROR： test_C (__main__.FooTest)
Test routine C
----------------------------------------------------------------------

======================================================================
FAIL： test_B (__main__.FooTest)
Test routine B
----------------------------------------------------------------------

======================================================================
FAIL： test_D (__main__.FooTest)
Test routine D
----------------------------------------------------------------------

----------------------------------------------------------------------
Ran 4 tests in 0.004s
```

用 0. 004s 时间运行 4 个测试用例

但是如果你创建如下的运行器：

FAILED (failures=2， errors=1， skipped=1)

But if you create the runner as follows：

fooRunner = unittest. TextTestRunner（description=False）

The docstr comment line from each test routine will not be included.

每个测试程序中的 docstr 标注行将不会被包括在内。

上面的控制台输出结果有一个冗余为 1 的水平值，从 sFEF 行开始，sFEF 代表 2 个失败的用例，一个跳过的和一个有错的用例。但是假如你创建如下测试运行器

fooRunner = unittest. TextTestRunner（verbosity=**0**）

Then, the sFEF line will not appear. If you create the test runner by typing

那么，sFEF 行将不会出现。如果你输入下面代码行

那么控制台输出结果显示为：

```
======================================================================
FAIL： test_B（__main__. FooTest）
Test routine B
----------------------------------------------------------------------


======================================================================
FAIL： test_D（__main__. FooTest）
Test routine D
----------------------------------------------------------------------


----------------------------------------------------------------------
Ran 4 tests in 0. 009s  运行 4 个用例，用时 0. 009s

FAILED（failures=2， errors=1， skipped=1）
```

```
test_A（__main__. FooTest）
Test routine A ... skipped ’FooTest：test_A：skipped’
test_B（__main__. FooTest）
Test routine B ... FAIL
test_C（__main__. FooTest）
Test routine C ... ERROR
test_D（__main__. FooTest）
Test routine D ... FAIL

======================================================================
ERROR： test_C（__main__. FooTest）
Test routine C
----------------------------------------------------------------------
```

注意 sFEF 行被四对代码行替换掉了，每对命名测试用例和测试程序，最终测试状态，每个程序的 docstr 标注和任意断言消息。

第二个输出结果类型是 TestResults 对象。这个对象是在所有测试运行完之后由运行器对象返回的。

Fo==oResult = fooRunner. run（fooSuite）

TestResult 对象有两组属性存取器。（图 6）

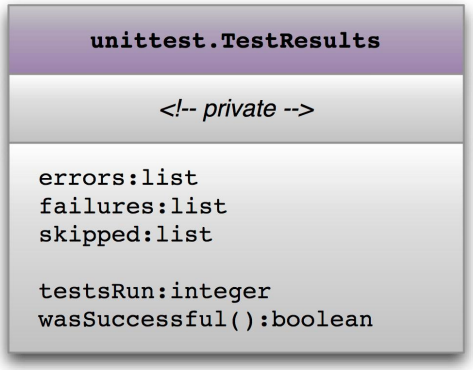


图 6：TestResult 对象（部分）

这不是一个完整的存取器集合，但是这些是你将最可能用到的。第一个集合返回一个数组列表。每个数组显露每个测试程序在执行时运行的怎样。错误存取器列出被识别的抛出异常的程序。每个数组都有测试用例和测试程序的名字，测试脚本的位置，错误的行位置，跟踪位置和错误原因。

失败存取器列出失败的测试程序。其数组包含与错误存取器相同的数字信息。跳过存取器列出那些跳过的程序，有条件跳过或者无条件跳过的程序。其数组命名那些测试用例和程序，并给出跳过的原因。

第二个存取器集提供了辅助的数据。testsRun 给出运行过的测试程序个数，不管结果是什么。wasSuccessful()返回一个 True 值，如果所有的程序都正确无误地运行完的话，反之返回一个 False 值只要有一个程序出问题的话。注意这最后一个存取器是写作一个函数的。

列表十五演示了 TestResult 的工作过程。

列表十五

```
import sys
import unittest

class FooTest(unittest.TestCase):
        """Sample test case"""

# Run the test case
if __name__ == '__main__':
        fooSuite = unittest.TestLoader().loadTestsFromTestCase(FooTest)

        fooRunner = unittest.TextTestRunner()
        fooResult = fooRunner.run(fooSuite)

```

```
14              print
15              print ″---- START OF TEST RESULTS″
16              print fooResult
17              print
18              print ″fooResult：： errors″
19              print fooResult. errors
20              print
21              print ″fooResult：： failures″
22              print fooResult. failures
23              print
24              print ″fooResult：： skipped″
25              print fooResult. skipped
26              print
27              print ″fooResult：： successful″
28              print fooResult. wasSuccessful()
29              print
30              print ″fooResult：： test-run″
31              print fooResult. testsRun
32              print ″---- END OF TEST RESULTS″
33              print
```

这个脚本使用同样的在列表十四定义过的 FooTest 用例。在其触发运行器对象 fooRunner 的 run()方法之后，这个脚本将运行结果存储到本地 fooResults 中。然后触发每个存取器并将测试结果打印在控制台窗口上。

这里是由 fooRunner 返回的测试结果：

```
---- START: Test Results:
<unittest2.runner.TextTestResult run=4 errors=1 failures=2>

fooResult:: errors
[(<__main__.FooTest testMethod=test_C>, 'Traceback (most
recent call last):\n  File "/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 49, in test_C\n
self.assertEqual(fooA, fooB, "A is not equal to B")\nNameError:
global name \'fooB\' is not defined\n')]

fooResult:: failures
[(<__main__.FooTest testMethod=test_B>, 'Traceback (most
recent call last):\n  File "/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 41, in test_B\n
self.assertEqual(fooA, fooB, "A is not equal to B")
\nAssertionError: 123 != 234 : A is not equal to B\n'),
(<__main__.FooTest testMethod=test_D>, 'Traceback (most
```

```
recent call last):\n  File "/Volumes/Projects/Pro_Articles/_ddj/
18_pyUnitTest/ddj18_code/foo_testRun.py", line 55, in test_D\n
self.fail("FooTest: test_D: fail_")\nAssertionError:
FooTest: test_D: fail_\n')]

fooResult:: skipped
[(<__main__.FooTest testMethod=test_A>, 'FooTest: test_A: skipped')]

fooResult:: successful?
False

fooResult:: test-run
4
```

第二行总结了测试结果，显示了测试程序的总数目和出错或者失败的程序数。下面这一段显露了错误的程序：test_C()。也揭露了导致该错误的原因：未定义的变量 fooB。

接着的一段文本揭露了失败的程序：test_B()和 test_D()，揭示了为什么 test_B()失败：两个不相等的值传递给 assertEqual()。也揭示了 test_D()明确地调用了实例方法 fail()。

下面展示的是跳过的程序，test_A()。接下来的代码行展示遇到的问题，最后两行报告共有四个测试程序运行，确认一开始汇报的内容。

结论

Python 有大量的资源能做单元测试。在本文中，我详细地介绍了 unittest 模块，检查那些对单元测试至关重要的类。展示怎样创建一个测试用例，怎样设计测试程序，以及怎样将几个测试用例集合进一个测试套。

Unittest 模块不仅仅可以运行基本的测试。用它，我们可以设计那些关注异常的测试或者做模式匹配的测试。我们可以装载，执行存储在分散的不同的文件中的测试用例。我们甚至可以设计使用模拟对象（以模仿著称）的测试用例。但是这些话题是后话。与此同时，“推荐参考”可以提供更先进的性能上的指导。

【测试技术】项目团队自动化

作者：郝强

1. 似曾相识的项目

做了多年的软件开发及软件测试项目，发现貌似大多数项目过程都是极其曲折，而且多数失败的项目极其类似，虽然实践过多种软件项目管理的方法学，但貌似大家都被工期，成本以及人员的能力限制着，除了疯狂的加班，以及救火队员式的服务着我们的项目，最终历经九九八十一难后项目上线了，大家也觉得可以松口气了，但发现上线后要保证线上应用稳定，也不是一件容易的事，貌似这事就没有尽头了。

我在之前的项目中就曾遇到过许许多多的问题，现简单列举一下：

1. 版本控制混乱，尤其是多个产品线共同存在时，特别是多分支的版本库。
2. 测试人员可支配的测试时间极少，人员也严重不足，一个项目基本就一两个人，多数的时候上线前只有不到一天的测试时间。
3. 手工多环境部署，重复劳动多，经常出现少配置这项参数，或那项参数的问题。
4. 大家都苦不堪言，团队凝聚力差，核心人员渐渐流失。

上述我只列举了几个比较重要且是核心问题，具体细节上的问题更多，再这就不在一一列举。对于上述问题，我一直也在项目中思索如何解决这问题。

2. 理想与现实离的不远

2.1 一个成功的项目状态

某年某月的某一天，在某个项目中，经过大家共同的努力， 那些似曾相识的糟糕的状态已经一去不复返，同时大家还积累一些方法与实践，通过努力，项目达到了如下的状态：

1. 版本控制有序。
2. 测试方面有了有效的自动化测试，手工人员可以关注在更重要的测试任务里。
3. 部署团队实现一键部署，多个环境共同一套部署脚本。
4. 任一环境出现失败，均有邮件告警。
5. 收到告警邮件时，有人立马响应并解决问题，形成了交付的生态圈。
6. 整个开发，测试，部署，运维是相辅相成并且做为整体的自动化呈现的。

7. 貌似即使有加班，大家也感觉很兴奋。

2.2 理想是如何变成现实的

理想与现实之间的距离有时隔的很远，当你为之努力的时候，它们之间的距离又离的如此之近。

当你遭遇到一个个糟糕的项目的时候，你是否想过如何改变它？如果你试想过，又努力过，那么方法是什么？你是否想听听我曾经的项目经验。

工作十几年来，可以说做了差不多几十个项目，那些失败的项目经历与前边提到的似曾相认的项目总是大同小异，然而我们也经历了例外，从糟糕的项目到最终改变它，让他处于一个“成功”的状态。你一定想问，到底是怎么做到的？说来话长，简而言之，三个要点。第一，对于所有糟糕的部分，就要使用最笨的办法，即手工的方式，先要稳定住，不要让它再往更坏的方向去发展。第二，如果第一点达到了，我们要想办法提高已经稳定住的部分的效率。对，你已经想到了，尽可能的让它能够自动化。第三，将整个项目生命周期的每个部分都自动化，达到整体自动化，即要几乎将所有事情都自动化。

接下来，我会从配置管理，测试过程，部署发布，以前开发过程等方面来讨论如何让梦想成为现实。

2.3 配置管理

把配置管理放在第一位来说，是要通过一系列的过程来保证所有与之相关的产出物以及它们之间的关系 都被唯一定义、修改、存储和检索。

您所在的项目可能是多产品线同时开发，且产品关联性很强，版本库也是多个分支同时存在，每次发布可能会基于某个分支发布或是全部合并到主干进行发布，如何能够做到有效的配置管理是极其关键的。

所以我们必须将所有内容进行版本控制，并管理其依赖关系，这其中包括不仅仅是源代码，还要对所有软件的配置信息进行版本控制，同时还包括对整个环境的配置管理。，即对应用程序所依赖的软件、硬件和基础设施进行配置管理。

对于配置管理，我们需要对二进制文件与配置文件进行分离，同时要将所有配置信息保存在一处，达到高效的目的。

实现高效的配置管理的目的是为了后续能够实现部署流水线，并确保软件任何时候都可以工作，达到整个交付过程几乎全部自动化的目的。只有将配置管理部分做好，才能保证后续测试过程自动化，部署过程自动化以及保证持续集成的有效性成为可能。

2.4 测试过程

对于测试过程自动化的考虑，我们首先要求项目团队在开发过程即要考虑系

统的可测试性，要保证尽可能高的单元测试覆盖率，尤其是敏捷项目实行的测试驱动开发。另外，对于 CS 结构的应用来讲，尽可能使用标准控件，以减少后期实现自动化测试时对非标准控制要做过多编程或处理。对于 BS 结构应用来讲，在没有特殊要求的情况，尽可能不使用浏览器插件。

在项目初始阶段，测试经理需要与项目经理沟通，要把握和实施好质量内嵌原则。其原则要从多个层次(即单元测试、组件测试和验收测试)上写自动化测试，并将其作为部署流水线的一部分来执行。每次应用程序的代码、配置或环境以及运行时所需软件发生变化时，都要执行一次该测试。同时手工测试也是质量内嵌的关键组成部分。当然质量内嵌还意味着，你需要不断地改进你的自动化测试策略。

实施自动化测试，要先尽可能的先自动化接口部分。因为这部分功能的自动化测试更容易实现。针对于每次构建过程，有构建被触发时，要能够触发自动化冒烟测试，这部分也是需要你优先考虑自动化的部分。因为你要验证每次发布是否有效，如果有效才能执行更大量的测试。针对于 UI 及回归测试的自动化实现，是放在最后考虑的，因为这部分很可能是变动最频繁和需要花人力去经常维护的。当然，创建回归测试套件的价值在于保证系统当前的功能正确，将这部分自动化放在最后考虑不是说其不重要，实际上它是同等重要的，只是优先级较低。

2. 5 部署发布

针对部署发布过程，最重要的是一定要实现部署流水线，即软件从版本控制库到用户手中这一过程的能够自动化。为了能够实现部署流水线，我们必须分析所在项目的整个流程，对标准流程进行分析，并创建一个可工作的简单框架实现。之后，我们需要对构建过程部署流程实现自动化。对于开发过程，我们要实现单元测试和代码分析自动化。对于测试过程，也必须要将验收测试自动化，最后我们要将发布过程实现自动化，这样一来，一个完整的部署流水线就形成了。

为了保证部署流水线有效，我们只生成一次二进制包，并对所有不同环境采用同一种部署方式，即部署方式是重复的。另外，我们需要对部署进行冒烟测试（自动化），以保证其有效。在向生产环境部署前要向生产环境的副本中部署。如果在整个部署流水线过程有变更出现，那么每次变更都要立即在流水线中传递，如果一旦有某个环节失败，就停止整个流水线，有相关人员立即介入解决，保证有效反馈和快速解决问题，让流水线能够正常运转。

2. 6　使用持续集成

从项目管理的角度来讲，使用持续集成是必须的，因为它能够加速你的交付，即使你的项目还未使用敏捷的方法，在开始持续集成之前，必须要保证项目中应用了版本控制系统，前边已经讲过其重要性。另外要实现自动化构建，最重要的

一点，整个团队要达成共识，不允许在持续集成过程中出现邮件告警时无人理，必须要做到快速反馈。

持续集成有效的前提，就是要频繁提交代码主干，然后要创建全面的自动化测试套件 。持续集成中会使用三类自动化测试：单完测试自动化套件，组件测试自动化套件和验收测试自动化套件。另外要保持较短的构建过程和测试过程。

在使用持续集成时，如果构建失败了，就不要提交新代码，开发人员先修复它。开发人员在提交代码前要在本地运行所有的提交测试，或让持续集成服务器完成此事。之后要运行提交测试，等提交测试通过后再继续工作。大家下班回家之前，构建必须处于成功状态。无论何时，我们必须时刻准备着回滚到前一版本或是必须有能力回滚到前一版本。在回滚之前要规定一个修复时间 （推荐 10 分钟之内）。在过程中如果发现有测试未通过，请不要将失败的测试注释掉。要立即检查失败的测试并解决相关问题。项目团队的中每一个要为自己导致的问题负责。

更多精彩内容请见 原创测试文章系列（三十三）（上篇）