# 为什么要做接口测试

更容易实现持续集成

自动化测试落地性价比高，比ui更稳定 维护速度更快

大型系统更多更复杂，系统间模块越来愈多

Bug更容易定位，因为分模块测试，模块间调用

降低研发成本、提高效率

接口测试的重点就是检查数据的交换，传递和控制管理过程，以及系统间的相互逻辑依赖关系，比如一个if条件要返回ture or flase,接口测试就要测试他们下一个逻辑业务分别是什么。尤其是现在我们的大部分系统都是前后端架构分离的，从安全的层面讲，只依赖前端就不能满足系统的安全性，需要从后端进行控制，在这种情况下就要进行接口测试了。就敏捷测试讲，只要前后端定义好接口，测试就可以编写自动化测试代码

# 自动化测试落地过程

需求阶段--研发阶段--测试阶段（与研发并行）--项目上线



# 接口测试的范围

**功能测试：**等价类划分法、边界值分析法、错误推断法、因果图法、判定表驱动法、正交试验法、功能图法、场景法

**异常测试：**数据异常（null、””、数据类型）、环境异常（负责均衡、冷热备份）

#热备份就是保证服务正常不间断运行，通过一台服务器对了一台服务器进行实时数据复制，且保障两边数据的一致性。我们将在线的备份称为热备份，而相对的，将脱机数据备份称为冷备份。冷备份是在系统不运作时，定时的将数据备份至备份服务器或存储

**性能测试：**负责测试、压力测试、并发测试、稳定性测试

# 测试用例设计

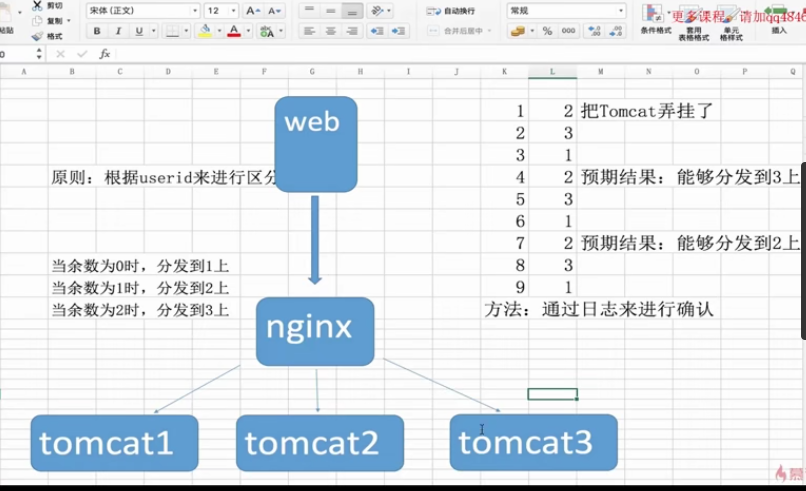
要求：

输入：用户名（邮箱或者是手机号）

输入：密码（6-16位 区分大小写 不能用空格）



# 环境异常的测试



# 自动化框架设计

**显示层** 测试报告

**控制层** 逻辑验证

**持久层** 测试用例存储

# 为什么要使用github

**github的特点：**

开源的分布式版本管理系统

开源项目集中的代码库

所有略有规模的公司都在使用

**github的功能介绍：**

记录多个版本

查看历史操作，可以进行版本回退和前进的控制

多端共享代码，自动合并

**Github与svn 的区别**

Svn版本集中管理，所有的代码都在中央服务器上

Github去中心化，每个服务器都有一个完整的代码库

# github的使用

/c/Users/89224/.ssh

[892243561@qq.com](mailto:892243561@qq.com) 892243561.lxW

**配置sshkey**

生成key : ssh-keygen -t rsa -C [“liuxu8”](mailto:\“89@qq.com\”)

Cd ~/.ssh

复制id\_rsa.pub的公钥内容到github网站中

**配置多个sshkey**

cd ~/.ssh

Vim config

添加Host/HostName/User/IdentityFile四项

# github的使用

**项目的克隆、推送与拉取**

克隆：

.ssh cd/workspace

Mkdir dir

Cd dir

Git clone [git@github.com: liuxu8/AutoTest.git](mailto:git@github.com:liuxu8/AutoTest.git)

Git status 查看仓库内容

Git add test.txt 添加文件至仓库

Git commit -m “添加测试文件” --为本次提交添加注解

Git push 将文件推送到网站（github.com）

Git pull 拉取文件或项目至本地（项目同步）

1. git status 查看工作区代码相对于暂存区的差别  
   2、git add . 将当前目录下修改的所有代码从工作区添加到暂存区 . 代表当前目录  
   3、git commit -m ‘注释’ 将缓存区内容添加到本地仓库  
   4、git push origin master 将本地版本库推送到远程服务器，  
   5、origin是远程主机，master表示是远程服务器上的master分支，分支名是可以修改的

**#将本地项目传到github上**

1. **在文件夹下点击git bash here 进入命令行 输入git init**
2. **Git add .**
3. **Git commit -m “提交项目”**
4. “git remote add origin [https://github.com/shench5612390/Test.git”](https://github.com/shench5612390/Test.git%E2%80%9D%EF%BC%88git" \t "https://www.cnblogs.com/shenchanghui/p/_blank)
5. git push -u origin master

# 项目分支

Git branch 查看当前分支

Git branch -a 查看远程分支

Git checkout -b branch\_One 创建分支（路径会自动进入到新的分支去，但是文件和主分支相同）

Vim进行编辑后

Git add xxx.xx

Git commit -m “做了什么改动”

git push --set-upstream origin branch\_One更新远网站库的分支

**删除分支**

Git checkout master 首先切换分支到master下

Git branch -d branch\_One 删除分支 （删除本地仓库中的分支，远程仍存在 -D强制删除）

Git branch -r -d origin/branch\_One 远程删除

Git push origin :branch\_One 刷新（有空格）

**分支合并**

Git merge branch\_One (前提是切换到其他分支下)

git是比较时间戳的。如果分支的代码比主干的代码新，那么无论代码多少都会应用分支的代码。

合并分支时要解决冲突

miss is moon.

<<<<<<< HEAD

i miss you in day and night,all the time .

master!

=======

i miss you in day and night,all the time~

branch!

>>>>>>> branch\_One

左尖括号下面的是master分支下的内容

等于号下边是branch\_One下的内容

之后可以将master下的内容 手动修改，然后再次进行提交

Git add xx.txt

Git commit -m “合并ing”

Git push

# 版本回退

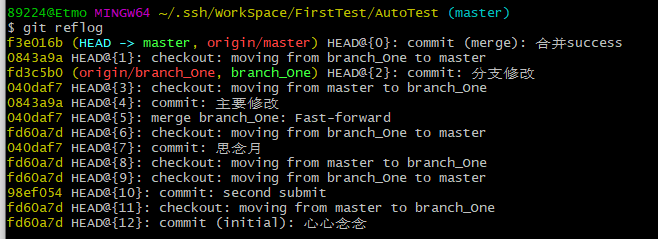
Git add . 上传本路径下的所有文件

Git commit -m “v1”

Git reset --hard HEAD^ (尖代表的是回退的次数，两个尖代表回退两个版本)

#Git reset --hard HEAD~10 （回退10个版本）

Git reflog 查看版本日志



Git reset --hard 040daf7 回退到指定版本

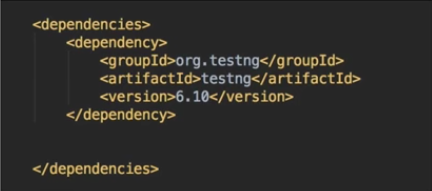
# testNG的介绍

比Junit 涵盖功能更全面的测试框架

Junit更适合隔离性比较强的单元测试

TestNGg更适合复杂的集成测试

在maven中使用



**在idea 中新增module 配置pom.xml**

# 注解实战@test标签

在java目录下新增包com.miss.testng

新增类，新增方法

**public class** BasicAuto {  
 *//最基本的注解，用来标记方法为测试的一部分* @Test  
 **public void** TestCase1(){  
 System.***out***.println(**"醉清风1！"**);  
 }  
 @Test  
 **public void** TestCase2(){  
 System.***out***.println(**"醉清风2！"**);  
 }  
 @BeforeMethod  
 **public void** BeforeMethod(){  
 System.***out***.println(**"方法前执行：乱了分寸的心动"**);  
 }  
 @AfterMethod  
 **public void** AfterMethod(){  
 System.***out***.println(**"方法后执行：和你醉后缠绵你可记得"**);  
 }  
}

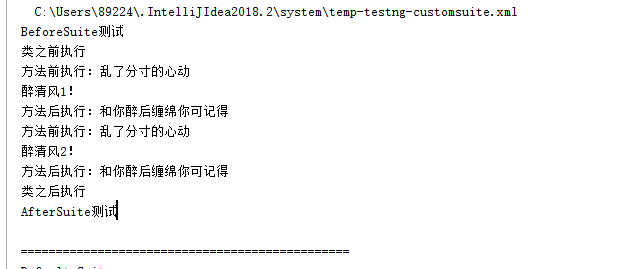
#只需要输入sout 即可;

BeforeMethod 与 AfterMethod 总会在所有方法的执行前（后）执行

@BeforeClass  
**public void** BeforeClass(){  
 System.***out***.println(**"类之前执行"**);  
}  
  
@AfterClass  
**public void** AfterClass(){  
 System.***out***.println(**"类之后执行"**);  
}

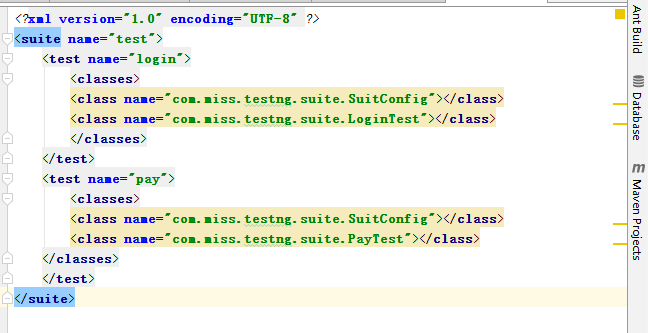
@BeforeSuite  
**public void** BeforeSuite(){  
 System.***out***.println(**"BeforeSuite测试"**);  
  
}  
@AfterSuite  
**public void** AfterSuite(){  
 System.***out***.println(**"AfterSuite测试"**);  
  
}

**BeforeSuite执行的顺序在类执行之前，然后执行BeforeClass 然后执行BeforeMethod方法，最后执行测试方法**



**测试套件**

在resources中新增suite.xml



# #Git提示fatal: remote origin already exists解决办法:之前添加过远程库

1、先删除远程 git 仓库

git remote rm origin

git 远程 移除 起源

$ git remote rm origin

2、再添加远程 Git 仓库

$ git remote add origin git@github.com:FBing/Java-code-generator

如果执行 git remote rm origin 报错的话，我们可以手动修改gitconfig文件的内容

$ vi .git/config

# 忽略测试

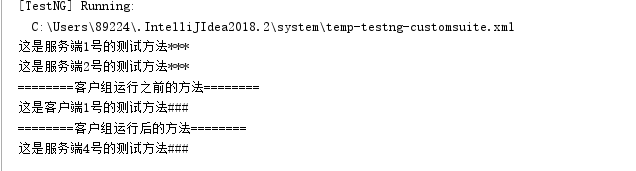
**package** com.miss.testng;  
  
**import** org.testng.annotations.Test;  
  
**public class** IgnoreTest {  
 @Test  
 **public void** IgnoreOne(){  
 System.***out***.println(**"执行测试"**);  
 }  
 @Test(enabled = **false**)  
 **public void** IgnoreTwo(){  
 System.***out***.println(**"忽略测试"**);  
 }  
}

当test的属性enabled没有标注时 默认是true

# TestNG组测试

**package** com.miss.testng.groups;  
  
**import** org.testng.annotations.AfterGroups;  
**import** org.testng.annotations.BeforeGroups;  
**import** org.testng.annotations.Test;  
  
**public class** GroupOnMethod {  
 @Test(groups = **"sever"**)  
 **public void** test1(){  
 System.***out***.println(**"这是服务端1号的测试方法\*\*\*"**);  
 }  
 @Test(groups = **"sever"**)  
 **public void** test2(){  
 System.***out***.println(**"这是服务端2号的测试方法\*\*\*"**);  
 }  
 @Test(groups = **"client"**)  
 **public void** test3(){  
 System.***out***.println(**"这是客户端1号的测试方法###"**);  
 }  
 @Test(groups = **"sever"**)  
 **public void** test4(){  
 System.***out***.println(**"这是服务端4号的测试方法###"**);  
 }  
 @BeforeGroups(**"client"**)  
 **public void** BeforeMethod(){  
 System.***out***.println(**"========客户组运行之前的方法========"**);  
 }  
 @AfterGroups(**"client"**)  
 **public void** AfterMethod(){  
 System.***out***.println(**"========客户组运行后的方法========"**);  
 }  
}

Console:

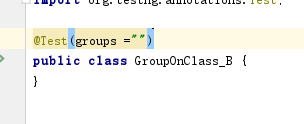


#1、github：是一个面向开源及私有软件项目的托管平台，因为只支持git 作为唯一的版本库格式进行托管。

2、git：是一个开源的分布式版本控制系统，可以有效、高速地处理从很小到非常大的项目版本管理。

**类分组测试：**

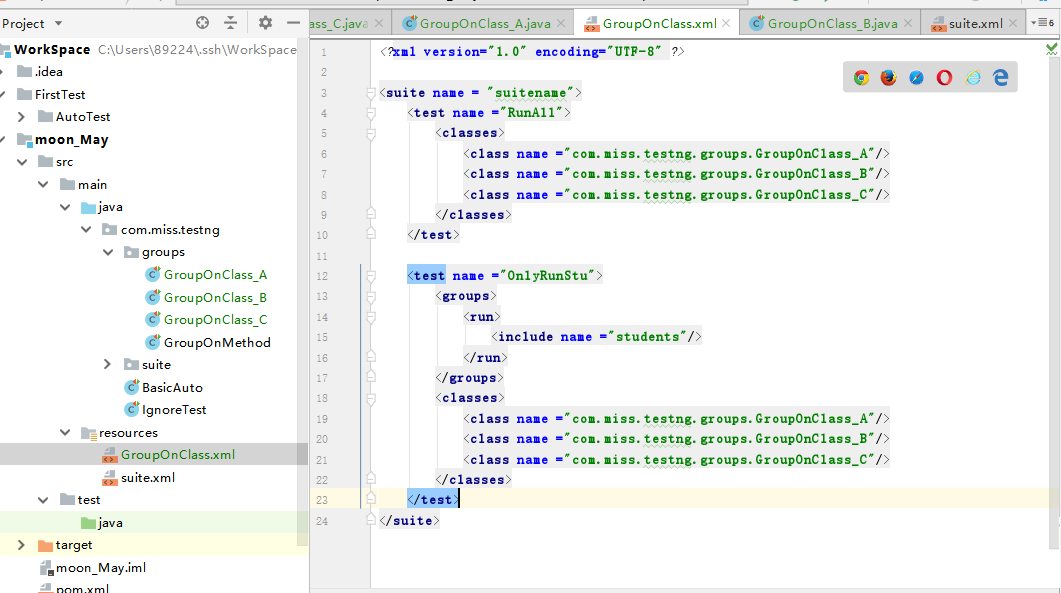
**！@Test 后面的 groups G是小写**



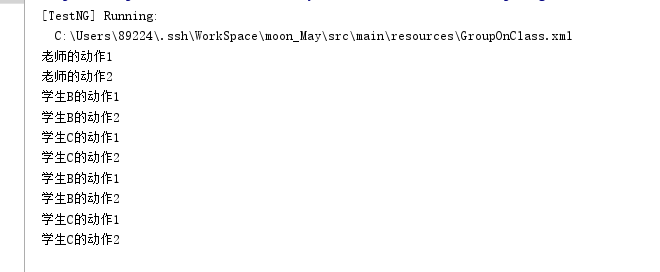
**package** com.miss.testng.groups;  
  
**import** org.testng.annotations.Test;  
  
@Test(groups = **"teacher"**)  
**public class** GroupOnClass\_A {  
 **public void** ter1(){  
 System.***out***.println(**"老师的动作1"**);  
 }  
 **public void** ter2(){  
 System.***out***.println(**"老师的动作2"**);  
 }  
}

**package** com.miss.testng.groups;  
  
**import** org.testng.annotations.Test;  
  
@Test(groups =**"students"**)  
**public class** GroupOnClass\_B {  
 **public void** stu1(){  
 System.***out***.println(**"学生B的动作1"**);  
 }  
 **public void** stu2(){  
 System.***out***.println(**"学生B的动作2"**);  
 }  
}

**在resources下新增GroupOnClass Xml文件 运行得到分组测试结果**



**得到测试结果：**



# TestNG异常测试

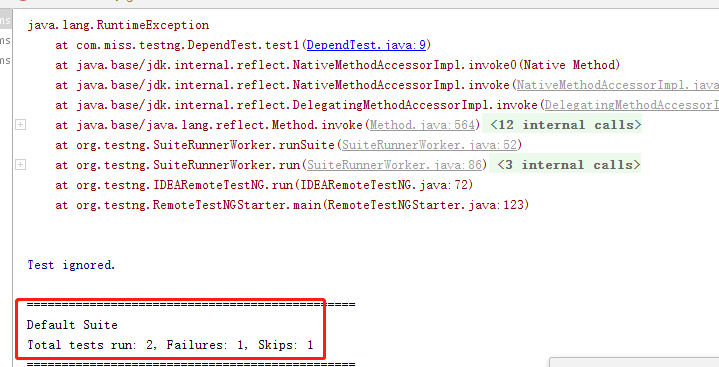
**package** com.miss.testng;  
  
**import** org.testng.annotations.Test;  
  
**public class** Exception {  
 */\*  
 \*什么时候会用到异常测试  
 \* 在我们的期望结果为一个异常的时候  
 \* 比如传入了一个不合法的参数，程序抛出了异常  
 \*/* @Test(expectedExceptions = RuntimeException.**class**)  
 **public void** RunTimeExceptionFail(){  
 System.***out***.println(**"异常测试"**);  
 }  
 @Test(expectedExceptions = RuntimeException.**class**)  
 **public void** RunTimeExceptionSuccess(){  
 System.***out***.println(**"异常测试Success!"**);  
 **throw new** RuntimeException();  
 }  
}



# 依赖测试

**package** com.miss.testng;  
  
**import** org.testng.annotations.Test;  
  
**public class** DependTest {  
 @Test  
 **public void** test1(){  
 System.***out***.println(**"测试一执行\*\*\*"**);  
 **throw new** RuntimeException();  
 }  
  
 @Test(dependsOnMethods = {**"test1"**})  
 **public void** test2(){  
 System.***out***.println(**"测试二执行\*\*\*"**);  
 }  
}

**当被依赖的方法1执行失败时，测试方法二自动被忽略**



# TestNG参数化测试

**创建传参的方法**

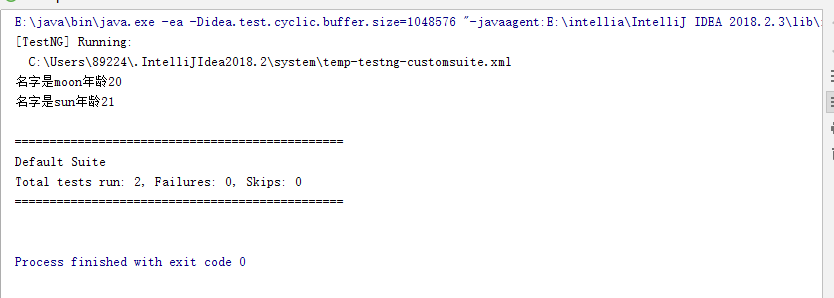
**package** com.miss.testng.params;  
  
  
**import** org.testng.annotations.Parameters;  
**import** org.testng.annotations.Test;  
  
**public class** ParamsTest {  
 @Test  
 @Parameters({**"name"**,**"age"**})  
 **public void** Test(String name,**int** age){  
 System.***out***.println(**"名字是"**+name + **"年龄是"** + age);  
 }  
}

**创建xml文件进行测试（suite 必须有名字属性 下面是test 下面是classes 下面是class）**

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**suite name="parames"**>  
 <**test name ="lvlvlv1"**>  
 <**classes**>  
 <**parameter name = "name" value ="moon"**/>  
 <**parameter name = "age" value ="25"**/>  
 <**class name="com.miss.testng.params.ParamsTest"**/>  
 </**classes**>  
 </**test**>  
</**suite**>

**Dateprovider参数化**

**package** com.miss.testng.params;  
  
**import** org.testng.annotations.DataProvider;  
**import** org.testng.annotations.Test;  
  
**public class** DataProviderTest {  
 @Test(dataProvider = **"data"**)  
 **public void** DataProviderTest(String name,**int** age){  
 System.***out***.println(**"名字是"**+ name + **"年龄"** + age);  
 }  
 @DataProvider(name =**"data"**)  
 **public** Object[][] ProData() {  
  
 Object[][] oj = **new** Object[][]{  
 {**"moon"**,20},  
 {**"sun"**,21}  
 };  
 **return** oj;  
 }  
}



**#Object在生命变量时有分号**

**Object[][] result = new Object[][]{**

**{“moon”,40},**

**{“sun”,41}**

**};**

**#@dataProvide在同一个类中不能重复**

@Test(dataProvider = **"moon"**)  
 **public void** Test1(String name,**int** age){  
 System.***out***.println(**"名字moon是"**+ name + **"年龄"** + age);  
 }  
 @Test(dataProvider = **"moon"**)  
 **public void** Test2(String name,**int** age){  
 System.***out***.println(**"名字是sun"**+ name + **"年龄"** + age);  
 }  
@DataProvider(name=**"moon"**)  
 **public** Object[][] Privider(Method method){  
 Object[][] result = **null**;  
 **if**(method.getName().equals(**"Test1"**)) {  
 result = **new** Object[][]{  
  
 {**"moon"**,17},  
 {**"sun"**,18}  
 };  
 }  
  
 **else if** (method.getName().equals(**"Test2"**)){  
 result = **new** Object[][]{  
  
 {**"moon"**,20},  
 {**"sun"**,21}  
 };  
 }  
  
  
 **return** result;  
  
 }

**在执行Test1是 方法名传递到了data provider方法中**

# TestNG多线程测试-xml文件实现

@1、print：将信息显示在命令窗口中，输出光标定位在最后一个字符之后。

2、printf：将信息进行格式化显示在命令窗口中，输出光标定位在最后一个字符之后。

3、println：将信息显示在命令窗口中，输出光标换行定位在下一行开头。

@%d,%c,%s,%x是程序汇编语言中的格式符，它们的含义：

1、%d表示按整型数据的实际长度输出数据。

2、%c用来输出一个字符。

3、%s用来输出一个字符串。

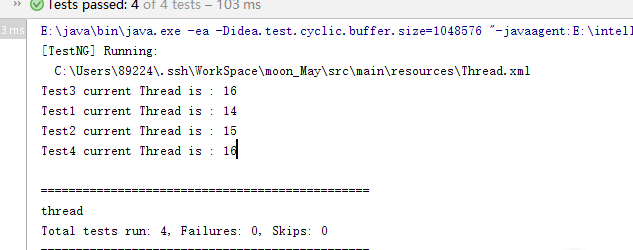
4、%x表示以十六进制数形式输出整数。

**package** com.miss.testng.thread;  
  
**import** org.testng.annotations.Test;  
  
**public class** MultThreadOnXml {  
@Test  
 **public void** Test1(){  
 System.***out***.printf(**"Test1 current Thread is : %s%n"**,Thread.*currentThread*().getId());  
 }  
 @Test  
 **public void** Test2(){  
 System.***out***.printf(**"Test2 current Thread is : %s%n"**,Thread.*currentThread*().getId());  
 }  
 @Test  
 **public void** Test3(){  
 System.***out***.printf(**"Test3 current Thread is : %s%n"**,Thread.*currentThread*().getId());  
 }  
 @Test  
 **public void** Test4(){  
 System.***out***.printf(**"Test4 current Thread is : %s%n"**,Thread.*currentThread*().getId());  
 }  
}

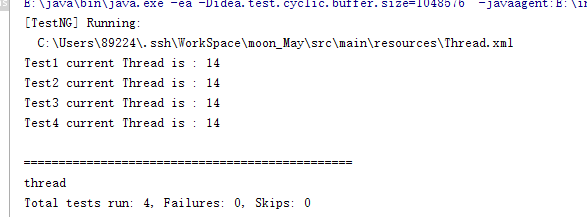
XML

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**suite name ="thread" parallel="tests" thread-count="3"**>  
 <**test name = "threadTest"**>  
 <**classes**>  
 <**class name="com.miss.testng.thread.MultThreadOnXml"**></**class**>  
</**classes**>  
  
  
  
</**test**>  
</**suite**>

#suite中的属性 parallel的值是有复数的（tests、methods但是class没有）



**如果改变parallel的属性为tests 输出结果如下**



这是因为

Tests级别：不同的test tag下的用例可以在不同的线程下执行

相同的test tag下的用例只能在相同的线程中使用

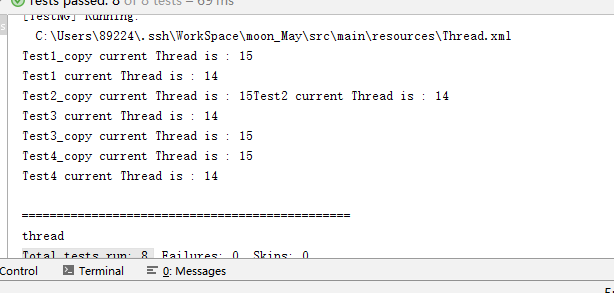
Methods级别：所有的用例都可以在不同的线程下执行

Class级别：相同的class tag下的用例在一个线程中执行

Thread-count 代表了最大并发线程数

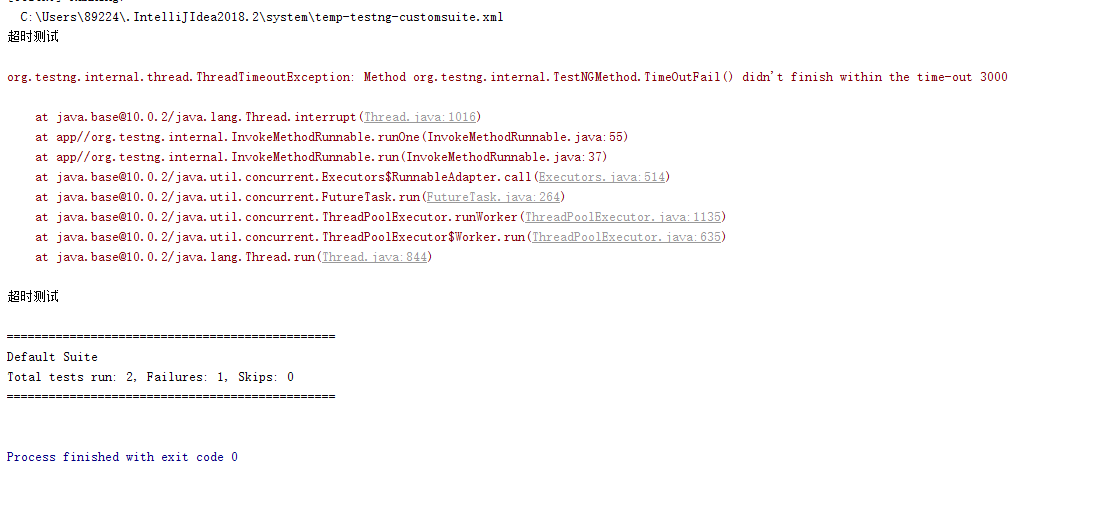
Xml文件这种配置方式不能指定线程池，只有方法上才可以指定线程池

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**suite name ="thread" parallel="tests" thread-count="3"**>  
 <**test name = "threadTest"**>  
 <**classes**>  
 <**class name="com.miss.testng.thread.MultThreadOnXml"**></**class**>  
</**classes**>  
</**test**>  
 <**test name = "threadTest1"**>  
 <**classes**>  
 <**class name="com.miss.testng.thread.MultThreadOnXml\_copy"**></**class**>  
 </**classes**>  
 </**test**>  
</**suite**>



# TestNG超时测试

**package** com.miss.testng;  
  
**import** org.testng.annotations.Test;  
  
**public class** TimeOutTest {  
 @Test(timeOut = 3000)  
 **public void** TimeOutSuccess() **throws** InterruptedException {  
 System.***out***.println(**"超时测试"**);  
 Thread.*sleep*(2000);  
 }  
 @Test(timeOut = 3000)  
 **public void** TimeOutFail() **throws** InterruptedException {  
 System.***out***.println(**"超时测试"**);  
 Thread.*sleep*(3001);  
 }  
}



# 测试报告

先在pom.xml中添加modules 标签

*<?***xml version="1.0" encoding="UTF-8"***?>*<**project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"**>  
 <**modelVersion**>4.0.0</**modelVersion**>  
  
 <**groupId**>com.miss.moon</**groupId**>  
 <**artifactId**>moon\_May</**artifactId**>  
 <**version**>1.0-SNAPSHOT</**version**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.testng</**groupId**>  
 <**artifactId**>testng</**artifactId**>  
 <**version**>RELEASE</**version**>  
 <**scope**>compile</**scope**>  
 </**dependency**>  
 </**dependencies**>  
  
<**modules**>  
 <**module**>May</**module**>  
</**modules**>  
</**project**>

**在双medule 下pom.xml 文件会自动加载属性值**

**1.在pom中配置依赖**

<**dependency**>  
 <**groupId**>com.relevantcodes</**groupId**>  
 <**artifactId**>extentreports</**artifactId**>  
 <**version**>2.41.1</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.vimalselvam</**groupId**>  
 <**artifactId**>testng-extentsreport</**artifactId**>  
 <**version**>1.3.1</**version**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.aventstack</**groupId**>  
 <**artifactId**>extentreports</**artifactId**>  
 <**version**>3.0.6</**version**>  
</**dependency**>

@添加 jar包 文件file ->project structure

在Project Structure界面中，

第一步，选择左侧的 **Modules**

第二步，点击  图示 **Dependencies**

第三步，点击 右上角的**+** 号

**2.添加测试方法**

**package** com.miss.testng.report;  
  
**import** org.testng.Assert;  
**import** org.testng.Reporter;  
**import** org.testng.annotations.Test;  
  
**public class** TestMethodDemo {  
@Test  
 **public void** test1(){  
 Assert.*assertEquals*(1,2);  
 }  
 @Test  
 **public void** test2(){  
 Assert.*assertEquals*(1,1);  
 }  
 @Test  
 **public void** test3(){  
 Assert.*assertEquals*(**"aaa"**,**"sss"**);  
 }  
 **public void** logDemo(){  
 Reporter.*log*(**"这是自己的日志！"**);  
 **throw new** RuntimeException(**"这是自己写的异常"**);  
 }  
}

**3.在resources中新增xml文件**

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**suite name ="自己的测试套件"**>  
 <**test name ="测试模块"**>  
<**classes**>  
 <**class name ="com.miss.testng.report.TestMethodDemo"**></**class**>  
 <**merhods**>  
 <**include name ="logDemo"**></**include**>  
 <**include name ="test1"**></**include**>  
 <**include name ="test2"**></**include**>  
 <**include name ="test3"**></**include**>  
  
 </**merhods**>  
</**classes**>  
 </**test**>  
  
 <**listeners**>  
 <**listener class-name= "com.vimalselvam.testng.listener.ExtentTestNgFormatter"**/>  
 </**listeners**>  
</**suite**>

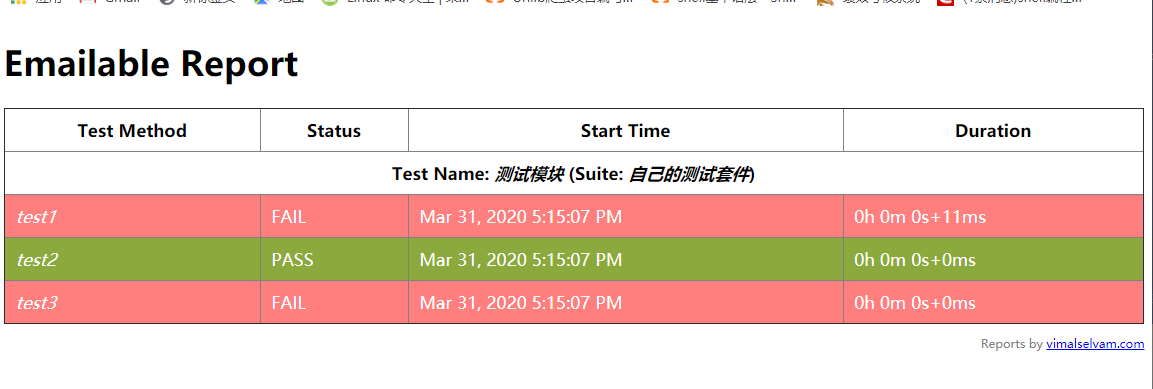
**4.新增类**

**package** com.miss.testng.report;  
  
**import** com.aventstack.extentreports.ExtentReports;  
**import** com.aventstack.extentreports.ExtentTest;  
**import** com.aventstack.extentreports.ResourceCDN;  
**import** com.aventstack.extentreports.Status;  
**import** com.aventstack.extentreports.model.TestAttribute;  
**import** com.aventstack.extentreports.reporter.ExtentHtmlReporter;  
**import** com.aventstack.extentreports.reporter.configuration.ChartLocation;  
**import** com.aventstack.extentreports.reporter.configuration.Theme;  
**import** org.testng.\*;  
**import** org.testng.xml.XmlSuite;  
  
**import** java.io.File;  
**import** java.util.\*;  
  
**public class** ExtentTestNGIReporterListener **implements** IReporter{  
 *//生成的路径以及文件名* **private static final** String ***OUTPUT\_FOLDER*** = **"test-output/"**;  
 **private static final** String ***FILE\_NAME*** = **"index.html"**;  
  
 **private** ExtentReports **extent**;  
  
 @Override  
 **public void** generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites, String outputDirectory) {  
 init();  
 **boolean** createSuiteNode = **false**;  
 **if**(suites.size()>1){  
 createSuiteNode=**true**;  
 }  
 **for** (ISuite suite : suites) {  
 Map<String, ISuiteResult> result = suite.getResults();  
 *//如果suite里面没有任何用例，直接跳过，不在报告里生成* **if**(result.size()==0){  
 **continue**;  
 }  
 *//统计suite下的成功、失败、跳过的总用例数* **int** suiteFailSize=0;  
 **int** suitePassSize=0;  
 **int** suiteSkipSize=0;  
 ExtentTest suiteTest=**null**;  
 *//存在多个suite的情况下，在报告中将同一个一个suite的测试结果归为一类，创建一级节点。* **if**(createSuiteNode){  
 suiteTest = **extent**.createTest(suite.getName()).assignCategory(suite.getName());  
 }  
 **boolean** createSuiteResultNode = **false**;  
 **if**(result.size()>1){  
 createSuiteResultNode=**true**;  
 }  
 **for** (ISuiteResult r : result.values()) {  
 ExtentTest resultNode;  
 ITestContext context = r.getTestContext();  
 **if**(createSuiteResultNode){  
 *//没有创建suite的情况下，将在SuiteResult的创建为一级节点，否则创建为suite的一个子节点。* **if**( **null** == suiteTest){  
 resultNode = **extent**.createTest(r.getTestContext().getName());  
 }**else**{  
 resultNode = suiteTest.createNode(r.getTestContext().getName());  
 }  
 }**else**{  
 resultNode = suiteTest;  
 }  
 **if**(resultNode != **null**){  
 resultNode.getModel().setName(suite.getName()+**" : "**+r.getTestContext().getName());  
 **if**(resultNode.getModel().hasCategory()){  
 resultNode.assignCategory(r.getTestContext().getName());  
 }**else**{  
 resultNode.assignCategory(suite.getName(),r.getTestContext().getName());  
 }  
 resultNode.getModel().setStartTime(r.getTestContext().getStartDate());  
 resultNode.getModel().setEndTime(r.getTestContext().getEndDate());  
 *//统计SuiteResult下的数据* **int** passSize = r.getTestContext().getPassedTests().size();  
 **int** failSize = r.getTestContext().getFailedTests().size();  
 **int** skipSize = r.getTestContext().getSkippedTests().size();  
 suitePassSize += passSize;  
 suiteFailSize += failSize;  
 suiteSkipSize += skipSize;  
 **if**(failSize>0){  
 resultNode.getModel().setStatus(Status.***FAIL***);  
 }  
 resultNode.getModel().setDescription(String.*format*(**"Pass: %s ; Fail: %s ; Skip: %s ;"**,passSize,failSize,skipSize));  
 }  
 buildTestNodes(resultNode,context.getFailedTests(), Status.***FAIL***);  
 buildTestNodes(resultNode,context.getSkippedTests(), Status.***SKIP***);  
 buildTestNodes(resultNode,context.getPassedTests(), Status.***PASS***);  
 }  
 **if**(suiteTest!= **null**){  
 suiteTest.getModel().setDescription(String.*format*(**"Pass: %s ; Fail: %s ; Skip: %s ;"**,suitePassSize,suiteFailSize,suiteSkipSize));  
 **if**(suiteFailSize>0){  
 suiteTest.getModel().setStatus(Status.***FAIL***);  
 }  
 }  
  
 }  
*// for (String s : Reporter.getOutput()) {  
// extent.setTestRunnerOutput(s);  
// }* **extent**.flush();  
 }  
  
 **private void** init() {  
 *//文件夹不存在的话进行创建* File reportDir= **new** File(***OUTPUT\_FOLDER***);  
 **if**(!reportDir.exists()&& !reportDir .isDirectory()){  
 reportDir.mkdir();  
 }  
 ExtentHtmlReporter htmlReporter = **new** ExtentHtmlReporter(***OUTPUT\_FOLDER*** + ***FILE\_NAME***);  
 *// 设置静态文件的DNS  
 //怎么样解决cdn.rawgit.com访问不了的情况* htmlReporter.config().setResourceCDN(ResourceCDN.***EXTENTREPORTS***);  
  
 htmlReporter.config().setDocumentTitle(**"api自动化测试报告"**);  
 htmlReporter.config().setReportName(**"api自动化测试报告"**);  
 htmlReporter.config().setChartVisibilityOnOpen(**true**);  
 htmlReporter.config().setTestViewChartLocation(ChartLocation.***TOP***);  
 htmlReporter.config().setTheme(Theme.***STANDARD***);  
 htmlReporter.config().setCSS(**".node.level-1 ul{ display:none;} .node.level-1.active ul{display:block;}"**);  
 **extent** = **new** ExtentReports();  
 **extent**.attachReporter(htmlReporter);  
 **extent**.setReportUsesManualConfiguration(**true**);  
 }  
  
 **private void** buildTestNodes(ExtentTest extenttest, IResultMap tests, Status status) {  
 *//存在父节点时，获取父节点的标签* String[] categories=**new** String[0];  
 **if**(extenttest != **null** ){  
 List<TestAttribute> categoryList = extenttest.getModel().getCategoryContext().getAll();  
 categories = **new** String[categoryList.size()];  
 **for**(**int** index=0;index<categoryList.size();index++){  
 categories[index] = categoryList.get(index).getName();  
 }  
 }  
  
 ExtentTest test;  
  
 **if** (tests.size() > 0) {  
 *//调整用例排序，按时间排序* Set<ITestResult> treeSet = **new** TreeSet<ITestResult>(**new** Comparator<ITestResult>() {  
 @Override  
 **public int** compare(ITestResult o1, ITestResult o2) {  
 **return** o1.getStartMillis()<o2.getStartMillis()?-1:1;  
 }  
 });  
 treeSet.addAll(tests.getAllResults());  
 **for** (ITestResult result : treeSet) {  
 Object[] parameters = result.getParameters();  
 String name=**""**;  
 *//如果有参数，则使用参数的toString组合代替报告中的name* **for**(Object param:parameters){  
 name+=param.toString();  
 }  
 **if**(name.length()>0){  
 **if**(name.length()>50){  
 name= name.substring(0,49)+**"..."**;  
 }  
 }**else**{  
 name = result.getMethod().getMethodName();  
 }  
 **if**(extenttest==**null**){  
 test = **extent**.createTest(name);  
 }**else**{  
 *//作为子节点进行创建时，设置同父节点的标签一致，便于报告检索。* test = extenttest.createNode(name).assignCategory(categories);  
 }  
 *//test.getModel().setDescription(description.toString());  
 //test = extent.createTest(result.getMethod().getMethodName());* **for** (String group : result.getMethod().getGroups())  
 test.assignCategory(group);  
  
 List<String> outputList = Reporter.*getOutput*(result);  
 **for**(String output:outputList){  
 *//将用例的log输出报告中* test.debug(output);  
 }  
 **if** (result.getThrowable() != **null**) {  
 test.log(status, result.getThrowable());  
 }  
 **else** {  
 test.log(status, **"Test "** + status.toString().toLowerCase() + **"ed"**);  
 }  
  
 test.getModel().setStartTime(getTime(result.getStartMillis()));  
 test.getModel().setEndTime(getTime(result.getEndMillis()));  
 }  
 }  
 }  
  
 **private** Date getTime(**long** millis) {  
 Calendar calendar = Calendar.*getInstance*();  
 calendar.setTimeInMillis(millis);  
 **return** calendar.getTime();  
 }  
}

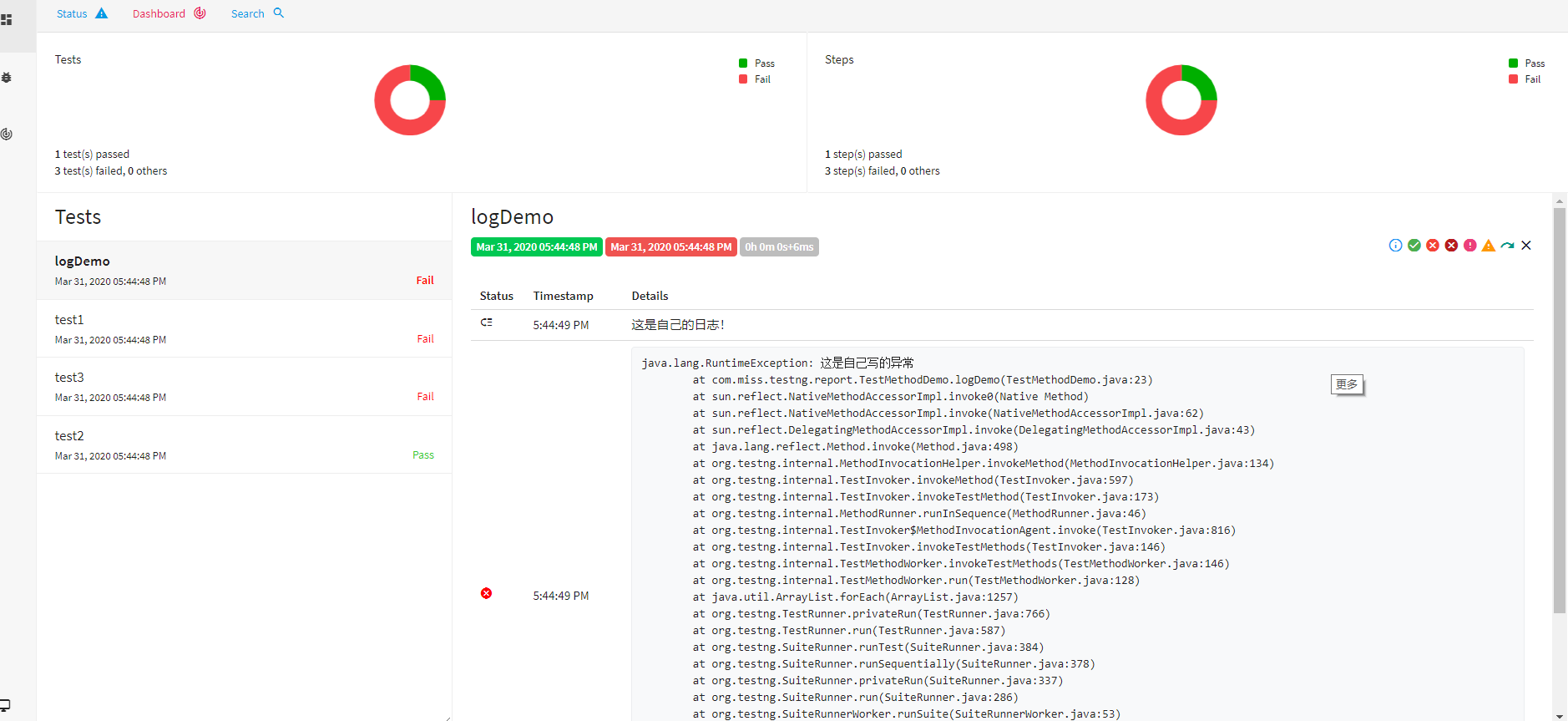
1. **修改监听项**

*<?***xml version="1.0" encoding="UTF-8"** *?>*<**suite name ="自己的测试套件"**>  
 <**test name ="测试模块"**>  
<**classes**>  
 <**class name ="com.miss.testng.report.TestMethodDemo"**></**class**>  
 <**merhods**>  
 <**include name ="logDemo"**></**include**>  
 <**include name ="test1"**></**include**>  
 <**include name ="test2"**></**include**>  
 <**include name ="test3"**></**include**>  
  
 </**merhods**>  
</**classes**>  
 </**test**>  
  
 <**listeners**>  
*<!--<listener class-name= "com.vimalselvam.testng.listener.ExtentTestNgFormatter"/> -->* <**listener class-name= "com.miss.testng.report.ExtentTestNGIReporterListener"**/>  
 </**listeners**>  
</**suite**>

**6.打开test-output下的html文件**



解决样式无法加载问题



# moco框架

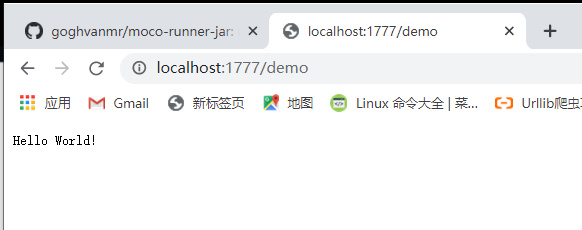
1. 新增json文件startup.json

[  
 {  
 **"description"**:**"第一个demo"**,  
 **"request"**:{  
 **"uri"**:**"/demo"** },  
 **"response"**:{  
 **"text"**:**"Hello World!"** }  
 }  
   
  
  
]

1. 在bash命令行中输入

java -jar ./moco-runner-0.10.0-standalone.jar http -p 8817 -c StartupGet.json

3.成功启动后即可在浏览器中通过 localhost:7777/demo 访问text中的内容



# Moco框架的http协议get方法Mock实现

Moco框架是热部署，无需重启自动更新 自动加载

Startup.json的内容

[  
 {  
 **"description"**:**"模拟一个没有参数的get请求"**,  
 **"request"**:{  
 **"uri"**:**"/GetDemo"**,  
 **"method"**:**"get"** },  
 **"response"**:{  
 **"text"**:**"这是个没有参数的Get请求"** }  
 },{  
 **"description"**:**"模拟一个有参数的get请求"**,  
 **"request"**:{  
 **"uri"**:**"/GetParamDemo"**,  
 **"method"**:**"get"**,  
 **"queries"**:{  
 **"name"**:**"liuxu"**,  
 **"age"**:**"26"** }  
 },  
 **"response"**:{  
 **"text"**:**"这是一个有参数的Get请求"** }  
  
}

在bash中启动；只有参数正常时 页面才会正常展示

java -jar ./moco-runner-0.10.0-standalone.jar http -p 8817 -c StartupGet.json

<http://localhost:8817/GetParamDemo?name=liuxu&age=26>

有时候不能写两个request 第二个会乱码

[  
{  
 **"description"**:**"模拟一个有参数的get请求"**,  
 **"request"**:{  
 **"uri"**:**"/GetParamDemo"**,  
 **"method"**:**"get"**,  
 **"queries"**:{  
 **"name"**:**"liuxu"**,  
 **"age"**:**"26"** }  
 },  
 **"response"**:{  
 **"headers"**: {  
 },  
 **"text"**:**"挖槽，这居然是一个带参数的moco"** }  
  
}  
  
  
  
]

浏览器访问中文乱码，可以在json文件中写明响应response的编码格式，在response中添加一下代码即可

"headers": {  
 "Content-Type": "text/html;charset=gbk"  
}

# Moco框架的http协议post方法mock实现

#有的module下无法新增package时 需要将src目录变为：Sources Root

新增json文件，内容如下

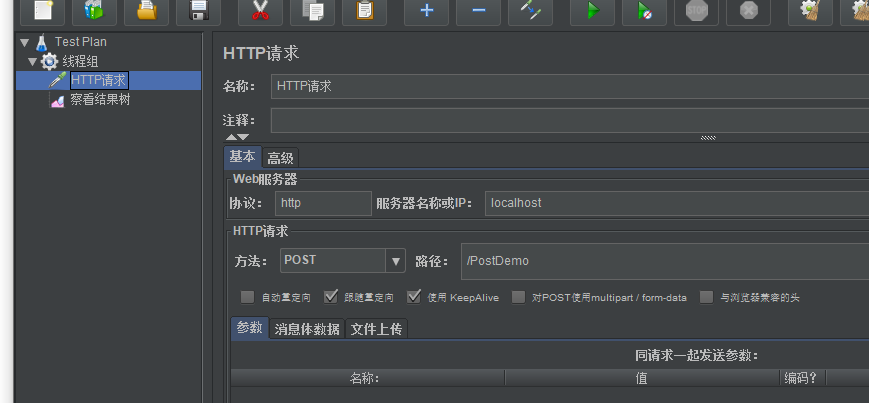
[  
 {  
 **"description"**:**"模拟一个post请求"**,  
 **"request"**:{  
 **"uri"**:**"/PostDemo"**,  
 **"method"**:**"post"** },  
 **"response"**:{  
 **"text"**:**"这可是一个post请求"** }  
  
  
 }  
  
]

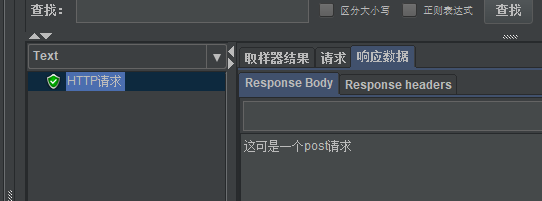
在bash运行

java -jar ./moco-runner-0.10.0-standalone.jar http -p 8817 -c StartupPost.json

**因为浏览器只能访问get请求，所以要使用jmter**

**新建线程组，添加http请求，添加查看结果树**





**带参数的Post方法实现**

#GBK和UTF8有什么区别？

UTF8编码格式很强大，支持所有国家的语言，正是因为它的强大，才会导致它占用的空间大小要比GBK大，对于网站打开速度而言，也是有一定影响的。

GBK编码格式，它的功能少，仅限于中文字符，当然它所占用的空间大小会随着它的功能而减少，打开网页的速度比较快。

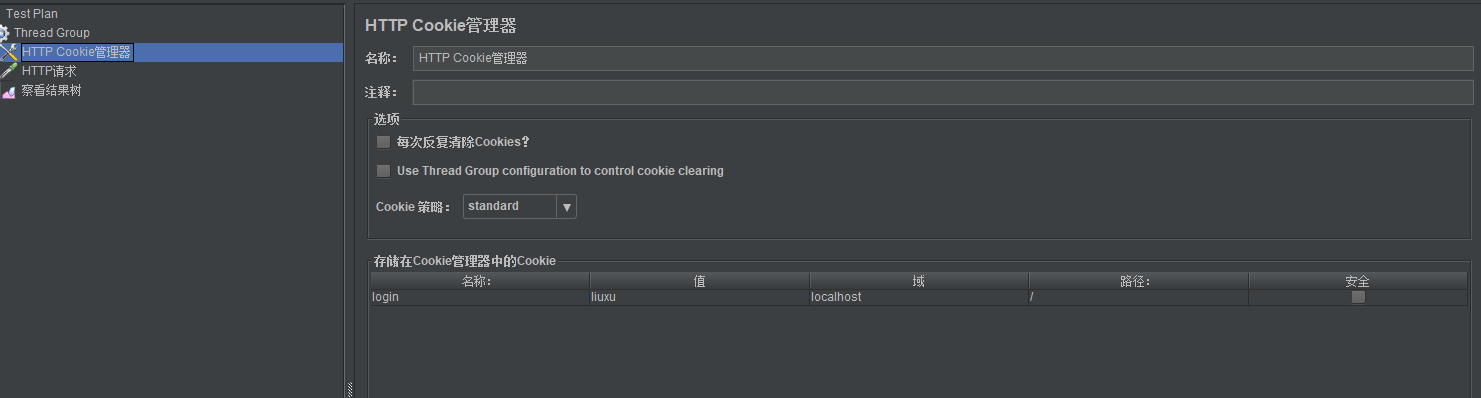
# 带有cookies的moco框架方法实现

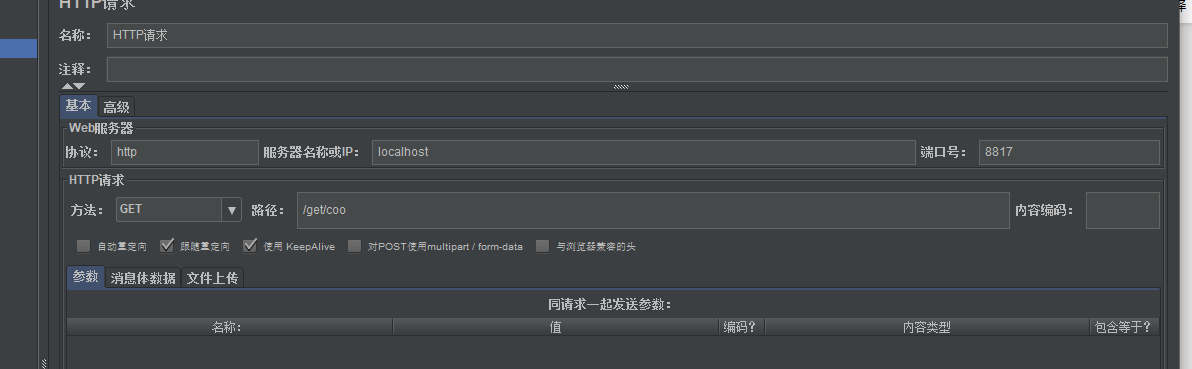
添加json文件

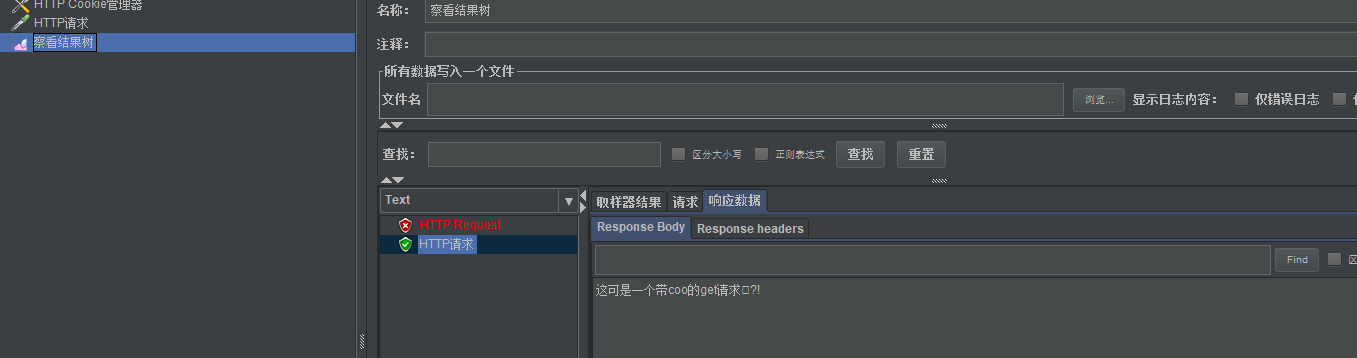
[  
 {  
 **"description"**: **"模拟一个带cookies的get请求"**,  
 **"request"**: {  
 **"uri"**: **"/get/coo"**,  
 **"method"**: **"get"**,  
 **"cookies"**: {  
 **"login"**: **"liuxu"** }  
 },  
 **"response"**: {  
 **"text"**: **"这可是一个带coo的get请求！!"** }  
 }  
]

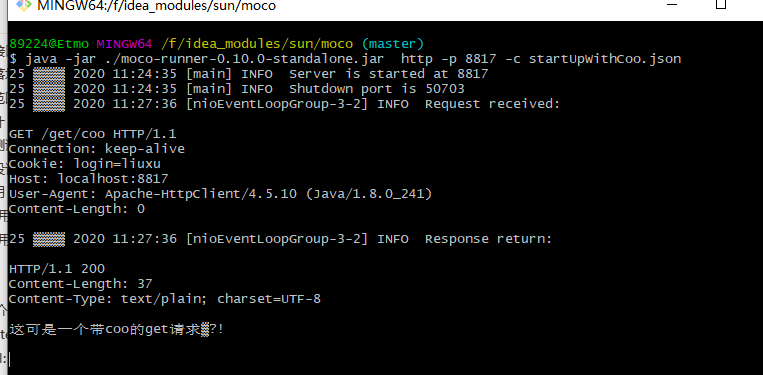
在bash中执行：java -jar ./moco-runner-0.10.0-standalone.jar http -p 8817 -c startUpWithCoo.json

在jmeter中做设置后可见





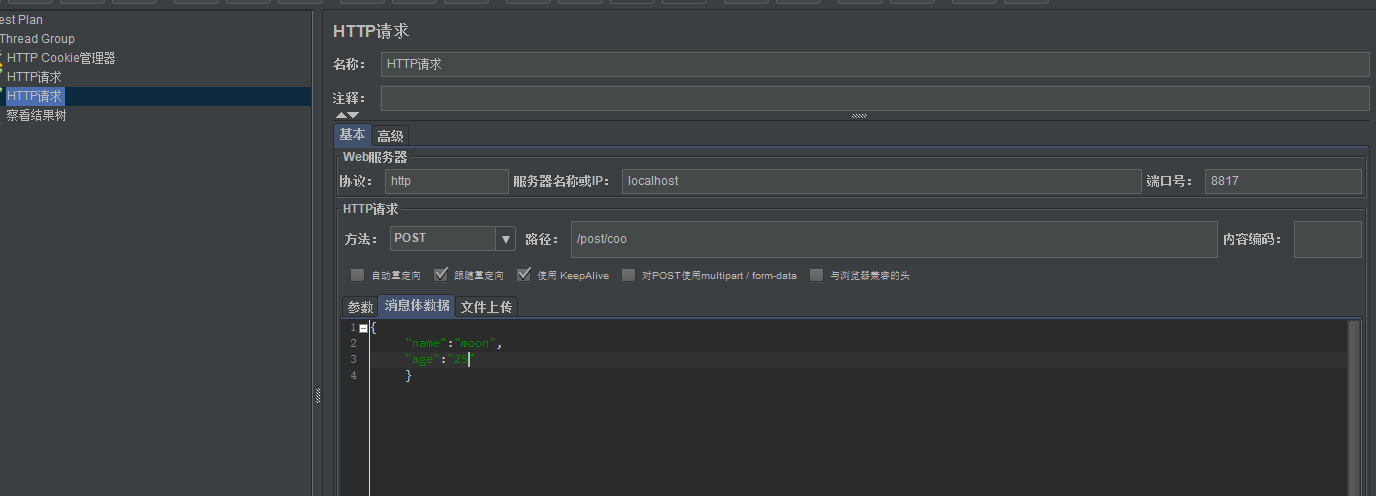




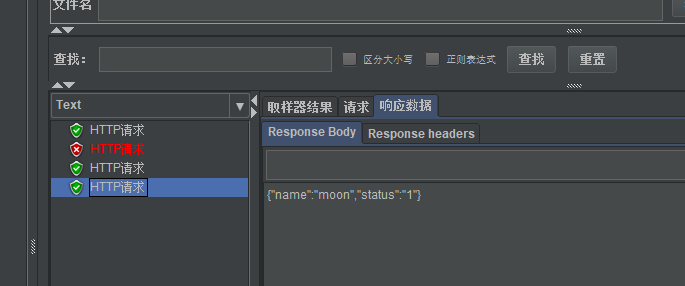
同理可得post方法

[  
 {  
 **"description"**: **"模拟一个带cookies的get请求"**,  
 **"request"**: {  
 **"uri"**: **"/get/coo"**,  
 **"method"**: **"get"**,  
 **"cookies"**: {  
 **"login"**: **"liuxu"** }  
 },  
 **"response"**: {  
 **"text"**: **"这可是一个带coo的get请求！!"** }  
 },  
 {  
 **"description"**: **"模拟一个带cookies的post请求"**,  
 **"request"**: {  
 **"uri"**: **"/post/coo"**,  
 **"method"**: **"post"**,  
 **"cookies"**: {  
 **"login"**: **"liuxu"** },  
 **"json"**:{  
 **"name"**:**"moon"**,  
 **"age"**:**"25"** }  
 },  
 **"response"**: {  
 **"status"**:**"200"**,  
 **"json"**:{  
 **"name"**:**"moon"**,  
 **"status"**:**"1"** }  
 }  
 }  
]

在jmeter中

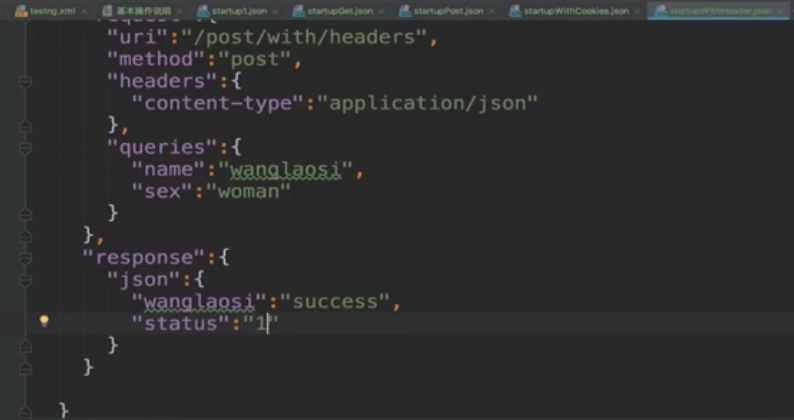


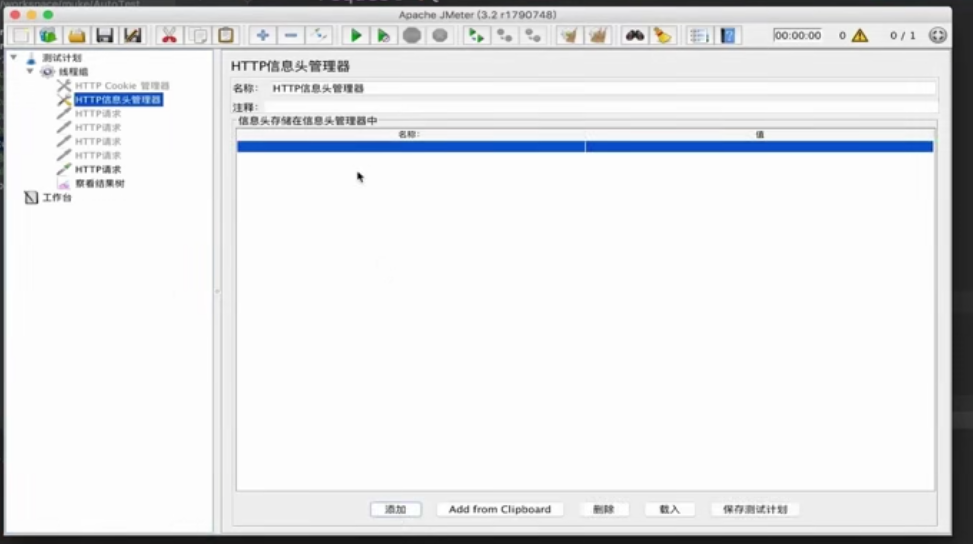
结果得出：



# moco框架中如何加入header

新增文件





# moco请求重定向

# 28.http协议进阶

**常用请求头**

Accept:浏览器告诉服务器它所支持的数据类型

Accept-Charset:浏览器告诉服务器它采用的字符集

Accept-Encoding:浏览器告诉服务器它所支持的压缩格式

Accept-Language:浏览器告诉服务器它所采用的语言

Host:浏览器告诉服务器我要访问哪台主机

If-Modified-Since:浏览器告诉服务器他缓存数据时间是多少

Referer:浏览器钙素服务器我是从哪个网页过来的（防盗链）

User-Agent:浏览器告诉服务器我使用的浏览器类版本等信息

Date：浏览器告诉服务器我什么时间访问的

常用响应头

Location:告诉浏览器你去找谁，配额302状态码使用

Server:告诉浏览器服务器的类型

Content-encoding:告诉浏览器回送的数据采用的压缩格式

Content-Type：告诉浏览器回送数据的数据类型

Last-Modified:告诉浏览器数据的最后修改时间

Refresh:用于控制浏览器定时刷新

**Cookie与Session**

cookie在客户端的头信息中

Session在服务端存储，文件、数据库都可以

一般来说session的验证需要cookie带一个字段来标识这个用户是哪一个session

当客户端禁用cookie时，session将失效

Cookie就是一小段文本信息

Cookie的格式是 key:value; key:value

cookie由服务端生成，客户端保存

再看第九章