目录 | (六十三期·上)

投稿邮箱：editor@51testing.com

Fiddler 也能像 Burp 一样使用了?

◆ 作者：王昌

Fiddler 作为一款 Web 调试工具，功能强大的同时还提供免费版本给用户使用，可以记录客户端和服务器之间的所有 HTTP/S 请求，针对特定的应用场景，提供了分析请求数据、设置断点、调试 web 应用、修改请求等功能。

Fiddler 的本质就是“中间人”，负责在浏览器将请求上送至服务器前，对请求进行拦截，修改后将请求发送至目标服务器；服务器收到请求后进行响应，将响应报文传送至浏览器前，Fiddler 进行响应拦截，将响应报文篡改后，再传给浏览器。Fiddler 的工作原理赋予了其在安全渗透测试场景下的能力，通过截获 HTTP/S 请求，将可变请求参数替换为攻击向量，通过特定的规则库匹配服务器的响应报文进一步判断是否存在安全问题。

Fiddler 也只是满足安全测试的基本条件，但是当面对自动化 Fuzzing 场景难免有些乏力。如何改造 Fiddler 以满足我们的需求呢？Fiddler 的可拓展性为我们提供了可能，用户可以定制化 Burp-Like Inspector 插件来实现自动化 Fuzzing 测试。接下来我们将着重介绍一下这个插件的用法。

一、下载和安装

Burp-Like Inspector 插件并不在 Fiddler 的官方插件库内，而是作为第三方插件方便安全渗透测试人员定制。该插件的下载地址为：http://yamagata.int21h.jp/tool/BurplikeInspector/，最新的版本为 BurplikeInspector-ver0_02.zip。将其解压后，可以看到插件是一个 dll 文件，需要将该文件导入到指定文件路径下才可以使用，该路径为：%userprofile%\Documents\Fiddler2\Scripts。

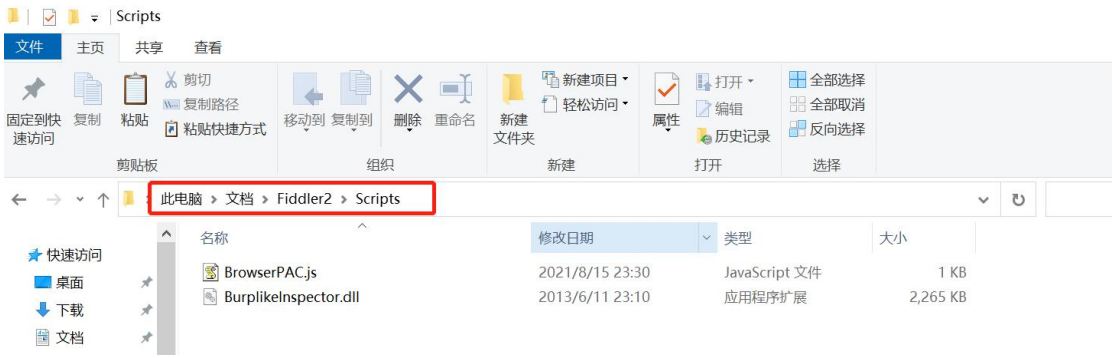


图 1 插件导入到制定文件夹下

至此，启动 Fiddler 会自动弹出插件的对话框，若不小心将其关闭，可以在 Rules 菜单栏中找到该插件的功能入口。

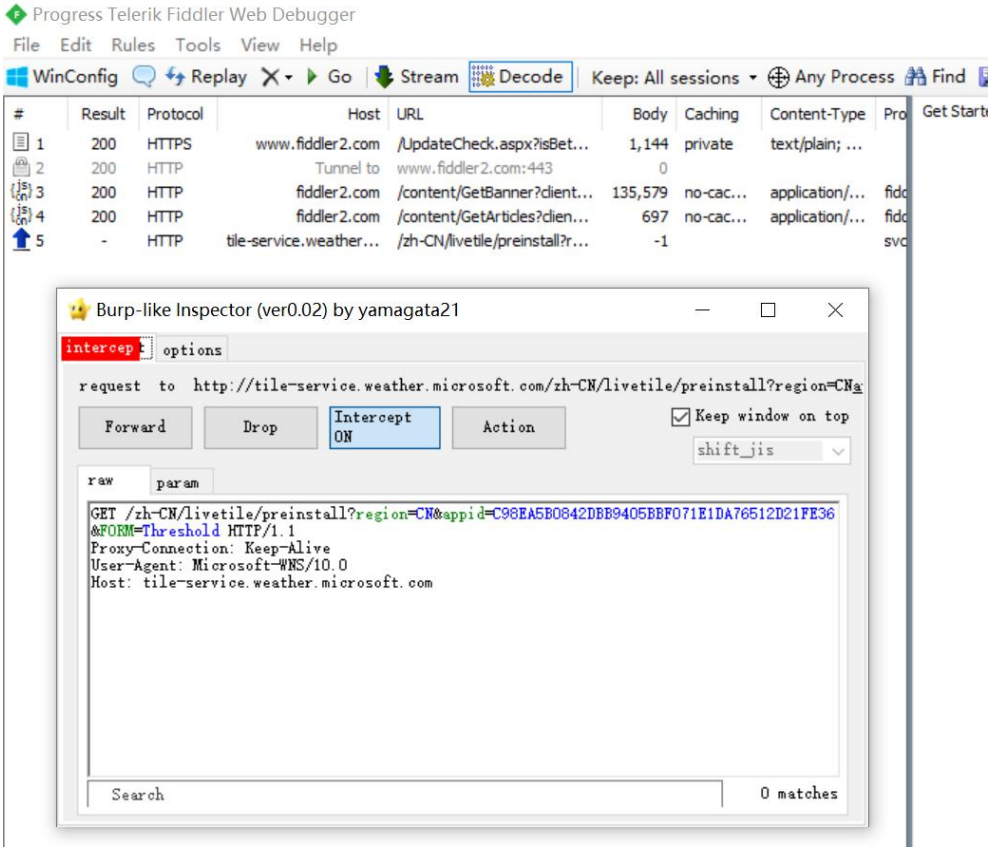


图 2 安装插件后 Fiddler 初始化页面

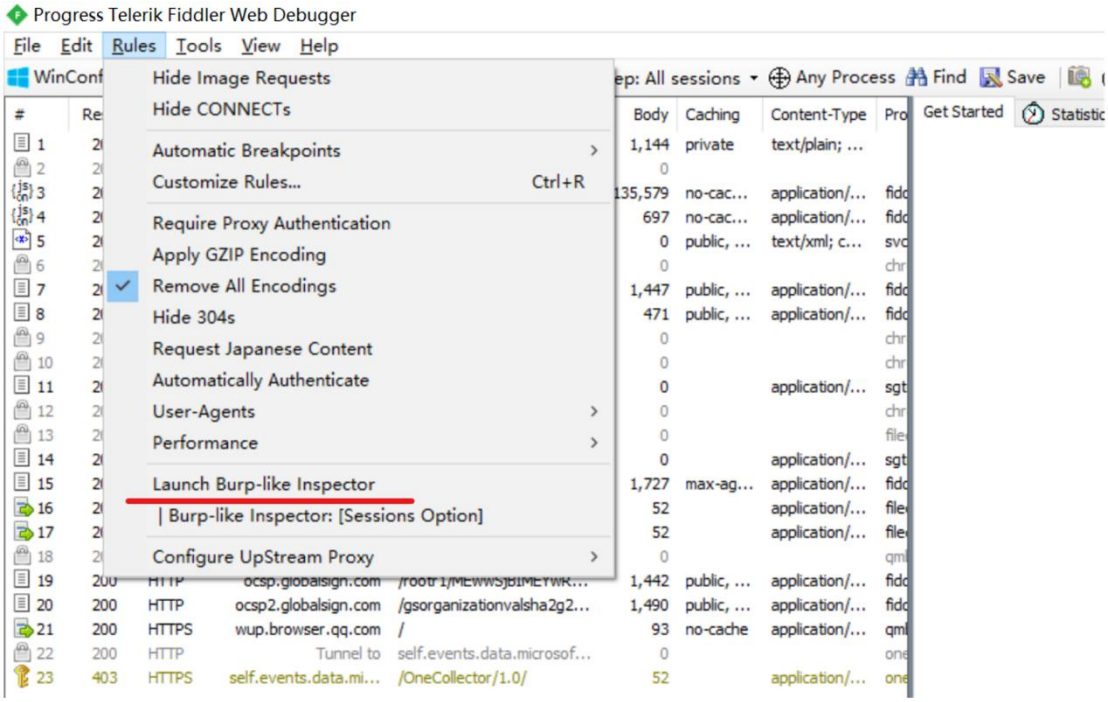


图 3 插件可以在 Rules 菜单栏中重新唤起

二、插件使用

该插件模仿了 Burpsuite Proxy 工具的 Intercept 标签页的界面布局及功能效果。

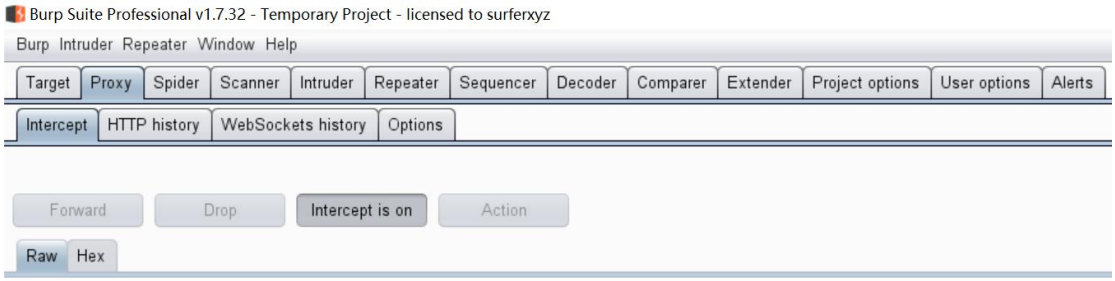


图 4 Burpsuite 工具的 Intercept 标签页界面效果

在安全渗透测试中经常需要对请求报文的参数进行替换并重放，以及在面对大量 Payloads 时需要使用自动化模块进行爆破操作，该插件刚好解决了原生 Fiddler 在重放、爆破、解码等安全渗透测试场景的不友好、不易用的问题。

下面就该工具的功能进行详细的介绍。

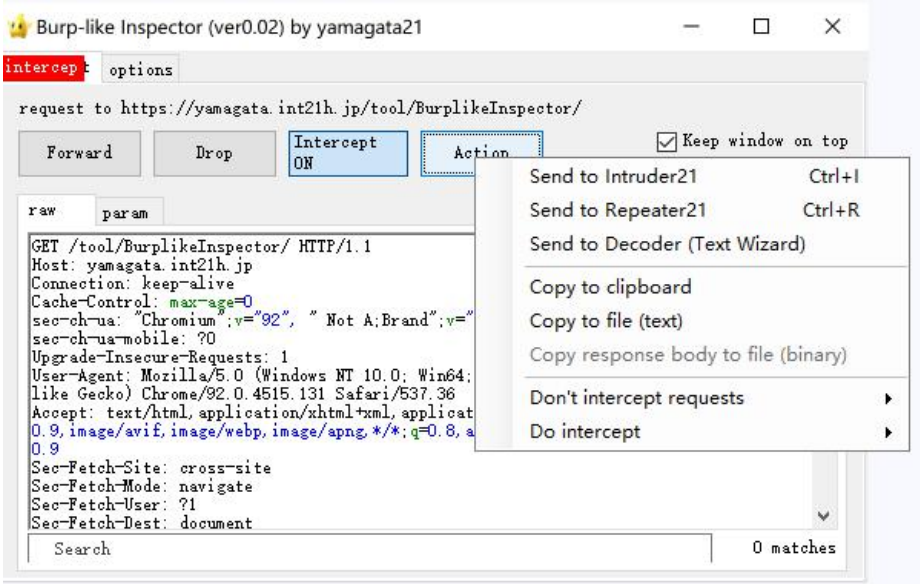


图 5 BurplikeInspector 插件的界面效果

1、Intercept 标签页

（1）、该插件可根据用户的需求进行开启和关闭，开启时开关状态为“Intercept On”表示正在拦截请求报文；关闭时开关状态为”Intercept Off“表示不再拦截请求报文，直接放行。

（2）、当截获请求报文后，Forward 和 Drop 按钮将高亮，两个按钮对应不同的处理动作，Forward 表示将截获的请求放行至服务器；Drop 表示丢弃该请求数据报文，服务器将收不到客户端发起的该请求。

（3）、Action 按钮，针对当前截获的报文发送到其他的处理模块进行改造，处理模块主要包括重放、爆破、解码等。

2、repeater 模块

用户在截获请求并发送到 repeater 模块后，可以对该请求进行重放操作，用户可以任意修改其中的请求方法、请求头、请求体等内容，编辑完成后可以点击“Go”按钮，将修改后的请求发送至服务器，在窗口下方可以查看服务器的响应报文，同时在左侧列表中会生成一条该请求记录。另外，在渗透测试过程中可能会遇到一个功能可能需要多步请求的情况，这时需要定制一个请求序列，可以使用该功能将多个请求定制为会话宏。

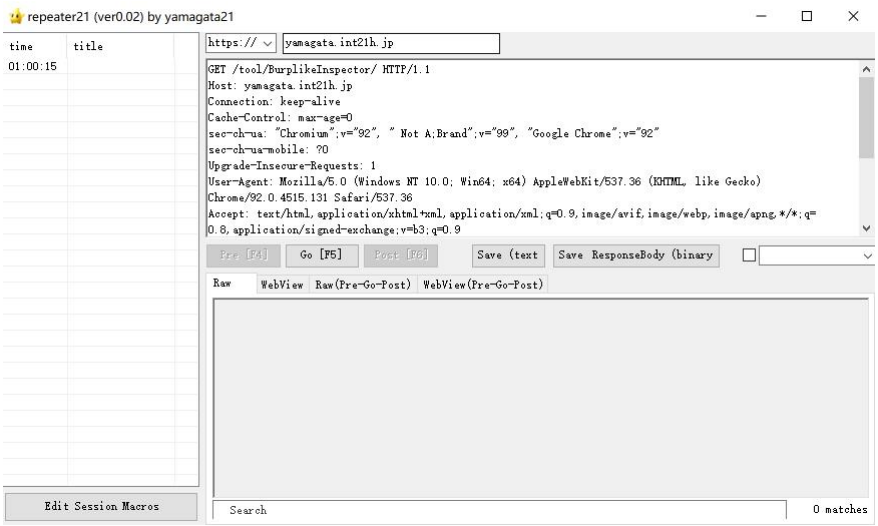


图 6 repeater 模块

3、Intruder 模块

用户在截获请求并发送到 Intruder 模块后，可以对该请求进行 Fuzzing 操作，用户需要对该请求的某个参数进行批量替换并发送至服务器。前提是用户已经构造了用于替换原始参数的攻击向量，我们将其称之为 payloads，可适用于口令爆破、手机号/银行卡号枚举等场景。

首先，选定需要 Fuzzing 的请求参数。Intruder 模块会自动识别请求中的动态参数，并为其前后缀添加$符号以表示对改参数进行 payloads 替换。

图 7 Intruder 模块 Target 标签页

其次，加载 Payloads 进行替换。Payloads 标签页会自动加载初始的 payloads，Comment 表示对每一条 payload 标注备注，Signature 表示 payload 的具体参数，用来替换原始请求中使用$符号标注的参数，SearchFor 表示在响应页面中搜索关键词，若可以搜索到，将在 Results 标签页中高亮显示。同时还可以通过设置进程数来控制 QPS，防止请求过于频繁导致 IP 封禁。

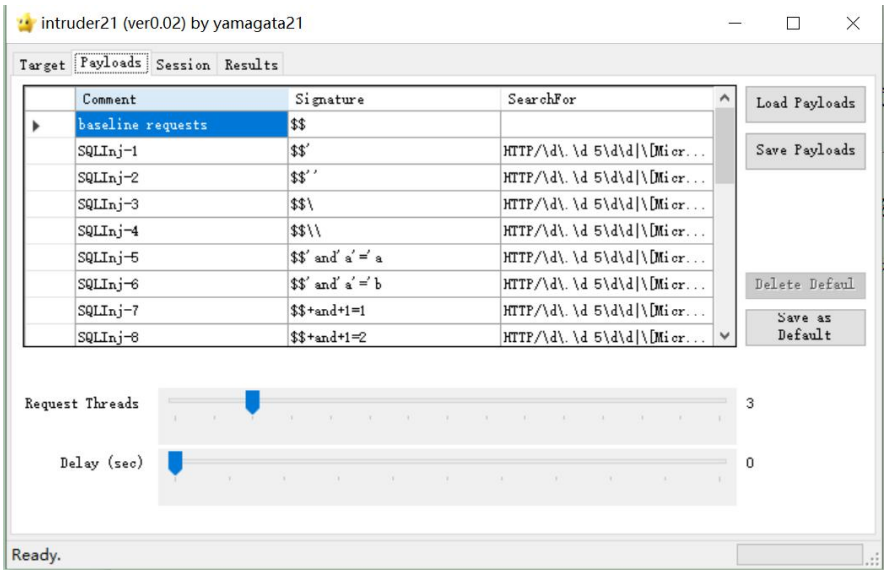


图 8 Intruder 模块 Payloads 标签页

最后，批量执行请求。在 Results 标签页，点击“Start Test”，即可完成 Fuzzing 测试。

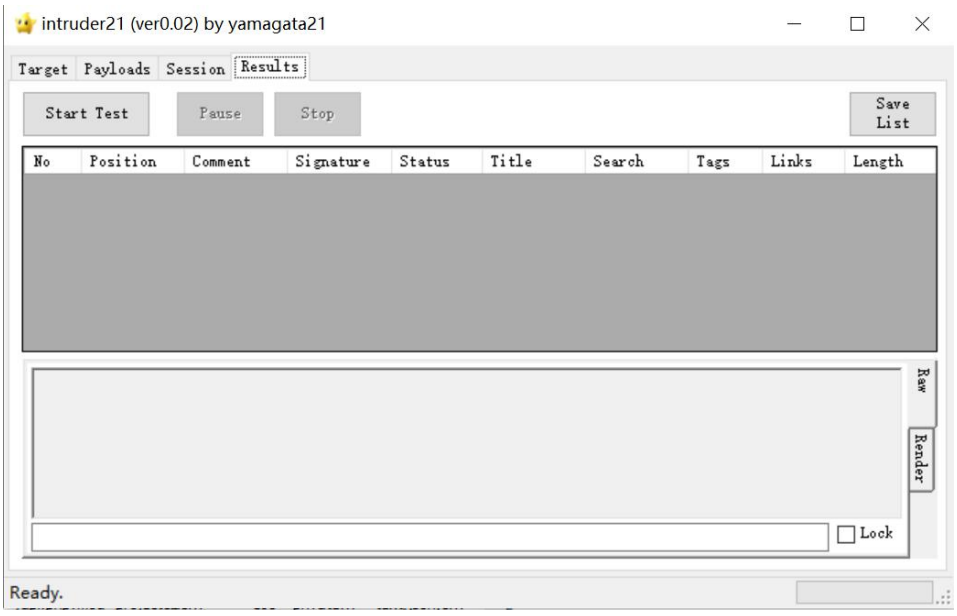


图 9 Intruder 模块 Results 标签页

GoogleTest 单元测试框架总结（上）

◆作者： Ron

在正式开启 GoogleTest 之旅前，先介绍一点术语，以便平滑过渡。

黑盒和白盒

所谓黑盒，将待测对象看成一个黑盒子，测试时不关心功能是如何实现的，仅关注输入数据和实际输出结果，核对实际输出是否与预期输出一致。功能测试采用的就是黑盒测试技术，如下就是一条手机的功能测试用例，测试步骤对应输入（点击图标）、预期结果对应输出（打开应用）。

预置条件	测试步骤	预期结果
手机已开机	点击天气图标	打开天气应用

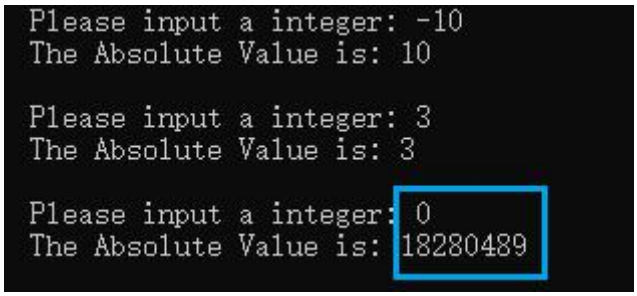
所谓白盒，简单来说就是代码层面的测试。测试人员需要了解功能是如何编码实现的（需要读懂代码）。单元测试就是代码层面的测试，举个栗子。

Absolute( )函数实现了求整数绝对值的功能：

```
1.  int Absolute(int n)
2.  {
3.      if (n > 0)
4.          return n;
5.      else if (n < 0)
6.          return n * -1;
7.  }
```

浏览 Absolute( )函数，可以看出它的代码存在明显 bug：没有考虑输入为 0 的情况。

运行程序，当输入数据为 0 时，输出结果是错误的：



通过上面的示例，相信大家对单元测试有了一个比较直观的印象了，下面给出单元测试的定义。

产品代码中的 一个函数、一个类 或 一个接口均可以看成一个单元，针对这些单元的代码级测试就是单元测试。

单元测试的对象是一个个'单元'，单元测试可以发现待测对象中的代码级故障，对产品整体性错误无能为力。

虽然单元测试有其局限性，但是良好的单元测试可以保障一个单元模块的代码正确性，即：该单元被其他模块调用时，自身是没有代码问题的。

下面介绍 GoogleTest

Google Test 是 Google 公司开发的一款 C++单元测试框架，Google Chrome 浏览器使用的测试框架就是 GoogleTest。对于使用 C++开发的产品，可以通过 GTest 编写单元测试用例进行单元测试。

为什么要编写单元测试用例，上面的 Absolute( )函数进行代码走查不就搞定了吗？

光是走查（没有对应的单元测试用例），当代码变更后，需要重新走查，之前的走查成果无法继承；随着单元模块逻辑复杂度的提升，必须编写测试代码进行代码质量的保障（大神除外）。对于拥有单元测试用例的模块，当此单元进行较大的代码优化后，可以通过已有的单元测试用例快速评估优化后的代码质量、及时发现代码错误。

为什么要使用测试框架，直接单元测试不香吗？

先说结论：不香😂，直接编写测试用例进行单元测试整体效率要低得多：测试框架是对整个测试系统的可重复使用设计，可重复意味着自动化；有了测试框架，测试人员不再需要编写琐碎的测试代码，从而可以专注于测试用例本身，使得测试更聚焦。

好的单元测试框架 及 Google Test 的优势（摘自 GoogleTest Primer.md，你理解为 GTest 自己吹捧自己也没错）：

1. 测试是独立和可重复的：GTest 使每个测试用例运行在不同的对象中从而使测试隔离。当一个测试失败时，GTest 允许你将它运行在隔离的环境下从而达到快速调试的目的。

2. 测试有良好的组织，可以反映被测试代码的结构：GTest 将相关测试划分到一个测试组内，组内的测试能共享数据，使测试易于维护。

3. 测试可移植和可复用：与平台无关的代码，其测试代码也应该和平台无关，GTest 能在不同的 OS 下工作，并且支持不同的编译器。

4. 当用例执行失败时，可以提供尽可能多的有效信息，以便定位问题。GTest 可以定制出错时的有效信息，使得更容易定位故障。

5. 测试资源可以复用：GTest 能在测试用例之间复用测试资源，使单元测试更高效。

啰嗦了这么久，下面正式开始：

阅读本文的收益

1. 熟悉 GoogleTest 单元测试框架

2. 掌握如何通过 GoogleTest 进行基础的单元测试

流畅阅读本文的前置条件

1. 熟练掌握 C++

2. 良好的英语阅读能力

主要内容

0.编写第一条单元测试用例

1.对已有项目展开测试

2.函数的单元测试

3.类的单元测试

4.Test Fixture 的使用场景

5.Test Fixture 的继承和差异化

6.接口的单元测试

由于内容比较多，分为上、下两篇，上篇主要介绍了 基础知识、环境配置和函数的单元测试，下篇包含剩余内容。所有示例均已在 Windows10 + Visual Studio 2019 上调试通过，函数到接口的单元测试示例为 GoogleTest 的附带示例，作者进行了标注并结合实际情况进行了少量改动。

零、编写第一条单元测试用例

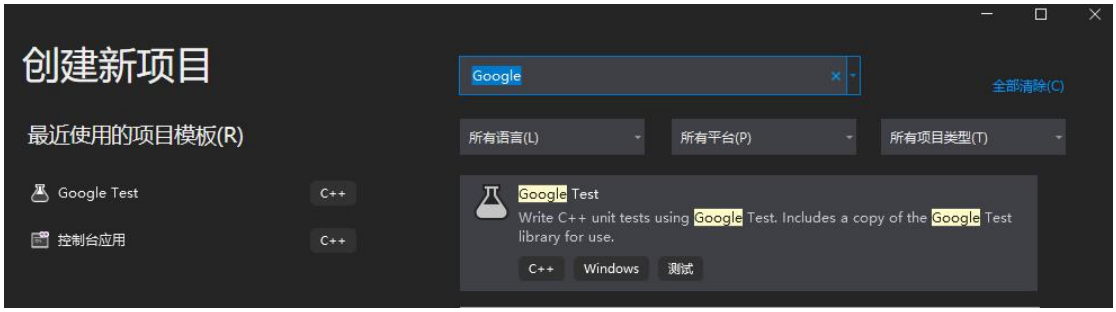
测试环境

Windows10 + Visual Studio 2019（vs2019 已集成 GTest）

创建项目

1. 打开 VS 2019，创建新项目。

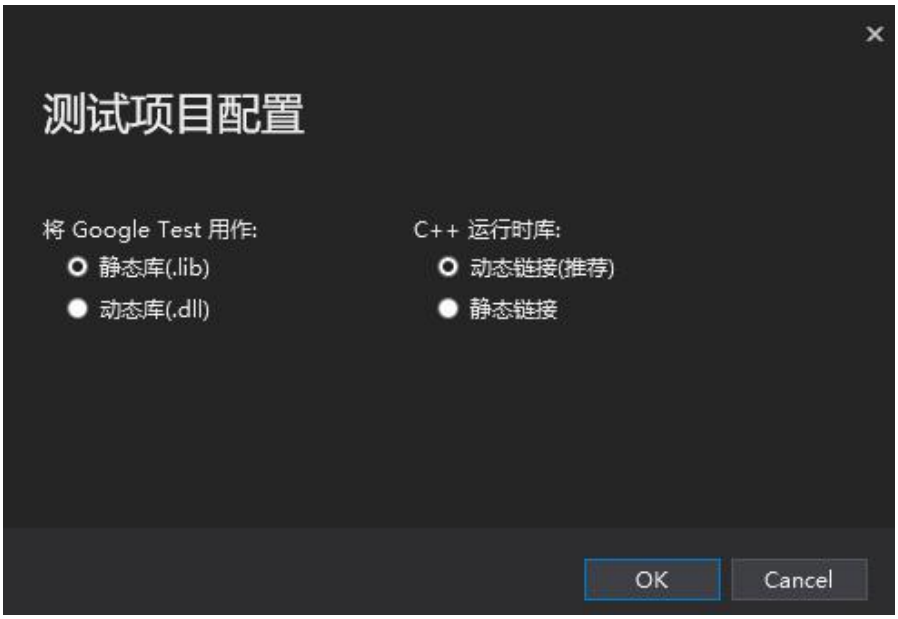
2. 搜索关键字 Google，可以得到 Google Test 的项目模板：Write C++ unit tests using Google Test。



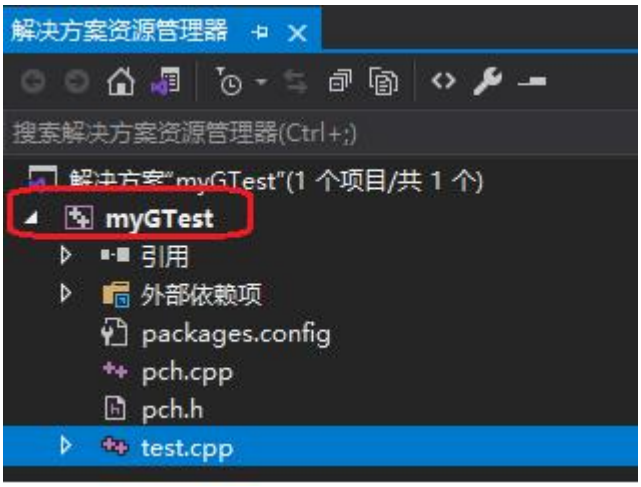
3. 选择项目模板 Google Test，点击下一步。

4. 配置项目名称和位置，点击创建。

5. 测试项目配置如下。



6. 等待 VS 创建好新项目：如图所示，创建好了名为 myGTest 的项目。



编写单元测试用例

1. 在 test.cpp 中编写函数 Factorial() ，并编写 Factorial( ) 对应的单元测试用例（Factorial( )函数用于计算整数 n 的阶乘）。

```
#include "pch.h"

int Factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }

    return result;
}

TEST(TestFactorialFunc, FirstGTest)
{
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(24, Factorial(4));
}
```

2. 观察测试用例：TEST(TestFactorialFunc, FirstGTest) ：

（1）编写测试用例使用了 TEST 宏，它有两个参数：TEST[TestCaseName，TestName]，TestCaseName 对应测试用例集名称，TestName 是归属的测试用例名称。

（2）对检查点的检查，使用了 EXPECT_EQ 宏，用来比较两个数字是否相等。可以看到，通过 GTest 编写用例还是蛮方便的。

Google 打包了一系列 EXPECT_* 和 ASSERT_* 宏，EXPECT 和 ASSERT 的区别：EXPECT_* 失败时，测试用例继续执行；ASSERT_*失败时，同一用例后面的语句将不再执行。

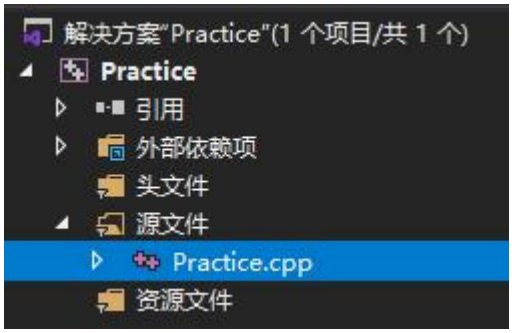
执行测试用例

```
Running main() from c:\a\_work\32\s\thirdparty\googletest\googletest\src\gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from TestFactorialFunc
[ RUN      ] TestFactorialFunc.FirstGTest
[       OK ] TestFactorialFunc.FirstGTest (0 ms)
[----------] 1 test from TestFactorialFunc (2 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran. (5 ms total)
[  PASSED  ] 1 test.
```

一、对已有项目展开测试

创建待测项目配套的 Google Test 项目

1. 在 VS2019 中创建新项目 Practice（待测项目）



其中的 Practice.cpp 包含 main( )函数

```
#include <iostream>
using namespace std;

int main()
{
    cout << "Start Google Test…\n";
    return 0;
}
```

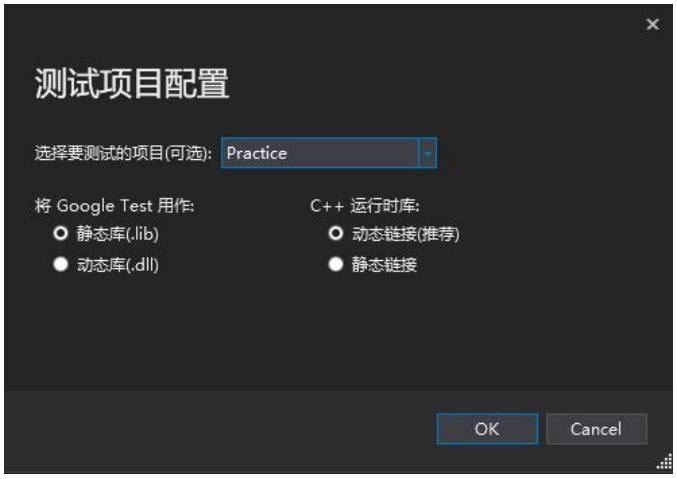
2. 创建 Practice 项目配套的 gtest 项目

（1）文件-新建-项目，打开 创建新项目 窗口。

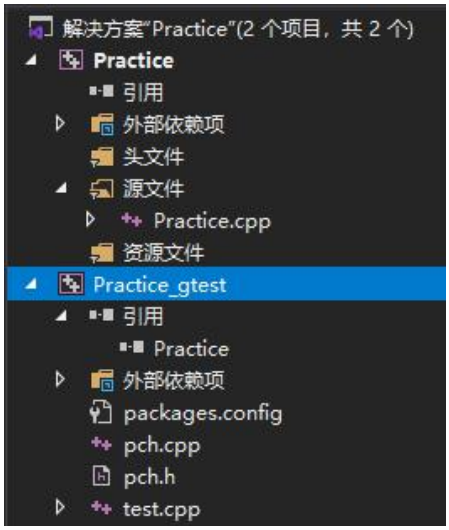
（2）选择创建 Google Test，然后点击下一步。

（3）配置新项目 窗口 ：项目名称自定，解决方案选择：添加到解决方案。然后点击创建。

（4）选择要测试的项目后，点击 OK



（5）如图所示，已经创建好了 Practice 项目 对应的 gtest 项目 practice_gtest



设置项目之间的依赖关系

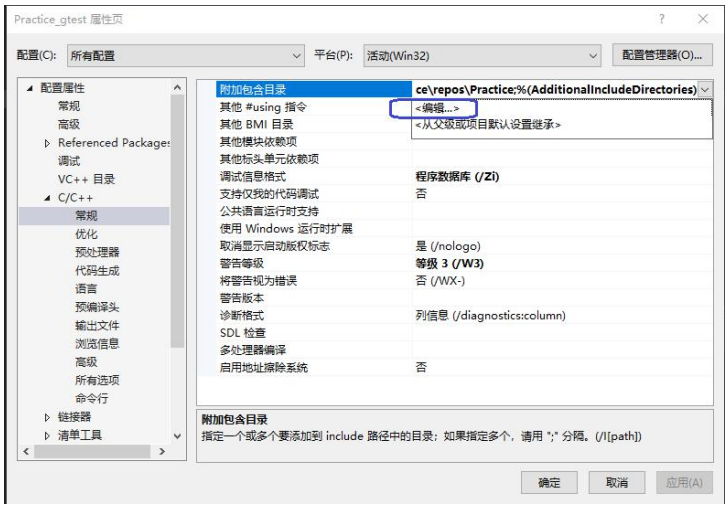
1. 设置项目之间的依赖，可以简化头文件的路径描述

```
1. //#include "../Practice/Calc.h"    //设置依赖前
2. #include "Calc.h" //设置依赖后
```

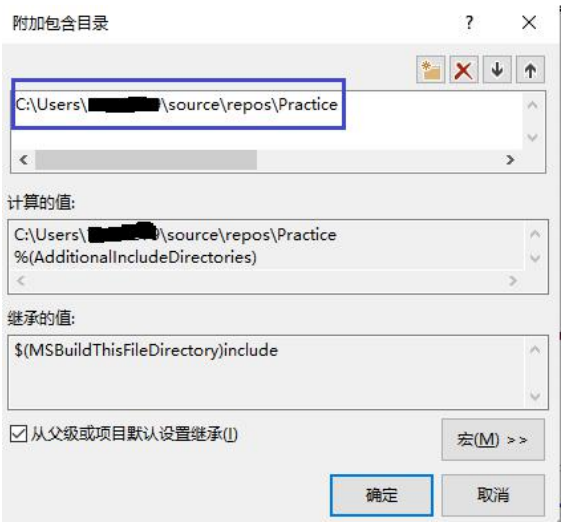
2. 设置依赖关系

(1) 右键点击 Practice_gtest，选择 属性。

(2) 在属性页中，依次定位到：配置属性 --- C/C++ --- 常规 --- 附加包含目录，点击下拉框后选择编辑。

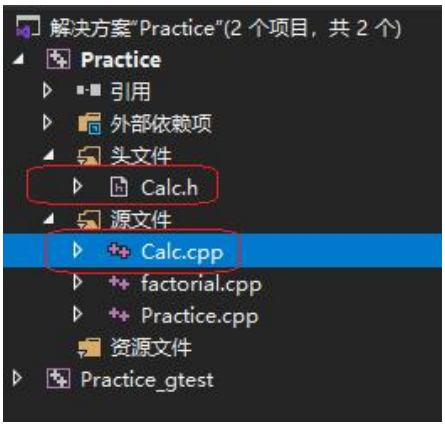


(3) 在弹出的附加包含目录窗口中设定依赖关系：输入待测项目的目录地址。



对已有项目展开测试

1. 在 practice 项目中新建文件 Calc.h、Calc.cpp，用于模拟加减乘除运算



Calc.h 的内容

```
1.  class Calc
2.  {
3.  public:
4.      int  Add(int a, int b);
5.      int  Minus(int a, int b);
6.      int  Multi(int a, int b);
7.      float Divide(float a, float b);
8.  };
```

Calc.cpp 的内容

```
1.  #include "Calc.h"
2.
3.  int Calc::Add(int a, int b)
4.      return a + b;
5.
6.  int Calc::Minus(int a, int b)
7.      return a - b;
8.
9.  int Calc::Multi(int a, int b)
10.     return a * b;
11.
12. float Calc::Divide(float a, float b)
13.     return a / b;
```

2. 在 gtest 测试工程中新建 ClassCalcTest.cpp，用于测试类 Calc。

这里使用的还是 TEST 宏，测试用例分别对应代码中的加减乘除。

一个问题，为什么测试对象是 Calc.h，而不是 Calc.cpp？因为调用的是接口(*.h 头文件)，而不是实现（*.cpp 文件）。

```
1.  #include "pch.h"
2.  #include "Calc.h"
3.
4.  Calc calculation;
5.
6.  TEST(CalcClassTest, add)
7.  {
8.      EXPECT_EQ(3,calculation.Add(1, 2));
9.  }
10.
11. TEST(CalcClassTest, minus)
12. {
13.     EXPECT_EQ(calculation.Minus(1, 2), -1);
14. }
15.
16. TEST(CalcClassTest, multi)
17. {
18.     EXPECT_EQ(calculation.Multi(1, 2), 2);
19. }
20.
21. TEST(CalcClassTest, devide)
22. {
23.     EXPECT_FLOAT_EQ(calculation.Divide(1, 2),0.5);
24. }
```

3. 正式启动测试前，需要先设定好对应目标文件的地址：

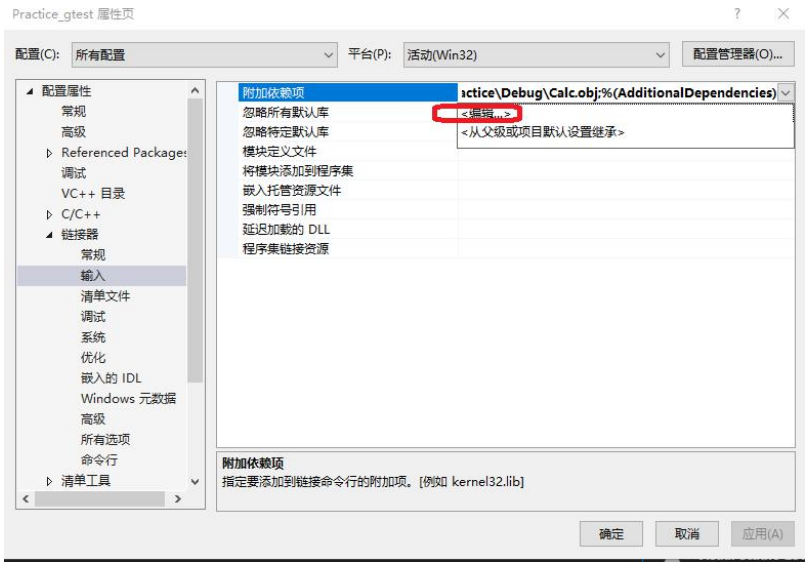
(1) 进入 测试工程 Practice_gtest 的属性设定界面(右键点击项目，在弹出菜单中选择属性)。

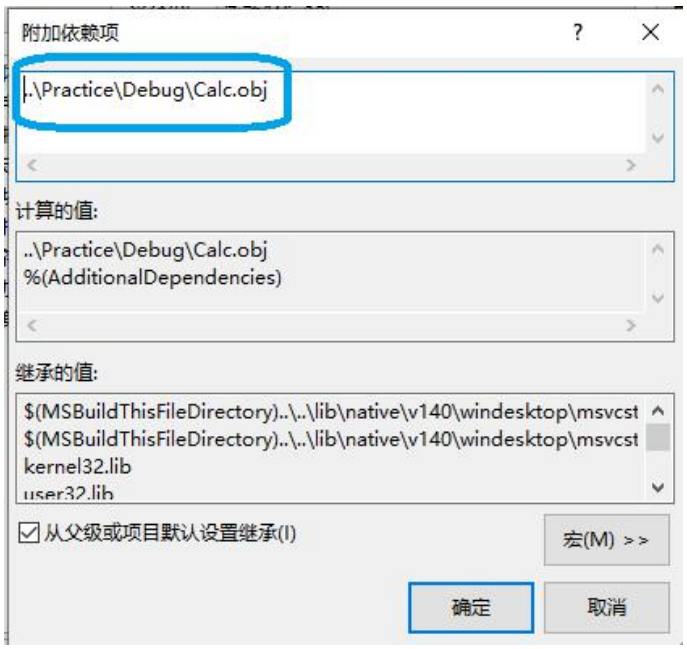
(2) 依次定位到链接器-输入-附加依赖项，点击下拉框，进行编辑。

(3) 输入类 Calc 的目标文件 Calc.obj 的地址。

注意：当测试多个类时，需要分别添加对应的 obj 文件，该场景下不能使用通配符 * ，

否则执行测试时会报错。





4. 打开测试资源管理器，执行测试

二、函数的单元测试

代码文件是 sample01.h， sample01.cpp，对应的单元测试文件是 sample01UnitTest.cpp

sample01.h 进行了函数声明。

```
1.  // Returns n! (the factorial of n).  For negative n, n! is defined to be 1.
2.  int Factorial(int n);
3.
4.  // Returns true if and only if n is a prime number.
5.  bool IsPrime(int n);
```

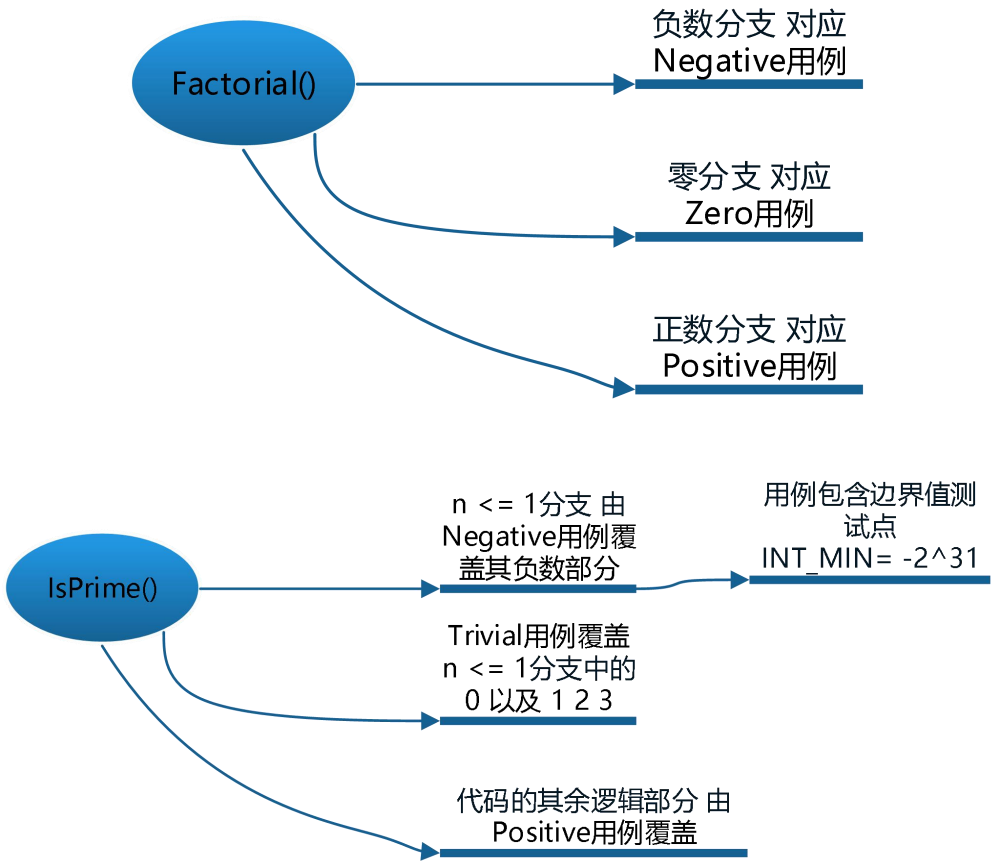
sample01.cpp 中撰写了待测试的函数（对应的接口文件是 sample01.h）：Factorial( ) 函数用于求一个数的阶乘，IsPrime( )函数用于判定一个数是否是素数。

```
1.  #include "sample01.h"
2.
3.  // Returns n! (the factorial of n).  For negative n and zero, n! is defined
    to be 1.
4.  int Factorial(int n)
5.  {
6.      int result = 1;
7.
8.      for (int i = 1; i <= n; i++)
9.          result *= i;
10.
11.     return result;
12. }
13.
```

```
14. // Returns true if and only if n is a prime number.
15. bool IsPrime(int n)
16. {
17.     if (n <= 1) return false;
18.
19.     if (n % 2 == 0) return n == 2;
20.
21.     for (int i = 3; ; i += 2)
22.     {
23.         if (i > n / i) break;
24.         if (n % i == 0) return false;
25.     }
26.
27.     return true;
28. }
```

注意，编写单元测试用例时，你的重点并不是读懂每一行原码，而是弄清楚这个单元对应的主体功能、调用方式 以及 代码的逻辑构成，你的目标是：编写的单元测试用例可以覆盖所有的代码逻辑分支。

在单元测试用例 sample01UnitTest.cpp 中，可以看到他覆盖了待测函数的各个分支（请参考下图）



大家可以通过阅读下面的源码和注释加深理解：

```
// Tests Factorial().

// Tests factorial of negative numbers.
TEST(FactorialTest, Negative) // This test is named "Negative", and belongs to the "FactorialTest" test case.
{
    EXPECT_EQ(1, Factorial(-5));
    EXPECT_EQ(1, Factorial(-1));
    EXPECT_GT(Factorial(-10), 0);  //GT mean Great Than
}

// Tests factorial of 0.
TEST(FactorialTest, Zero)
{
    EXPECT_EQ(1, Factorial(0));
}

// Tests factorial of positive numbers.
TEST(FactorialTest, Positive)
{
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(6, Factorial(3));
    EXPECT_EQ(40320, Factorial(8));
}


// Tests IsPrime()

// Tests negative input.
TEST(IsPrimeTest, Negative)
{
    EXPECT_FALSE(IsPrime(-1));
    EXPECT_FALSE(IsPrime(-2));
    EXPECT_FALSE(IsPrime(INT_MIN));
}

// Tests some trivial cases.
TEST(IsPrimeTest, Trivial)
{
    EXPECT_FALSE(IsPrime(0));
    EXPECT_FALSE(IsPrime(1));
    EXPECT_TRUE(IsPrime(2));
    EXPECT_TRUE(IsPrime(3));
}

// Tests positive input.
TEST(IsPrimeTest, Positive)
{
    EXPECT_FALSE(IsPrime(4));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_FALSE(IsPrime(6));
    EXPECT_TRUE(IsPrime(23));
}
// Note : For this sample(vs 2019) , you just need open "Test resource manager" and execute the start, don't need Call RUN_ALL_TESTS() in main().
```

类测试、接口测试 以及 Test Fixture 的使用将在下篇介绍。

To Be Continue…

GoogleTest 单元测试框架总结（中）

◆ 作者：Ron

前导文档：GoogleTest 单元测试框架总结（上）

本篇介绍 类的测试 和 Test Fixture

一、类的测试

我们先从一个简单的计数器开始：

1.1 头文件 Counter.h 仅定义功能，不进行具体实现：增加计数 Increment()?、减少计数 Decrement()、打印 。

```
1.  class Counter
2.  {
3.  private:
4.      int counter_;
5.
6.  public:
7.      // Creates a counter that starts at 0.
8.      Counter() : counter_(0) {}
9.
10.     // Returns the current counter value, and increments it.
11.     int Increment();
12.
13.     // Returns the current counter value, and decrements it.
14.     int Decrement();
15.
16.     // Prints the current counter value to STDOUT.
17.     void Print() const;
18. };
```

1.2 配套的 cpp 文件，负责成员函数的具体实现

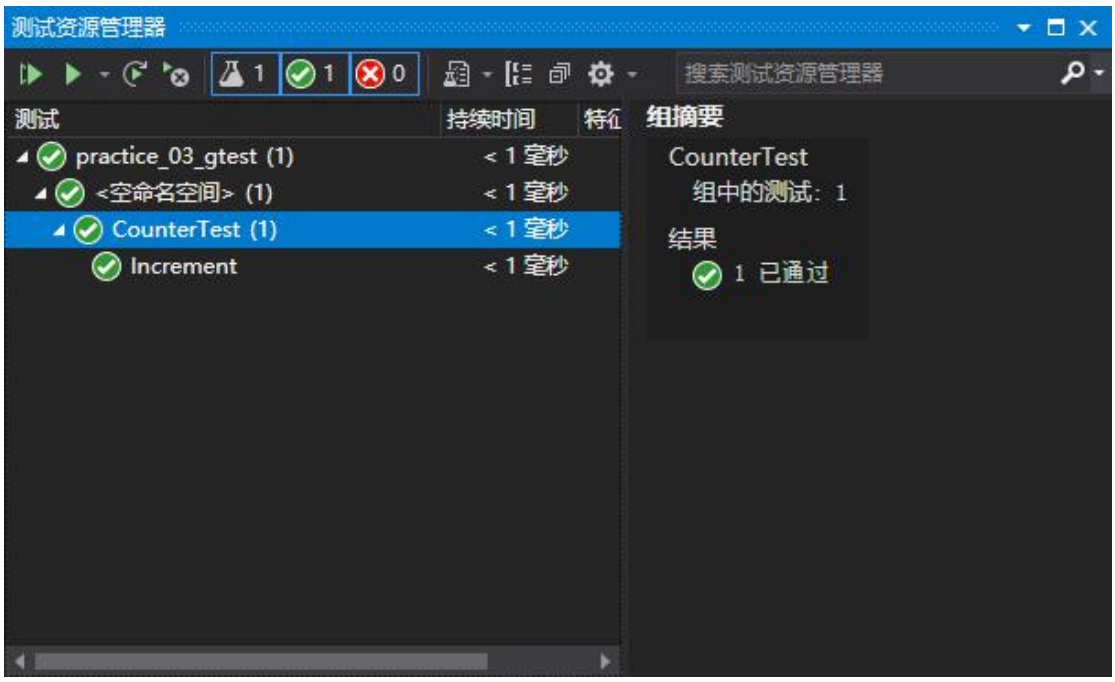
```
#include "Counter.h"
// Returns the current counter value, and increments it.
int Counter::Increment()
{
    return counter_++;
}

// Returns the current counter value, and decrements it.
// counter can not be less than 0, return 0 in this case
int Counter::Decrement()
{
    if (counter_ == 0) {
        return counter_;
    }
    else {
        return counter_--;
    }
}

// Prints the current counter value.
void Counter::Print() const {
    cout << counter_;
}
```

1.3 单元测试用例

先实例化对象，然后对类的成员函数进行测试

```
TEST(CounterTest, Increment)
{
    Counter c;

    // Test that counter 0 returns 0
    EXPECT_EQ(0, c.Decrement());

    EXPECT_EQ(0, c.Increment());
    EXPECT_EQ(1, c.Increment());
    EXPECT_EQ(2, c.Increment());

    EXPECT_EQ(3, c.Decrement());
}
```

用例执行结果如下：

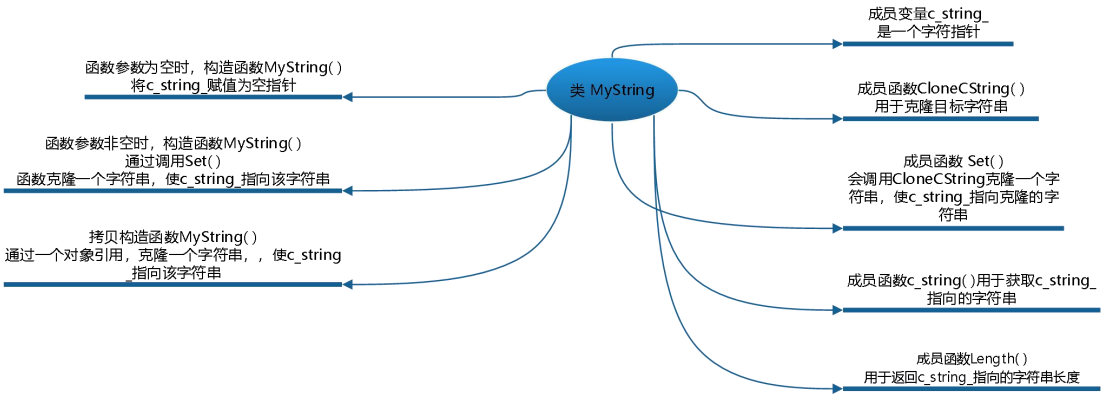


接下来的示例复杂一些

1.1 sample02 由三个部分组成：sample02.h， sample02.cpp， sample02UnitTest.cpp

1.2 sample02.h 定义了一个 string 类 MyString，阅读代码时，重点关注各成员的逻辑关系、厘清功能点，为用例的代码覆盖做准备。

提炼的逻辑关系和功能点如下图所示：



sample02.h 源码如下：

```
#include <string.h>

// A simple string class.
class MyString
{
    private:
        const char* c_string_; //声明一个字符指针常量 c_string_
        const MyString& operator=(const MyString& rhs);
        //赋值运算符重载函数
    public:
        // Clones a 0-terminated C string, allocating memory using new.  //克隆一个以 \0 结尾的字符串，通过 new 申请内存空间
        static const char* CloneCString(const char* a_c_string); //CloneCString() 函数将在 sample02.cpp 中进行具体的实现

        // The default c'tor constructs a NULL string.
        MyString() : c_string_(nullptr) {} //构造函数，将 c_string_ 赋值为 空指针 nullptr（相当于 NULL。C++中推荐 nullptr）

        // Constructs a MyString by cloning a 0-terminated C string.
        explicit MyString(const char* a_c_string) : c_string_(nullptr)
        { //构造函数，克隆一个以 \0 结尾的字符串
            Set(a_c_string);
        }

        // Copy constructor Func.
        MyString(const MyString &string) : c_string_(nullptr) //string 是一个对象引用，用于初始化另一个对象
        { //拷贝构造函数
            Set(string.c_string_);
        }
```

```
    //MyString is intended to be a final class, so the d'tor doesn't need to be virtual.
    ~MyString() { delete[] c_string_; }   //析构函数  deconstructor Func.

    // Gets the 0-terminated C string this MyString object represents.
    const char* c_string() const { return c_string_; } // c_string()函数用于获取字符串  c_string_

    // Length()函数用于返回字符串  c_string_的长度
    size_t Length() const { return c_string_ == nullptr      0 : strlen(c_string_); }

    //Sets the 0-terminated C string this MyString object represents.
    void Set(const char* c_string); //set()函数将在  sample02.cpp 中进行具体的实现
};
```

1.3　sample02.cpp 对成员函数 CloneCString( )、Set( )进行了具体实现。

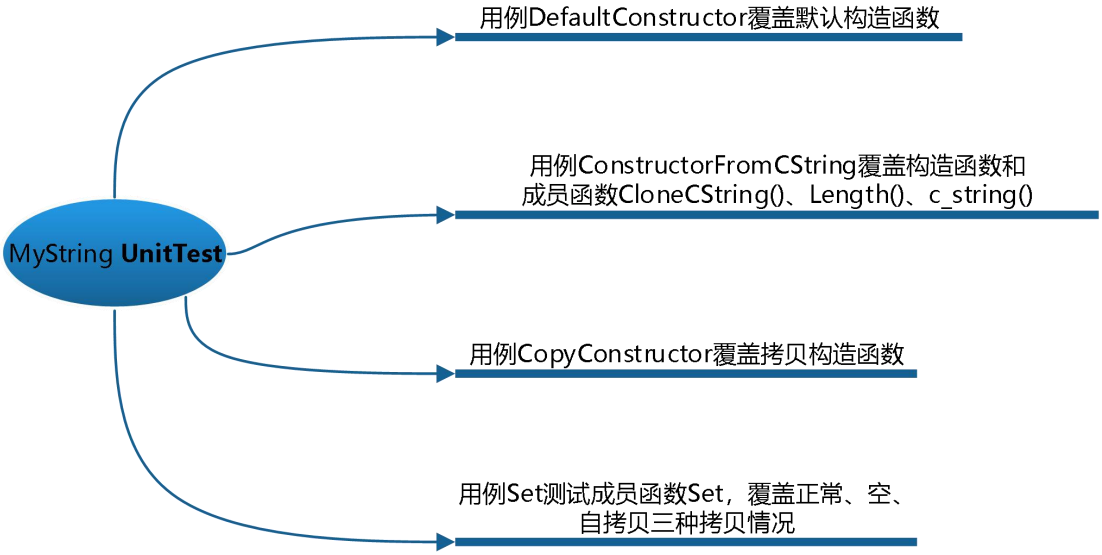
CloneCString( )用于克隆目标字符串；　Set( ) 调用 CloneCString 克隆一个字符串，使 c_string_指向克隆的字符串。

```
#include "sample02.h"
#include <string.h>

// Clones a 0-terminated C string, allocating memory using new.
const char* MyString::CloneCString(const char* a_c_string)
{
    if (a_c_string == nullptr) return nullptr; //空指针的情况

    const size_t len = strlen(a_c_string); //size_t 类似于 int
    char* const clone = new char[len + 1];
    memcpy(clone, a_c_string, len + 1);

    return clone;
}
```

```
// Sets the 0-terminated C string this MyString object represents.
void MyString::Set(const char* a_c_string)
{
    // Makes sure this works when c_string == c_string_
    const char* const temp = MyString::CloneCString(a_c_string);
    delete[] c_string_; //执行内存释放
    c_string_ = temp;
}
```

1.4 单元测试用例 sample02UnitTest.cpp

各用例覆盖的成员函数如下图所示：

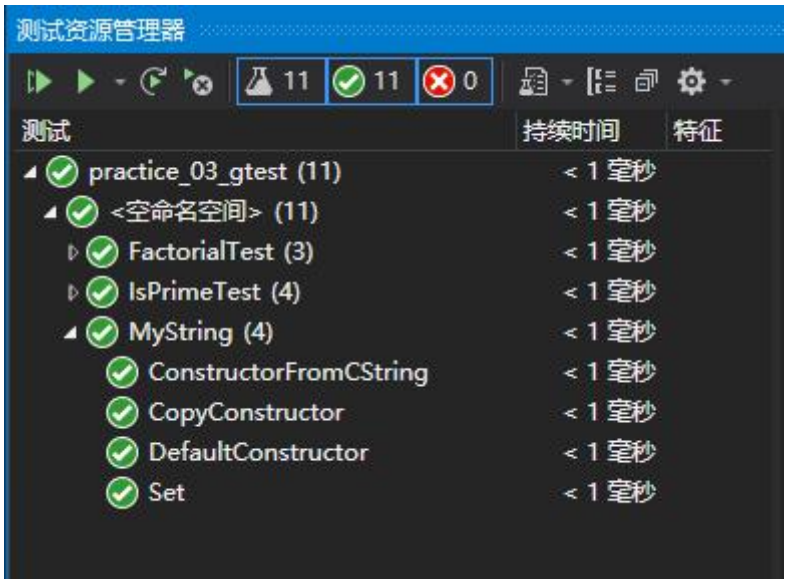



```
#include "pch.h"
#include "sample02.h"

TEST(MyString, DefaultConstructor) // 1.测试默认构造函数    MyString() : c_string_(nullptr) {}
{
    const MyString s;

    EXPECT_STREQ(nullptr, s.c_string()); //const char* c_string() const { return c_string_; }
```

```
        EXPECT_EQ(0, s.Length());
    }
    //
    const char kHelloString[] = "Google Test Framwork!";
    //
    // Tests the c'tor that accepts a C string.
    TEST(MyString, ConstructorFromCString) // 2.测试构造函数，该测试点包含成员函数 CloneCString( )、Length()、c_string()
    { //explicit MyString(const char* a_c_string) : c_string_(nullptr) { Set(a_c_string);}
        const MyString s(kHelloString);
        EXPECT_EQ(0, strcmp(s.c_string(), kHelloString));
        EXPECT_EQ( sizeof(kHelloString) / sizeof(kHelloString[0]) - 1, s.Length() );
    }
    //
    // Tests the copy c'tor.
    TEST(MyString, CopyConstructor) //3.测试拷贝构造函数
    {
        const MyString s1(kHelloString); //构造函数
        const MyString s2 = s1; //拷贝构造函数  MyString(const MyString &string) : c_string_(nullptr) { Set(string.c_string_); }
        EXPECT_EQ(0, strcmp(s2.c_string(), kHelloString));
    }

    // Tests the Set method.
    TEST(MyString, Set) //4.测试 Set()
    {
        MyString s;

        s.Set(kHelloString);
        EXPECT_EQ(0, strcmp(s.c_string(), kHelloString));

        // Set should work when the input pointer is the same as the one already in the MyString object.
        s.Set(s.c_string()); //5.测试 Set()的特殊情况：自身
```

```
    EXPECT_EQ(0, strcmp(s.c_string(), kHelloString));

    // Can we set the MyString to NULL
    s.Set(nullptr);   //6.测试特殊情况：NULL
    EXPECT_STREQ(nullptr, s.c_string());
}
```

用例执行结果如下：



二、Test Fixture

Test Fixture 介绍

对于单词 fixture，柯林斯词典是这样定义的：Fixtures are fittings or furniture which belong to a building and are legally part of it, for example, a bathtub or a toilet. 房间中的固定装置，如浴缸（搬家一般不带走。好吧，强行带走也不是不行….）

对于 Test Fixture，他就是每个测试用例执行时都要用到的、相同的测试资源，如果对每个测试用例都单独进行编码准备测试资源，代码冗余量太大，因此 Test Fixture 对于 code sharing 意义重大。

Test Fixture 示例

使用 Test Fixture，需要继承 testing::Test 类，重点是定制化自己的 SetUp( ) 函数，以便不同的测试用例可以复用相同的测试代码。

下面代码以 类 QueueTest 为例：

QueueTest 继承了类 testing::Test，用于测试 sample03.h 。 QueueTest 的成员变量包括 q0 q1 q2（创建了三个 sample03.h 中定义的队列）；QueueTestSmpl3 定制了 SetUp( )，用于锁定用例的预置条件；其中的定制化函数 MapTester()用于测试 sample03.h 中的 Map( ) 函数，后面会说明。

Google Test 每一条测试用例执行前都会调用 SetUp( )，每一条用例执行完毕都会调用 TearDown( )清理数据（如果没有特殊的清理需求，不需要重写 TearDown）。

```
class QueueTest : public testing::Test
{
    protected:
        Queue<int> q0; //创建队列
        Queue<int> q1;
        Queue<int> q2;

        void SetUp() override
        {
            q1.Enqueue(1); //入队
            q2.Enqueue(2);
            q2.Enqueue(3);
        }
       static int Double(int n)   //辅助测试函数
        {
             return 2 * n;
        }

        void MapTester(const Queue<int>* q)
        {
```

```
                // 生成一个新队列，和源队列相比，新队列的每一个元素值都是源队列对应元素值的两倍，这就是 Double 函数的作用
                const Queue<int>* const new_q = q->Map(Double);
                //把检查点写在类的声明中，供测试用例调用
                ASSERT_EQ(q->Size(), new_q->Size()); //核对队列大小
                for (const QueueNode<int>* n1 = q->Head(), *n2 = new_q->Head(); n1 != nullptr; n1 = n1->next(), n2 = n2->next())
                {  //核对每一个队列元素值：new_q 的元素值是对应 q 元素值的两倍
                    EXPECT_EQ(2 * n1->element(), n2->element());
                }

                delete new_q;  //释放资源
            }
    };
```

使用 TEST Fixture 需要使用宏 TEST_F；另外，测试用例集名字 必须和定义的类名保持一致。

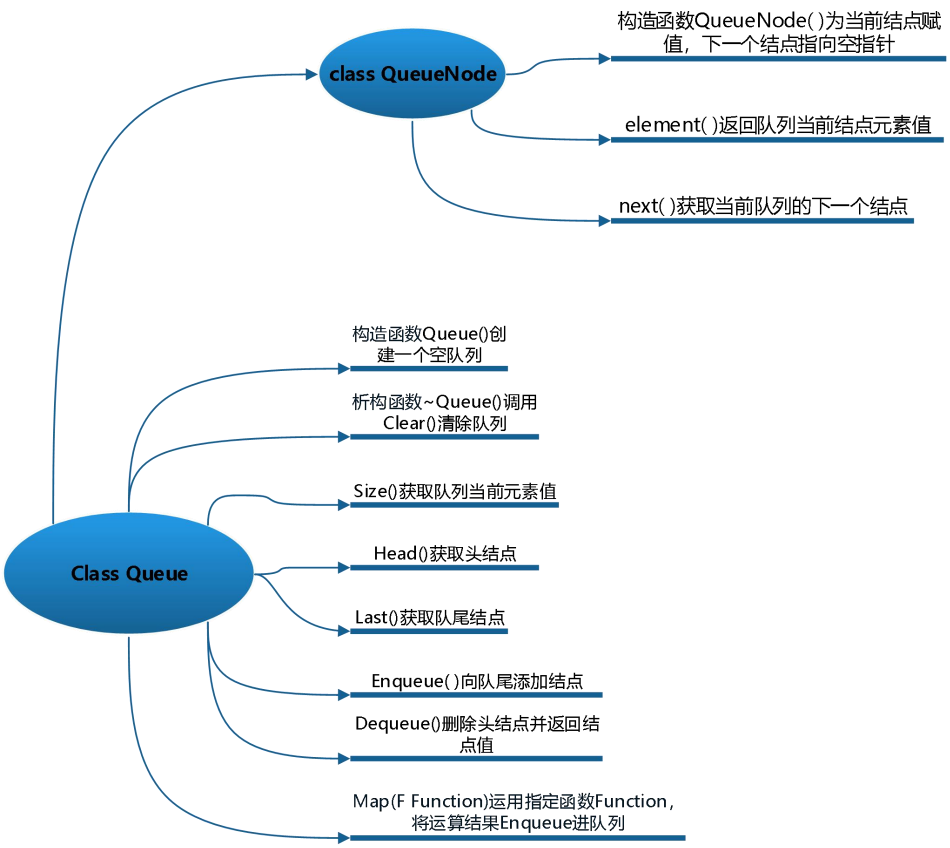
```
1. class QueueTest : public testing::Test{    };
2. TEST_F(QueueTest, DefaultConstructor){    };
```

各个测试用例 TEST_F( )使用相同的测试资源，同时测试资源不会受到上一条测试用例的影响。

Test Fixture 调用示例

1. sample03.h 源码如下

示例 3 理解起来比较难，我们不需要理解每一行代码，重点关注成员函数提供的功能和代码的逻辑关系：类 Queue 实现了队列功能（先进先出 如：超市排队付款），通过调用类 QueueNode 创建队列结点，除了普通的入队和出队函数外，他有一个 Map 函数（以函数为参数）




```
#include <stddef.h> //template

template <typename E> class Queue; // Queue 队列


template <typename E> class QueueNode // QueueNode 是队列中的结点，结点中包含 类型为 E 的元素 和 指向下一个结点的指针

{

    friend class Queue<E>; //Queue 是 QueueNode 的友元类，因此 Queue 可以访问 QueueNode 的所有成员


private:

    E element_;

    QueueNode* next_; //指向类的指针 指向队列中的下一个结点


    explicit QueueNode(const E &an_element) :element_(an_element), next_(nullptr) {} //构造函数


    const QueueNode& operator= (const QueueNode&);
```

```
    QueueNode(const QueueNode&);

public:
    const E &element() const // 返回当前结点的元素值(类型 E 的一个引用)
    {
        return element_;
    }

    QueueNode* next() //得到队列中的下一个结点
    {
        return next_;
    }
    const QueueNode* next() const
    {
        return next_;
    }
};

template <typename E>
class Queue   //定义类 Queue
{
private:
    QueueNode<E>* head_;   //队列首元素
    QueueNode<E>* last_;   //队列的最后一个元素.
    size_t size_;   //队列的元素个数

    Queue(const Queue&);
    const Queue& operator = (const Queue&);   //取消队列的赋值功能(队列拷贝)

public:
    Queue() : head_(nullptr), last_(nullptr), size_(0) {} //创建空队列

    ~Queue() { Clear(); }   //析构函数，用于清除队列
```

```
void Clear() //  具体的清除函数
{
    if (size_ > 0)
    {
        QueueNode<E>* node = head_;
        QueueNode<E>* next = node->next();
        for (; ;)
        {
            delete node;
            node = next;
            if (node == nullptr)
                break;
            next = node->next();
        }

        head_ = last_ = nullptr;
        size_ = 0;
    }
} //清除函数完毕

size_t Size() const { return size_; }

QueueNode<E>* Head() { return head_; }
const QueueNode<E>* Head() const { return head_; }

QueueNode<E>* Last() { return last_; }
const QueueNode<E>* Last() const { return last_; }

void Enqueue(const E& element) //使用拷贝构造函数向队列尾部添加元素
{
    QueueNode<E>* new_node = new QueueNode<E>(element);
    if (size_ == 0)
```

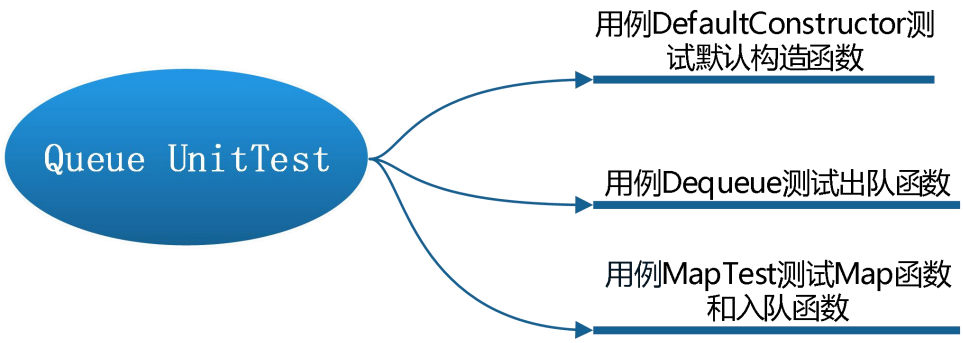
```
    {
        head_ = last_ = new_node;
        size_ = 1;
    }
    else
    {
        last_->next_ = new_node;
        last_ = new_node;
        size_++;
    }
}

E* Dequeue() //删除队列的头元素并返回元素值
{
    if (size_ == 0)
    {
        return nullptr;
    }

    const QueueNode<E>* const old_head = head_;
    head_ = head_->next_;
    size_--;

    if (size_ == 0)
    {
        last_ = nullptr;
    }

    E* element = new E(old_head->element());
    delete old_head;

    return element;
}  //Dequeue()函数完毕
```

```
    template <typename F> Queue* Map(F function) const
    { //在每一个队列元素上 运用指定函数 function，并将函数结果保存到新队列
        Queue* new_queue = new Queue();

        for (const QueueNode<E>* node = head_; node != nullptr; node = node->next_)
        {
            new_queue->Enqueue(function(node->element()));
        }

        return new_queue;
    }
};
```

2. 对应的单元测试用例

调用 Test Fixture 时，使用宏 TEST_F，并且 测试用例集名字 必须和定义的类名保持一致（本次示例使用的 Test Fixture 是 QueueTest）

理解了示例三提供的功能和代码的逻辑关系后，单元测试用例编写起来还是比较便捷的：请大家参考图示、源码和注释进一步理解。




```
TEST_F(QueueTest, DefaultConstructor)
{
    EXPECT_EQ(0, q0.Size()); //预期结果是 Pass，因为 SetUP()中 q0 未添加数据
```

```
}

TEST_F(QueueTest, Dequeue) //2.测试 Dequeue()函数
{
    int* n = q0.Dequeue();
    EXPECT_TRUE(n == nullptr); //测试结果 Pass：SetUP()中 q0_ 未添加元素

    n = q1.Dequeue(); //Dequeue()的返回值是一个 E 类型的指针
    ASSERT_TRUE(n != nullptr); //预期 pass, SetUP()中 q1 中添加了一个元素
    EXPECT_EQ(1, *n);  //q1.Enqueue(1);  pass, 元素值为 1
    EXPECT_EQ(0, q1.Size()); //弹出一个元素后，队列的元素个数为 0
    delete n;

    n = q2.Dequeue();
    ASSERT_TRUE(n != nullptr);
    EXPECT_EQ(2, *n);
    EXPECT_EQ(1, q2.Size()); //1u equal to 1
    delete n;
}

TEST_F(QueueTest, MapTest) //3.测试 Map()函数（Map 会调用 Enqueue 函数）
{
    MapTester(&q0); //传入的实际是队列的引用
    MapTester(&q1);
    MapTester(&q2);
    EXPECT_EQ(0,q0.Size());
    EXPECT_EQ(1, q1.Size());
    EXPECT_EQ(2, q2.Size()); //各个测试用例 TEST_F()使用相同的测试资源，测试资源不会受到上一条测试用例的影响
}
```

Test Fixture 的继承和差异化 以及 接口测试将在后面介绍。

GoogleTest 单元测试框架总结（下）

◆作者：Ron

前导文档：GoogleTest 单元测试框架总结（上）（中）

下面介绍 Test Fixture 的继承和差异化 和 接口测试

一、Test Fixture 的继承和差异化

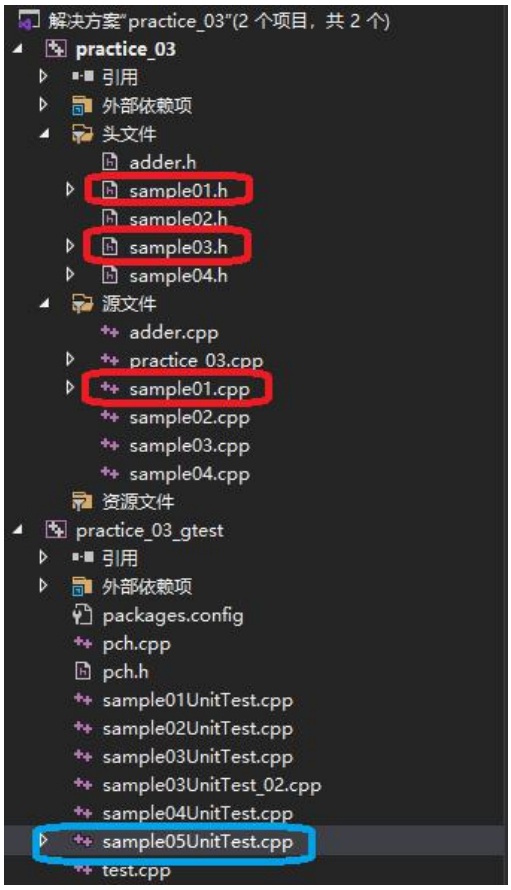
之前介绍了 Test Fixture，下面通过示例 sample05 说明下如何进行 Test Fixture 的继承。

对已有的 Test Fixture 进行继承，往往发生在如下两个场景 (1) 定义的 Test Fixture 可用于指定的一套单元测试用例集，当有多套测试用例时，如何处理？ (2)当不同的测试用例集对应的测试资源基本相同、但又存在少量差异，又如何处理？

被继承的 Test Fixture 在 Google Test 中被称为 Super Fixture、继承者被称为 sub fixture。Super Fixture /Test Fixture 本质就是一个类，类当然可以被继承啦。

sample05 的测试对象

1. 测试对象是 sample01.h + sample01.cpp, sample03.h，文件结构如下图所示：



sample01.h 对 Factorial( ), IsPrime( )函数进行了声明、sample01.cpp 对 Factorial( ) 、IsPrime( )两个函数进行了具体实现：Factorial 用于求一个整数的阶乘、IsPrime 用于判定一个整数是否为素数。

sample03.h 完整定义了类 Queue（包含成员函数的具体实现）：类 Queue 实现了队列的基本功能，同时他包含了一个定制化的 Map( )函数。

上述源码文件在之前的文档中已经展示过，这里不再重复。

对应的单元测试用例

1. sample05UnitTest.cpp 中定义了名为 QuickTest 的 Test Fixture，用来判定用例的执行时间是否超过 5 秒：从 SetUp( )调用时启动计时，从 TearDown( )调用时结束计时，期间时间就是用例执行时间。

GoogleTest 的每一条用例，正式启动前先调用 SetUp( )，执行结束后调用 TearDown( )

```
class QuickTest : public testing::Test
    {
    protected:
        time_t start_time_;
        void SetUp() override
        {
            start_time_ = time(nullptr);
        }
        void TearDown() override
        {
            const time_t end_time = time(nullptr);
            EXPECT_TRUE(end_time - start_time_ <= 5) << "The test took too long.";
        }
};
```

2. Test Fixture 的继承

IntegerFunctionTest 继承自测试夹具 QuickTest，该继承解决了上面提到的问题一：只定义了一个 Test Fixture，但是需要编写多套测试用例？

```
class IntegerFunctionTest : public QuickTest{ };
TEST_F(IntegerFunctionTest, Factorial)
{
        EXPECT_EQ(1, Factorial(-5)); //每条用例都会判定是否超过 5 秒
        EXPECT_EQ(1, Factorial(-1));
 }
```

QueueTest 也继承自测试夹具 QuickTest，他的目标是测试类 Queue，因此进行了必要的差异化：在 SetUp( )中为类 Queue 的实例化对象赋初值。

该继承解决了上面提到的问题二：不同的测试用例集使用相同的测试资源、但又存在差异？

```
class QueueTest : public QuickTest
    {
```

```
protected:
    Queue<int> q0_; //差异化
    Queue<int> q1_;
    Queue<int> q2_;
    void SetUp() override
    {
        QuickTest::SetUp();
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
    }
};
```

使用 QueueTest 的宏 TEST_F，由于继承自 QuickTest，同样会判定用例的执行是否超过 5 秒。

```
TEST_F(QueueTest, DefaultConstructor)
    {
        EXPECT_EQ(0, q0_.Size());
    }
```

QueueTest, IntegerFunctionTest 也可以被继承：可以进行 Test Fixture 继承的继承，此类继承没有层数限制。

3. 单元测试用例的完整源码

下面通过测试夹具 QuickTest, QueueTest, IntegerFunctionTest 完成 sample01.h 和 sample03.h 的测试。

```
#include "pch.h"
#include <limits.h>
#include <time.h>
#include "sample01.h"
#include "sample03.h"
```

```
namespace {
    class QuickTest : public testing::Test     //首先定义测试夹具 QuickTest
    {
    protected:
        time_t start_time_;
        void SetUp() override
        {
            start_time_ = time(nullptr);
        }
        void TearDown() override
        {
            const time_t end_time = time(nullptr);
            EXPECT_TRUE(end_time - start_time_ <= 5) << "The test took too long.";
        }
    };

    class IntegerFunctionTest : public QuickTest
    {
        // IntegerFunctionTest 继承自 QuickTest，由于不需要差异化，因此代码为空
    };

   //通过 IntegerFunctionTest 完成 Factorial() 和 IsPrime()的单元测试
   TEST_F(IntegerFunctionTest, Factorial)
     {
      // Tests factorial of negative numbers.
      EXPECT_EQ(1, Factorial(-5));
      EXPECT_EQ(1, Factorial(-1));
      EXPECT_GT(Factorial(-10), 0);

      // Tests factorial of 0.
      EXPECT_EQ(1, Factorial(0));

      // Tests factorial of positive numbers.
```

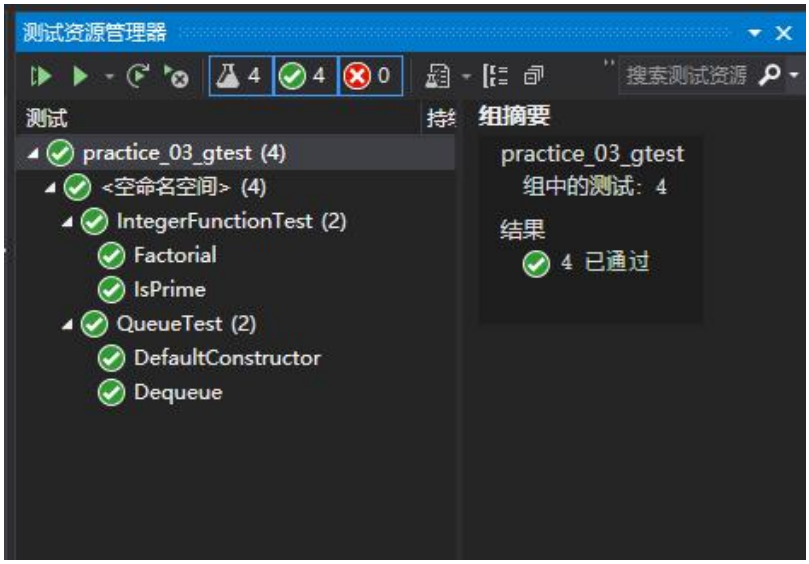
```
    EXPECT_EQ(1, Factorial(1));
    EXPECT_EQ(2, Factorial(2));
    EXPECT_EQ(6, Factorial(3));
    EXPECT_EQ(40320, Factorial(8));
    }


// Tests IsPrime()
TEST_F(IntegerFunctionTest, IsPrime)
{
    // Tests negative input.
    EXPECT_FALSE(IsPrime(-1));
    EXPECT_FALSE(IsPrime(-2));
    EXPECT_FALSE(IsPrime(INT_MIN)); //INT_MIN 是边界值

    // Tests some trivial cases.
    EXPECT_FALSE(IsPrime(0));
    EXPECT_FALSE(IsPrime(1));
    EXPECT_TRUE(IsPrime(2));
    EXPECT_TRUE(IsPrime(3));

    // Tests positive input.
    EXPECT_FALSE(IsPrime(4));
    EXPECT_TRUE(IsPrime(5));
    EXPECT_FALSE(IsPrime(6));
    EXPECT_TRUE(IsPrime(23));
}

class QueueTest : public QuickTest
{   // QueueTest 也继承自 QuickTest，由于是测试 Queue，进行了部分差异化
protected:
    Queue<int> q0_;
    Queue<int> q1_;
```

```
    Queue<int> q2_;
    void SetUp() override
    {
        QuickTest::SetUp();
        q1_.Enqueue(1);
        q2_.Enqueue(2);
        q2_.Enqueue(3);
    }
};

//通过 QueueTest 完成类 Queue 的测试，进行代码测试并评估用例的执行时间是否超过 5 秒
TEST_F(QueueTest, DefaultConstructor)
{
    EXPECT_EQ(0, q0_.Size());
}

// Tests Dequeue().
TEST_F(QueueTest, Dequeue)
{
    int* n = q0_.Dequeue();
    EXPECT_TRUE(n == nullptr);

    n = q1_.Dequeue();
    EXPECT_TRUE(n != nullptr);
    EXPECT_EQ(1, *n);
    EXPECT_EQ(0u, q1_.Size());
    delete n;

    n = q2_.Dequeue();
    EXPECT_TRUE(n != nullptr);
    EXPECT_EQ(2, *n);
    EXPECT_EQ(1u, q2_.Size());
    delete n;
```

```
    }
} // namespace
```

4. 用例执行结果如下

从测试结果来看，所有用例的执行时间都在 5 秒内。



二、接口测试

接口

1. GoogleTest 示例 6 演示了如何测试接口，示例中的源码关系如下：

1.1 头文件 PrimeTable.h 定义了接口 PrimeTable，该接口提供了两个功能： bool IsPrime(int n) 判定一个数是否为素数 、int GetNextPrime(int p) 返回比当前数大的最小素数。

1.2 PrimeTable.h 虽然定义了接口，但未具体实现，它的具体实施例有两个 OnTheFlyPrimeTable、PreCalculatedPrimeTable ：采用不同算法实现相同功能。

1.3 PrimeTable.h 对应的单元测试文件是 sample06UnitTest.cpp 。

2. 接口文件 PrimeTable.h

PrimeTable.h 具体代码及注释如下，该代码难度比较高，需要掌握 C++ 类、继承、

模板相关知识。但是，我们的目的是编写对应单元测试用例、保障软件代码的质量，重点理解其提供的功能和代码逻辑就可以。

```
#pragma once


// 本头文件提供了一个接口 interface 素数表 PrimeTable，用于判定一个数是否是素数 及 返回下一个素数
#ifndef GTEST_SAMPLES_PRIME_TABLES_H_ //#ifndef #define #endif
#define GTEST_SAMPLES_PRIME_TABLES_H_


#include <iostream>
using namespace std;


//class PrimeTable 定义了接口，未具体实现
class PrimeTable
{
    public:
        virtual ~PrimeTable() {} //析构函数
         // IsPrime()用于判定一个数是否为素数
        virtual bool IsPrime(int n) const = 0;
        // GetNextPrime( ) 返回比 p 大的最小素数；若返回值超过素数表的范围，返回-1
        virtual int GetNextPrime(int p) const = 0;
};


//接口的具体实施例一：直接计算素数
class OnTheFlyPrimeTable : public PrimeTable
{
    //OnTheFlyPrimeTable 继承自类 PrimeTable，具体实现了类 PrimeTable 的成员函数 IsPrime( ) 和 GetNextPrime( ),
    public:
        bool IsPrime(int n) const override
        {
            if (n <= 1) return false;
            for (int i = 2; i * i <= n; i++)
```

```
                if ((n % i) == 0) return false;
            return true;
        }

        int GetNextPrime(int p) const override
        {
            if (p < 0) return -1;

            for (int n = p + 1;; n++)
            {
                if (IsPrime(n)) return n;
            }
        }
};

//接口的具体实施例二：引入数组，对素数进行标记
class PreCalculatedPrimeTable : public PrimeTable
{
    private:
        const int is_prime_size_; //is_prime_size_ 定义了素数表中的最大数
        bool* const is_prime_; //定义布尔型数组 is_prime_

    public:
        // max 定义了素数表的最大数
        explicit PreCalculatedPrimeTable(int max) : is_prime_size_(max + 1), is_prime_(new
bool[max + 1])
        {   //构造函数：is_prime_size_ = max+1 ; 定义了数组的大小 is_prime_[max + 1]
            CalculatePrimesUpTo(max); //声明了函数 CalculatePrimesUpTo( ) 并在下方的
private 中实现
        }

        ~PreCalculatedPrimeTable() override { delete[] is_prime_; } //析构函数，释放数组 is_prime_
占用的内存空间. delete 和 new 对应
```

```
        bool IsPrime(int n) const override
        { //在 PrimeTable 中声明了接口，在实施例中进行具体实现
            return 0 <= n && n < is_prime_size_ && is_prime_[n];
        } // is_prime_[n]为素数返回 true; 0 <= n 处理了负数的情况; n < is_prime_size_ 限定了范围

        int GetNextPrime(int p) const override //返回比 p 大的最小素数，若超出范围，返回-1
        {
            for (int n = p + 1; n < is_prime_size_; n++)
            {
                if (is_prime_[n]) return n;
            }

            return -1;
        }

    private:
        void CalculatePrimesUpTo(int max)
        {
            //布尔型数组 is_prime_[max+1], is_prime_size_ 数组元素个数
            fill(is_prime_, is_prime_ + is_prime_size_, true); // ::std::fill()函数，将 is_prime_数组中的所有元素均填充为 true

            is_prime_[0] = is_prime_[1] = false; //0 和 1 非素数

            // 对 2 至 max 进行素数的预判定（2 是偶数中唯一的素数）
            for (int i = 2; i * i <= max; i += i % 2 + 1) //i=2 3 5 7 9 11 13 15
            {
                if (!is_prime_[i]) continue;
                for (int j = i * i; j <= max; j += i) 数
                {
                    is_prime_[j] = false;
```

```
                }
            }
        }
        void operator=(const PreCalculatedPrimeTable &rhs);
};
#endif  // GTEST_SAMPLES_PRIME_TABLES_H_
```

梳理得到的重点信息：从上述代码可以看到，接口 PrimeTable 提供了两个功能：IsPrime( ) 用于判定一个数是否为素数 、GetNextPrime( ) 返回比当前数大的最小素数。为实现上述功能，PrimeTable 接口提供了两个实施例 OnTheFlyPrimeTable、PreCalculatedPrimeTable（实施例如何实现具体的功能，不用过于关注。重点是单元测试用例需要覆盖这两个实施例）。

OnTheFlyPrimeTable 的实现方法：根据素数定义，编写代码判定一个数是否为素数。

PreCalculatedPrimeTable 的实现方法：采用空间换时间的方案，先通过数组标示出素数（数组下标代表数字、数组元素值代表是否为素数），然后执行判定即可。

3. 单元测试文件 sample06UnitTest.cpp

3.1 sample06UnitTest.cpp 中介绍了接口测试的两种方法：I.接口的实施例在编写用例时是明确的，可以使用名为 typed tests 的方法。II.接口的实施例不明确，可以使用名为 type-parameterized tests 的方法。

3.2 在使用具体的方法前，需要先编写 factory function CreatePrimeTable<T>( ) 用于获取实例对象。

这里插几句：大家可以看到，除良好的逻辑分析能力外，单元测试也需要良好的代码编写能力（所以单元测试一般由开发人员自己完成），单元测试门槛还是蛮高的……

```
template <class T> PrimeTable* CreatePrimeTable(); //模板函数 CreatePrimeTable<T>() 返回一个指向 PrimeTable 类的指针

template <> PrimeTable* CreatePrimeTable<OnTheFlyPrimeTable>()
{
```

```
        return new OnTheFlyPrimeTable;
    }

    template <> PrimeTable* CreatePrimeTable<PreCalculatedPrimeTable>()
    {
        return new PreCalculatedPrimeTable(10000);
    }
```

3.3 然后定义 Test Fixture PrimeTableTest（Test Fixture 继承自类 testing::Test），typed tests 和 type-parameterized tests 中都会用到 Test Fixture。

注意，通过接口测试实施例才符合真实场景“实施例通过接口调用”，这样也不会遗漏多个实施例的情况。

```
    template <class T> class PrimeTableTest : public testing::Test
    {
    protected:
        PrimeTable* const table_;
        // 构造函数调用上面的工厂函数，工厂函数根据参数 T 创建一个接口的实施例对象
        PrimeTableTest() : table_(CreatePrimeTable<T>()) {}

        ~PrimeTableTest() override { delete table_; }

        // 经由接口 测试 实施例符合实际场景（实施例是经由接口调用的），这样做可以避免遗漏实施例（当有多个实施例时）
    };
```

3.4 typed tests 的步骤如下

(1) 首先定义 types（即接口的所有实施例），比如：我们知道接口 PrimeTable 的实施例是 OnTheFlyPrimeTable 和 PreCalculatedPrimeTable。

```
using testing::Types;
typedef Types<OnTheFlyPrimeTable, PreCalculatedPrimeTable> Implementations;
```

(2) 使用宏 TYPED_TEST_CASE(TestCaseName, TypeList)

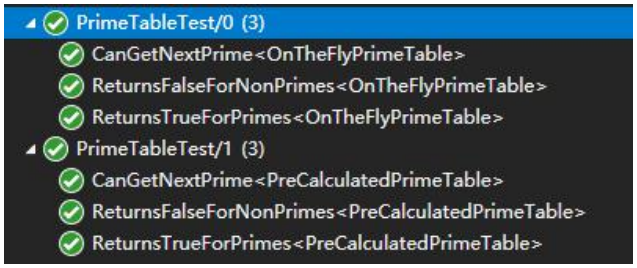
和宏 TEST_F 的使用一样：TestCaseName=test fixture name 。注意，在 vs2019 环境下是 TYPED_TEST_CASE。

```
template <class T> class PrimeTableTest : public testing::Test{};
//...
TYPED_TEST_CASE(PrimeTableTest, Implementations);
```

(3) 最后，使用宏 TYPED_TEST(TestCaseName, TestName) 编写具体的测试用例，这里，TestCaseName 也是定义的 Test Fixture 的名称。

```
template <class T> class PrimeTableTest : public testing::Test{};
//...
TYPED_TEST(PrimeTableTest, ReturnsTrueForPrimes)
    {
        EXPECT_TRUE(this->table_->IsPrime(2));
        EXPECT_TRUE(this->table_->IsPrime(3));
        EXPECT_TRUE(this->table_->IsPrime(5));
        EXPECT_TRUE(this->table_->IsPrime(7));
        EXPECT_TRUE(this->table_->IsPrime(11));
        EXPECT_TRUE(this->table_->IsPrime(131));
    }
```

(4) 用例执行的时候，Google Test 会根据 TYPED_TEST_CASE 中定义的 Types：OnTheFlyPrimeTable 和 PreCalculatedPrimeTable，重复执行 TYPED_TEST 中定义的用例。



3.5 type-parameterized tests 的步骤如下：

(1) type-parameterized tests 适用于仅知道接口，不知道具体的实施例的情况。

(2) 第一步还是要定义 test fixture。

```
template <class T> class PrimeTableTest2 : public PrimeTableTest<T> { };
```

(3) 第二步使用 宏 TYPED_TEST_CASE_P(TestFixture) 。

```
TYPED_TEST_CASE_P(PrimeTableTest2);
```

(4) 第三步使用宏 TYPED_TEST_P(TestFixture,TestCaseName)编写测试用例。

```
TYPED_TEST_P(PrimeTableTest2, ReturnsFalseForNonPrimes)
    {
        EXPECT_FALSE(this->table_->IsPrime(-5));
        EXPECT_FALSE(this->table_->IsPrime(0));
        EXPECT_FALSE(this->table_->IsPrime(1));
        EXPECT_FALSE(this->table_->IsPrime(4));
        EXPECT_FALSE(this->table_->IsPrime(6));
        EXPECT_FALSE(this->table_->IsPrime(100));
    }
```

(5) 最后，通过 REGISTER_TYPED_TEST_SUITE_P 枚举上面的测试用例 //VS2019 应该是 REGISTER_TYPED_TEST_CASE_P。

```
// Type-parameterized tests involve one extra step: you have to enumerate the tests you defined:
REGISTER_TYPED_TEST_CASE_P( PrimeTableTest2, ReturnsFalseForNonPrimes, ReturnsTrueForPrimes, CanGetNextPrime);
// The first argument is the test case name. The rest of the arguments are the test names.
```

(6) 注意：通过上述步骤，完成了一个 test pattern 的定义，但实际上并未执行具体的测试。

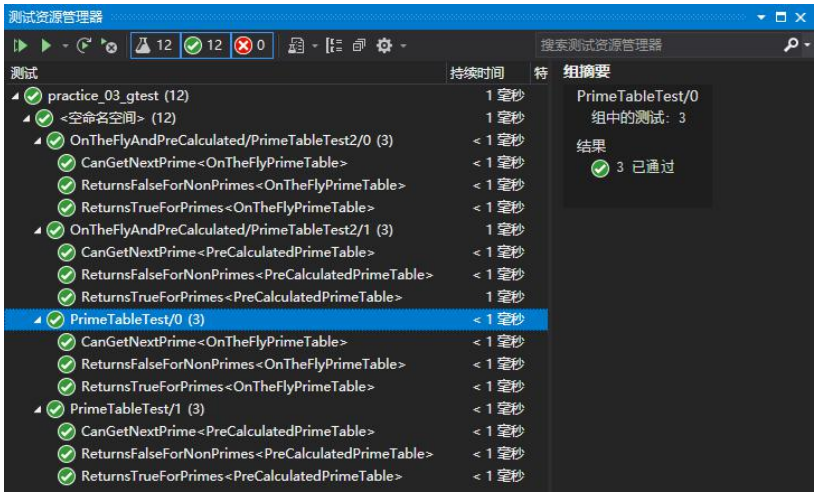
通常情况下，(2)-(5)的工作会包含在一个头文件中，由接口的作者编写，用于验证具体的实施例，实施例的作者可以包含该头文件，然后执行测试：看实施例是否能满足接

口的需求。

(7) test pattern 的调用方法如下，非 VS2019 下，需要将 CASE 变更为 SUITE

```
typedef Types<OnTheFlyPrimeTable, PreCalculatedPrimeTable>  PrimeTableImplementations; //Define Type
INSTANTIATE_TYPED_TEST_CASE_P(OnTheFlyAndPreCalculated, PrimeTableTest2, PrimeTableImplementations);
```

sample06 的执行结果如下：



通过上面的示例，相信大家对单元测试、GoogleTest 已经有了一个比较直观的认识了。如果感兴趣，想进一步的熟悉 GoogleTest，请参考 https://github.com/google/googletest/ 配套的说明文档很不错的。

测试技术的学习是一件长期的、痛苦的、枯燥的事情，但是豁然开朗的那一刻会让你感觉很充实，这也会为自己后续的职业生涯的拓展和延续提供一定的帮助，大家彼此加油。

由于本人能力有限，上述总结难免有疏漏错误的地方，恳请大家指正。

JMeter 中如此多的控制器，你还傻傻分不清吗

◆ 作者：罗狮小钉

JMeter Controller，即 JMeter 控制器，是我们构建测试场景中不可或缺的原件。它对于构建实时。

JMeter 测试计划来说是非常重要的，JMeter 控制器可以指定将请求发送到服务器的逻辑顺序。

在 JMeter 中最常用的控制器有：

- Simple Controller
- Loop Controller
- Once Only Controller
- Throughput Controller
- Interleave Controller
- If Controller
- While Controller

下面我们对这些控制器逐一介绍，结合简单应用，来梳理每个控制器的不同功能。

1. Simple Controller

Simple Controller，即"简单控制器"，本身不具备任何特定功能。

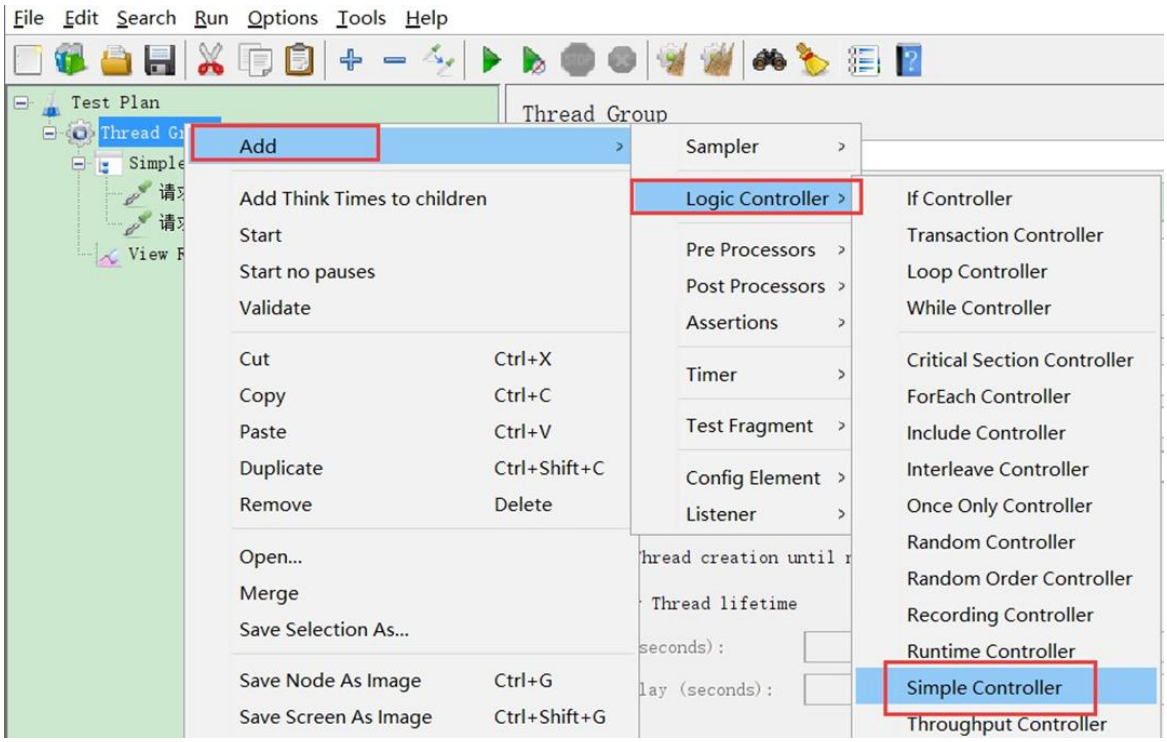
Simple Controller 可以视作为一个容器，把某个业务中涉及的接口请求放在一起，便于理解。

例如，我们可以把登录业务涉及到的一系列接口放在一个 Simple Controller， 把注册业务涉及到的

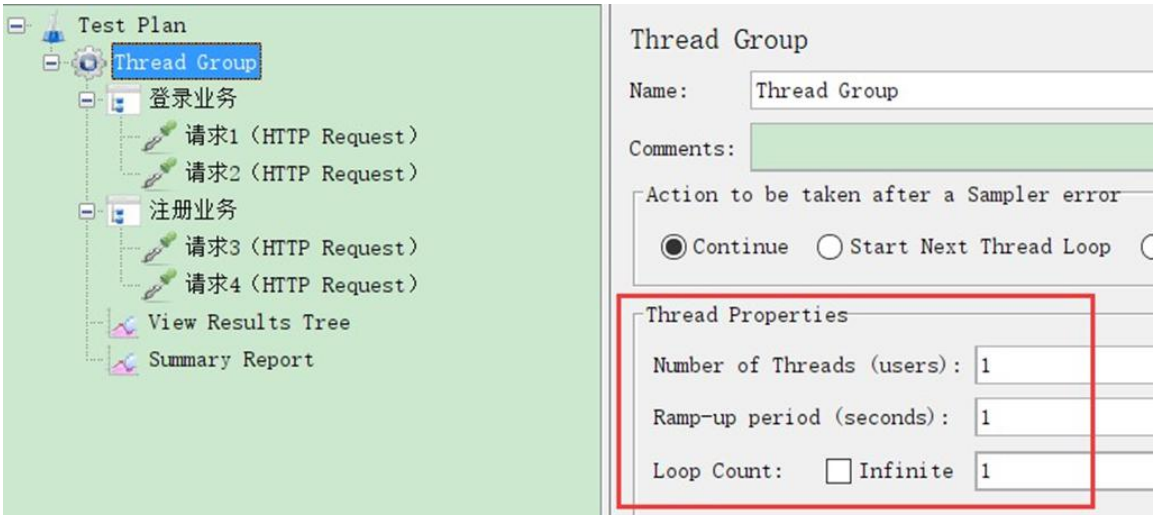
一系列接口放在另一个 Simple Controller 中。

【 Simple Controller _ Demo】

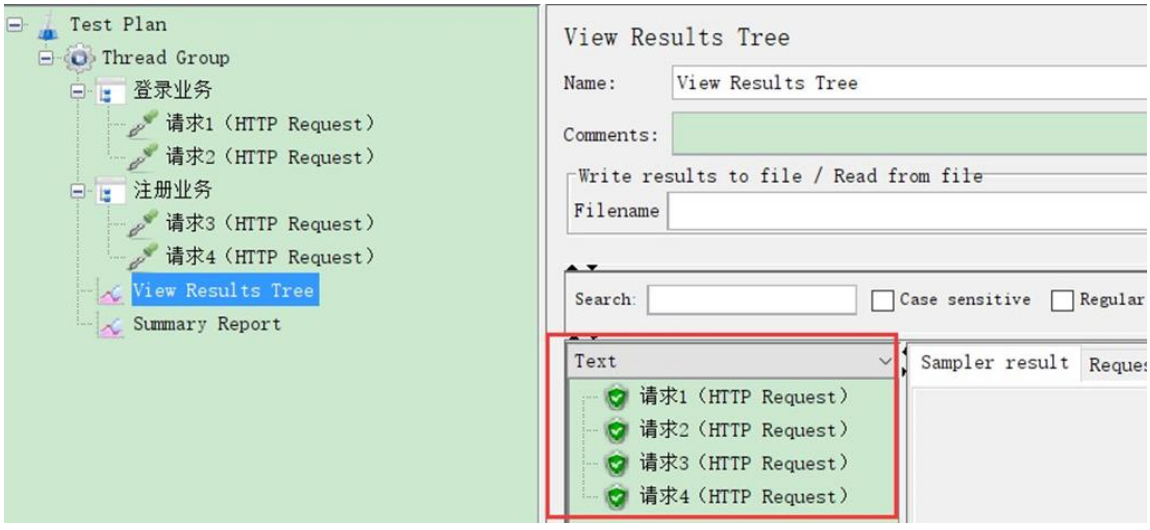
• Simple Controller 设置



- 当前线程组设置



- 执行结果当前仅一个线程，每个请求仅执行一次



2. Loop Controller

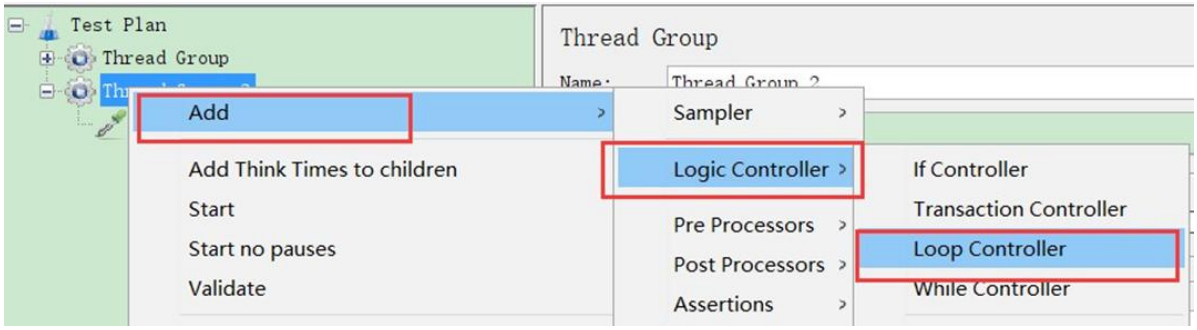
Loop Controller， 即"循环控制器"。

当你希望某些特定请求运行的次数，多余线程组中指定的迭代次数，这个时候，就可以把这些请

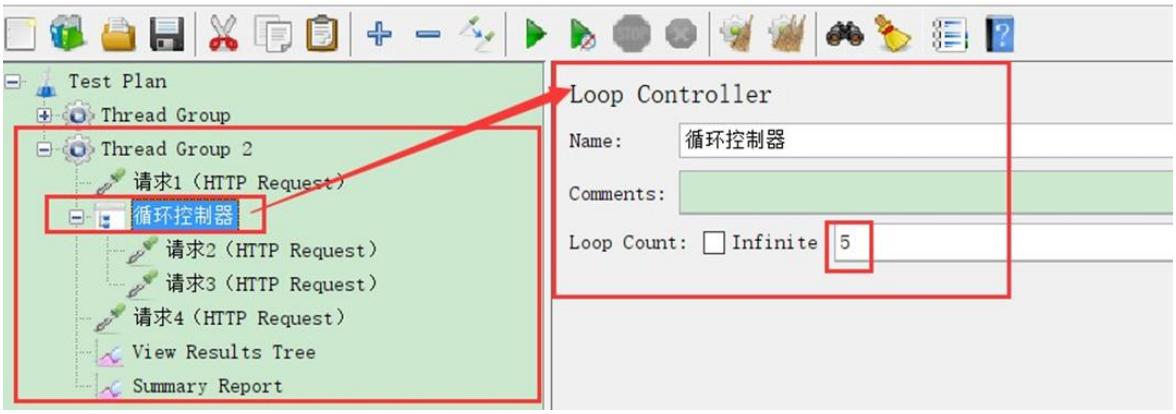
求放到循环控制器中，并在控制器设置中对应的循环次数。

【 Loop Controller _ Demo】

• Loop Controller 设置

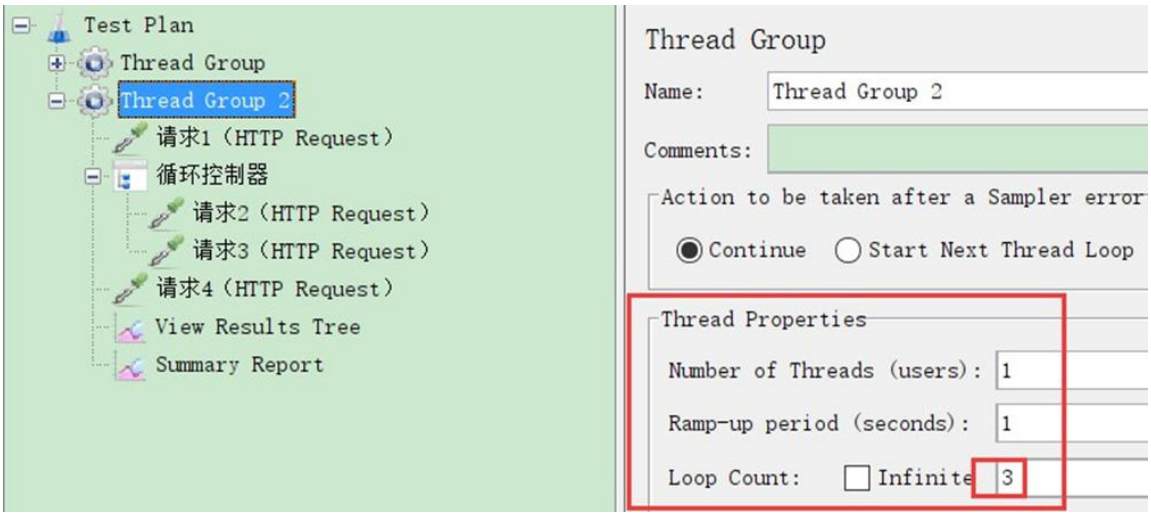


当前循环控制器内有两个接口请求 ———— 请求 2 和请求 3； 循环控制器计数为 5，即当前循环控制器中的请求会迭代 5 次； 所以，请求 2 和请求 3 都会执行 5 次。



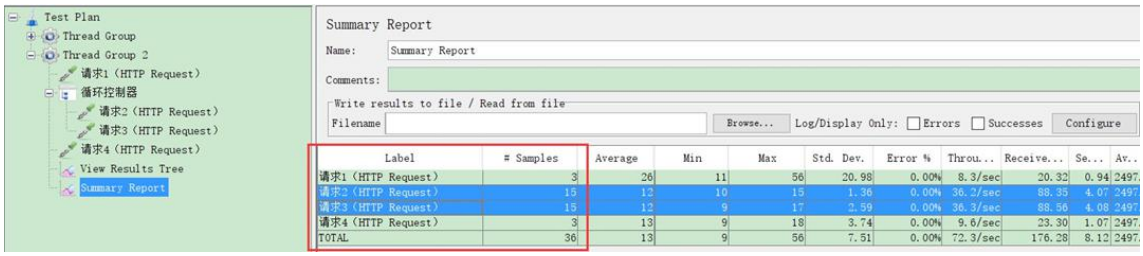
• 当前线程组设置

当前线程组中仅一个线程，整个线程组中的请求需要迭代执行 3 次； 当前线程组中共有 4 个请求，请求 1 和请求 4 不在循环控制器内。



• 执行结果

当前线程组设置循环次数为 3 ———— 针对每一个当前线程，请求 1 和请求 4 会执行 3 次； 循环控制器中设置循环次数为 5 ———— 针对每一个当前线程，请求 2 和请求 3 会执行 3*5， 即 15 次。



3. Once Only Controller

Once Only Controller，即"只执行一次控制器"。

即便当前线程组中设置了多次循环，如果有这样一个请求，你只想它执行一次即可，这时就需要

用 Once Only Controller。

例如 “首页访问”，实际应用中，我们只会发起一次获取首页的请求，而其他请求（例如搜索/删除

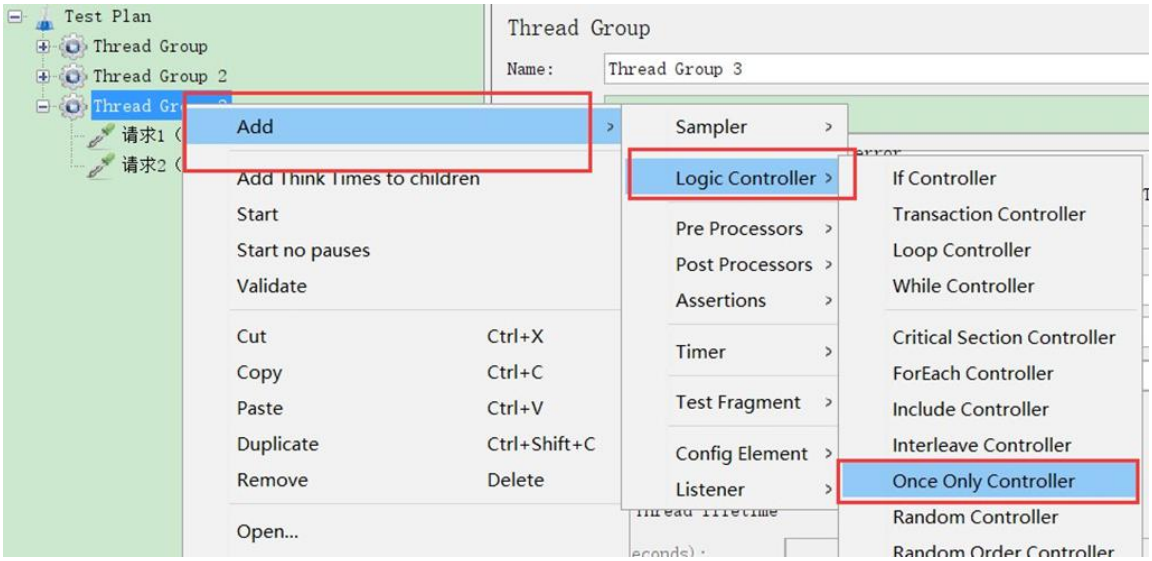
等）会多次执行。我们可以把仅希望执行一次的接口请求放在 Once Only Controller

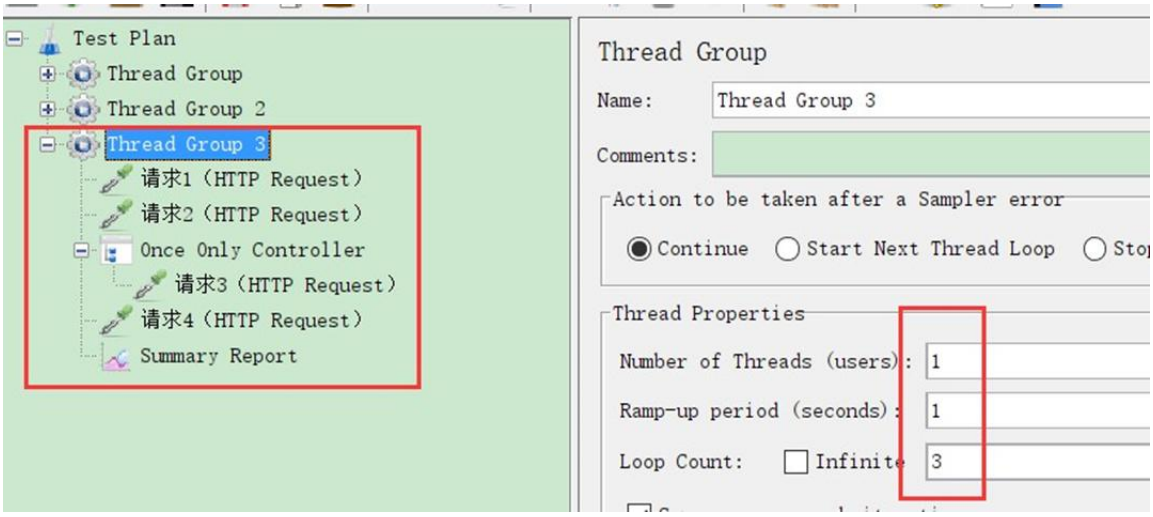
中，在该控制。

器下的请求，当前线程只会执行一次，即便父类线程组设置了多次循环。

【Once Only Controller _ Demo】

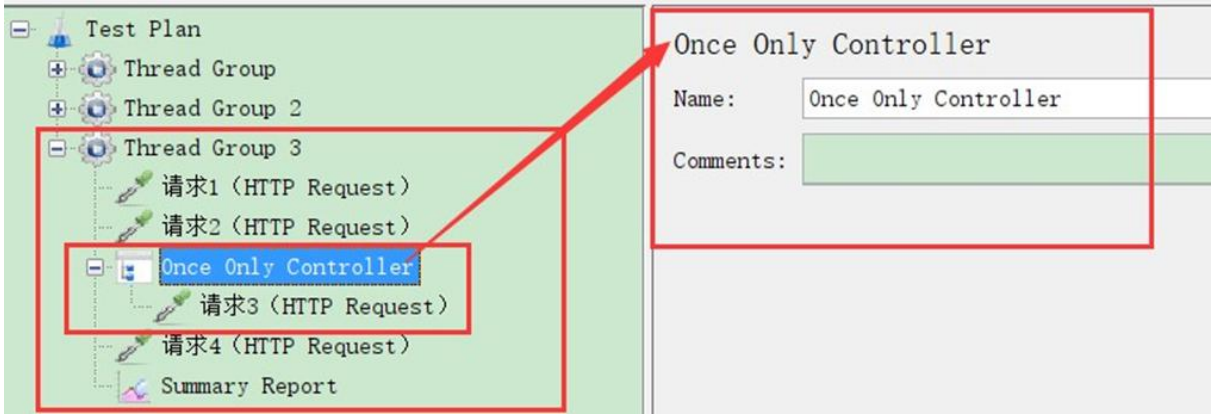
• Once Only Controller 设置





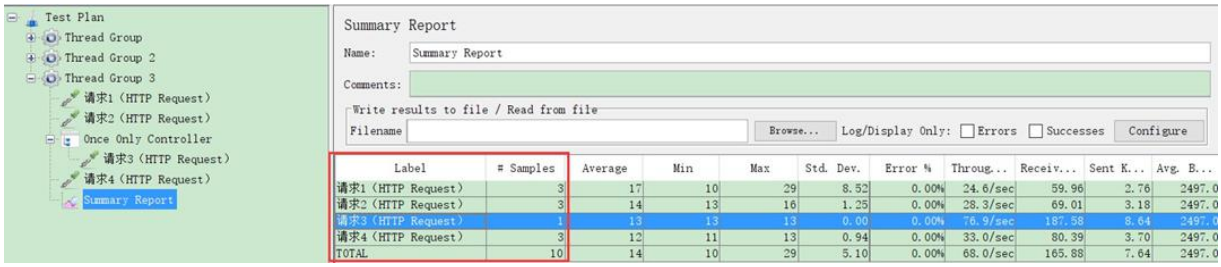
• 当前线程组设置

当前线程组中仅一个线程，整个线程组中的请求需要迭代 3 次； 当前线程组中共有 4 个请求，请求 3 在 Once Only Controller 中。



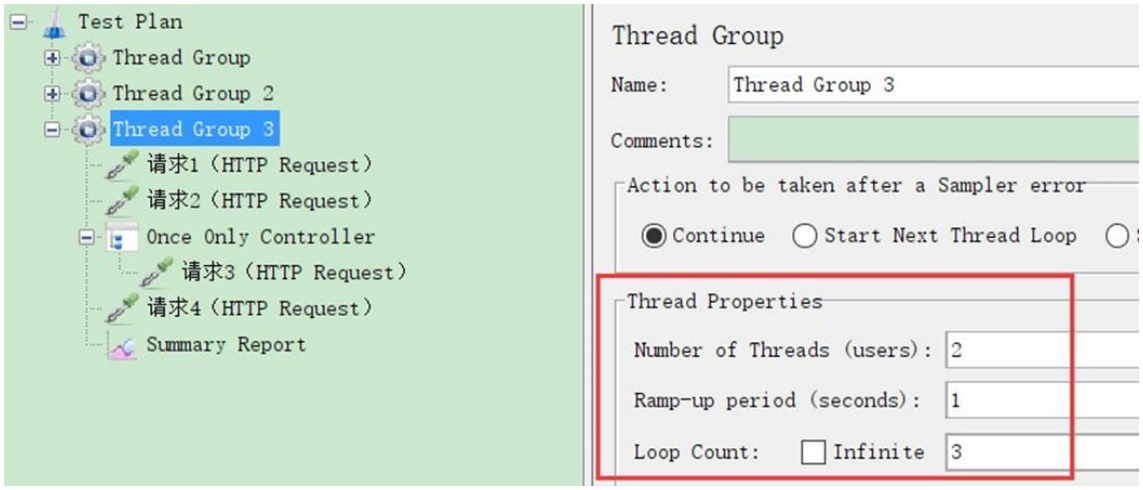
• 执行结果_1

当前线程组设置循环次数为 3 ———— 针对每一个当前线程，请求 1，请求 2 和请求 4 会执行 3 次；线程组中有一个 Once Only Controller ———— 针对每一个当前线程，请求 3 只会执行 1 次。



• 执行结果_2

如果把当前线程组中的线程数改为 2， 线程组的循环次数还是 3。



那么有如下结果：

当前线程组设置循环次数为 3

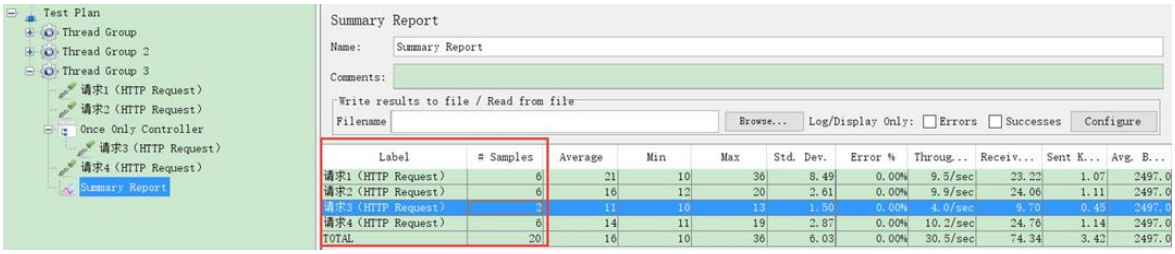
———— 针对每一个当前线程，请求 1，请求 2 和请求 4 会执行 3 次；

———— 由于线程组中有 2 个线程，请求 1，请求 2 和请求 4 一共会执行 2*3， 即 6 次。

线程组中有一个 Once Only Controller

———— 针对每一个当前线程，请求 3 只会执行 1 次；

———— 由于线程组中有 2 个线程，请求 3 一共会执行 2*1，即 2 次。



4. Throughput Controller

Throughput Controller，即"吞吐量控制器"。

用于性能测试中，控制当前虚拟用户发起接口请求的迭代次数，或迭代比率。这个控制器分为两个部分实现：

• Percent Execution —— 用来指定百分比

在该控制器下的采样器将执行总迭代数的特定百分比。

通过选中“Per User”复选框，可以在用户级别进行控制（当前线程，即为当前虚拟用户）。

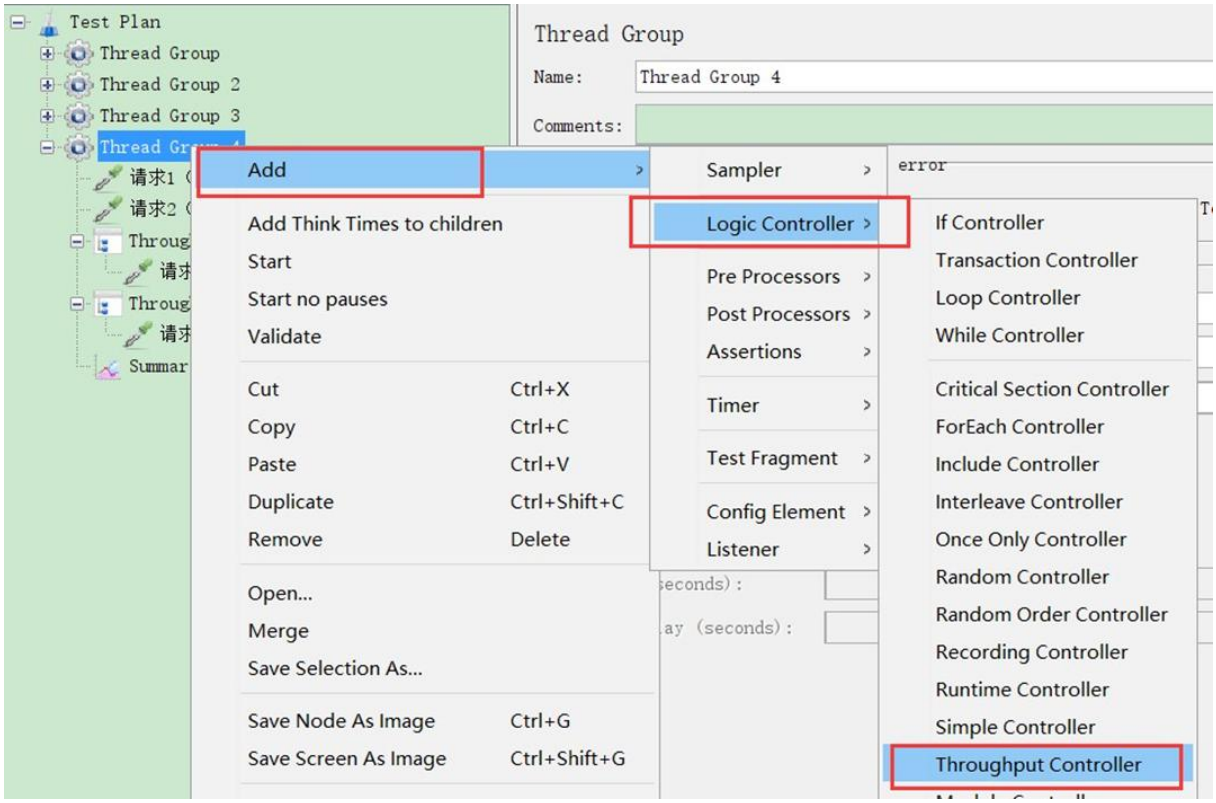
例如，线程组配置有 10 个用户，循环次数为 5，总迭代次数为 50。如果 Percent Execution 设置为 50%，则吞吐量控制器下的所有采样器将仅进行 25 次迭代（50 中的 50%）。

• Total Executions —— 用来指定迭代次数

在该控制器下的采样器将执行指定的迭代次数；通过选中“Per User”复选框，可以在用户级别进行控制（当前线程，即为当前虚拟用户）。

【 Throughput Controller _ Demo】

• Throughput Controller 设置



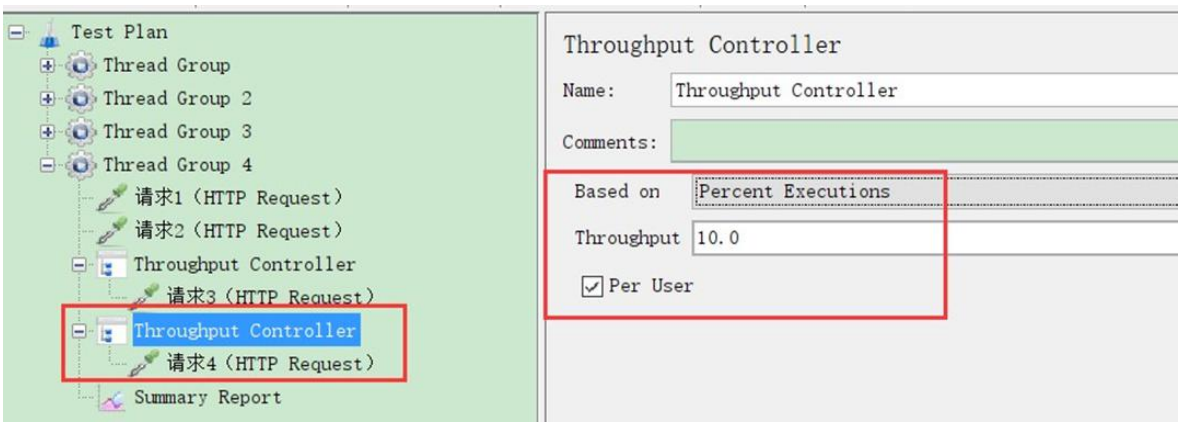
（1）添加“Throughput Controller”吞吐量控制器 ———— 选择 Total Executions

（Throughtput：15） ———— 勾 选“Per User”；

（2）在该控制器下，添加请求取样器（此处为“请求 3”）。表示：当前线程/当前虚拟用户对于指定的请求（此处为“请求 3”）只迭代 15 次；

(3) 添加“Throughput Controller”吞吐量控制器 ———— 选择 Percent Executions（Throughtput：10） ———— 勾 选“Per User”；

(4) 在该控制器下，添加请求取样器（此处为“请求 4”） 表示： 当前线程/当前虚拟用户对于指定的请求（此处为“请求 4”）只迭代线程组配置中的 10%；



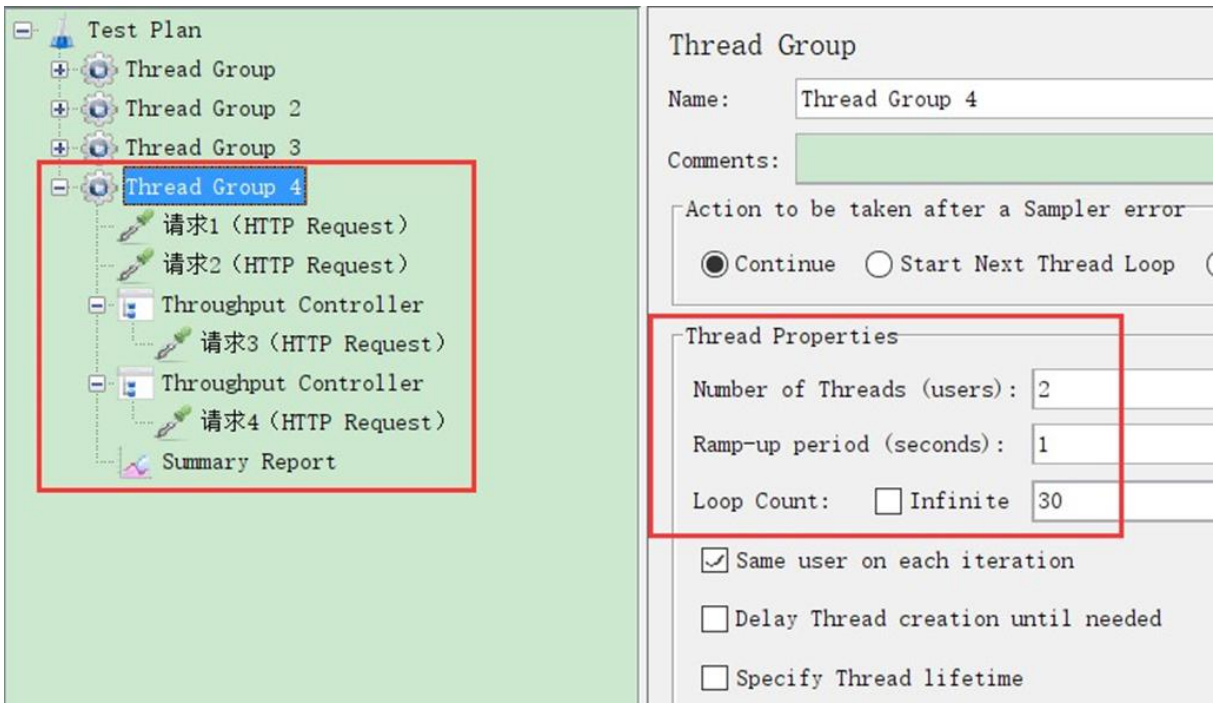
• 当前线程组设置

当前线程组中有 2 个线程（即 2 个虚拟用户），整个线程组中的每个线程需要迭代执行 30 次；

当前线程组中共有 4 个请求；

请求 3 在 Throughput Controller 中；

请求 4 在 Throughput Controller 中；



• 执行结果

总迭代数：2 * 30 = 60

请求 1， 请求 2 都分别执行了 60 次 。

请求 3

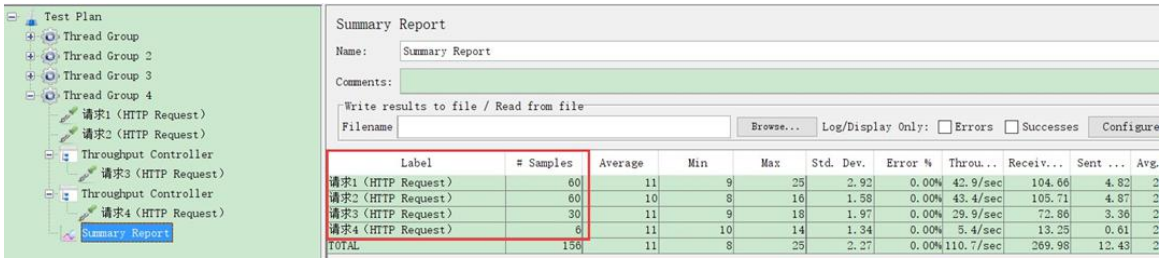
— 受到 Throughput 控制，每个线程（虚拟用户）只迭代 15 次， 线程组中一共有 2 个线程（2 个虚拟用户）；

— 总迭代次数为：15 * 2 = 30 次。

请求 4

— 受到 Throughput 控制，迭代次数是线程组配置次数的 10%；

— 总迭代次数为：60 * 10% = 6 次。



5. Interleave Controller

Interleave Controller， 即"交替控制器"。

在性能测试中，为了模拟更多交互性场景，该控制器可以通过多种方式修改请求序列，对其下的采样器进行替代选择，从而获取不同请求序列对服务端发起请求后的负载情况。

• Ignore sub-controller blocks —— 忽略子控制器模块

（1）勾选这个选项后，则会忽略当前节点下的所有控制器（交替控制器、随机控制器例外），将每个取样器作为一个单独子节点来看待；

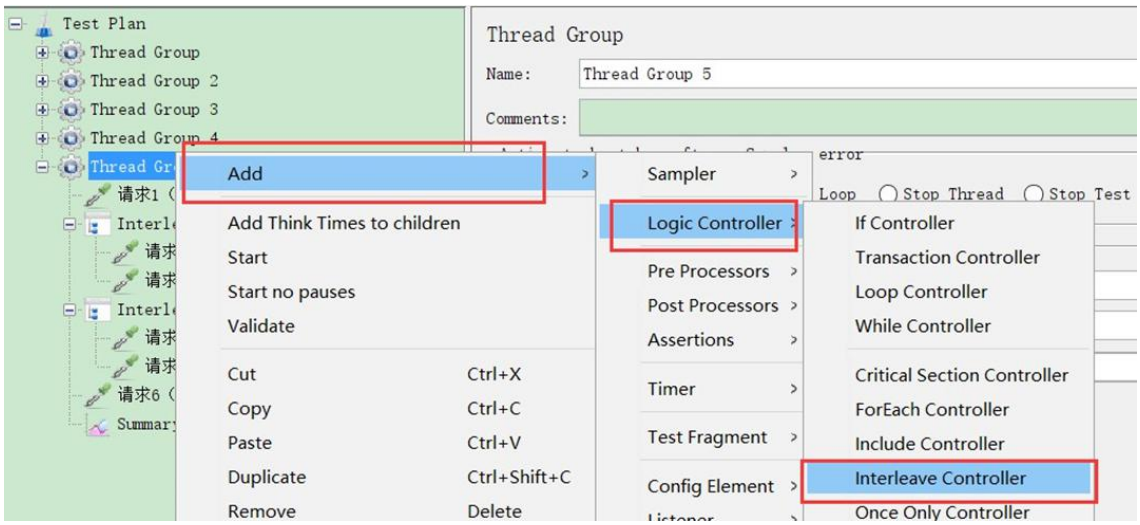
（2）如果不勾选，则在交替执行时，当前节点下，子节点中的每个取样器、逻辑控制器都认为是一个单独的子节点来交替执行。

• Interleave across threads —— 允许跨线程交替执行

勾选这个选项后，如果当前线程组中的线程数大于 1，当前线程组中，线程的首次执行，则会根据线程数顺序进行交替，后续执行按自己所属线程的上一次执行的位置进行交替。

【 Interleave Controller _ Demo】

• Interleave Controller 设置



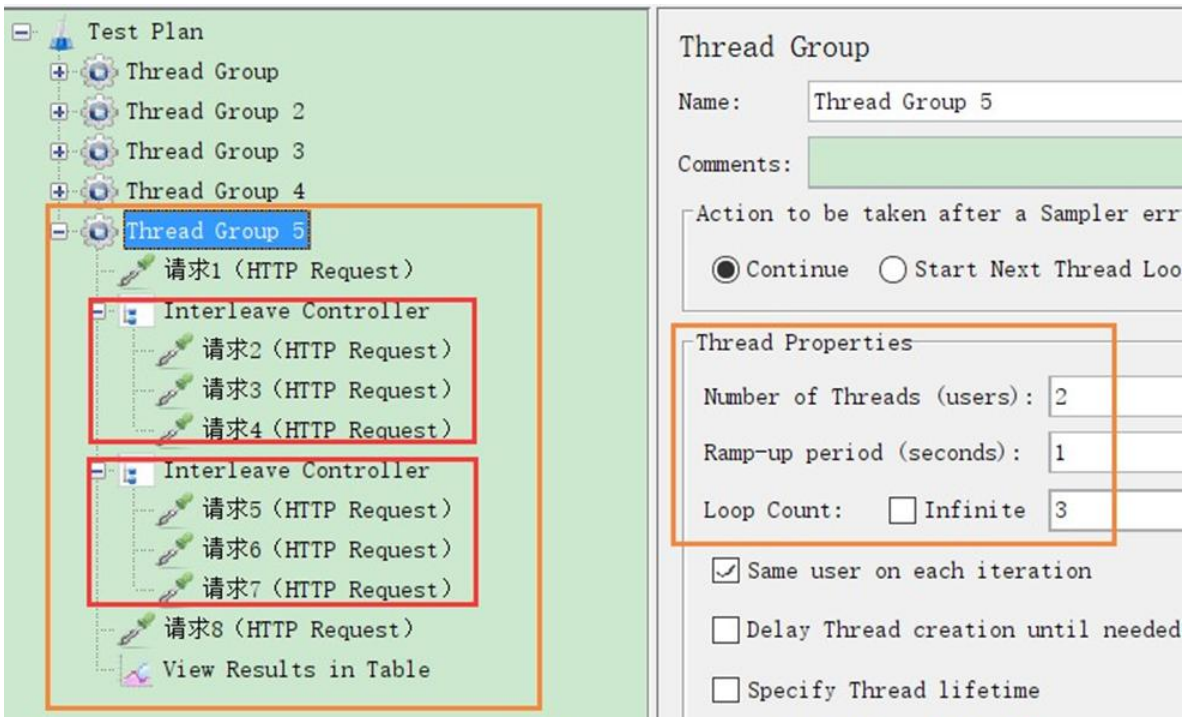
• 当前线程组设置

当前线程组中有 2 个线程（即 2 个虚拟用户），整个线程组中的每个线程需要迭代执行 3 次；

当前线程组中共有 8 个请求；

请求 2，请求 3，请求 4 在 Interleave Controller 中；

请求 5，请求 6，请求 7 在另一个 Interleave Controller 中；



• 执行结果

线程 1 和线程 2 分别都迭代了 3 次，每个迭代都发送了 4 次请求：

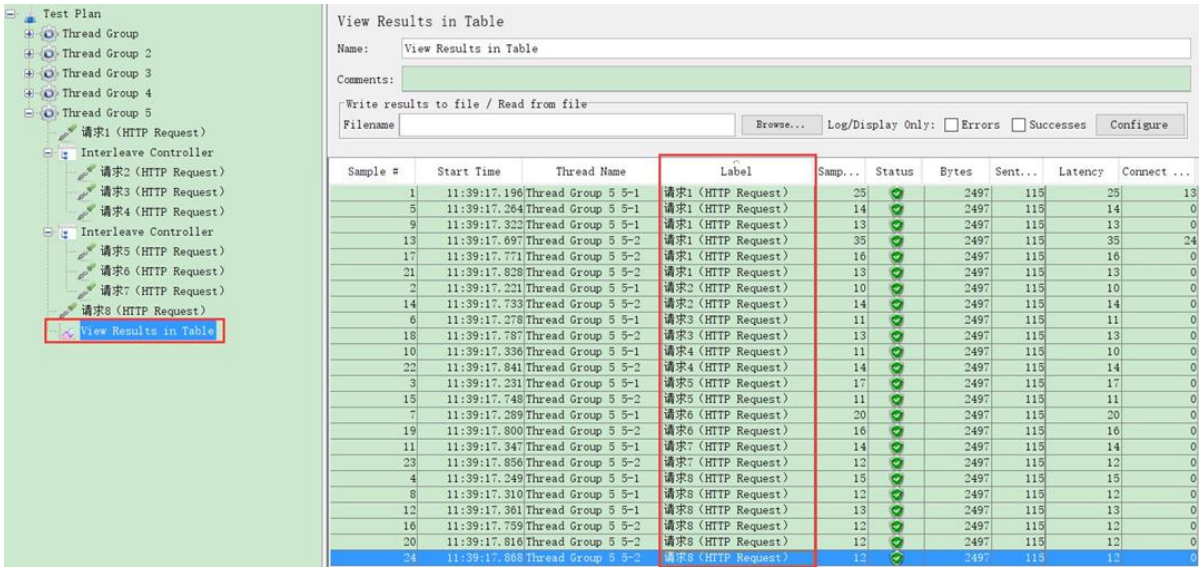
请求 1 —— 按顺序交互替换（请求 2/请求 3/请求 4） —— 按顺序交互替换（请求 5/请求 6/请求 7） —— 请求 8。

每一次迭代情况如下：

第一次迭代：请求 1 —— 请求 2 —— 请求 5 —— 请求 8

第二次迭代：请求 1 —— 请求 3 —— 请求 6 —— 请求 8

第三次迭代：请求 1 —— 请求 4 —— 请求 7 —— 请求 8



在交替控制器中分别勾选 "Ignore sub-controller blocks" 或 "Interleave across threads" 情况，大家可以自行实践验证效果，这里不多累赘了。

6. If Controller

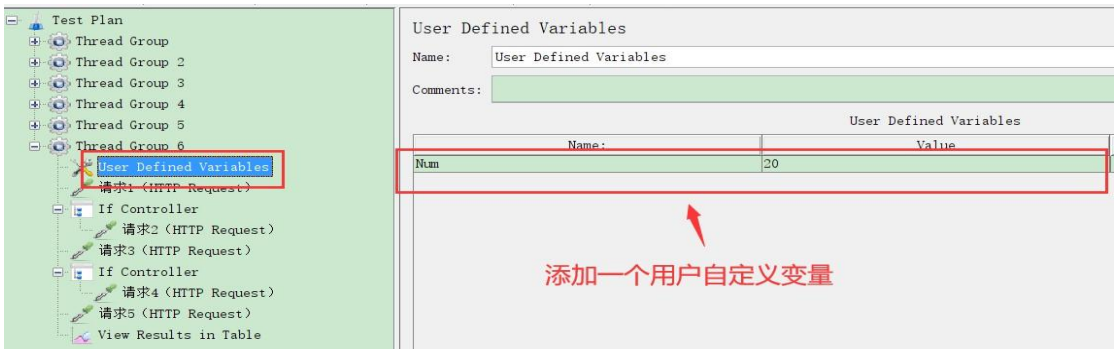
If Controller, 即"If 控制器"。

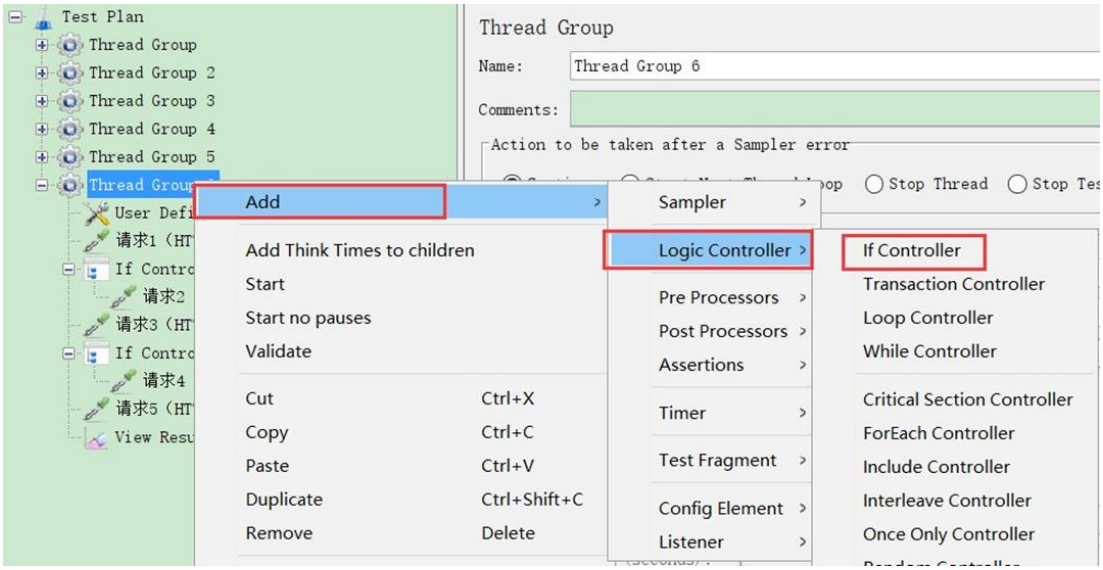
和任何编程语言处理 If 表达式一样，先验证条件，如果条件为 True，则执行此容器下的组件，否则将跳过 If 控制器，执行后续操作。

【 If Controller _ Demo】

• If Controller 设置

为了方便起见，我们在当前线程组中添加一个用户自定义变量；添加 If 控制器；针对之前用户自定义变量 "Num"， 分别设置两个 If 控制器：

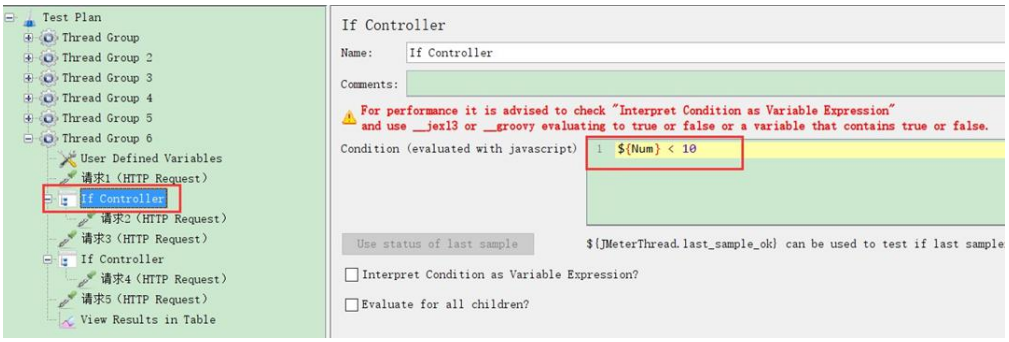


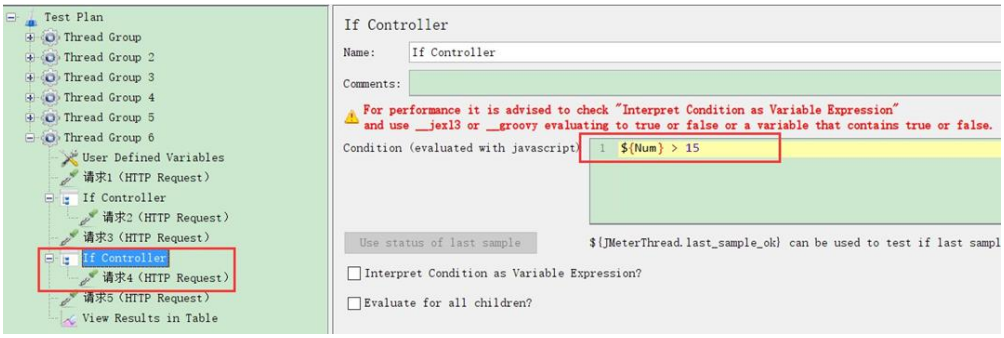


针对之前用户自定义变量“Num”，，分别设置两个 If 控制器；

${Num}<10 ———— 如果 Num 值 ＜10，则请求 2 会被执行，否则该请求不执行；

${Num}>15 ———— 如果 Num 值 ＞15，则请求 4 会被执行，否则该请求不执行；



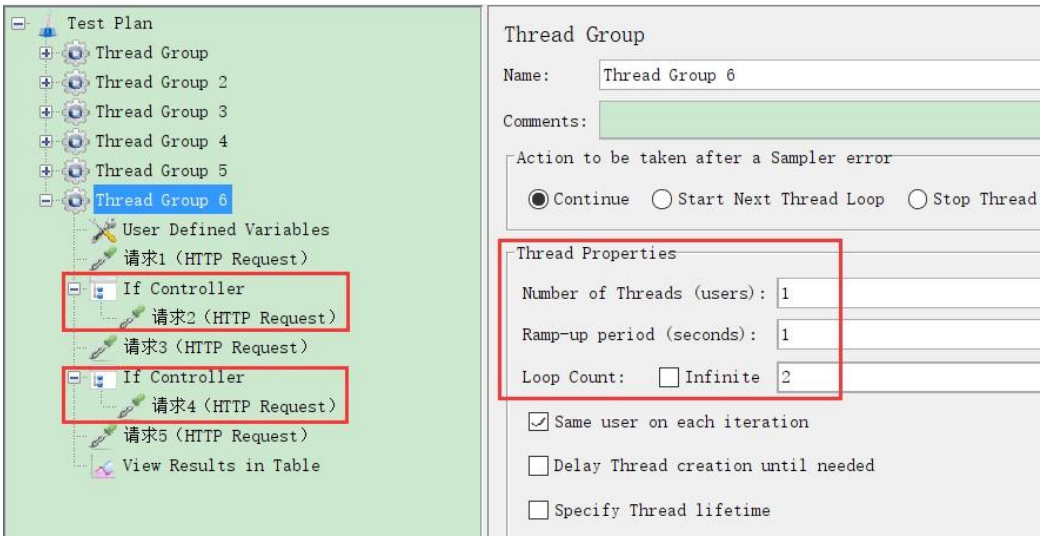


• 当前线程组设置

当前线程组中有 1 个线程（即 1 个虚拟用户），整个线程组中的每个线程需要迭代执行 2 次；

当前线程组中共有 5 个请求。

请求 2，请求 4 在 If Controller 中，只有当条件满足时，才会被执行；



• 执行结果

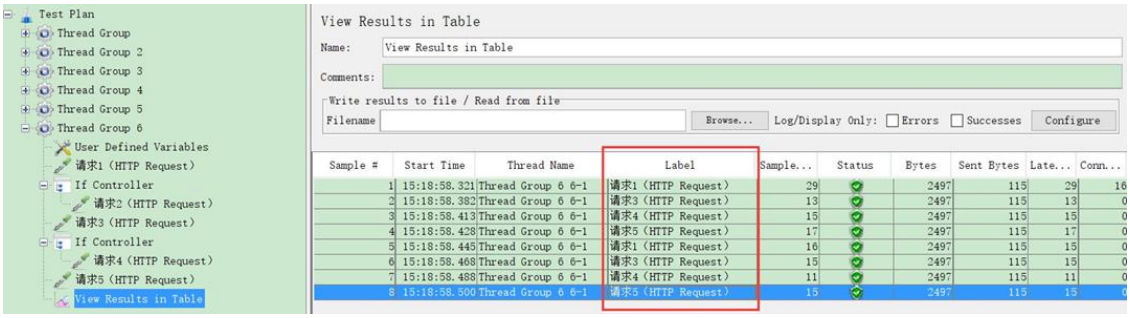
当前线程迭代了 2 次，每个迭代都发送了 4 次请求：

每一次迭代情况如下（两次请求相同）：

第一次迭代：请求 1 —— 请求 3 —— 请求 4 —— 请求 5

第二次迭代：请求 1 —— 请求 3 —— 请求 4 —— 请求 5

请求 2 由于在第一个 If 控制器中，且不满足控制器的条件，所以请求 2 不会被执行。



7. While Controller

While Controller， 即"While 循环控制器"。

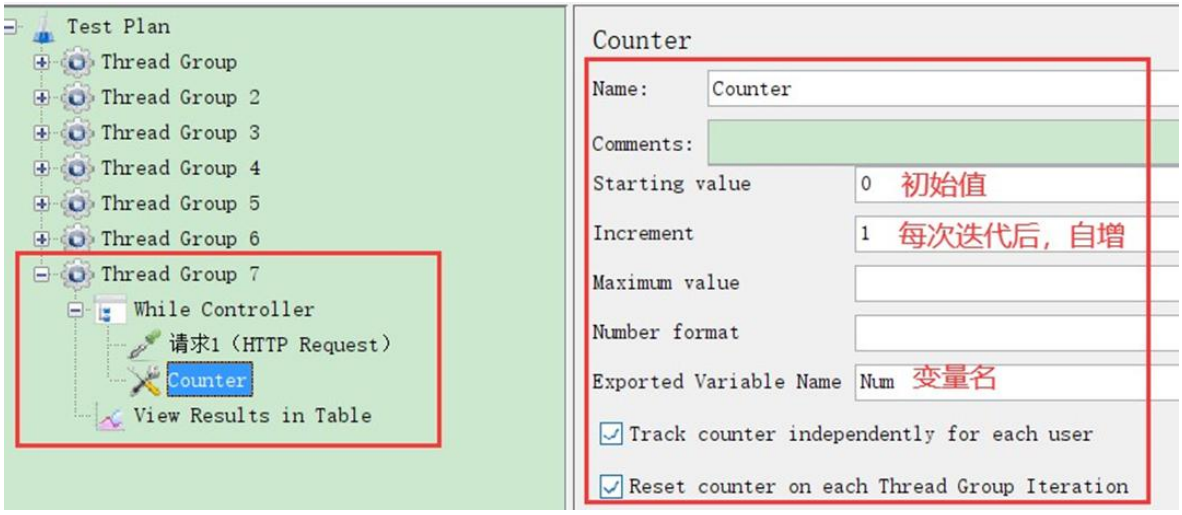
和上面的 If 控制器类似，也和任何编程语言处理 While 表达式一样，先验证条件，如果条件为 True，则执行此容器下的组件，直到条件变为假时，跳出当前 While 循环。

假设有一个条件 while($(num)<5)，若循环条件满足，则将执行该控制器下的组件，直到条件为假。下面我们结合计数器，在每次迭代时递增 num 的数量，然后评估条件。

【 While Controller _ Demo】

• While Controller 设置

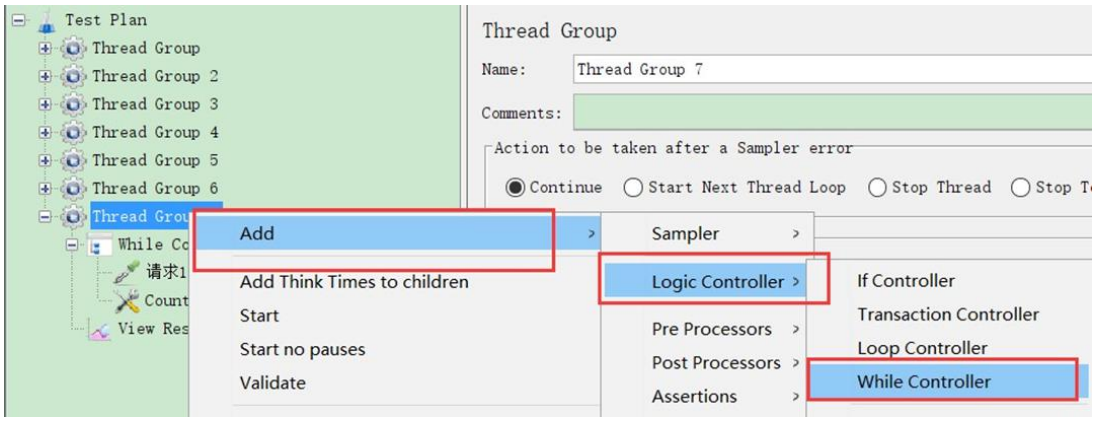
为了方便起见，我们在当前线程组中添加一个计数器，用于控制循环条件；

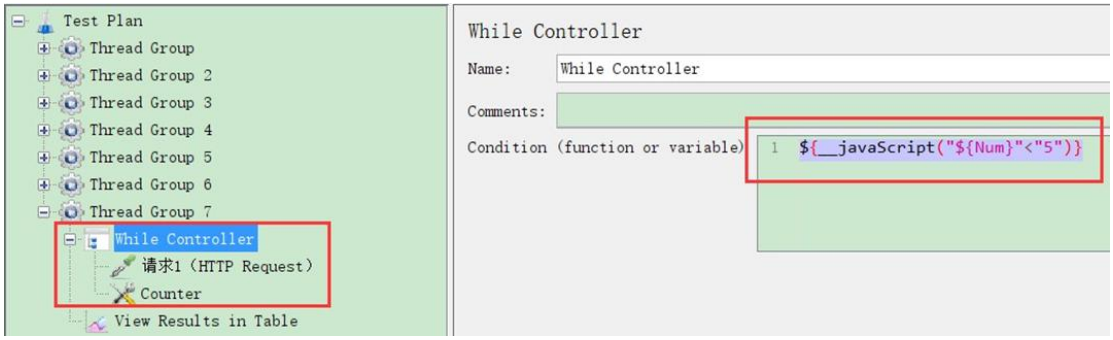


添加 While 循环控制器，并将计数器放到 While 循环控制器中，便于每次循环后变量更新；

设置循环控制条件为当前计数不足 5 的时候，进入循环，否则终止当前循环，跳出 While 循环控制器：

${__javaScript("${Num}"<"5")}

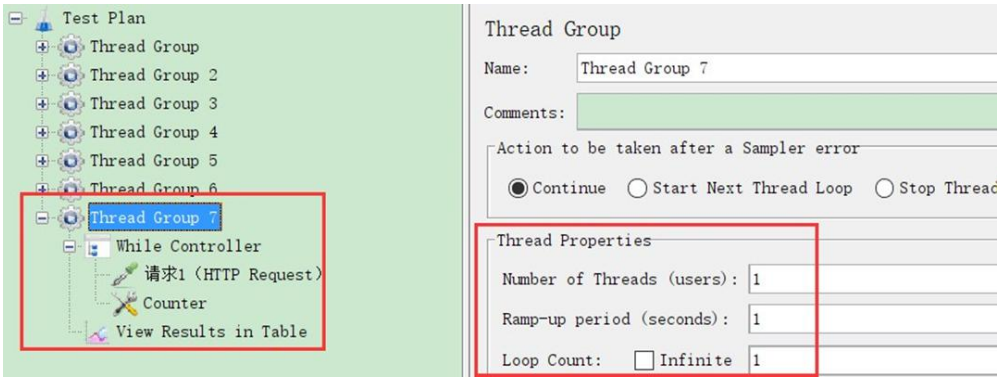




• 当前线程组设置

简单设置当前线程组，只有一个线程，即一个虚拟用户；

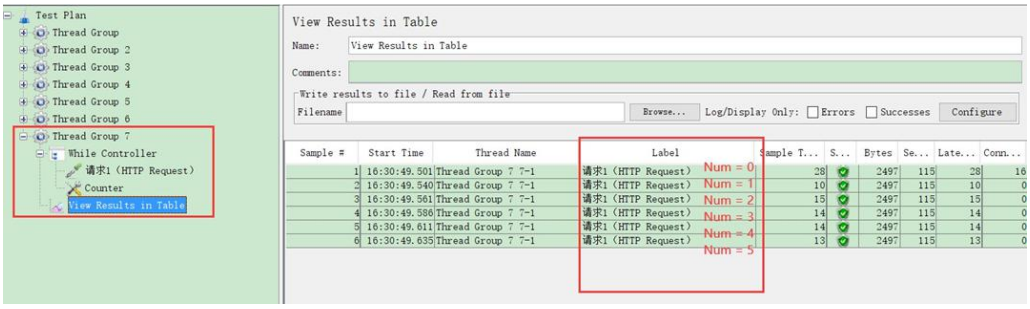
线程组中有一个 While 循环控制器，该控制器下面包含一个请求，一个计数器；



• 执行结果

对于 While 循环，迭代器的值，每次都是进入下一次循环后才增加的，而不是进入之前增加的。

当前设置 while 循环条件为 Num 值小于 5 才进入循环，但是发现 Num 值等于 5 的时候依然会发送一个请求。



以上我们就 JMeter 中常见的逻辑控制器做了综合梳理。

在具体接口性能测试任务中，往往需要通过逻辑控制器控制不同场景下的测试脚本，同时也会存在控制器嵌套的使用场景。但无论业务逻辑多么复杂，每个控制器的作用域仅针对当前范畴内的组件，且各司其职。

希望通过本文的学习，能够有助于你对 JMeter 常用逻辑控制器的理解，有效落实到今后的工作中去。”

Loadrunner 脚本中对 header 的处理实践

◆作者：云竹

一.问题描述

近期在某项目的性能测试中，录制脚本时出现一个问题，脚本回放时报 500 错误，采用关联、参数化等各种技术，都无法解决。

二.问题分析与解决

• 问题分析

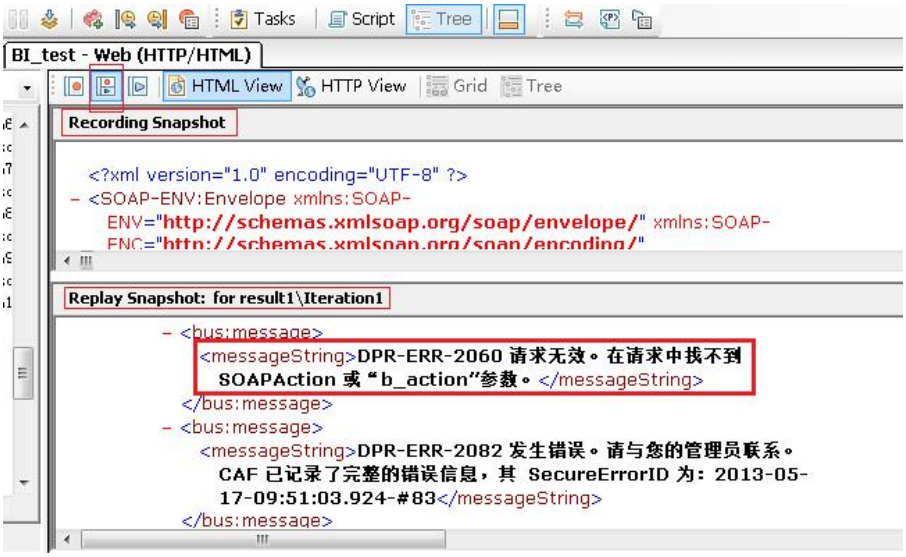
脚本回放时的错误如下所示：

```
web_custom_request("dispatch_4",
    "URL=http://10.233.73.37:9080/p2pd/servlet/dispatch",
    "Method=POST",
    "Resource=0",
    "RecContentType=text/xml",
    "Referer=http://10.233.73.37:9080/p2pd/explore/explore.htm",
    "Snapshot=t147.inf",
    "Mode=HTTP",
```

Replay Log | Recording Log | Correlation Results | Generation Log

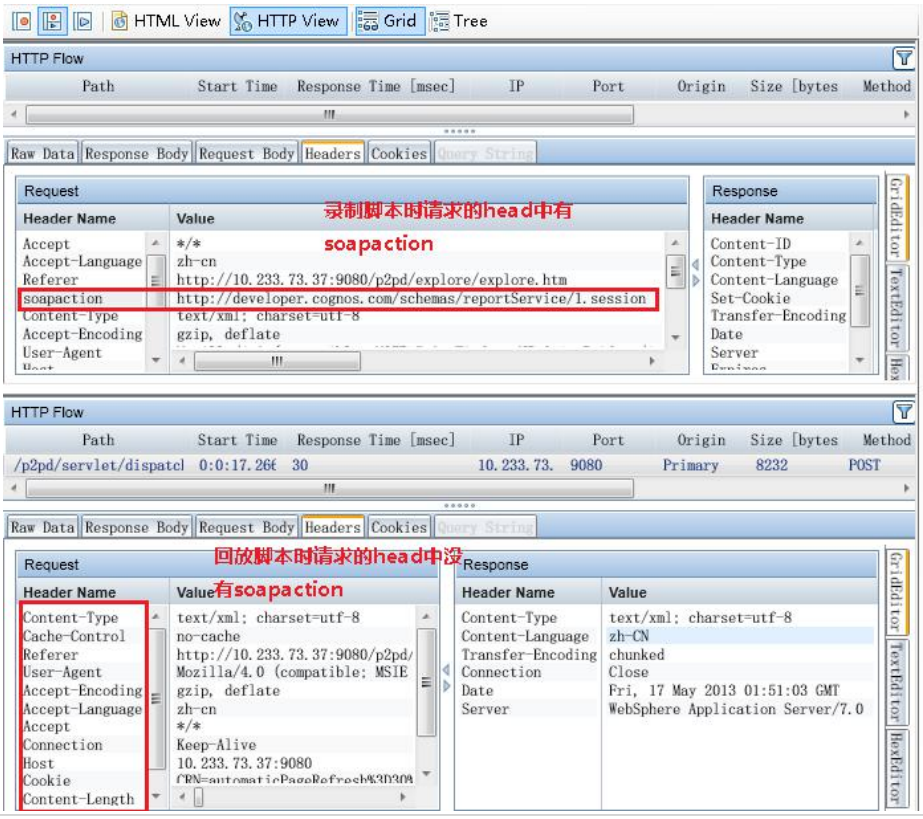
```
: Registering web_url("icon_chart.gif") was successful     [MsgId: MMSG-26390]
: Registering web_url("calculation16x16.gif") was successful     [MsgId: MMSG-26390]
: Registering web_url("Custom_set.gif") was successful     [MsgId: MMSG-26390]
: Registering web_url("swapRowColumn.gif") was successful     [MsgId: MMSG-26390]
: Registering web_url("tool_main_saveas.gif") was successful     [MsgId: MMSG-26390]
: web_concurrent_end was successful, 4548 body bytes, 5372 header bytes   [MsgId: MMSG-26386]
: Notify: Parameter Substitution: parameter "WCSParam6" =  "CAFW00000070Q0FGQTNjMDAwMDAwMGFGQUFBQUhCMi1YQ1ZwcONMZjRNaE
: Notify: Parameter Substitution: parameter "WCSParam7" =  "71ee5c3a0e9280f26e4dcfc7317bc8bbd7525c6b"
: Notify: Parameter Substitution: parameter "WCSParam8" =  "cad16b72f448a5f8908327a802a55d7948df9a27"
: Notify: Parameter Substitution: parameter "WCSParam9" =  "1434586338"
: Notify: Parameter Substitution: parameter "WCSParam10" =  "8b8d2ea9524213304cc237bfcec40ade0e1278b2"
: Error -26612: HTTP Status-Code=500 (Internal Server Error) for "http://10.233.73.37:9080/p2pd/servlet/dispatch"
: web_custom_request("dispatch_4") highest severity level was "ERROR", 8232 body bytes, 228 header bytes, 13 chunking
: Notify: Transaction "renyuanjiegoufenxi" ended with "Fail" status (Duration: 2.6534 Wasted Time: 0.4163).
```

通过脚本的 tree 视图，对比录制和回放时的页面快照，可以看到回放时的快照中有明显的错误信息，如下所示：



通过这个错误提示，应该是在 web_custom_request("dispatch_4",…..)这个请求的语句中缺少了 SOAPAction 或 b_action 参数，这个参数应该加到哪里呢？为什么没有录制出来呢？又尝试手工将这个参数放到脚本语句中，也尝试采用 port mapping 的方式录制脚本，都没有效果。

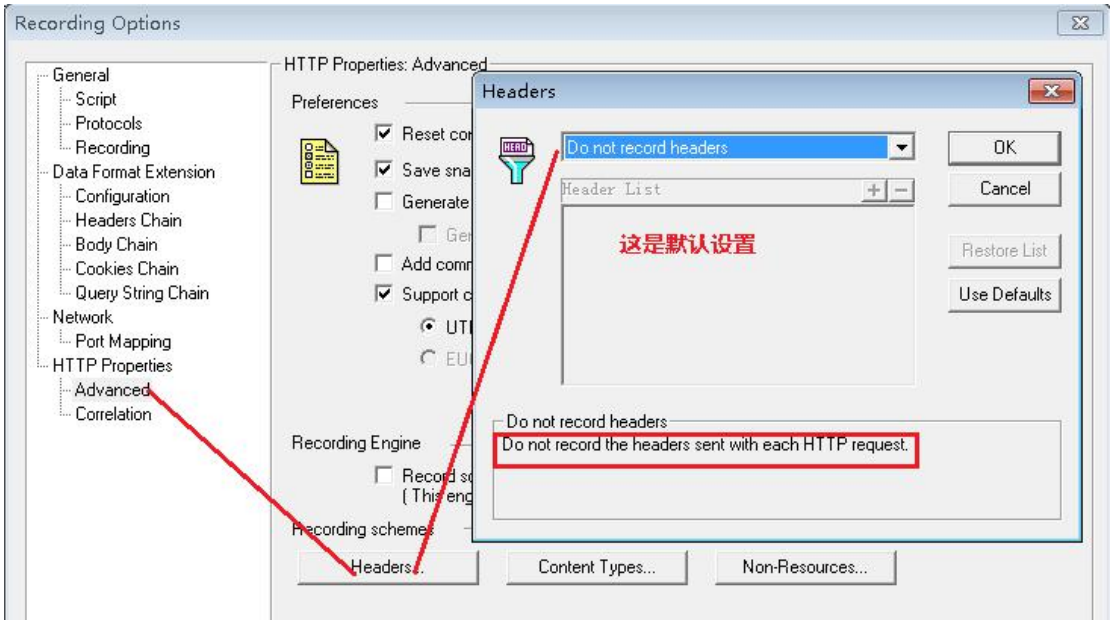
上面对比的是录制和回放时的快照视图，还可以对比 http 视图，在该视图中，终于发现了问题所在，对比 http 请求的 head 部分时，在录制时有一个 soapaction 的 header，而在回放时没有这个 header，脚本回放中正是缺少 soapaction：



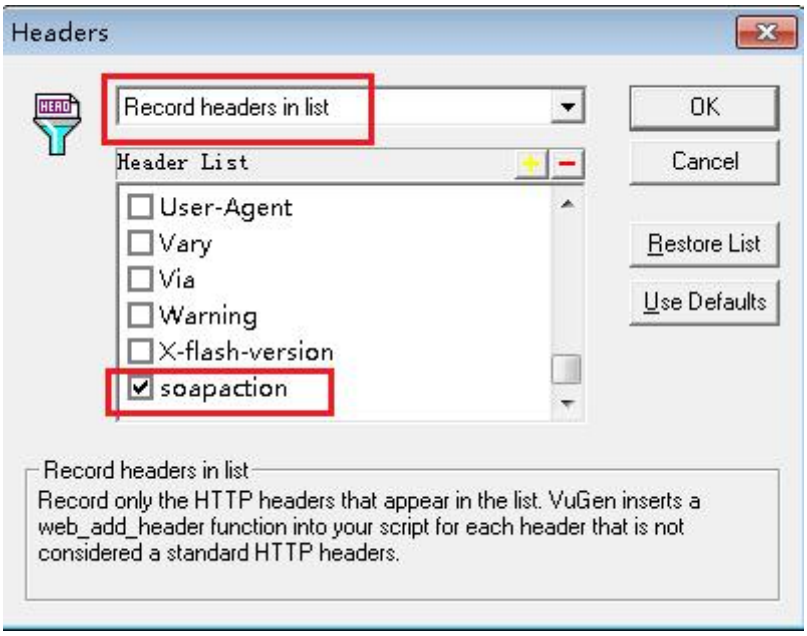
分析到这里已经知道，只要将 soapaction 这个 header 加入到脚本中，问题应该就可以解决了。

• 问题解决

在 loadrunner 脚本的录制中，默认是不录制 header 信息的，如下所示：



现在需要将 soapaction 这个 header 信息录制到脚本中，需要选择录制 header 选项，如下所示：



由于默认的 header 信息中没有 soapaction，手动加上一个并选中，保存设置后重新录制脚本，在 web_custom_request("dispatch_4",…..)前面就多了一条 web_add_header 语句，如下所示：

```
web_add_header("soapaction",
    "http://developer.cognos.com/schemas/reportService/1.session");

web_custom_request("dispatch_4",
    "URL=http://10.233.73.37:9080/p2pd/servlet/dispatch",
    "Method=POST",
    "Resource=0",
    "RecContentType=text/xml",
    "Referer=http://10.233.73.37:9080/p2pd/explore/explore.htm",
    "Snapshot=t147.inf",
    "Mode=HTTP",
    "EncType=text/xml; charset=utf-8",
    "Body=<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=\"http://schemas.xmlsoa
    ":type=\"bus:CAF\"><bus:contextID xsi:type=\"xs:string\">{WCSParam6}</b
    ":string\">productLocale</bus:name><bus:value xsi:type=\"xsd:string\">z
    "appname&gt;&lt;requestId&gt;request_i15&lt;/requestId&gt;&lt;/clientst
    "parameter&gt;&lt;parameter&gt;&lt;name&gt;default&lt;/name&gt;&lt;valu
    "xsi:type=\"bus:encodingEnum\">base64</bus:value></item><item xsi:type=
    "!__/content/folder[@name=&#39;BI_REPORT&#39;]/package[@name=&#39;
    LAST);
```

重新回放脚本错误消失，问题得到解决。

三.总结

本次问题分析过程中，重点使用了录制和回放的对比功能，对比了 html 快照视图和 http 请求响应视图，这对于分析脚本回放的错误尤为重要，header 处理也是脚本回放中需要重点关注的，由于 loadrunner 默认不做录制，可能经常会被忽视，希望通过本文，大家在遇到类似问题时可以多一条思路。

LoadRunner 负载测试系列二

◆ 作者：枫叶

LoadRunner 协议推荐器

我们来认识一下如何使用强大的 LoadRunner 协议推荐器。

首先 LoadRunner 提供了一个典型的工作流，这个工作流可以利用协议推荐器来找到可以录制我们的应用程序最适合的协议，该工作流如下：

1)启动协议推荐器。

进入 LoadRunner 软件首页，选择文件>协议推荐器>分析应用程序，弹出协议推荐器对话框；

2)运行典型的逻辑业务流程；

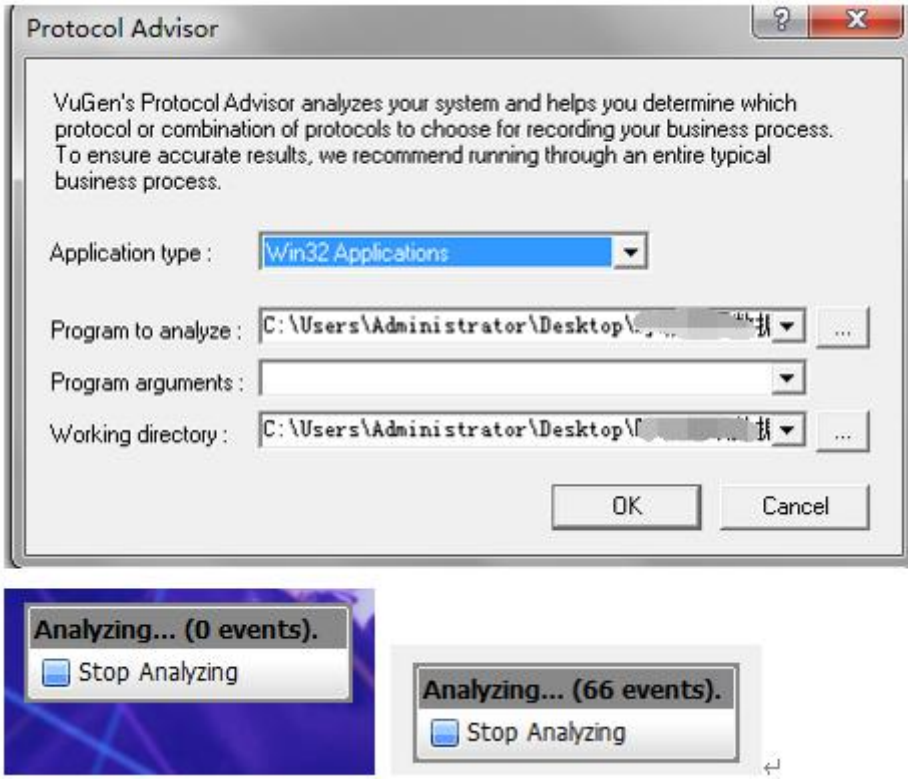
3)保存运行结果；

4)选择一个协议并且创建一个新虚拟用户的脚本；

5)进行优化、调试，并且验证回放；

6)如果不能成功回放，选择另一个协议并且重复上述的步骤。

我们来具体实践一下如何使用这个便捷的 LoadRunner 协议推荐器。打开 LoadRunner 软件，选择菜单项 Create/Edit Script，打开首页，选择“ File > Protocol Advisor > Analyze Application”，在弹出的协议推荐器对话框中填写应用程序类型、程序安装位置、工作目录，点击 OK 按钮。



选择合适协议

我们看到协议推荐结果：

Application name: C:\Users\Administrator\Desktop\软件***.exe\

Program arguments:

We detected the following protocols in your business process:

• Web (HTTP/HTML)

通过推荐，我们准确地得到了 LoadRunner 应该选择 Web（HTTP/HTML）协议。

创建脚本

现在我们开始创建脚本。点击 New script 按钮，录制应用程序业务逻辑后，得到的 vuser_init 和 vuser_end。

vuser_init

vuser_init 部分：

```
vuser_init()
{
//访问登录接口，方法 POST
  web_submit_data("login",
        "Action=http://***/login",
        "Method=POST",
        "EncType=multipart/form-data",
        "RecContentType=application/json",
        "Referer=",
        "Snapshot=t1.inf",
        "Mode=HTML",
        ITEMDATA,
          //填写登录名、密码，type=2 代表在客户端上登录
        "Name=username", "Value=user", ENDITEM,
        "Name=password", "Value=pwd", ENDITEM,
        "Name=type", "Value=2", ENDITEM,
        "Name=api_os_code", "Value=F0:BF:97:E2:C9:B3", ENDITEM,
        "Name=api_os_model", "Value=6.1.7601.65536", ENDITEM,
        "Name=api_version_code", "Value=Win32NT", ENDITEM,
        EXTRARES,
        "Url=getauth
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf", "Referer=", ENDITEM,
        LAST);
//登录用户 token1 验证
  web_submit_data("checktoken",
        "Action=http://***/users/checktoken",
        "Method=POST",
        "EncType=multipart/form-data",
        "RecContentType=application/json",
```

```
            "Referer=",
            "Snapshot=t2.inf",
            "Mode=HTML",
            ITEMDATA,
            "Name=uid", "Value=user", ENDITEM,
            "Name=tr_id", "Value=trid", ENDITEM,
            "Name=oauth_token", "Value=817f64a82c54f47ad974311cb9308baa", ENDITEM,
            "Name=oauth_token_secret", "Value=12770d793561a92c79628d2104d653cf", ENDITEM,
            "Name=type", "Value=2", ENDITEM,
            EXTRARES,
            "Url=../cases/mylock
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf&type=2&page=1&limit=30&status=0&caseid=0&case_type=0", "Referer=",
ENDITEM,
            LAST);
    //登录用户 token2 验证
      web_submit_data("checktoken_2",
            "Action=http://***/users/checktoken",
            "Method=POST",
            "EncType=multipart/form-data",
            "RecContentType=application/json",
            "Referer=",
            "Snapshot=t3.inf",
            "Mode=HTML",
            ITEMDATA,
            "Name=uid", "Value=userid", ENDITEM,
            "Name=tr_id", "Value=trid", ENDITEM,
            "Name=oauth_token", "Value=817f64a82c54f47ad974311cb9308baa", ENDITEM,
            "Name=oauth_token_secret", "Value=12770d793561a92c79628d2104d653cf", ENDITEM,
            "Name=type", "Value=2", ENDITEM,
            EXTRARES,
            "Url=../chronic/mbpatpist
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf& type=2&page=1&limit=30", "Referer=", ENDITEM,
```

```
            LAST);
        //登录用户 token3 验证
          web_submit_data("checktoken_3",
            "Action=http://***/users/checktoken",
            "Method=POST",
            "EncType=multipart/form-data",
            "RecContentType=application/json",
            "Referer=",
            "Snapshot=t4.inf",
            "Mode=HTML",
            ITEMDATA,
            "Name=uid", "Value=userid", ENDITEM,
            "Url=../devices/getlist
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf&type=2&page=1&limit=30", "Referer=", ENDITEM,
            LAST);
        //登录用户 token4 验证
          web_submit_data("checktoken_4",
            "Action=http://***/users/checktoken",
            "Method=POST",
            "EncType=multipart/form-data",
            "RecContentType=application/json",
            "Referer=",
            "Snapshot=t5.inf",
            "Mode=HTML",
            ITEMDATA,
            "Name=uid", "Value=userid", ENDITEM,
            "Name=tr_id", "Value=trid", ENDITEM,
            "Name=oauth_token", "Value=817f64a82c54f47ad974311cb9308baa", ENDITEM,
            "Name=oauth_token_secret", "Value=12770d793561a92c79628d2104d653cf", ENDITEM,
            "Name=type", "Value=2", ENDITEM,
            EXTRARES,
            "Url=../getlist
```

```
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf&type=2&page=1&limit=30", "Referer=", ENDITEM,
                LAST);
        //登录用户 token5 验证
          web_submit_data("checktoken_5",
                "Action=http://***/users/checktoken",
                "Method=POST",
                "EncType=multipart/form-data",
                "RecContentType=application/json",
                "Referer=",
                "Snapshot=t6.inf",
                "Mode=HTML",
                ITEMDATA,
                "Name=uid", "Value=", ENDITEM,
                "Name=tr_id", "Value=", ENDITEM,
                "Name=oauth_token", "Value=817f64a82c54f47ad974311cb9308baa", ENDITEM,
                "Name=oauth_token_secret", "Value=12770d793561a92c79628d2104d653cf", ENDITEM,
                "Name=type", "Value=2", ENDITEM,
                EXTRARES,
                "Url=../getlist
uid=&tr_id=&oauth_token=817f64a82c54f47ad974311cb9308baa&oauth_token_secret=12770d793561a92c
79628d2104d653cf&type=2&page=1&limit=30", "Referer=", ENDITEM,
                LAST);
          return 0;
        }
```

vuser_end

vuser_end 部分：

```
        vuser_end()
        {
          lr_think_time(8);
          web_submit_data("logout",
                "Action=http://***/users/logout",
```

```
        "Method=POST",
        "EncType=multipart/form-data",
        "RecContentType=application/json",
        "Referer=",
        "Snapshot=t7.inf",
        "Mode=HTML",
        ITEMDATA,
        "Name=uid", "Value=uid", ENDITEM,
        "Name=tr_id", "Value=trid", ENDITEM,
        "Name=oauth_token", "Value=817f64a82c54f47ad974311cb9308baa", ENDITEM,
        "Name=oauth_token_secret", "Value=12770d793561a92c79628d2104d653cf", ENDITEM,
        "Name=type", "Value=2", ENDITEM,
        LAST);
    return 0;
}
```

由此可以反过来验证，LoadRunner 推荐的协议是适合的。

检查点

检查点函数，我们来了解两个常用得函数：web_find()和 web_reg_find()。

这两个函数均用于内容的查找，但两者也有本质的区别。

web_find()

该函数的作用是“在页面中查找相应的内容”，常用参数及含义如下：

```
web_find("web_find",    //定义该查找函数的名称    "RightOf=a",    //定义查找字符的右边界
"LeftOf=b",    //定义查找字符的左边界
"What=name",    //定义查找内容
LAST);
```

- 使用该函数注意以下事项：

1、位置

该函数在页面内容显示出来以后，在页面中进行查找，所以只能写在要查找内容之后

2、录制模式

该函数只能在基于 HTML 模式录制的脚本中进行查找

3、必须启用内容检查选项

在 runtime setting->Preferences 里面，把 Enable image and text check 选中，否则不执行该查找函数。

4、在 VB 和 JAVA 语法中不支持该函数。

该函数有以下一个缺点：

1、执行效率较低。

2、不返回查找结果情况，如想在执行该函数后根据查找结果做进一步操作时，没有返回值可以依据。

例如：

在页面中查找“登录成功”的字符串，如果找到该字符串在日志中输出“登录成功”，如果找不到该字符串，则在日志中输出“登录失败”，此时使用该函数没有依据来做此判断，但使用 web_reg_find()函数，使用它其中的 SaveCount 可以进行判断，具体方法我们下面介绍。

web_reg_find()

该函数的作用是“在缓存中查找相应的内容”，常用参数及含义如下：

```
web_reg_find("Search=Body",   //定义查找范围
"SaveCount=ddd",              //定义查找计数变量名称     "Text=aaaa",                  //定义查找内容
LAST);
```

• 使用该函数注意以下事项：

1、位置

该函数写在要查找内容的请求之前，通常情况下写在如下六个函数之前：

Web_castom_request();web_image();web_link();web_submit_data();web_submit_form();web_url()

检查点验证

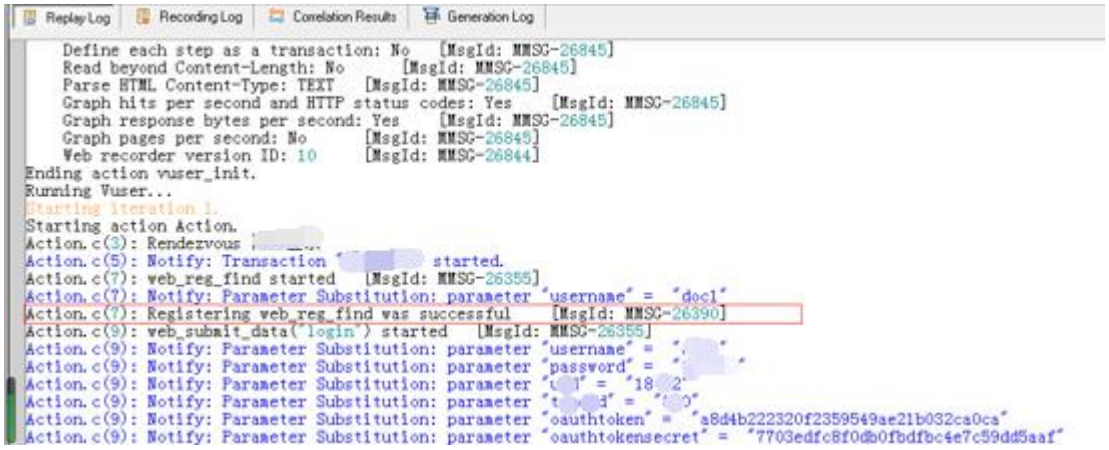
设置一个检查点，运行场景通过.如何验证检查点是否运行正常呢？

我们可以在 verify replay 结束之后，查看 log。

当 log 中出现“Registering web_reg_find was successful”说明查找成功了。

对脚本进行参数化之后，需要再次 verify replay，查看 Replay Log。

当我们看到“Registering web_reg_find was successful”，说明参数化后的脚本也没有问题了。



再运行 Controller 时，不需要再进行检查点检查了，把检查点函数注释掉，同时也能避免 26366: "Text=" not found for web_reg_find 的错误。

参数化

登录用户参数化代码如下：

```
Action()
{
  lr_rendezvous("同时登录");
  lr_start_transaction("登录事务");
  web_reg_find("Text={username}","Search=Body",LAST);
  web_submit_data("login",
        "Action=http://***/v2/users/login",
        "Method=POST",
        "EncType=multipart/form-data",
        "RecContentType=application/json",
        "Referer=",
        "Snapshot=t8.inf",
        "Mode=HTML",
        ITEMDATA,
        "Name=username", "Value={username}", ENDITEM,
        "Name=password", "Value={password}", ENDITEM,
        "Name=api_type", "Value=2", ENDITEM,
        "Name=api_os_code", "Value=F4:8E:38:A2:61:41", ENDITEM,
        "Name=api_os_model", "Value=6.1.7601.65536", ENDITEM,
        "Name=api_version_code", "Value=Win32NT", ENDITEM,
        EXTRARES,
        "Url=http://***/v2/users/getauth
uid={uid}&tree_id={treeid}&oauth_token={oauthtoken}&oauth_token_secret={oauthtokensecret}&api_type=2", "Referer=", ENDITEM,
        LAST);
  lr_end_transaction("登录事务",LR_AUTO);
  lr_start_transaction("退出事务");
  web_submit_data("logout",
        "Action=http://***/v2/users/logout",
        "Method=POST",
        "EncType=multipart/form-data",
        "RecContentType=application/json",
        "Referer=",
```

```
        "Snapshot=t14.inf",
        "Mode=HTML",
        ITEMDATA,
        "Name=uid", "Value={uid}", ENDITEM,
        "Name=tree_id", "Value={treeid}", ENDITEM,
        "Name=oauth_token", "Value={oauthtoken}", ENDITEM,
        "Name=oauth_token_secret", "Value={oauthtokensecret}", ENDITEM,
        "Name=api_type", "Value=2", ENDITEM,
        LAST);
    lr_end_transaction("退出事务",LR_AUTO);
    return 0;
}
```

并发测试的经验教训

去年项目有一个性能测试需求，即在最低配置下单一系统上最大并发用户数应该满足至少 30 个用户同时稳定运行。

刚开始我们粗略地理解为 1 个用户登录后同时创建 30 个用户，走了一个弯路，经过沟通明白项目的性能测试要求是 30 个用户同时登录并操作提交表单，又重新设计性能测试计划和编写测试脚本。我们在项目性能测试实施之前，需要进行充分沟通，了解并确认性能的需求，就如同了解功能测试的需求一样重要，以正确的性能测试目标作为导向是迈出性能测试成功的关键一步。

接下来，我们进行本次性能测试分析并得出性能测试的方案：

需要模拟 30 个用户，那么在后台网站中需要添加 30 个测试用户数据。添加用户之后，还有个认证的业务流程，认证之后用户才可以同时登录并进行并发操作。

如果手动添加 30 个测试用户尚可在十多分钟内完成，但是添加用户之后“认证”这一步过程，不能不说是一项重复性的工作。我们可能会提出这样一个问题：添加用户并认证的过程可以进行用自动化实现吗？这个问题问得很好，只要我们想到了，就可以实现的。

结合 selenium 自动化测试技术，我们把添加用户并认证这一部分用自动化测试脚本

实现了。于是相对轻松地产生了批量的测试数据。从可扩展性来讲，后续如果需要模拟更多用户，也是能够延用这种方法的。

回到并发测试 loadrunner 实践中，我们想要实现并发测试，主要通过以下三个步骤：

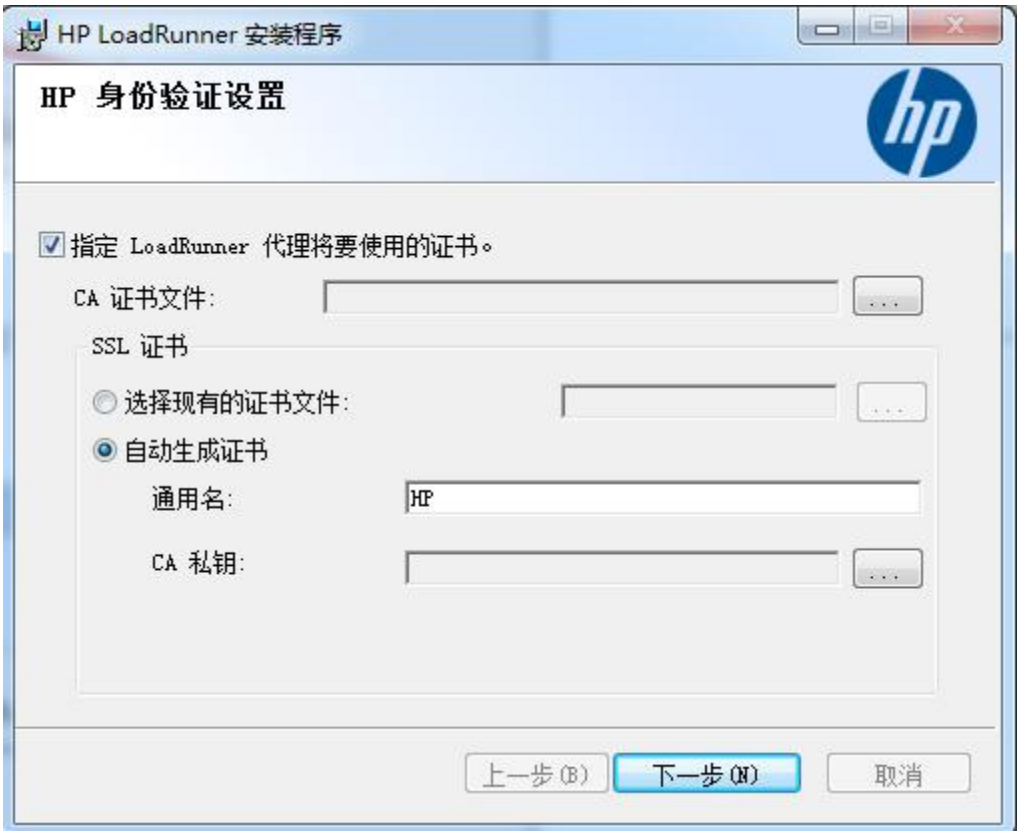
步骤 1 录制产生脚本，进行脚本优化，加入集合点并插入事务；

步骤 2 回放，并运行场景，生成报告，确保单个用户的脚本可用；

步骤 3 继续优化脚本，进行参数化；

步骤 4 回放，并运行场景，最终生成性能测试报告。

我们通过图解来回顾一下整个 LoadRunner 性能测试过程。

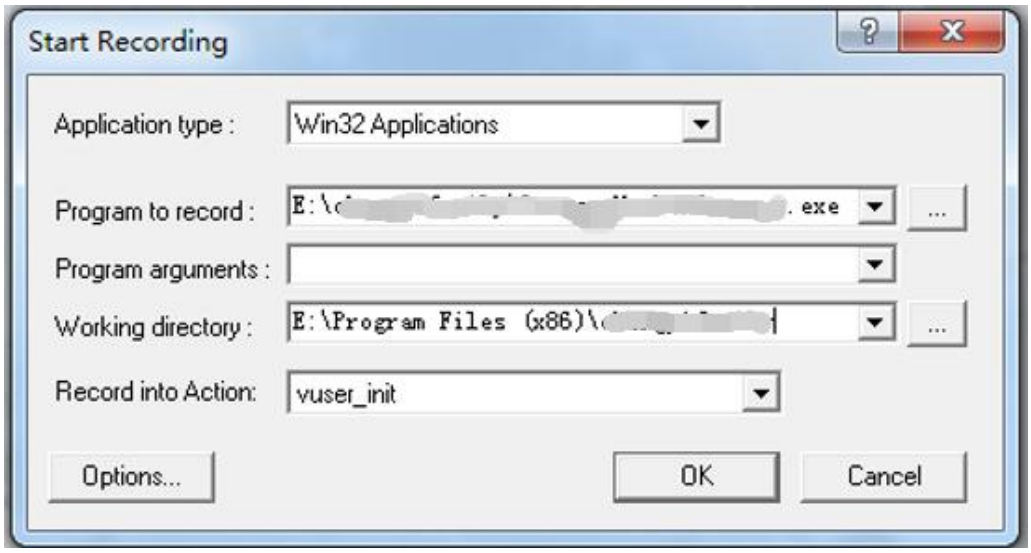


安装

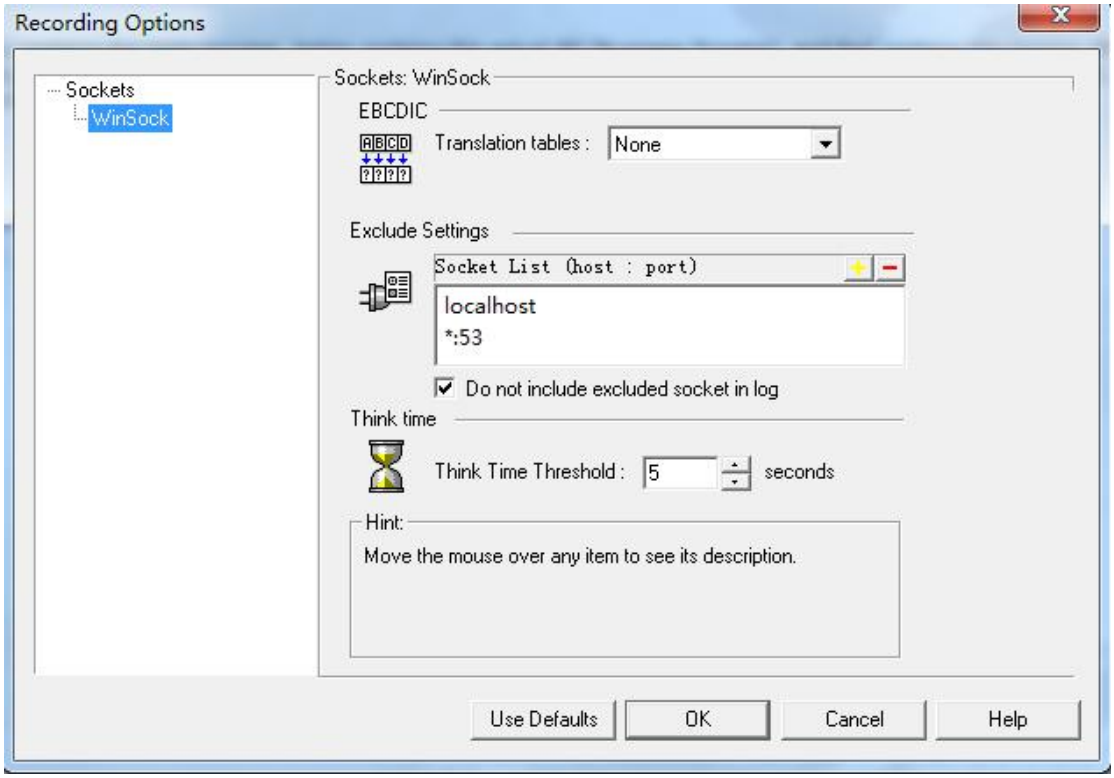
注意：

去掉勾选”指定 LoadRunner 代理将要使用的证书”选项；

不能同时装两个版本，装低版本时需要卸载掉高版本。

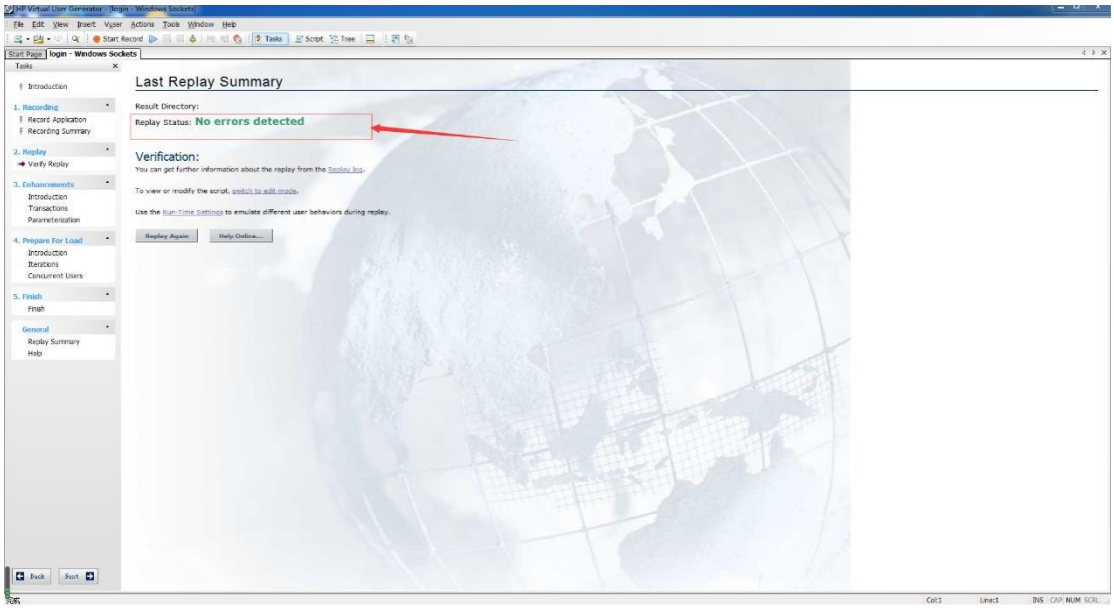


选择协议

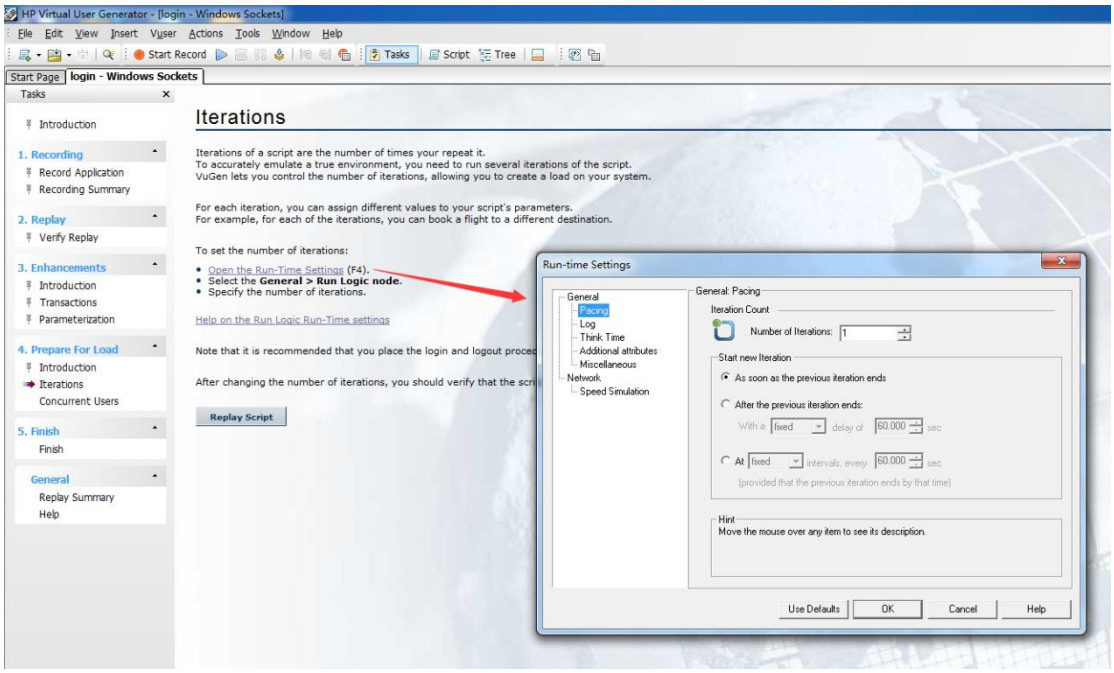


录制设置

录制后，在 action 里创建用户之前加上集合点，进行回放。回放成功，显示 No Errors detected。


Last Replay Summary
Result Directory:
Replay Status: No errors detected
Verification:

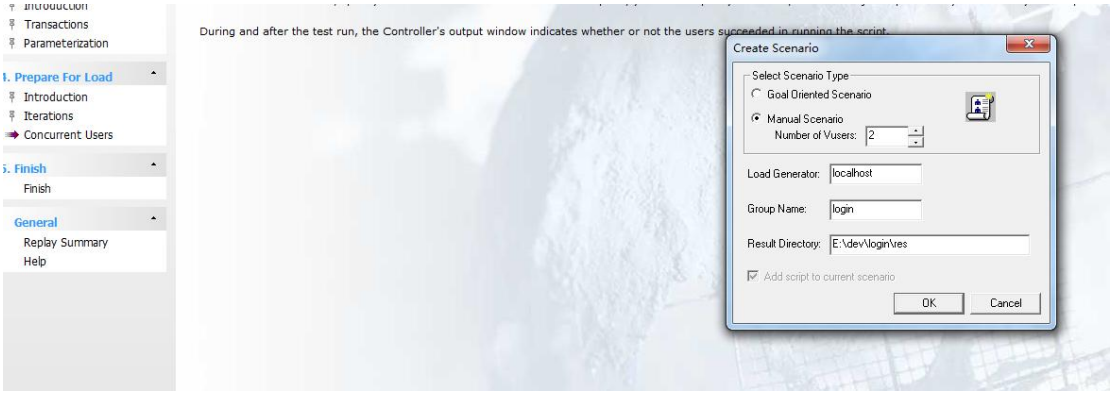
回放


HP Virtual User Generator - [login - Windows Sockets]
Iterations
Iterations of a script are the number of times your repeat it.
To accurately emulate a true environment, you need to run several iterations of the script.
VuGen lets you control the number of iterations, allowing you to create a load on your system.
For each iteration, you can assign different values to your script's parameters.
For example, for each of the iterations, you can book a flight to a different destination.
To set the number of iterations:
Open the Run-Time Settings (F4).
Select the General > Run Logic node.
Specify the number of iterations.
Help on the Run Logic Run-Time settings
Replay Script
Run-time Settings
Number of Iterations:
Use Defaults
OK
Cancel
Help

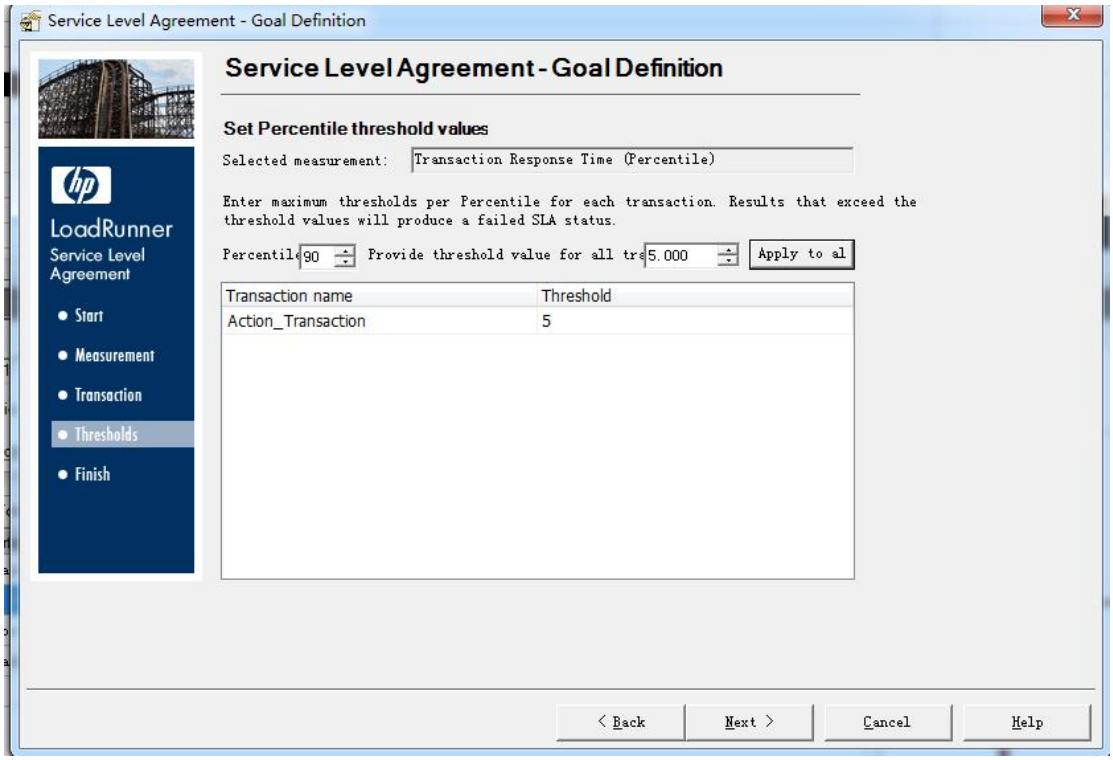
设置运行时

设置迭代 2 次，Replay 看是否不断执行 action。

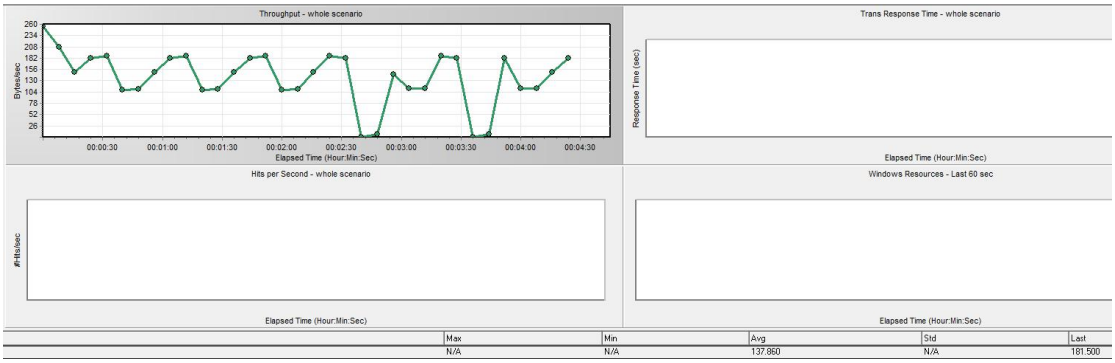
```
Action.c(118): lrs_receive(socket0, buf99)
Action.c(120): lrs_send(socket0, buf100)
Action.c(122): lrs_receive(socket0, buf101)
Action.c(122): Mismatch in buffer's length (expected 638 bytes, 726 bytes actually received, difference in 88 bytes)
Action.c(124): lrs_send(socket0, buf102)
Action.c(126): lrs_receive(socket0, buf103)
Action.c(128): lrs_send(socket0, buf104)
Action.c(130): lrs_receive(socket0, buf105)
Action.c(130): Mismatch in buffer's length (expected 1905 bytes, 726 bytes actually received, difference in 1179 bytes)
Ending action Action.
Ending iteration 1.
Starting iteration 2.
Starting action Action.
Action.c(14): lrs_send(socket0, buf50)
Action.c(16): lrs_receive(socket0, buf51)
Action.c(18): lrs_send(socket0, buf52)
Action.c(20): lrs_receive(socket0, buf53)
```



设置并发用户数



定义性能测试目标 SLA



性能测试运行中吞吐量 Throughput

性能测试结果

扩展知识：

• 吞吐率指单位时间内处理的客户请求数量。通常情况下，用“字节数/秒”来衡量；也可以“请求数/秒”“页面数/秒”来衡量，其本质都是在网络上传输的数据，而数据的单位就是字节数；从业务角度来讲，吞吐率还可以用“业务数/小时或天”，“访问人数/小时或天”，“页面访问量/小时或天”来衡量。

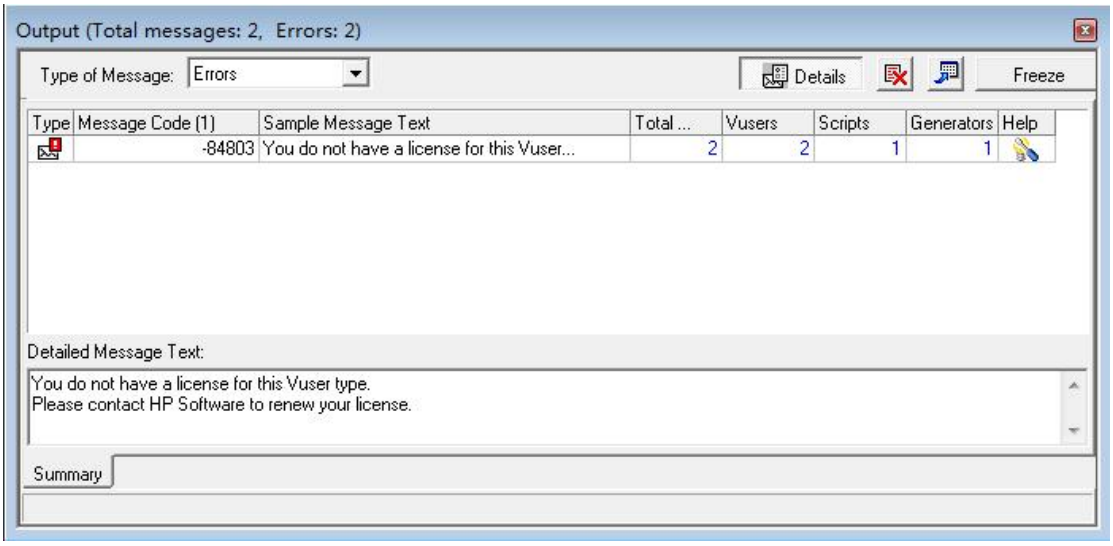
• 设置 SLA。如果性能需求比较具体，针对详细的需求可以设置尽可能适宜的 SLA。像网上提到的“测试的要求是验证在 30 分钟内完成 2000 次用户登录系统，然后进行业务，最后退出，在业务操作过程中页面的响应时间不超过 3 秒，并且服务器的 CPU 使用率、内存使用率分别不超过 75%、70%”，都是好例子。

常见问题解决

运行场景时，LoadRunner 报错：

You do not have a license for this Vuser type.

Please contact HP Software to renew your license.



解决办法：

LoadRunner 不同协议需要不同的 License。

退出 LoadRunner--->以管理员身份启动 LoadRunner--->Configuration--->LoadRunner License--->New License

输入 Global-100 注册码：

AEAMAUIK-YAFEKEKJJKEEA-BCJGI。

注册之后，我们能跑 100 个 vuser。

总结

LoadRunner 性能测试我们从设计性能测试方案开始，确定性能测试需求和目标，过程中经历选择协议、创建脚本、设置检查点、参数化，我们通过了解一次并发测试的实践经验，认识到性能测试全过程，同时我们也了解到在实践中遇到的常见问题，那么还犹豫什么，动手实践吧！

测试人必会--Python 带你上手 WebSocket

◆作者：罗狮小钉

1.闲聊：关于 Socket 那些事

Socket，即网络套接字，是双向通信通道的端点（是抽象的）。套接字可以在一个进程内、同一台机器上的进程之间，或者在不同机器上的进程之间进行通信。网络套接字可以通过多种不同的通道类型得以实现，例如 TCP、UDP 等。

换句话说，网络套接字，是在计算机网络中，两个运行程序之间通信流中的一个端点。这个端点是一个虚拟抽象的概念，并不存在任何硬件。

网络套接字可以通过 IP 地址和端口号的组合来唯一标识。下面我们对于 Socket 套接字相关术语做简单介绍，以便于后续内容的理解。

（1）Domain（域）

作为网络中的传输机制，域是协议中的一员。这些域的值是常量，例如 AF_INET、PF_INET、PF_UNIX、PF_X25 等，这些常量都属于地址系列，用于指定套接字可以与之通信的地址类型。

（2）Type（类型）

表示两个端点之间的通信类型，通常有 SOCK_STREAM 和 SOCK_DGRAM。

• SOCK_STREAM

基于 TCP，数据传输比较有保障，是有保障的面向连接的 Socket，即能保证数据正

确传送到对方；基于数据流的传输；使用 TCP/IP 协议的网络编程。

• SOCK_DGRAM 基于 UDP，用于无向连接协议，是无保障的面向消息的 socket，主要用于在网络上发放广播信息；基于数据包的传输；使用 UDP 协议的网络编程。

（3）Protocol（协议）

用于标识域和类型内的协议变形。它的默认值是 0，通常被省略。

（4）Hostname（主机名）

网络接口的标识符。主机名可以是字符串、点分四组地址或冒号（也可能是点）表示法的 IPV6 地址。

（5）Port（端口）

每个服务器监听调用一个或多个客户端的端口。

2.引入：PythonSocket 模块

在 Python 中实现 Socket 编程，就需要用到 Socket 模块。我们可以通过以下语法创建 Socket：

```
# 导入socket库，创建一个简单的socket
import socket
s = socket.socket (socket family, socket type, protocol = 0)
```

创建 Socket 时，可以使用不同参数（相关概念可参见上文）：

• socket_family：AF_UNIX 或 AF_INET

• socket_type：SOCK_STREAM 或 SOCK_DGRAMprotocol：这通常被忽略，默认为 0

3.普及：SocketMethods--套接字方法

Python 中为套接字提供了三种不同的方法，分别是"服务端套接字方法"，"客户端套接字方法"，"通用套接字方法"。

（1）服务端套接字方法

在客户端-服务器架构中（C/S 架构），有一个中央服务器，来提供服务，众多不同的客户端则从该中央服务器接收服务。众多不同客户端也会向该中央服务器发出请求。

C/S 架构中的一些重要服务器套接字方法有：

• socket.bind()

将服务器地址（主机名，端口号）绑定到套接字。

• socket.listen()

监听与套接字的连接，该方法启动了 TCP 监听器；此方法中的参数用于指定排队链接的最大数量（最小值为 0，最大值为 5）。

• socket.accept()

接受 TCP 客户端连接，该方法将返回一对值（conn,address），其中 conn 是一个新的套接字对象，用于发送和接受数据；address 是绑定到套接字的地址。在使用该方法之前，必须先使用 socket.bind()和 socket.listen()这两个方法。

（2）客户端套接字方法

在客户端-服务器架构中（C/S 架构），客户端会请求服务器，并且也会从服务器接收服务。

socket.connect(address)该方法主动建立服务器连接，简而言之，该方法的作用就是将客户端连接到服务器。

（3）通用套接字方法

除了客户端和服务端套接字方法，还有一些通用的套接字方法，这些方法在 Socket 套接字编程中也非常有用。

常用的通用套接字方法有：

• .recv(bufsize)

用于从套接字接收 TCP 消息；

参数 bufsize 代表缓冲区大小，定义了该方法在任何时候可以接收的最大数据量。

• socket.send(bytes)

用于将数据发送到连接着远程机器的套接字上；参数 bytes 表示发送到套接字的字节数。

• socket.recvfrom(data,address)

用于从套接字接收数据；

该方法返回两对值（data，address）；其中，data 表示接收到的数据，address 表示发送数据的套接字地址。

• socket.sendto(data,address)用于向套接字发送数据；

该方法返回两对值（data，address）；其中，data 表示发送的字节数，address 表示远程机器的地址。

• socket.close()

用于关闭套接字。

• socket.gethostname()返回主机的名称。

• socket.sendall(data)

将所有数据发送到连接着远程机器的套接字，直到发生错误；

期间，如果发生错误，则使用 socket.close()方法关闭套接字。

4.应用：通过 Python 在服务器和客户端之间建立连接

为了能够在服务器和客户端之间建立连接，我们需要通过 Python 编写两个程序，分别用于服务器，和客户端。

【服务端程序】

（1）socket.bind()方法

在服务端 Socket 套接字程序中，通过使用 socket.bind()方法，将其绑定到指定的 IP 地址及端口号上，这样一来，就可以监听到传入该 IP 和端口上的请求。

（2）socket.listen()方法

然后，使用 socket.listen()方法将服务器设置为监听模式。该方法有一个参数，表示当服务器正忙时，当前允许几个连接保持等待；例如 socket.listen(2)，则表示当前允许 2 个连接保持等待，如果第 3 个套接字尝试连接，则该连接请求会被拒绝。

（3）socket.send()方法

使用 socket.send()方法向客户端发送消息。

（4）socket.accept 方法

使用 socket.accept()方法启动连接。

（5）socket.close()方法

使用 socket.close()方法关闭连接。

【服务端程序_Demo】

```
import socket

def Server side ():
    host = socket.gethostname ()
    port = 23333
    serversocket = socket.socket ()
    serversocket .bind((host, port))
    serversocket .listen (1)
    print('socket 监听中')

    while True:
        conn, addr = serversocket .accept ()
        print("收到连接请求 %s" % str(addr))
        msg = '连接已创建' + "\r\n"
        conn.send(msg.encode('ascii'))
        conn.close ()

if __name__ == '__main__':
    Server side ()
```

【客户端程序】

（1）创建 socket 对象

在客户端 socket 程序中，需要先创建一个 socket 对象。然后将该 socket 对象连接到正在运行着的服务器端口（即上面的 23333 端口）。

（2）socket.connect()方法

使用 socket.connect()方法建立连接。

（3）socket.recv()方法

socket.recv()方法用于客户端接收来自服务器的消息。

（4）socket.close()方法

socket.close()方法关闭客户端。

【客户端程序_Demo】

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

host = socket.gethostname()
port = 23333

s.connect((host, port))
msg = s.recv(1024)

s.close()
print (msg.decode('ascii'))
```

【分别启动服务端和客户端程序】在运行服务端程序后，在终端上获得以下输出：

```
socket is listening
Got connection from ('XXX.XX.XXX.XXX', 65426)  # 此处IP为客户端IP
```

在运行客户端程序后，在终端上获得以下输出：

```
Connecting Established
```

【网络套接字异常处理】

try 和 except，可用于处理网络套接字异常，示例脚本如下：

```
import socket

host = socket.gethostname()
port = 23333

# 创建套接字对象
s = socket.socket(socket.AF INET, socket.SOCK DGRAM)

try:
   # 绑定服务端IP及端口
   s.bind((host,port))

   # 设置客户端的等待时间
   s.settimeout(3)
   data, addr = s.recvfrom(1024)
   print ("recevied from ",addr)
   print ("obtained ", data)
   s.close()
except socket.timeout :
   print ("No connection between client and server")
   s.close()
```

脚本解析：

（1）通过当前服务端IP及指定端口号（23333），创建了一个套接字对象。
（2）在 try - except 异常处理块中，使用 socket.bind() 方法，尝试绑定服务端IP及端口；
（3）使用 socket.settimeout() 方法设置客户端的等待时间，示例中设置为 3 秒；
（4）在except块中对异常进行处理，如果服务器和客户端之间没有建立连接，将打印一条消息。

以上程序运行后，在终端上获得以下输出：

```
No connection between client and server
```

5.总结

WebSocket 是测试人员知识储备库中不可或缺的一项，本次分享对 WebSocket 来龙去脉做了梳理，结合 Python 进行了实战演练。希望通过本文的学习，能够加深你对 WebSocket 认知，让它不再成为你的知识盲区。

《51 测试天地》(六十三) 下篇 精彩预览

- Python 自动化测试框架，谁才是你的唯一
- 测试体系搭建连载之 MQ 消息测试
- 测试体系搭建连载之安全测试基础
- 测试体系搭建连载之正交用例设计
- 黑客易筋经之修炼逆向功底
- 妙用 Mitmproxy 实现加密加签环境下渗透测试
- 谈谈软件项目中的评审
- 我靠 Jmeter 的录制功能摆脱了搬运工的角色

马上阅读