# AutoTest Studio入门系列1：初识AutoTest Studio

**AutoTest Studio 简介**

AutoTest Studio是一款优秀的软件自动化测试平台，以python语言为用例编写语言，集成用例开发，调试，管理，任务调度，自动生成测试日志和测试报告等功能，满足了大多数项目的自动化实施需求，并且它还是免费的，相比其他同类的自动化测试框架，比如Robot Framework,TestNg，它更加灵活，简单，不需要手工安装众多的依赖包和复杂的环境搭建。

官网：<https://www.autoteststudio.com/>

下载地址：<https://www.autoteststudio.com/?page_id=1259>

最新的版本是2.0.3。

官方给出的特性介绍：

**Logs and Reports**

测试日志记录了测试用例运行的细节，测试报告可以统计每个测试任务中测试用例运行的结果并以图表的形式呈现。

[**Data Dictionary**](https://www.autoteststudio.com/?page_id=1258&c=t_008)

AutoTest Studio 为每个测试项目维护一个独立的数据配置文件。通过数据字典管理工具，您可以配置多达数百万条数据记录，并在测试用例中通过API接口访问这些数据。

**Package Manager**

Python 包管理器允许您通过图形界面安装、删除、升级、导入和导出 Python 包。

**Code Editor**

AutoTest Studio拥有专业的代码编辑器，可以进行关键词高亮、代码折叠、展开、代码自动补全、代码格式化等操作。

**Debugger**

AutoTest Studio 集成了强大的 Python Debugger，可以让您更轻松地发现程序运行中的异常和错误信息。

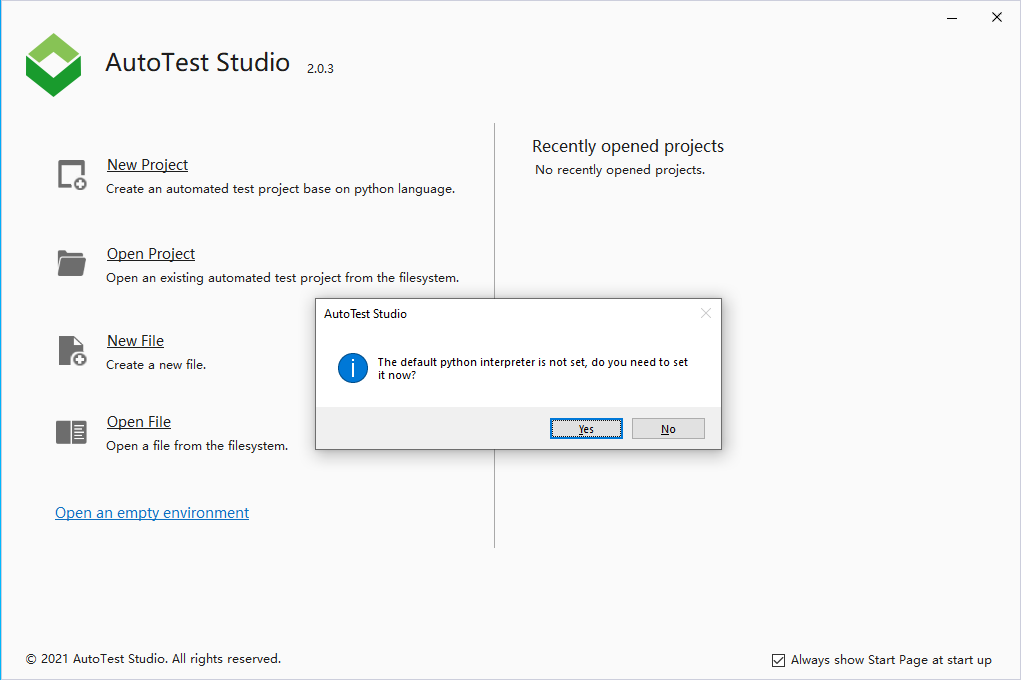
**Test case Runner**

AutoTest Studio 拥有先进的任务引擎，可以提供多种任务运行调度方式，如定时任务、循环运行、指定状态下的测试用例运行等。

**AutoTest Studio 安装与配置**

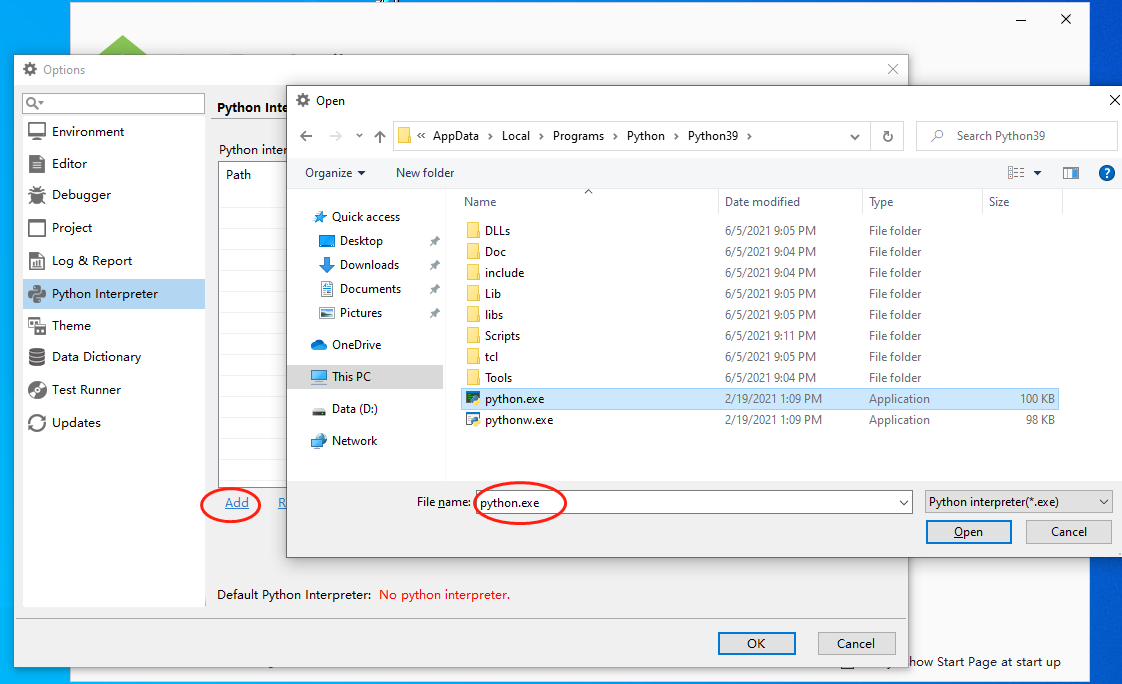
从官方下载安装包后，点击“Next”很容易就能完成安装。

完成安装后，首次运行程序，会提示配置python解释器路径，需要提醒一下，这个python解释器是程序后台一些插件运行需要，并非项目运行的python环境，在第二节中，我会介绍如何配置项目运行的python环境。

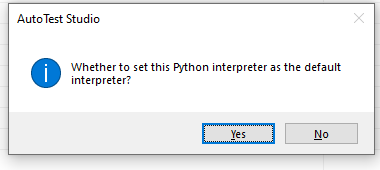


当然这里也可以直接跳过，以后再配置，这里我们按照提示配置python解释器。

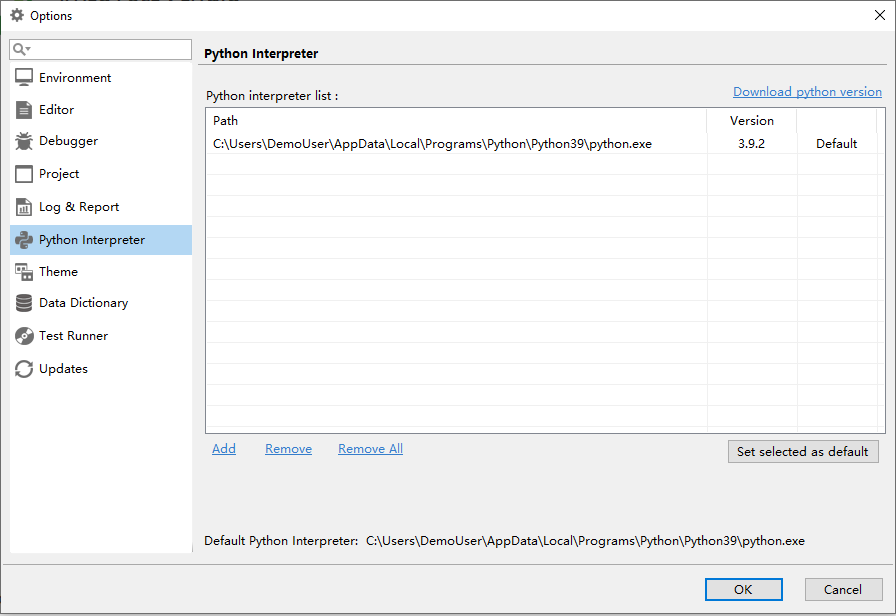
点击“Add”添加python解释器路径。



在这里可以添加多个python解释器，你可以通过设置默认python解释器来选择程序使用的python解释器，如果你没有设置默认的python解释器，添加的时候会提示你是否设置当前的python解释器为默认的python解释器。



设置python解释器为默认解释器后，程序会自动检查并安装程序需要的python包。



设置完成后，点击OK.

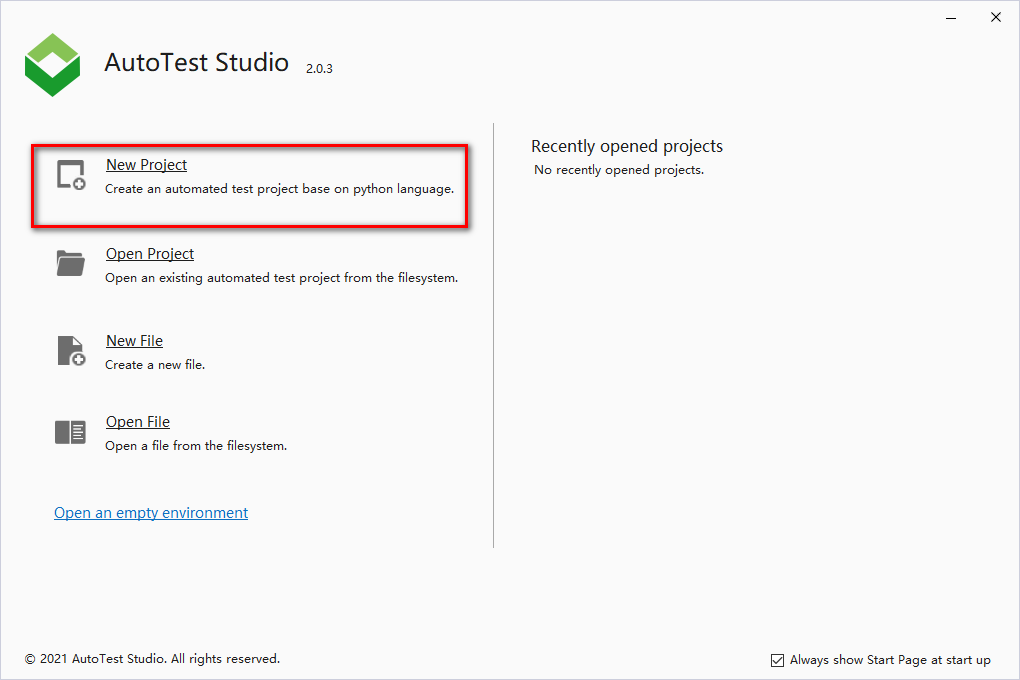
至此，已经完成AutoTest Studio安装和基本配置。

# AutoTest Studio入门系列2：从一个项目开始

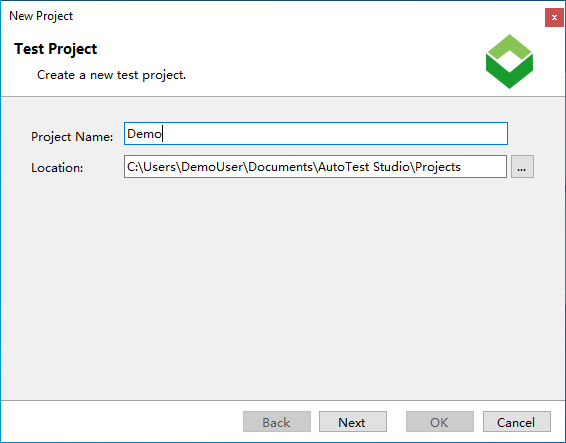
在这一节中，我将以一个简单的demo为例，介绍如何开始一个测试项目，demo的场景为测试一个查询ip地址位置信息的web service接口，通过判断接口调用后的状态码来验证接口是否正常，接口原型：http://ip-api.com/json/X.X.X.X（X.X.X.X是要查询的ip地址）。

**新建测试项目**

打开AutoTest Studio，在起始页选择“New Project”，打开新建项目向导。



在新建项目向导对话框输入项目名称和项目保存位置，本文以“Demo”为例，保存位置为默认路径。



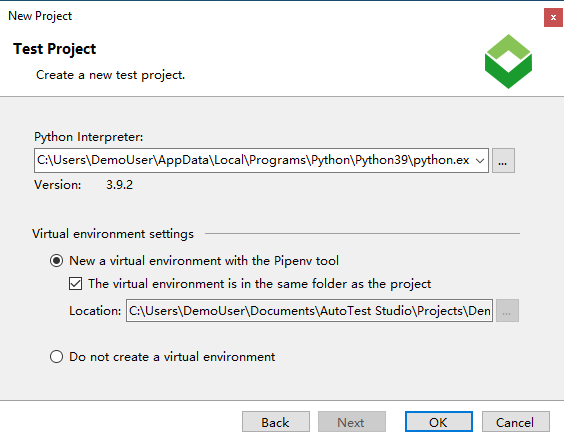
配置项目的python环境，“Python Interpreter”可以从预先配置的python列表中选择一个，也可以通过后面的“…”按钮选择一个新的python环境。

AutoTest Studio支持以虚拟环境运行，当你选中“New a virtual environment with the Pipenv Tool”，AutoTest Studio将会以选中的“Python Interpreter”为基础创建一个虚拟环境，当同一台PC上有多个测试项目时，虚拟环境可以将项目运行的python环境隔离开来，让每个项目都有独立的python运行环境。

“The virtual environment is in the same folder as the project”，将虚拟环境的配置信息与项目保存在同一个目录。

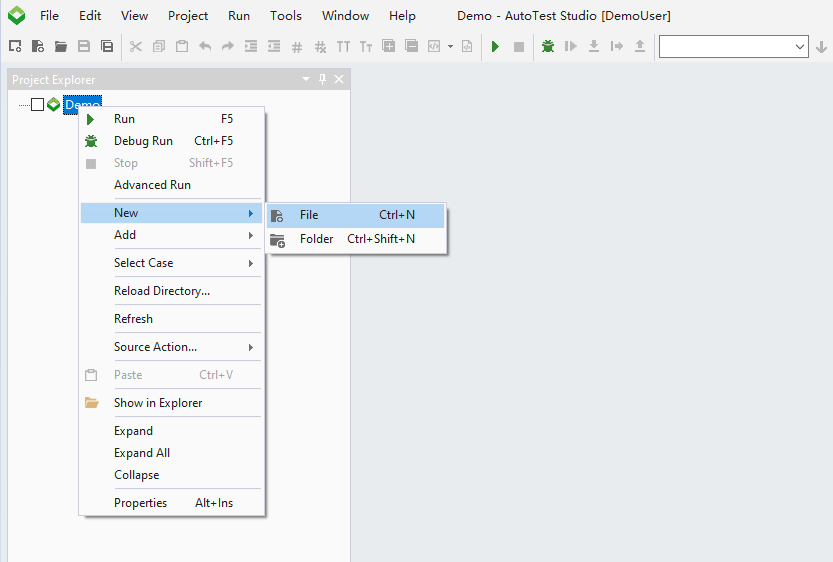
**注意：**Python环境配置设置完成后，如果后续需要修改，也可以在项目属性中修改。

点击“OK”按钮，AutoTest Studio将自动创建项目，包括python环境的初始化。



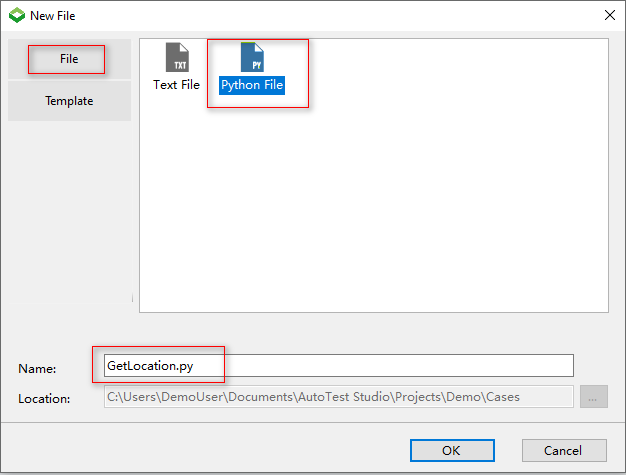
**新建测试用例**

在AutoTest Studio主界面，在右键菜单中，点击“New”🡪“File”弹出测试用例创建向导，这里需要注意是，只有通过项目目录的右键菜单创建的测试用例或者文件才能与当前测试项目关联，并出现在项目的目录结构中。



通过创建向导，选择“File”🡪”Python File”新建测试用例文件，填写用例用例名称，文件保存位置不可改变。

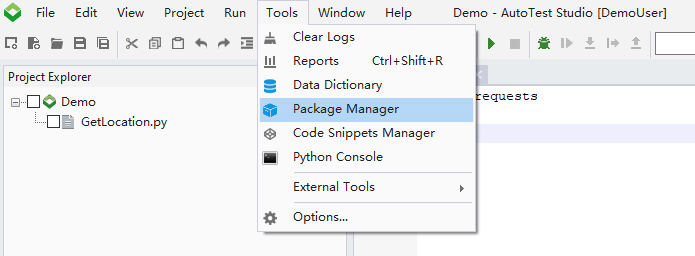
创建测试用例也可以通过模板创建，AutoTest Stduio允许用户根据需要创建不同的模板文件，并添加到程序中。



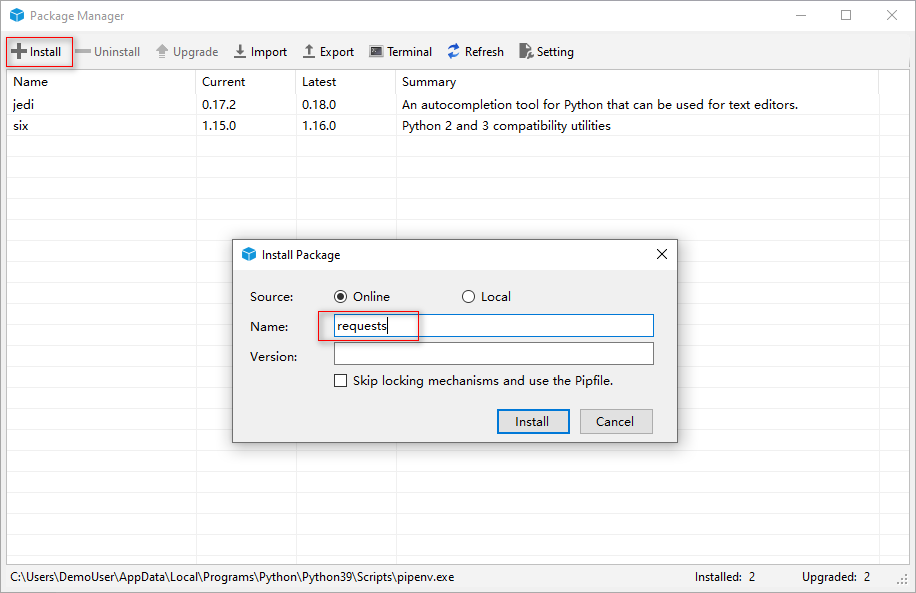
**安装requests包**

因为demo中的测试用例会用到python的requests包，因此需用通过python包管理器工具添加requests包，在后续章节中，我会详细介绍Package Manager的使用。

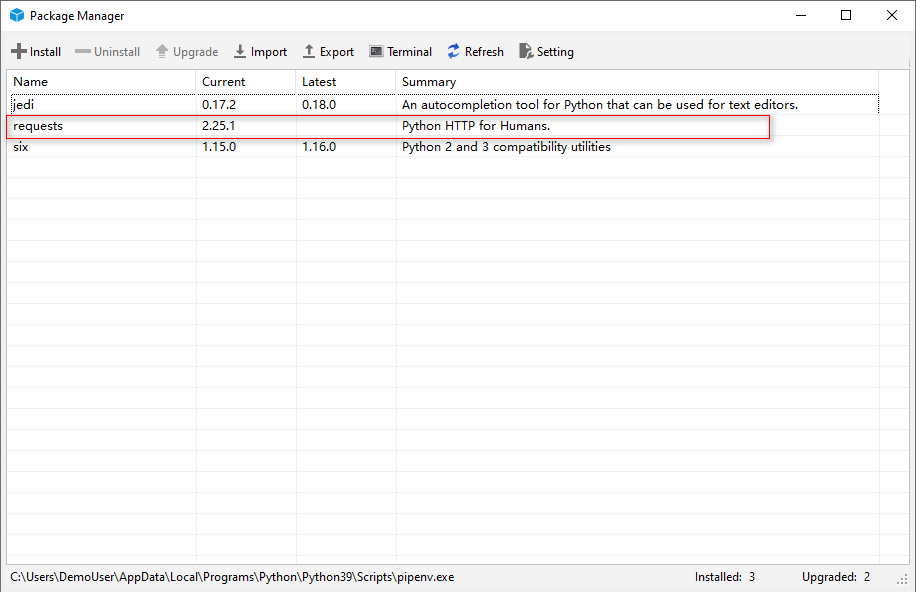
在AutoTest Studio的主菜单中点击“**Tools**”—>“**Package Manager**”打开python包管理器。



在“**Package Manager**”中点击“Install”在弹出的对话框，输入“requests”，点击“Install”, “**Package Manager**”会自动按照requests包以及相关依赖项。



安装成功后，可以在列表中看到已经安装的requests包。



**编写用例代码**

在测试用例中输入以下代码，测试代码比较简单，通过接口查询8.8.8.8这个ip地址的信息，并通过AssertEqual判断结果。

import requests

from autotest import \* *#引入AutoTest Studio内置的基础包*

SetCase("TEST-1","Get location by ip","1") *#设置测试用例的CaseID,CaseName,CaseLevel*

ip="8.8.8.8"

ipQueryUrl="http://ip-api.com/json/{0}".format(ip)

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

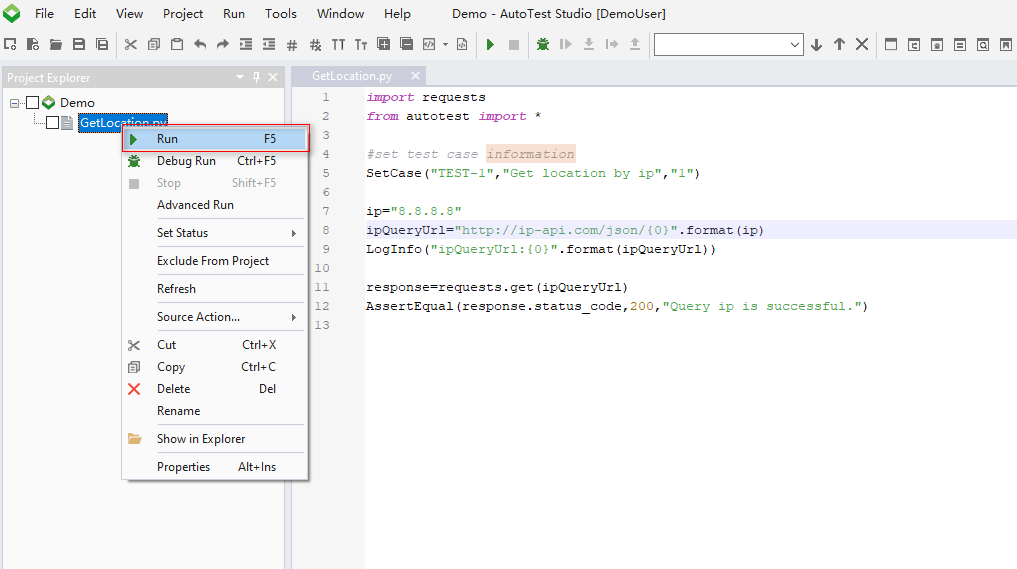
response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.") *#判断查询结果*

**运行测试用例**

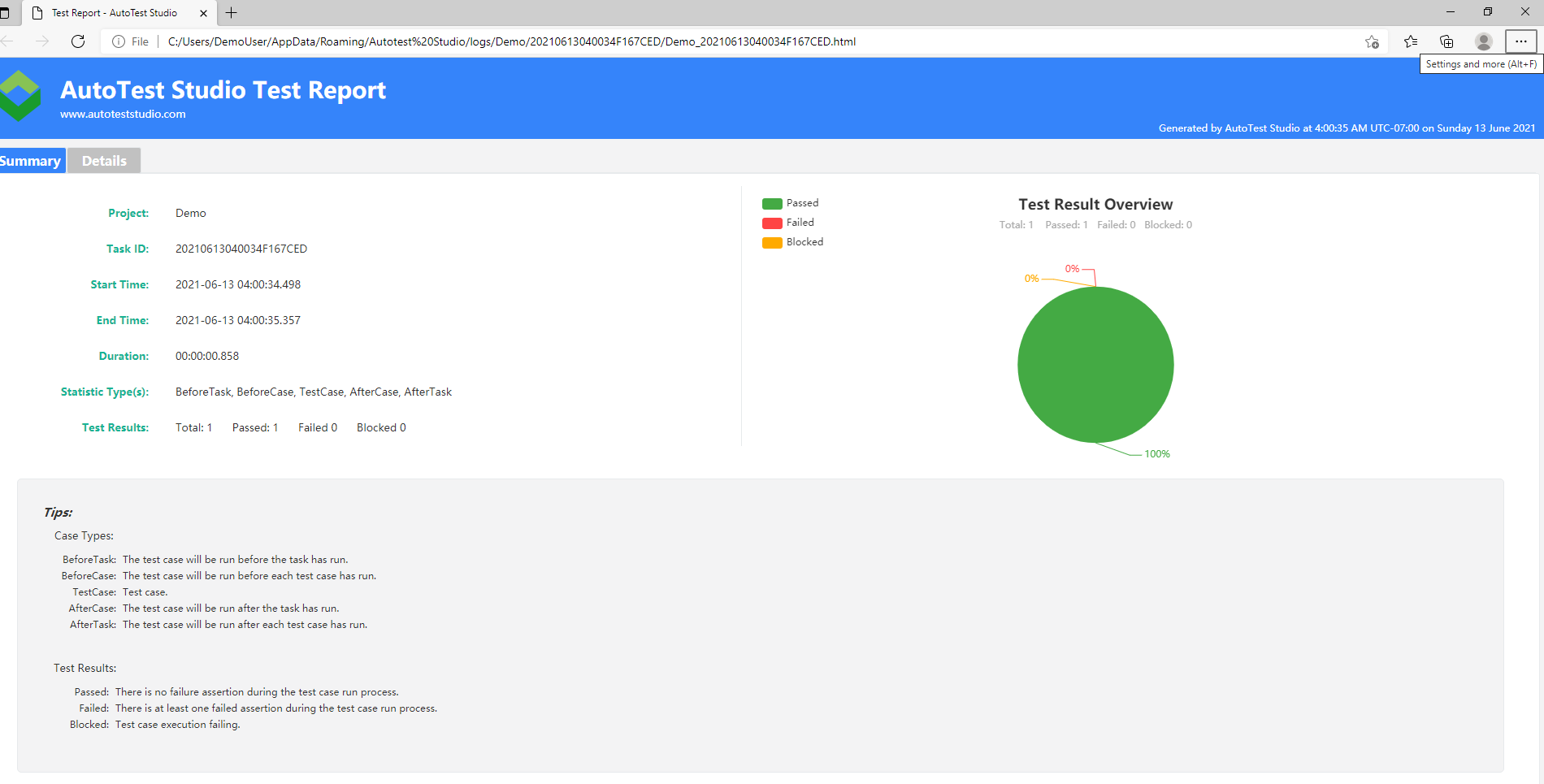
在项目的目录结构中选中用例，点击右键菜单中“Run”，运行测试用例，运行完成后，AutoTest Studio会自动打开测试报告。

**注意：**运行单个测试用例，直接选中用例后点击右键菜单中“Run”即可，如果是运行多个测试用例，则需要选中要运行的测试用例，并在父目录的右键菜单中点击“Run”。

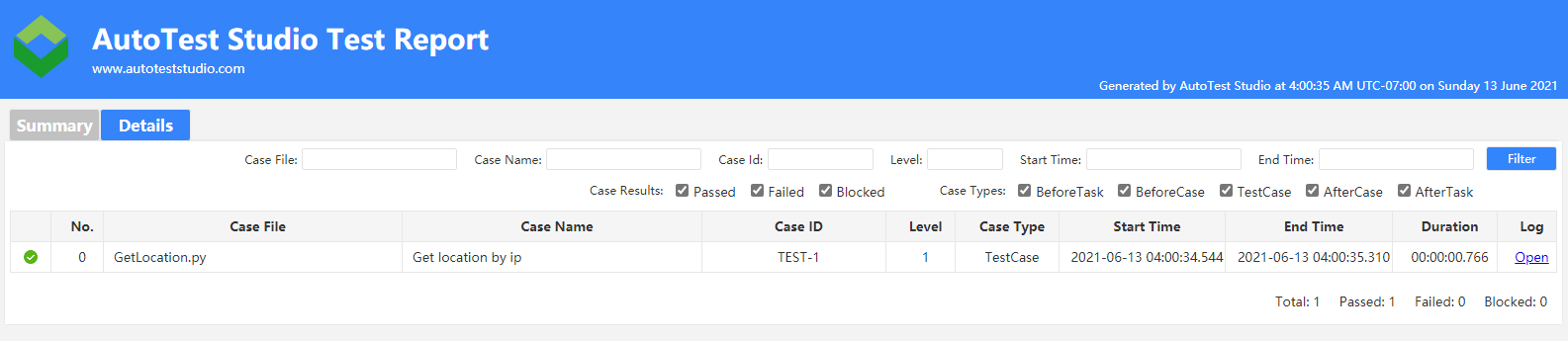


**查看测试报告和日志**

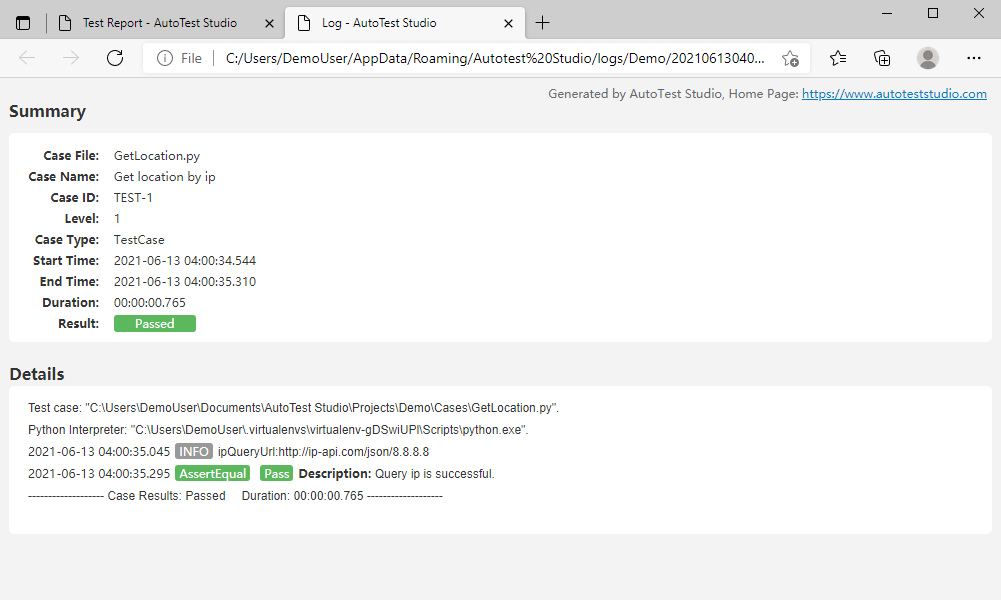
测试报告中包括当前任务运行的基本信息和统计信息。



点击测试报告的“Details”，可查看测试用例运行信息。



在测试用例列表项右侧，点击“Open”，可以打开用例运行的日志文件。



自此，已经介绍完了如何通过AutoTest Studio创建测试任务和测试用例，并运行了测试任务。

# AutoTest Studio入门系列3：Package Manager

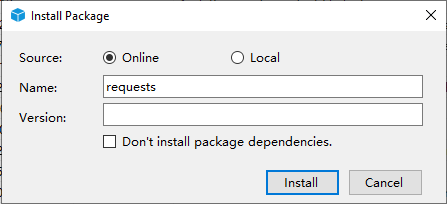
AutoTest Studio提供一个可视化的python包管理器- Package Manager，可以非常便捷的完成项目中python包的安装，查看和卸载操作，避免了命令行操作的繁琐。

Package Manager操作的python环境是当前项目的python环境，如果使用了虚拟环境，则是指虚拟环境的python包，对用户而言，无需关心python环境是否为虚拟环境，安装，卸载，查看操作都是相同的。

本文以上一节中用到的requests包为例，介绍如果安装，卸载，查看，升级，配置安装源的操作。

**在线安装**

在线安装，即通过网络直接安装python包，推荐使用这种方式。



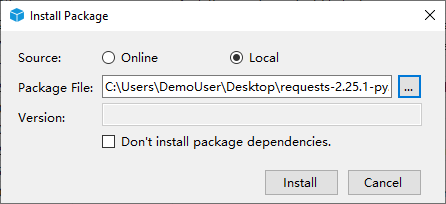
**Name**: python包名称；

**Version**: 包的版本，为空时，默认安装最细的版本;

**Don’t install package dependencies**: 不安装相关的依赖，只安装指定的包，建议不要勾选。

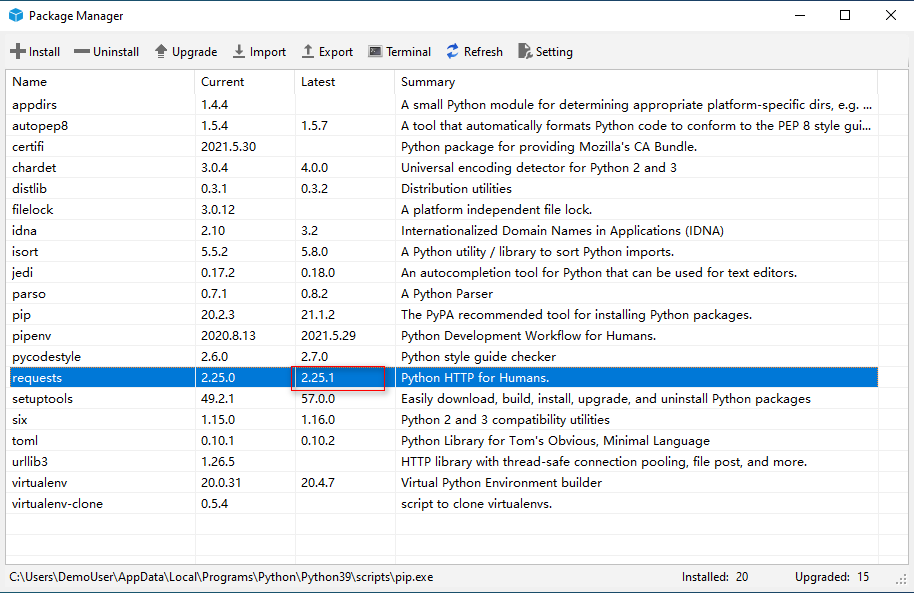
**离线安装**

如果包已经在本地存在，则可以使用本地安装，选择“Local”,点击“…”选择Python包的路径。

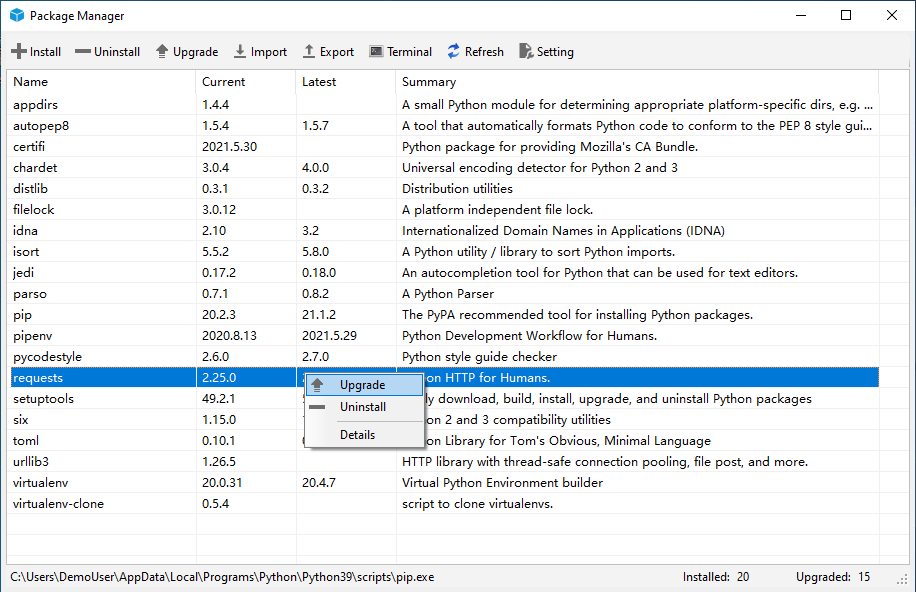


**升级**

Package Manager会自动检查最新的版本，并在Latest中列出最细的版本，为了演示升级操作，可以尝试先安装requests的2.25.0版本，在安装完成后，如下。



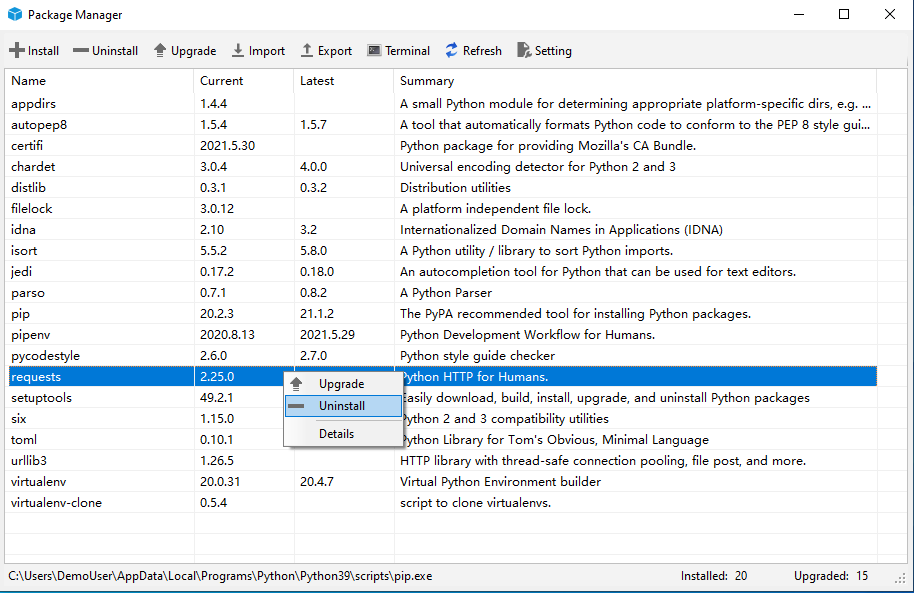
在列表选中request，点击“Upgrade”，Package Manager会自动升级到最新版本。



**卸载**

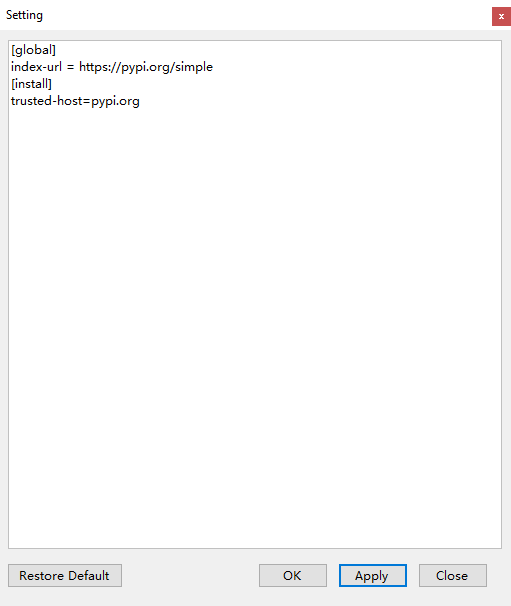
选中需要卸载的包，点击“Uninstall”，即可卸载包。

**注意：**卸载包时，不会卸载依赖的包。



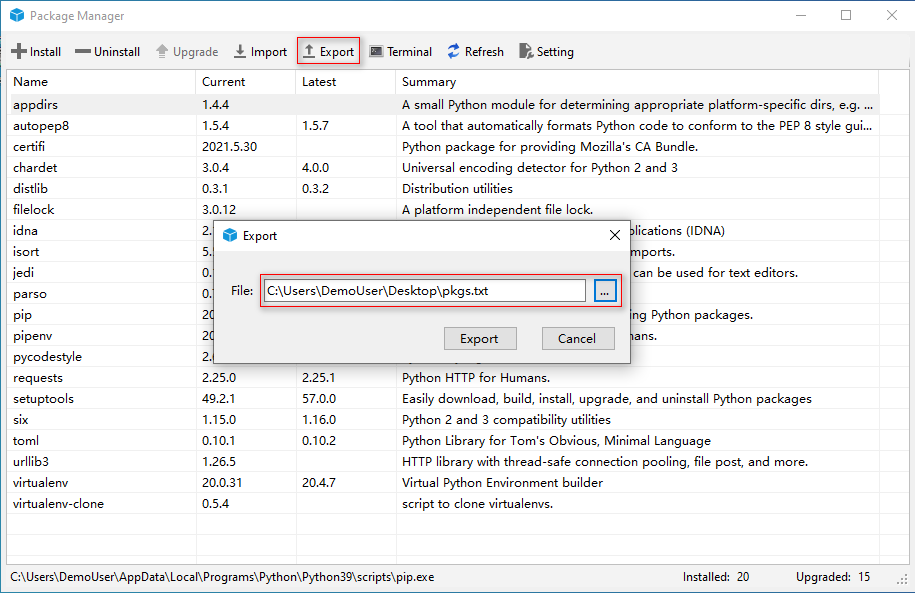
**配置安装源**

Package Manager允许用户配置安装源，实际就是配置pip.ini文件，在工具栏点击“Setting”，弹出的对话框中编辑安装源信息后，点击“OK”保存。如果需要恢复到默认安装源，可以点击底部的“Restore Defualt”按钮，默认安装源为python官方安装源。



**导出包列表**

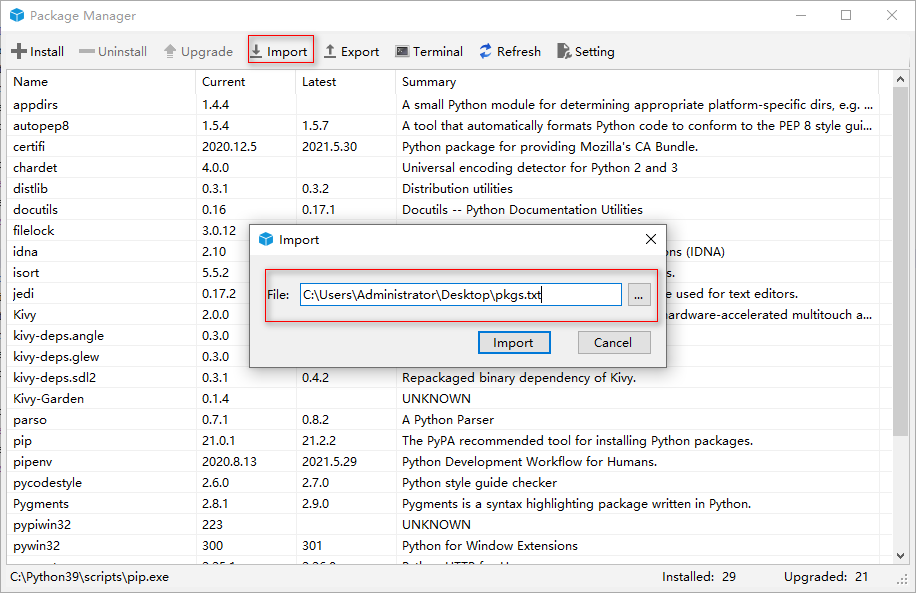
Package Manager可以导出当前环境已经安装package列表。



导出的列表实际上是一个包含了python包名和版本的文本文件。

**导入包列表**

如下图，可以将文本文件中的包导入到当前环境中，导入，导出功能主要是用来将当前环境包信息迁移到另外一个python环境中。需要注意的是，导入包时实际上就是在线安装，因此需要环境连接到互联网。



Package Manager避免了通过命令行管理包的繁琐，如果你对pip，pipenv命令足够熟悉，你也可以直接通过命令行的方式管理python包。

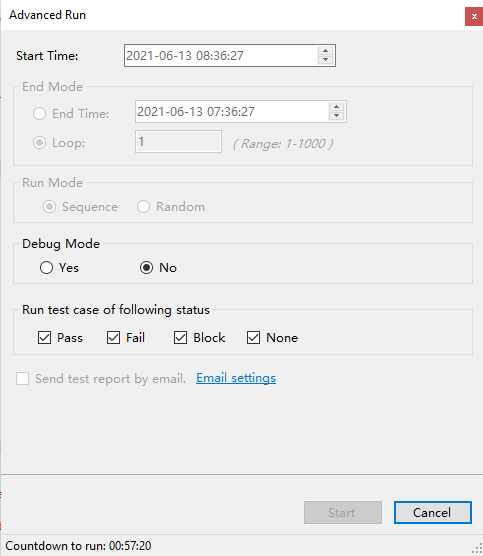
# AutoTest Studio入门系列4：任务调度

AutoTest Studio有一个强大的任务调度引擎，可以让用户灵活的运行测试任务，如，定时运行任务，定量运行任务，运行指定状态的测试用例。

为例便于演示，我们将第二节中的测试用例复制5份，同时将caseID也对应修改为：TEST-1, TEST-2, TEST-3, TEST-4, TEST-5。

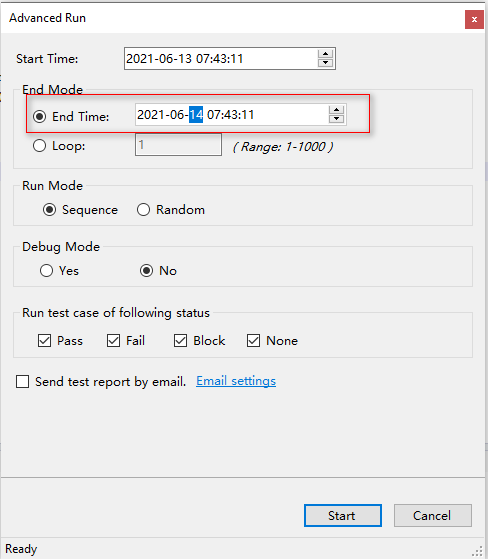
**定时运行**

当设定的Start Time大于当前时间时，AutoTest Stduio会进入倒计时，直到到达指定时间时，才会运行，如果start time小于当前时间时，则会立即运行任务。



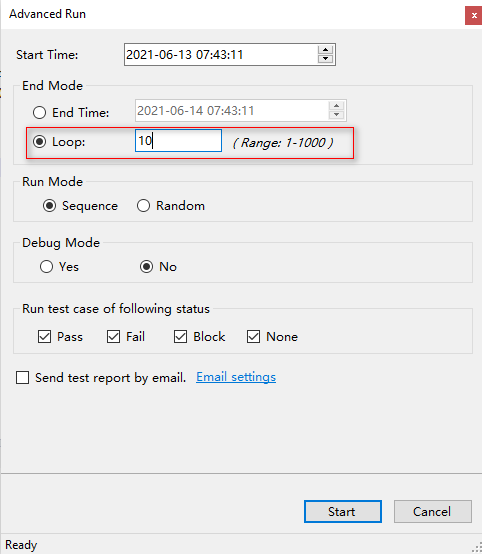
**定时结束**

AutoTest Studio允许设定结束时间，当到达结束时间时，程序立即终止任务运行，这里你不用担心AutoTest Studio会暴力的结束任务，事实上，结束时间到达后，AutoTest Studio会运行完当前测试用例后在终止任务。



**指定运行的次数**

AutoTest Studio除了提供指定时间结束任务，也提供指定运行次数，如下图，AutoTest Studio会将任务中的所有用例运行10次后终止任务。

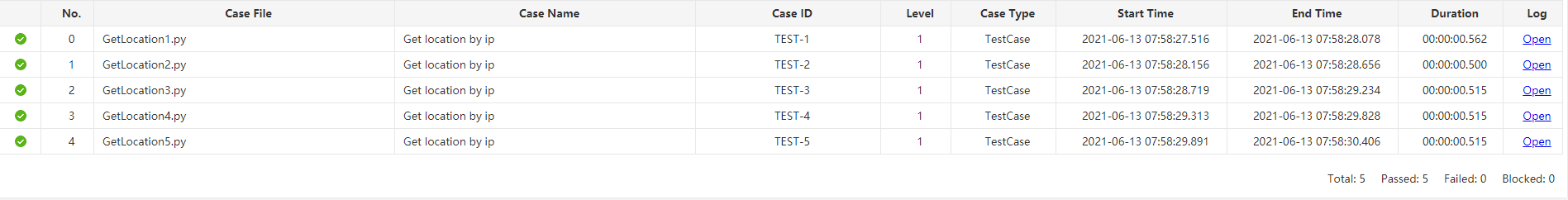


**运行模式**

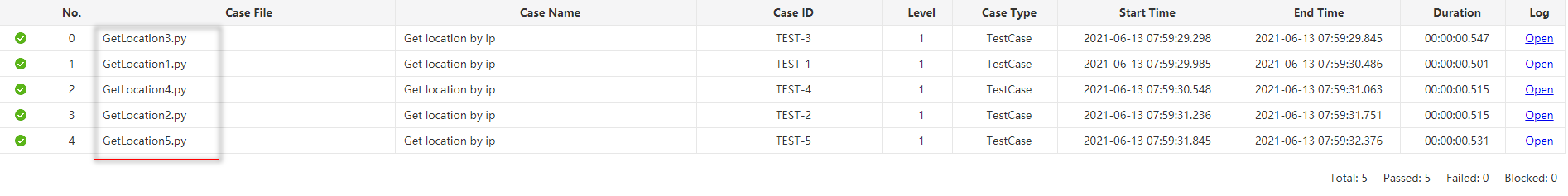
AutoTest Studio默认情况下会按顺序运行所有用例，但也提供了随机方式运行用例。

**Sequence**，顺序运行用例；**Random**，随机方式运行用例。

顺序运行的结果：



随机运行的结果

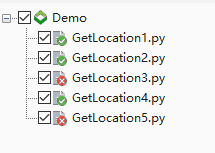


**调式运行**

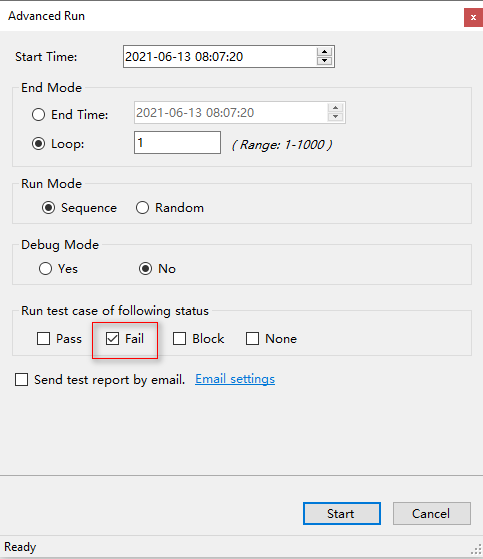
当Debug Mode 设置为“Yes”时，AutoTest Studio可以以调式方式方式运行测试用例，这个功能在我们定位一些“奇怪”问题时非常有用，我们可以让用例运行到特殊条件时进入断点。

**运行指定状态的用例**

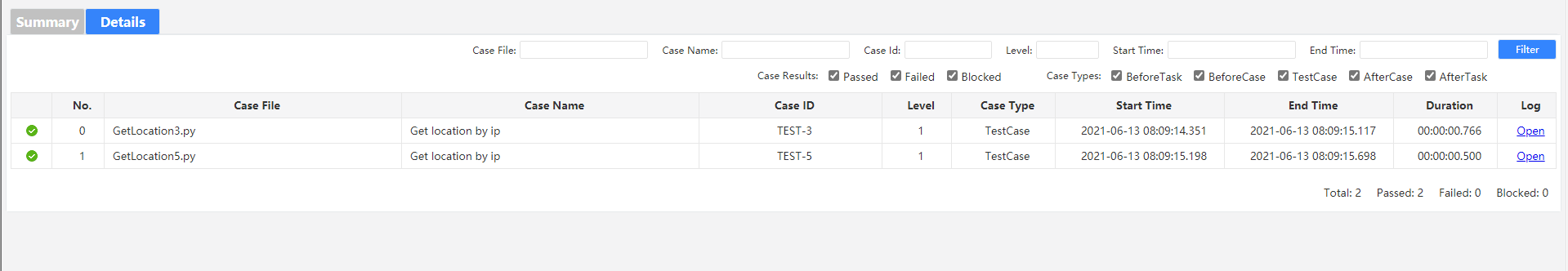
AutoTest Studio可以运行指定状态的用例，比如运行上一轮fail的用例。我们假定上一轮运行的结果如下图所示。



在“Advanced Run”的“Run test case of following status”设定只运行“Fail”状态的用例。

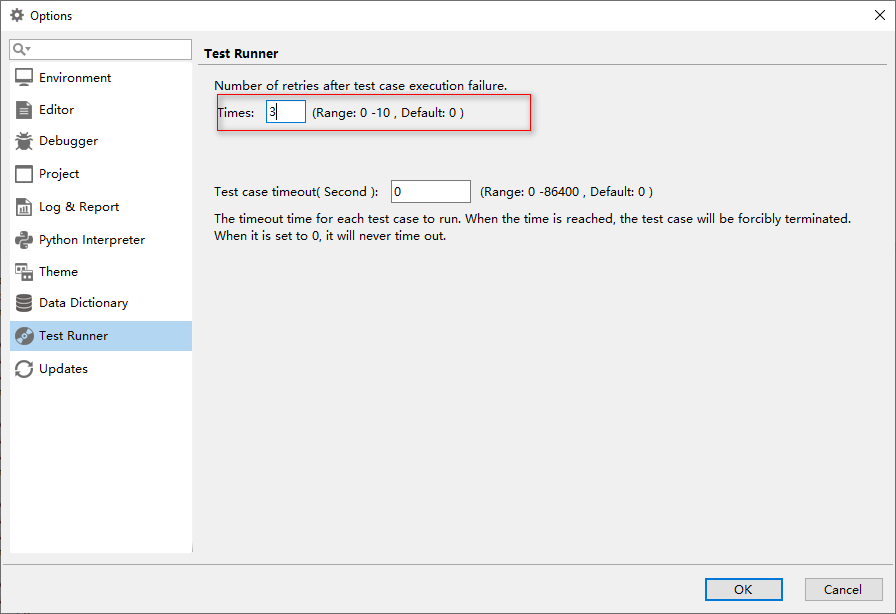


下面是运行的结果，可以看到AutoTest Studio只运行了“GetLocation3.py”，“GetLocation5.py”两个,即上一轮运行失败的用例。



**失败后重新运行**

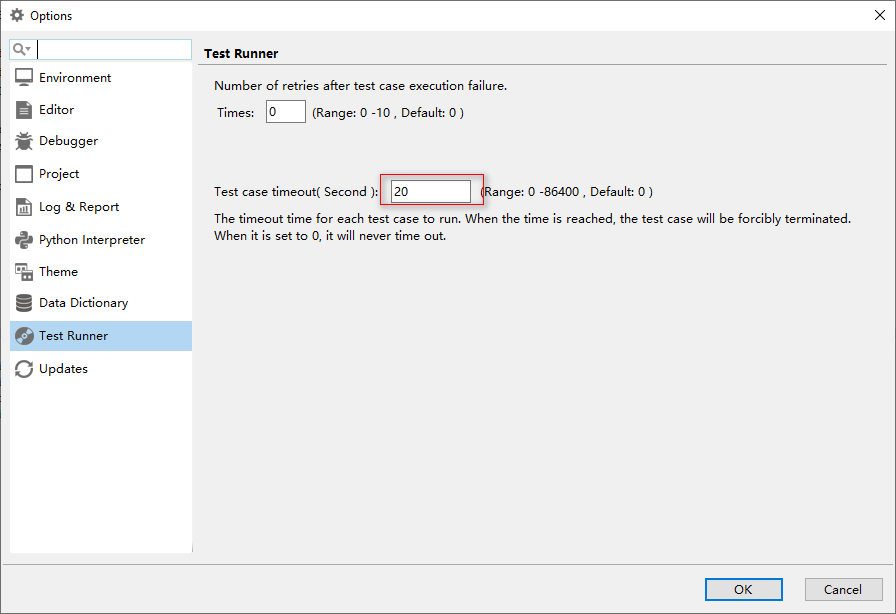
AutoTest Studio提供一个全局配置参数，允许用例运行失败后，重新运行，并能够指定重复运行的次数。默认为0，即不重复。



**用例运行超时设置**

在很多时候，可能因为一些异常原因，导致测试用例运行阻塞，但我们不希望因为这个用例的阻塞导致整个任务被阻塞，因此我们可以设置测试用例超时时间来强制中止阻塞的测试用例，继续运行其他用例，但这里需要注意的是，超时时间不能设置得过小，因为即使用例正常运行时，如果超过这个超时时间，也同样会被终止运行。

假定我们设置用例用例的超时时间为20秒，我们在用例使用time.sleep(25)来阻塞用例。

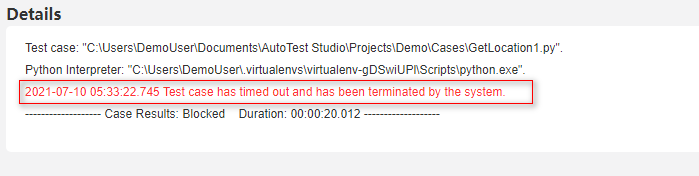


示例代码：

import time

time.sleep(25)

运行日志结果：



# AutoTest Studio入门系列5：预置用例

预置用例可以让你在测试任务或者测试用例运行前后，做一些额外的事情，例如，任务运行前检查环境或者设置某些参数，任务运行完成后清理环境或者恢复参数设置。

AutoTest Studio提供四种预置用例类型。

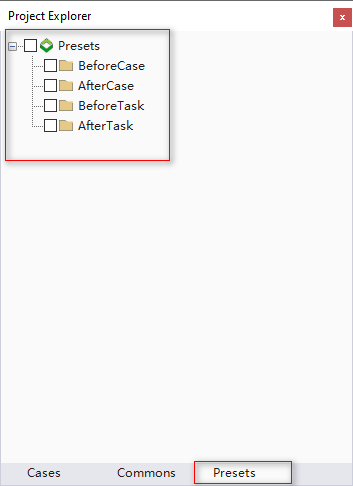
**BeforeTask**，每次任务开始时执行，即在所有用例运行前执行。

**BeforeCase**, 每个用例运行前执行。

**AfterCase**, 每个用例运行后执行。

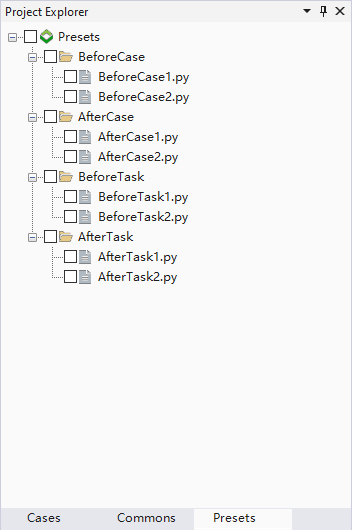
**AferTask**，每次任务结束时执行，即在所有用例运行结束后执行。

新建一个项目后，默认在“Presets”界面会有四个默认的文件夹，对应四种预置用例类型，如下图所示。

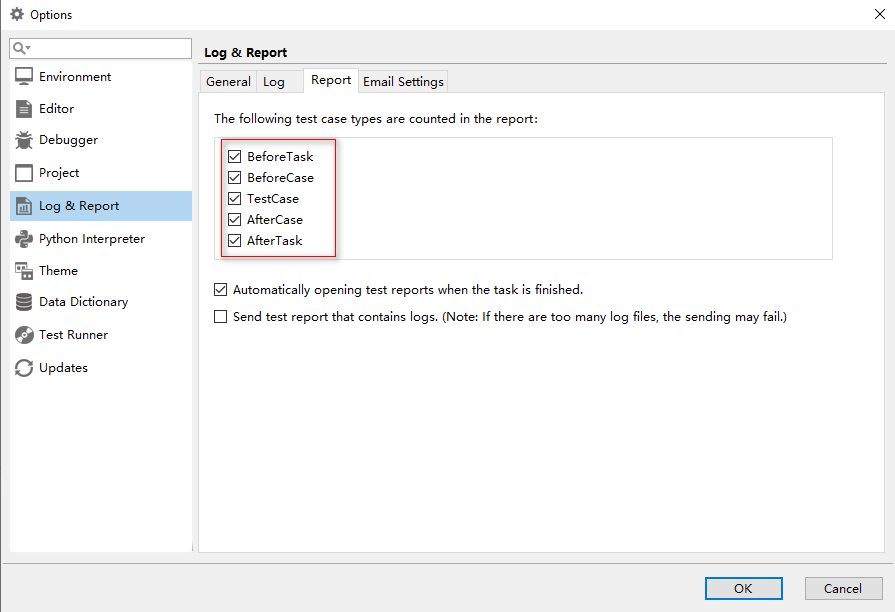


用户只需要在相应的文件夹下编写相应的用例即可，你可以在同一类型中编写多个用例，也可以一个也没有，这取决于的你需要。

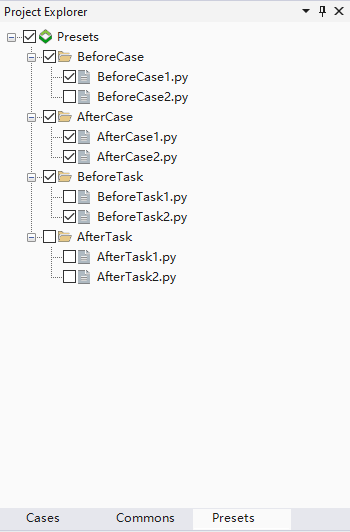
这里我们为每种类型编写两个用例，来演示预置用例的调用顺序，用例内容随意编写即可，预置用例在编写方式上与普通用例没有区别。



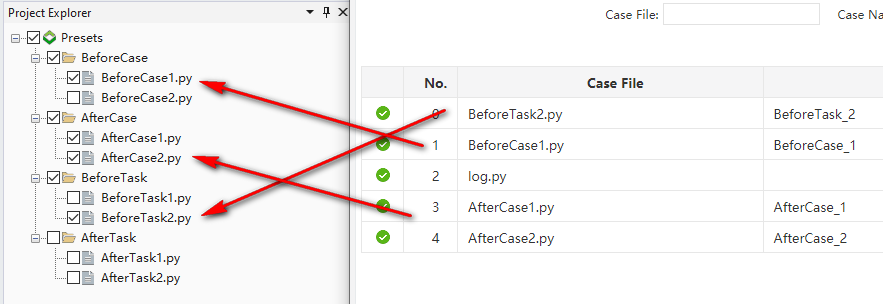
在设置中，我们选择统计所有类型。



在任务执行前，我们只需要勾选需要运行的一个或者多个预置用例即可，当然也可以一个都不选择，为了演示，我们如下勾选。

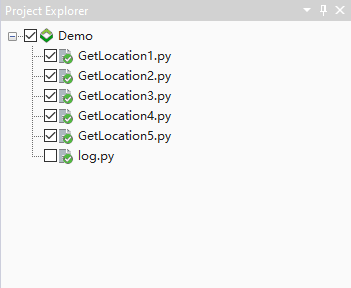


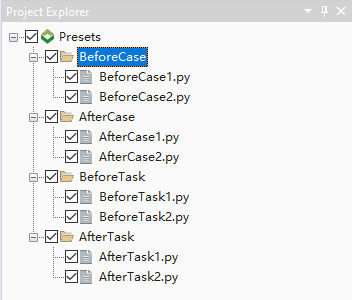
选择用例执行一次任务，执行结果如下。



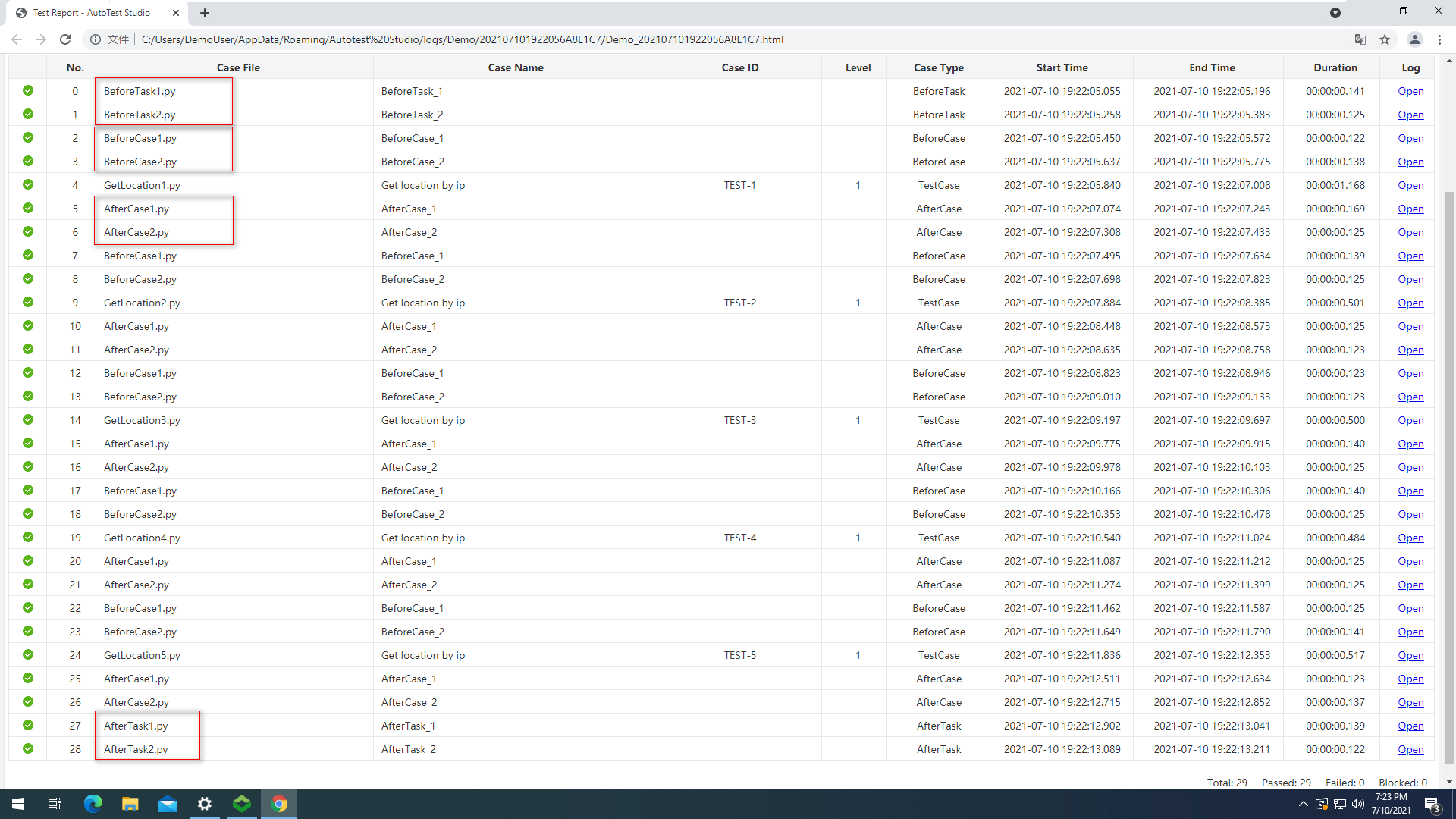
从以上截图可以看出，我们选择的预置用例都按照正确的逻辑执行，AfterTask目录下没有选择要执行的预置用例，所以日志中没有相应的记录。

我们再执行一次，将示例中所有预置用例勾选上，同时将前面章节编写的多个用例勾选上，看看效果。





运行结果如下（注意观察每个用例运行前后都执行了那些预置用例）。



# AutoTest Studio入门系列6：日志和报告

日志和报告是衡量一款自动化测试框架是否优秀的重要标准之一，AutoTest Studio也拥有完善的日志模块和报告模块，通过这些日志和报告可以评估和分析自动化测试任务运行的详细情况。

**日志**

AutoTest Studio拥有四种级别的日志，分别是DEBUG,INFO,WARN,ERROR,这四种级别的日志，用户可以根据自己的实际情况灵活应用，并且可以根据需要，决定显示哪些级别的日志。在实际应用中，我们可以在开发调试测试用例的时候，输出所有日志，但在任务运行时，选择性的输出部分日志，以减少日志的输出量。

以一段测试代码为例，来介绍如何输出日志和配置显示的日志级别。

from autotest import \*

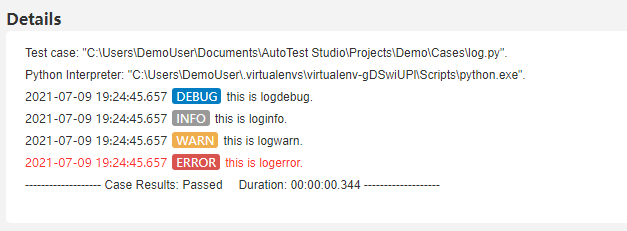
LogDebug("this is logdebug.")

LogInfo("this is loginfo.")

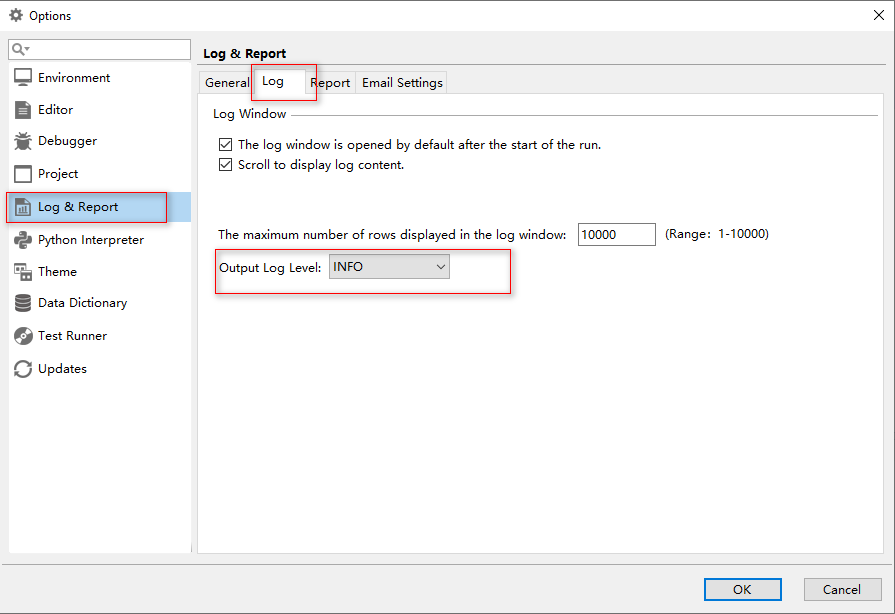
LogWarn("this is logwarn.")

LogError("this is logerror.")

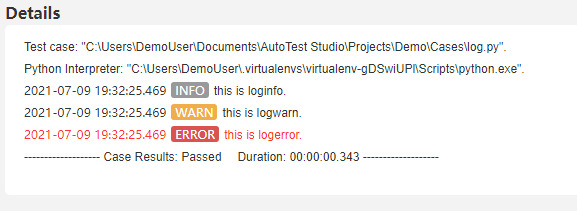
正常运行后，默认情况下，AutoTest Studio会显示四种级别的日志。



在配置管理器中，我们可以定制输出的日志级别，比如，我们只想输出INFO及以上级别的日志，配置如下图所示。



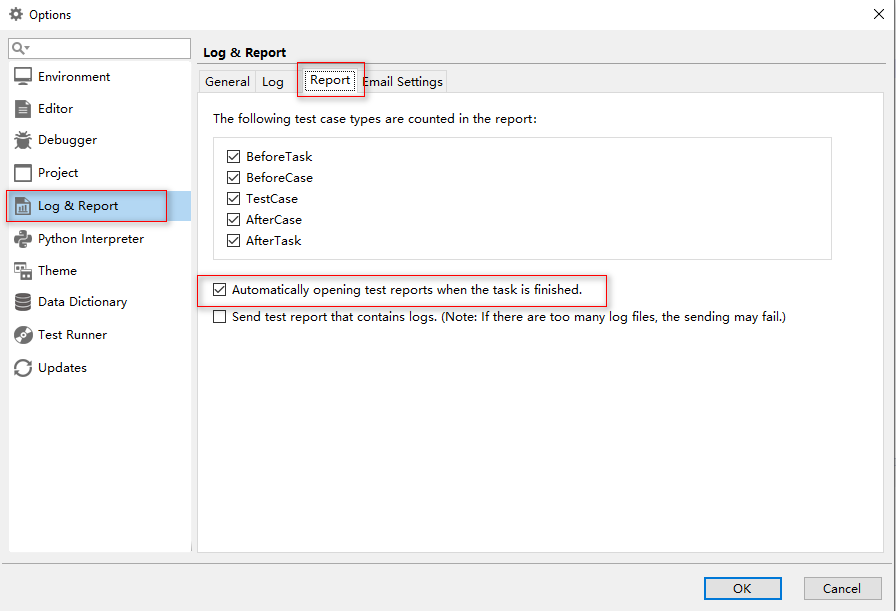
重新运行代码后，输出的日志如下，可以看到已经没有DEBUG级别的日志了。



**报告**

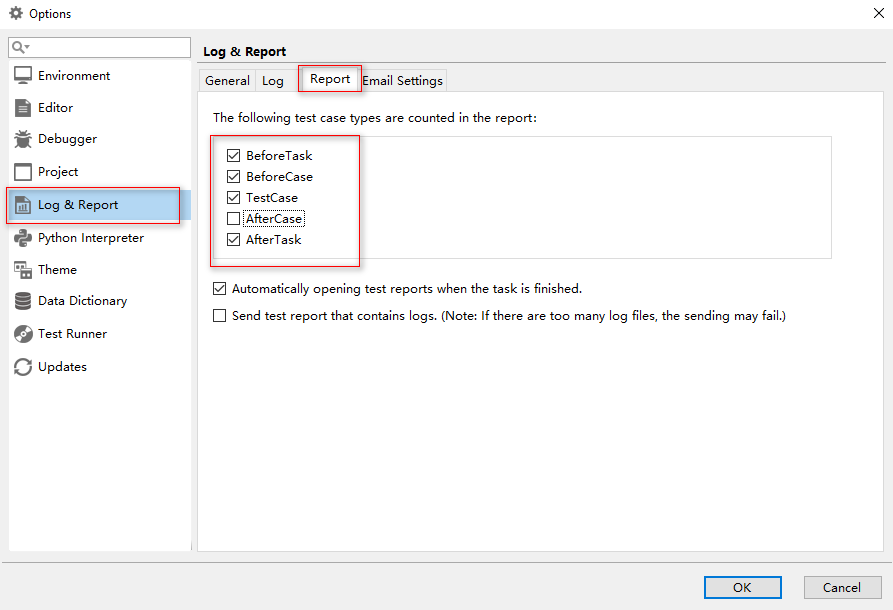
AutoTest Studio以任务为运行单位，每一次运行都是一次任务，即便你只运行一个测试用例。

任务运行完成后，默认会自动打开本次任务的测试报告，当然你也可以设置为不自动打开测试报告，设置方法如下。

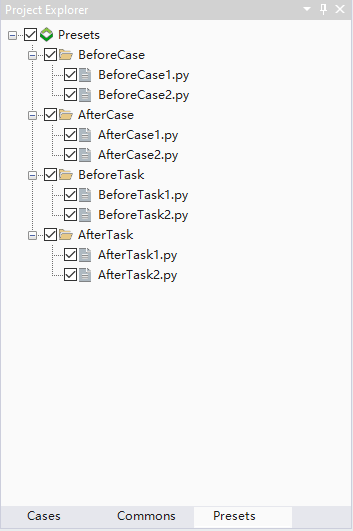


**设置报告统计类型**

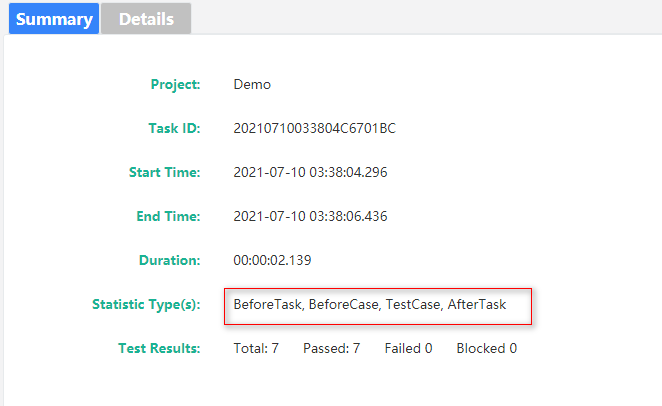
用户可以根据需要决定报告统计的用例类型，但必须在任务运行前设置，设置如下，这里为了对比，我们假定不统计AfterCase这种类型。



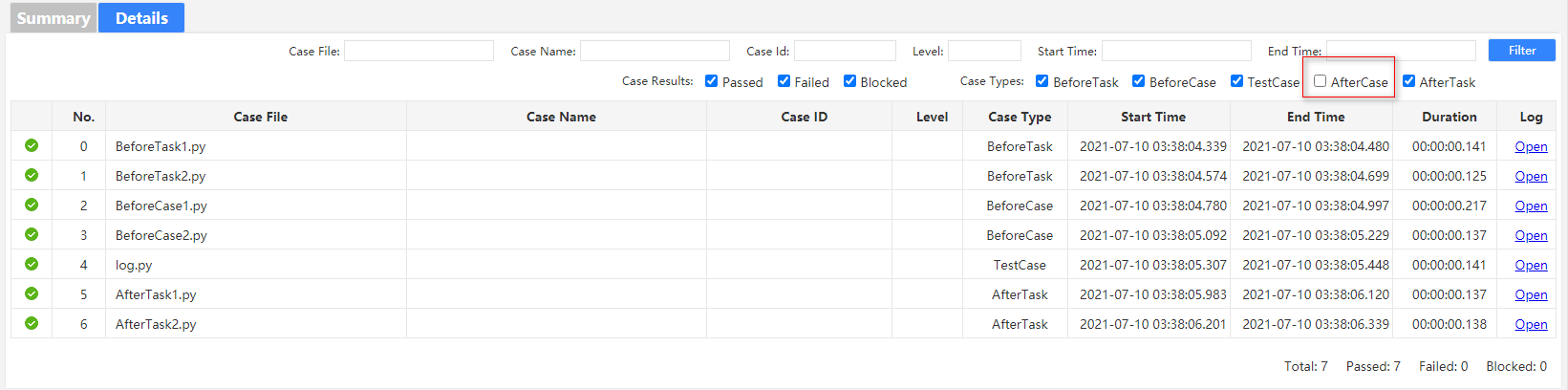
在“Presets”选中预置用例。



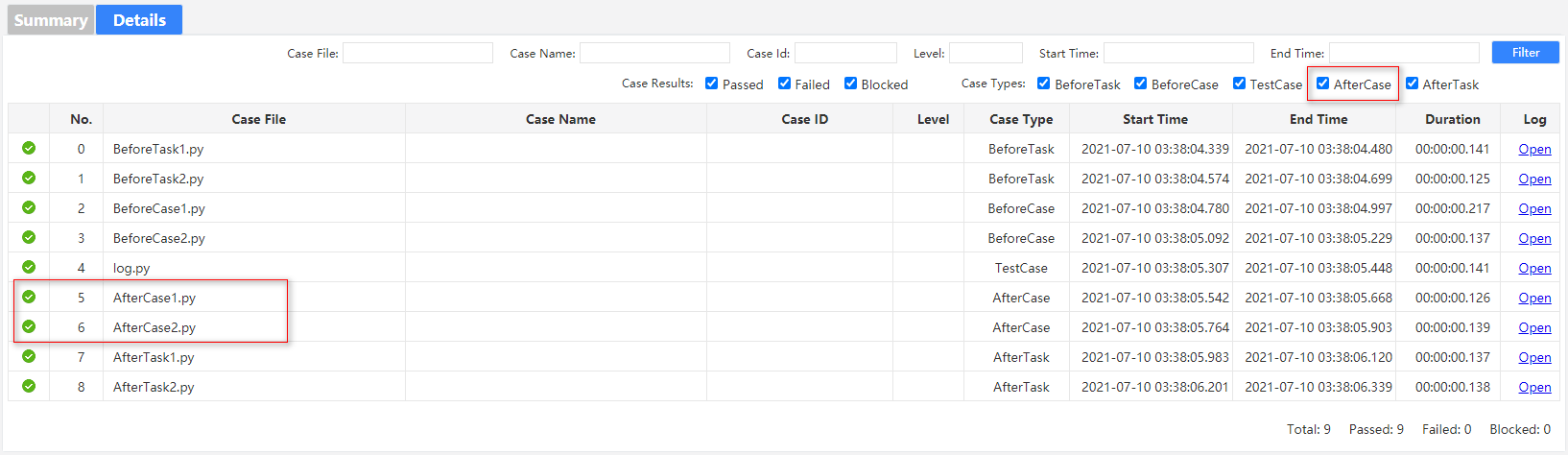
运行一次任务，如下图所示，在统计报告中，已经没有了AfterCase。



在测试日志记录中默认与报告统计一样，不会显示没有统计的类型，如下图所示。

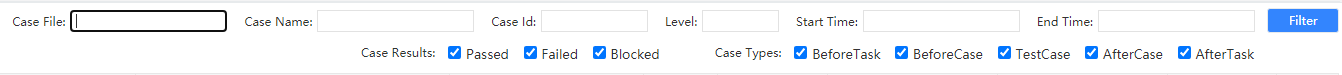


但这并非意味着AfterCase这种类型日志丢失，你可以在过滤中选中没有统计的类型，重新过滤。

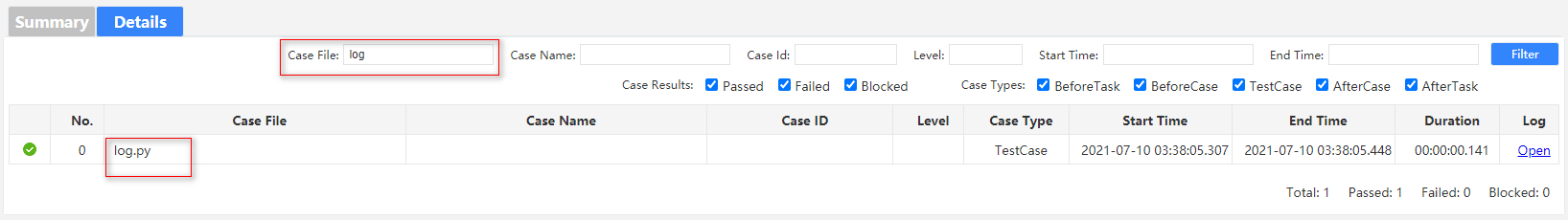


**日志过滤**

在测试报告中，用户可以通过多种条件过滤日志记录，过滤条件如下。

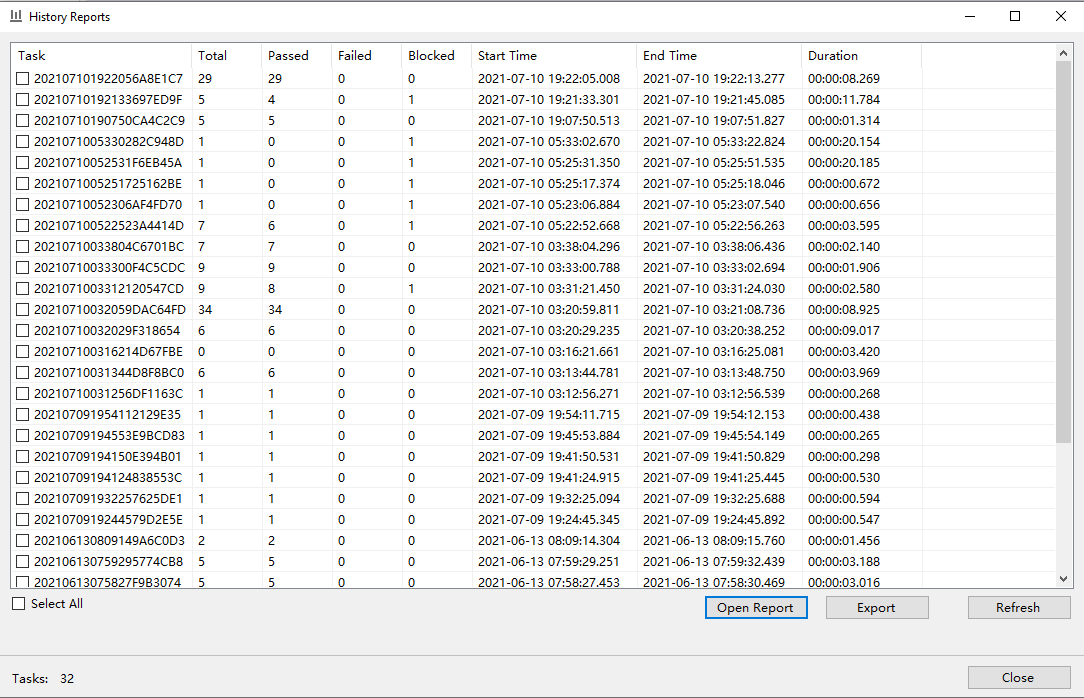


假如我们要过滤CaseFile中包含“log”关键字的用例。



**历史报告**

在工具栏“Tools”->”Reports”可以打开历史报告列表，通过该工具，用户可以打开之前运行的任务报告，也可以通过Export按钮以压缩包形式导出一个或者多个任务报告。



# AutoTest Studio入门系列7：数据字典

AutoTest Studio提供一个性能非常卓越的数据字典模块，与其他测试框架使用的文本方式存储数据不同，AutoTest Stduio使用了Sqlite作为内部存储引擎，Sqlite的性能和稳定性是业界公认的。按照官方的介绍，AutoTest Stduio即便是存储上百万条记录，依然能提供良好的读取性能。

需要指出的是，AutoTest Studio的数据字典在任务运行期间，是只读的，即用户不能通过函数接口向数据字典中写入或者修改数据，只能通过Data Dictionary工具预先写入或者修改数据，这种设计保证任务运行期间不污染测试数据。

AutoTest Studio的数据字典以目录结构组织数据层次关系，这种设计使得数据字典组织关系具有良好的可读性，在数据字典中，有两个概念，数据集(Dataset)和数据项（Dataitem）。

**Dataset**，同一类数据的集合，如一个测试用户的基本信息，Dataset在全局是唯一的，即同一个数据字典中，数据集的名称不能重复，即便是不同的目录下也不行。

**Dataitem**，数据集下的一条数据记录，即一个key-value记录。

在项目创建的时候，AutoTest Studio默认为每个项目创建一个空的数据字典。

数据字典模块提供两个操作接口。

读取整个数据集，返回一个python字典结构。

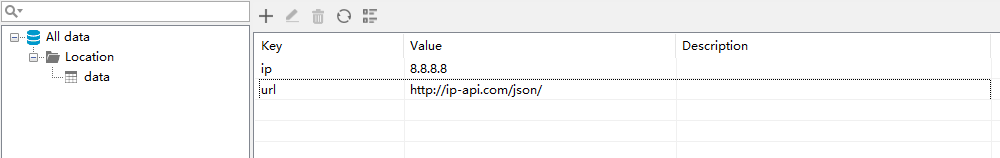
*GetDataset (dataset)*

读取数据集中key对应的value。

*GetDataValue(dataset, key)*

我们以前面章节的demo为例，将demo中直接定义ip和ipQueryUrl两个变量的值存放到数据字典中，然后通过数据字典接口来读取这两个变量的值。

首先，我们先建立一个目录（Location），再建立一个数据集(data),在“data”下新建两个数据项ip，ipQueryUrl，如下图所示。



修改原来的代码代码。

原来的代码：

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

ip="8.8.8.8"

ipQueryUrl="http://ip-api.com/json/{0}".format(ip)

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

修改后的代码：

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

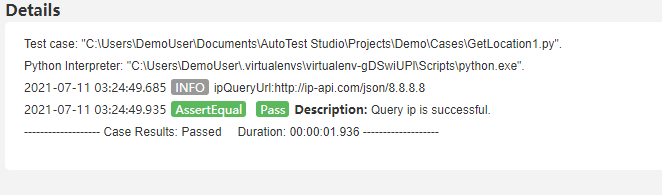
ipQueryUrl="{0}{1}".format(GetDataValue("data","url"),GetDataValue("data","ip"))

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

运行结果如下，从日志看出，与原来的代码运行结果一致。



再次修改代码，这次，我们将整个数据集读取出来。

import requests

from autotest import \*

#set test case information

SetCase("TEST-1","Get location by ip","1")

data=GetDataset("data")

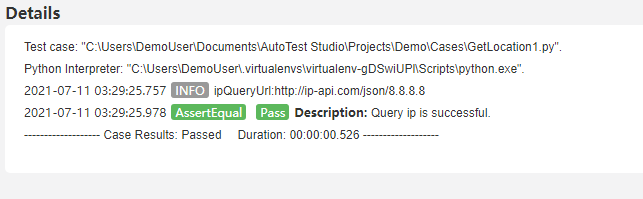
ipQueryUrl="{0}{1}".format(data["url"],data["ip"])

LogInfo("ipQueryUrl:{0}".format(ipQueryUrl))

response=requests.get(ipQueryUrl)

AssertEqual(response.status\_code,200,"Query ip is successful.")

运行结果：



从上面的demo可以看出，数据字典的使用非常简单。

# AutoTest Studio入门系列8：代码调试

AutoTest Studio内置一个强大的代码调试引擎，这在测试用例开发过程中非常有用。

代码调试方式多种多样，但总的来说，常用的就三个步骤，断点，监视变量，修改变量。本节以一个简单的demo为例介绍AutoTest Studio调试功能的使用。

新建一个名为“debugtest.py”的测试用例，输入以下内容。

from autotest import \*

a=1

b=2

c=3

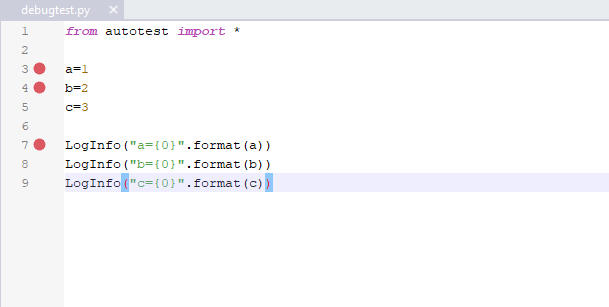
LogInfo("a={0}".format(a))

LogInfo("b={0}".format(b))

LogInfo("c={0}".format(c))

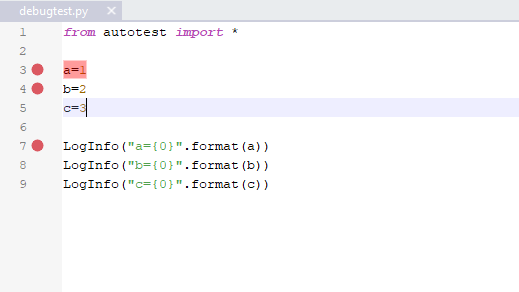
**断点**

在代码运行前，我们设置一些断点。



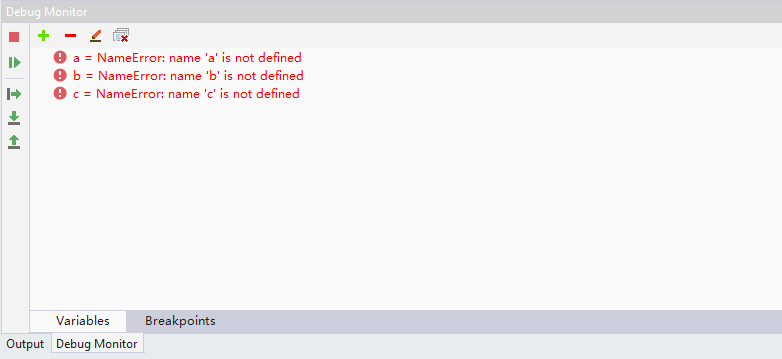
**运行代码**

以“Debug Run”方式运行代码到第一个断点处。

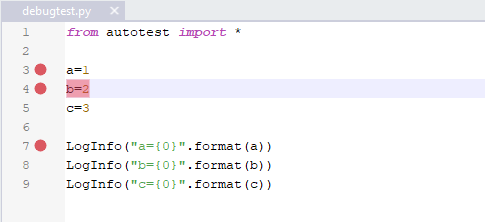


**监视变量**

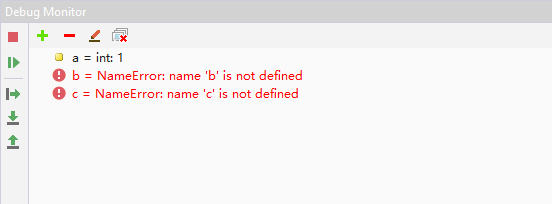
运行代码到第一个断点处时，我们新建三个监视变量分别为a，b，c，当然你也可以在代码运行前新建监视变量。



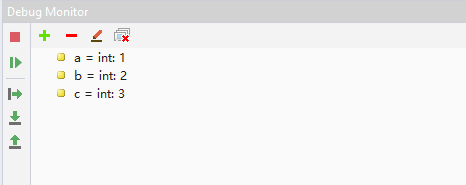
新建变量后，调试器会立即获取当前变量的值，但本示例中此时提示了“NameError”错误，这是因为代码还没有运行对a，b，c赋值操作。这里继续运行代码到第二个断点处。



在Debug Monitor中可以看到a已经有正确的值。

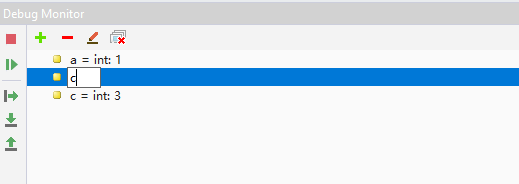


继续运行代码至下一个断点处，a，b，c完成赋值操作，Debug Monitor可以看到a，b，c的值。

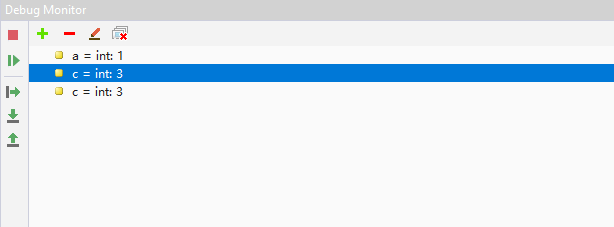


**修改变量**

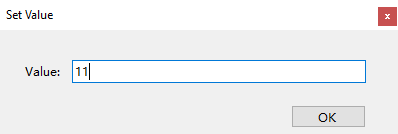
调试器允许用户在代码运行期间修改监视变量的名称和值，接着上面的示例，点击右键菜单“Edit Name”，在编辑状态下，将b的名称修改为c。



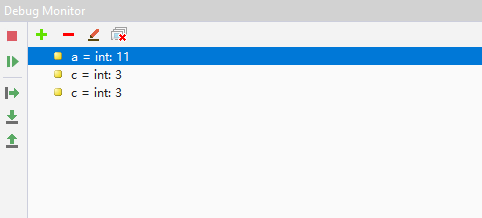
修改完成后，鼠标双击空白区域，让修改生效，生效后，调试器会立即获取新变量的值。



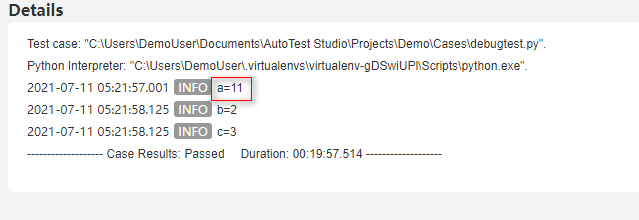
除了修改变量名称外，调试引擎也允许用户修改变量的值，选中a变量，点击右键菜单“Set Value”，设置a变量的值为“11”。



点击OK后，Debug Monitor显示，a的值已经变成“11“了。



为了验证修改的变量值是否生效，我们将代码运行至结束，查看日志打印。



从日志中可以看到，代码执行到“LogInfo("a={0}".format(a)) ”时已经是修改后的值了。

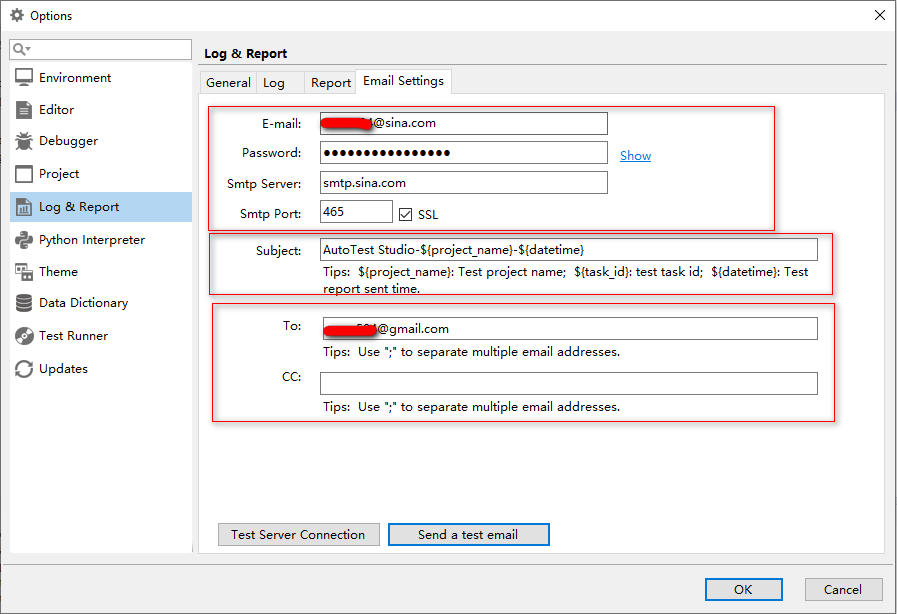
代码调试是开发过程中非常重要的一项技能，本文只是简单介绍了AutoTest Studio调试功能的基本使用，更复杂的使用技巧，大家可以在使用过程中自行探索。

# AutoTest Studio入门系列9：发送测试报告

AutoTest Studio内置了一个邮件发送模块，可以在任务运行结束后，自动将任务的测试报告发送到指定的邮箱。

**配置邮件参数**

配置分三个部分，如下图所示，从上到下，依次为发送者的配置，中间部分为邮件正文标题格式，下面部分为接收者邮箱地址，可以多个接收地址。



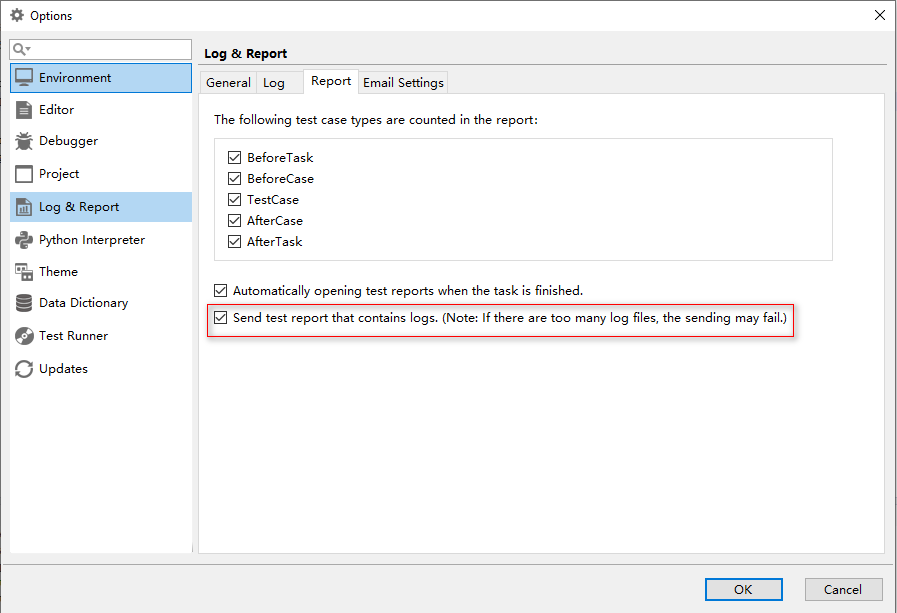
**测试配置**

当我们完成邮箱的参数配置后，可以点击“Send a test email”来检查配置信息是否正确，如上图所示的配置，我们打开接收的邮箱，可以收到一封主题为“AutoTest Studio Mail Test”的邮件。



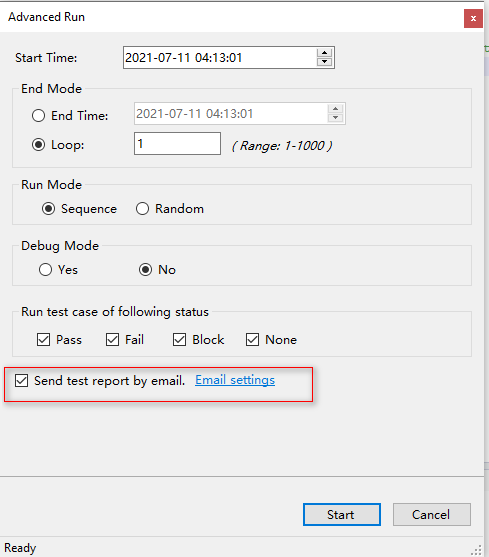
**携带日志**

AutoTest Studio发送的报告，默认是不携带日志文件，因为日志文件往往比较大，可能导致发送失败，如果你的邮箱支持较大的附件，你可以设置邮件携带日志文件。设置如下。

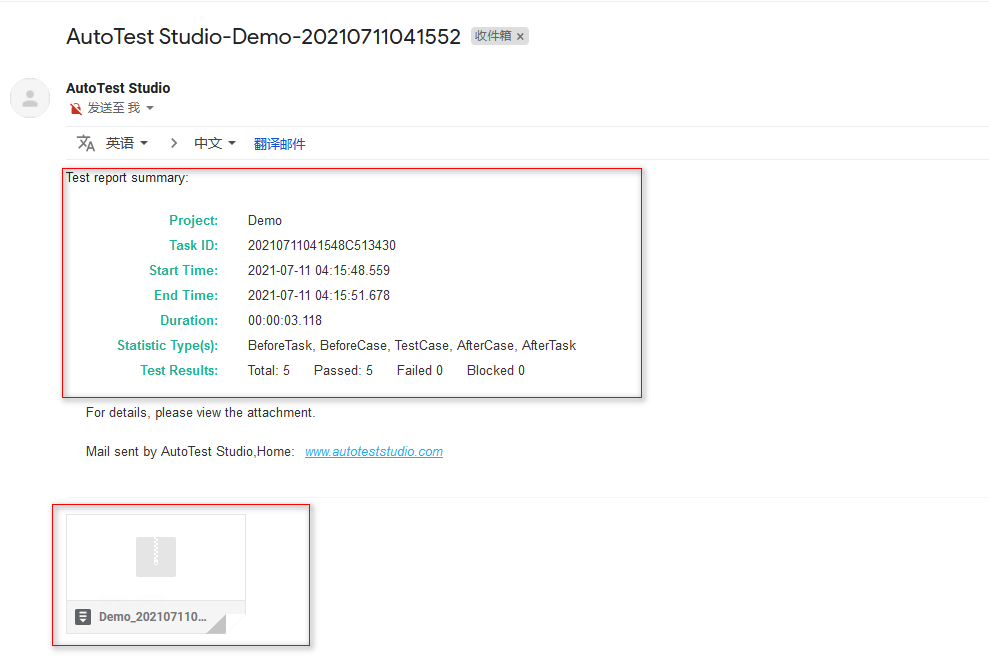


**任务结束后发送报告**

发送测试报告功能只有在“Advanced Run”方式运行时才能使用，如下图所示，勾选上“Send test report by email”，点击“Start”运行任务。



任务运行完成后，我们检查接收邮箱，可以看到收到了一封主题为“AutoTest Studio-Demo-20210711041552”的邮件，附件即为测试报告和日志的压缩包，邮件正文如下。



# AutoTest Studio入门系列10：Jenkins集成AutoTest Studio

随着敏捷开发的盛行，在很多的项目中都有着自己的CI环境，本节我们将介绍如何实现在Jenkins集成AutoTest Studio。在阅读本节内容前，假定你已经对Jenkins有一定的了解，并且本文的内容也仅仅是围绕Jenkins和AutoTest Studio集成，对一些基本操作不会做过多描述。

Jdk和Jenkins安装包：

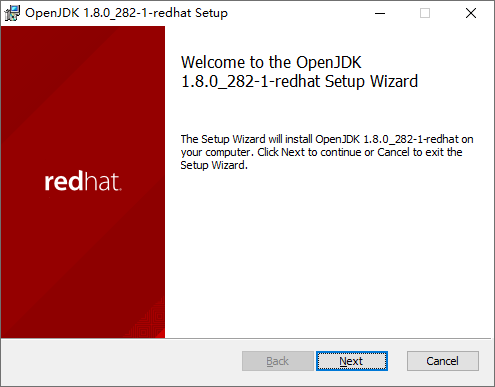
java-1.8.0-openjdk-1.8.0.282-1.b08.dev.redhat.windows.x86\_64.msi

Jenkins.war

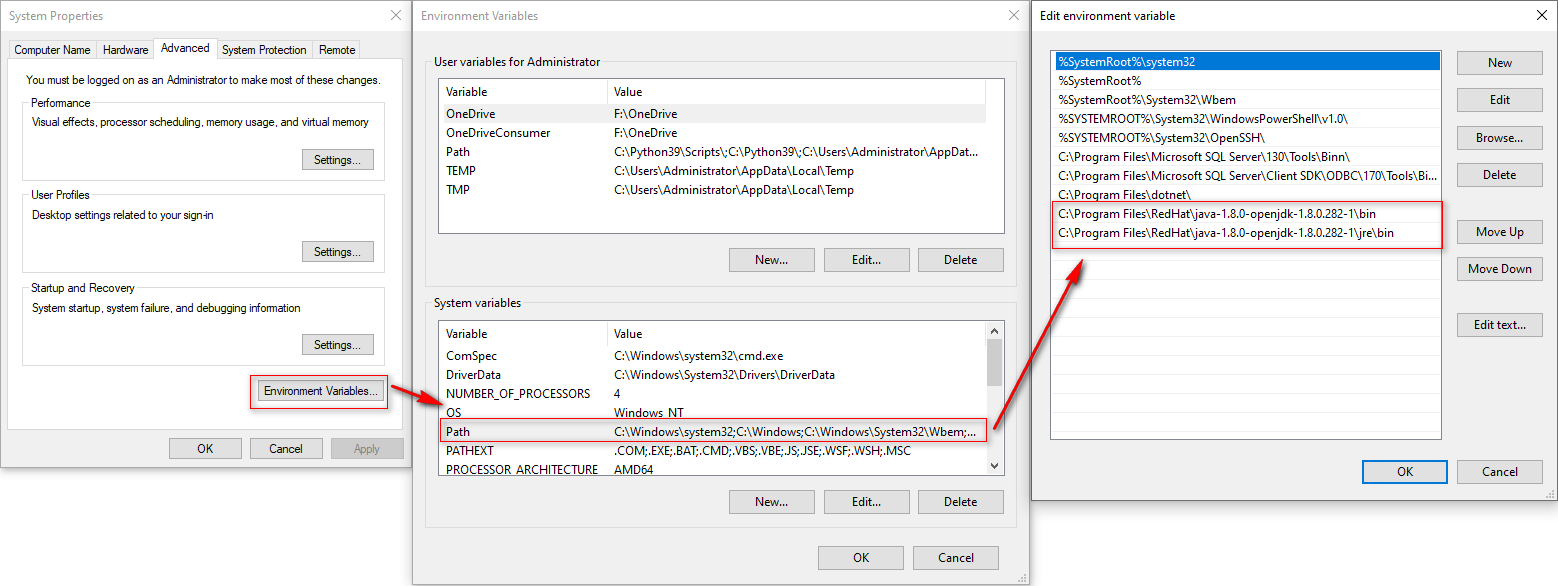
**注意**:jdk可以选用其它的jdk版本，但在本文的demo中Jenkins必须使用war包以命令行方式启动，因为AutoTest Studio是桌面应用程序，需要桌面环境中运行，如果Jenkins以服务形式安装，将无法正常运行AutoTest Studio。

**安装jdk**

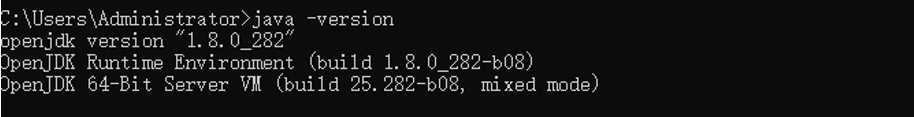
Jdk的安装非常简单，一直点击“Next”按钮即可。



安装完成后，检查系统环境变量，本文中使用jdk安装包会自动配置环境变量，如果其他jdk的版本没有配置环境变量，需要手工配置。

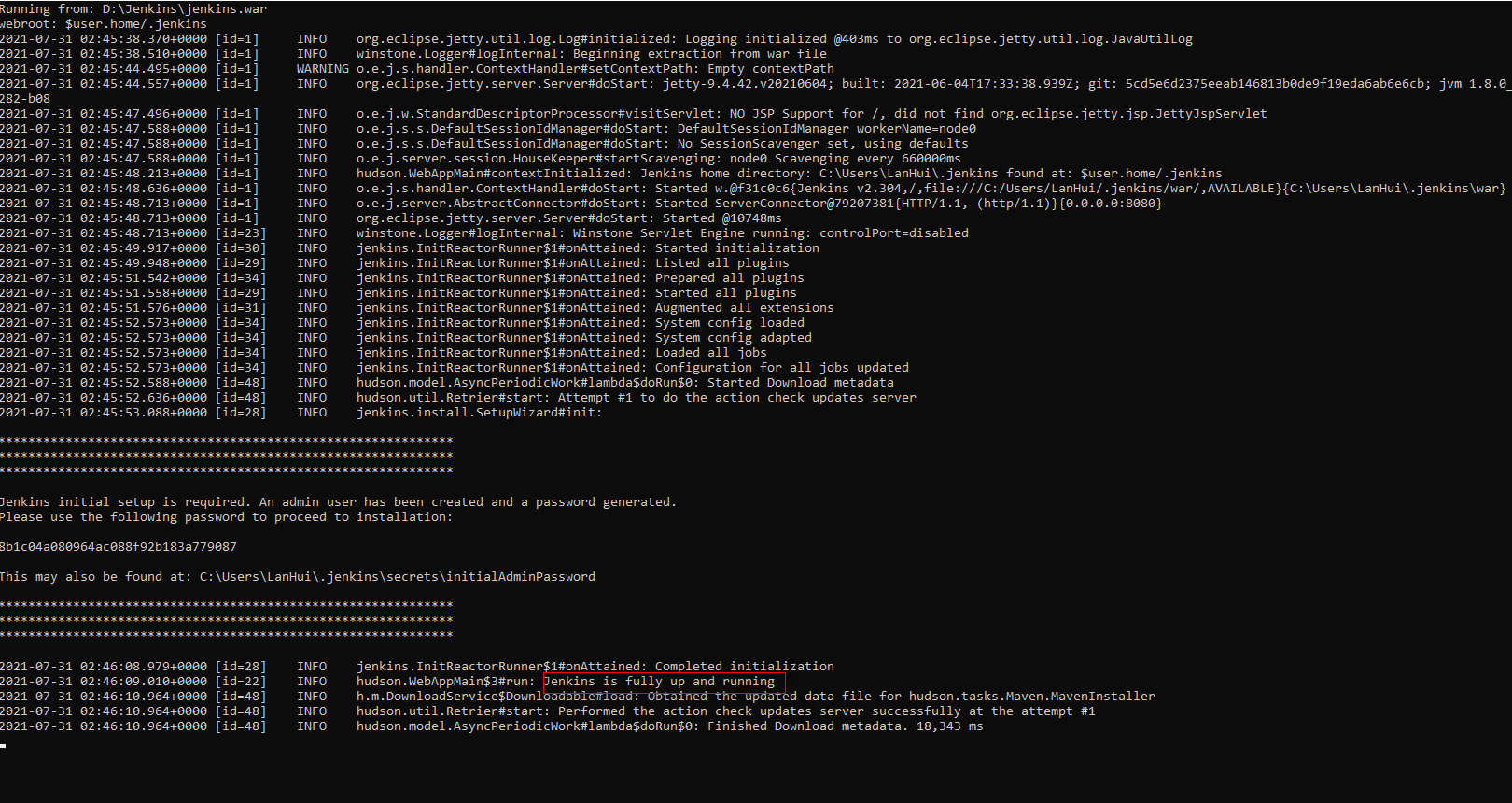


在cmd中运行“java -version”返回如下版本信息，说明jdk安装正常，并且已经配置好环境变量。

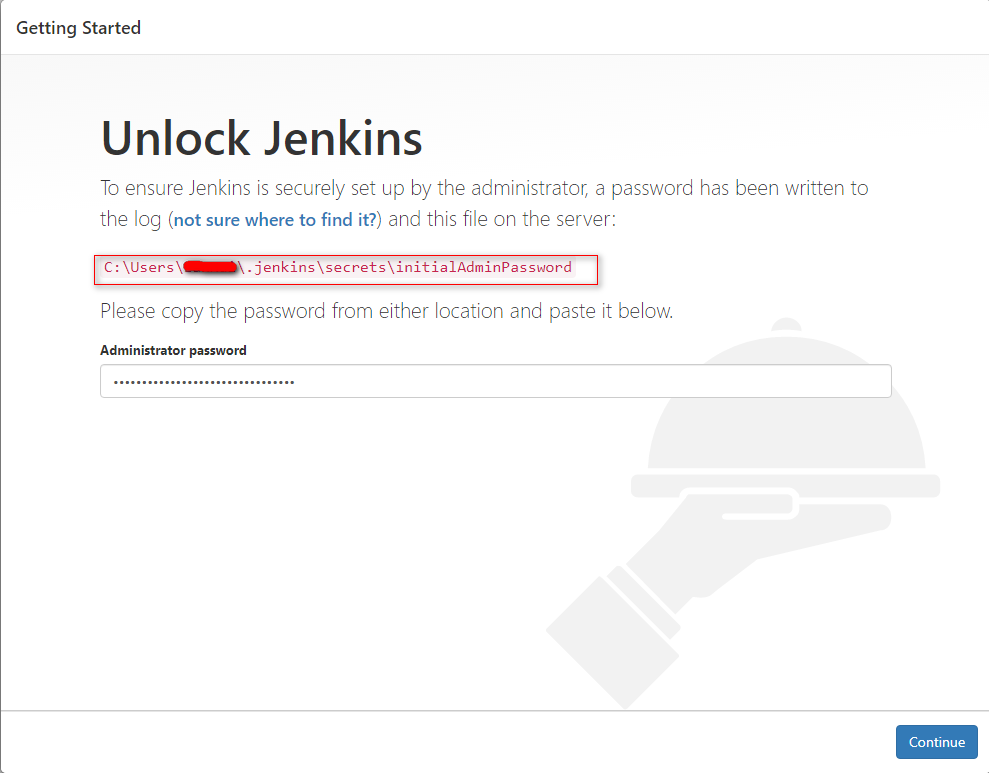


**启动Jenkins**

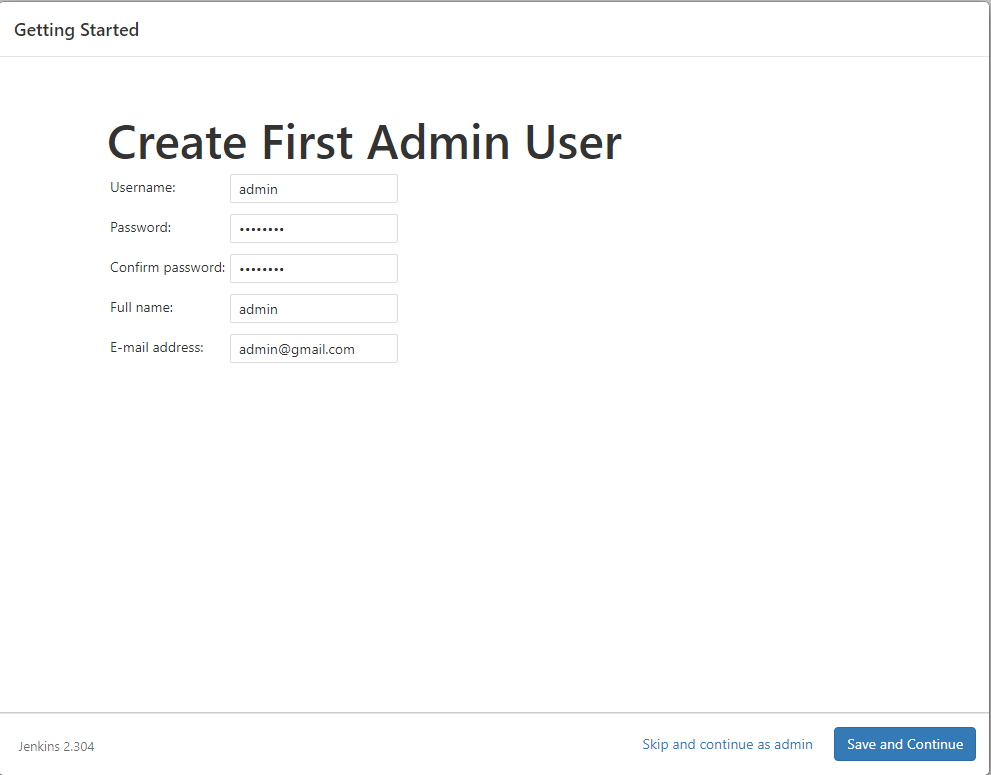
通过命令行方式启动Jenkins，启动命令“*java -jar "D:\Jenkins\Jenkins.war*"”，当启动信息中出现“Jenkins is fully up and running.”说明Jenkins已经正常启动。



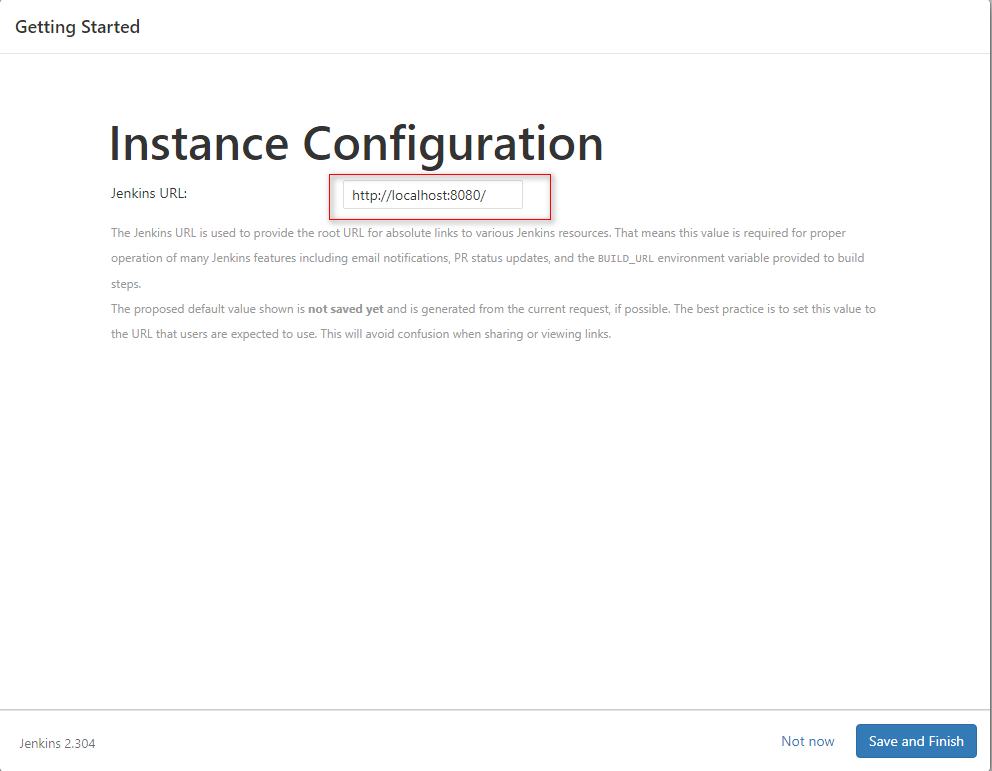
使用浏览器登陆“http://localhost:8080”。Administart password在提示的文件中，如下图所示。



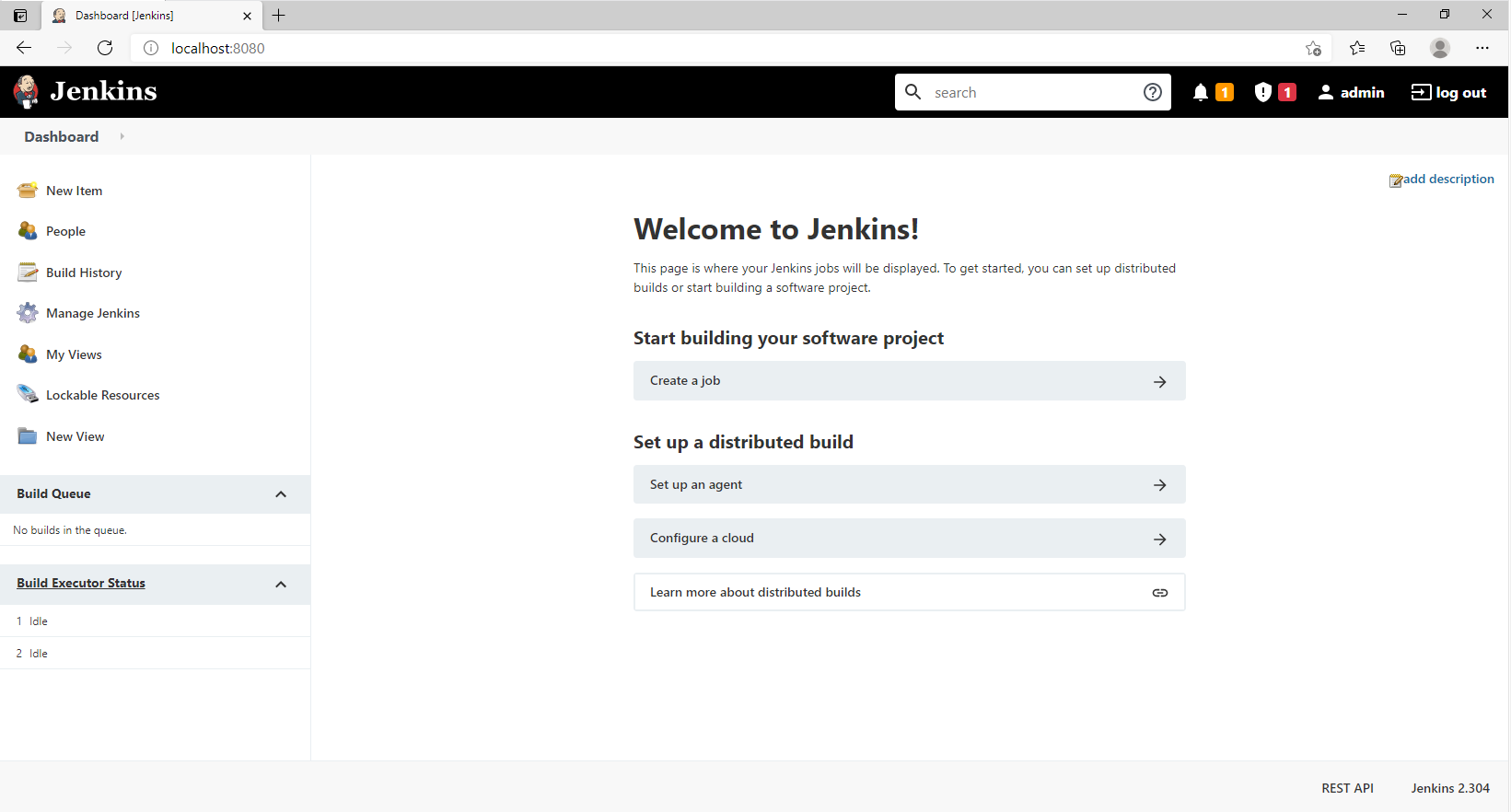
首次使用Jenkins时，需要安装默认的插件，按照提示安装即可。安装完成后，需要创建一个用户。



用户创建完成后，Jenkins会要配置登录ip地址和端口，这里我们使用默认的URL,即“http://localhost:8080/”；

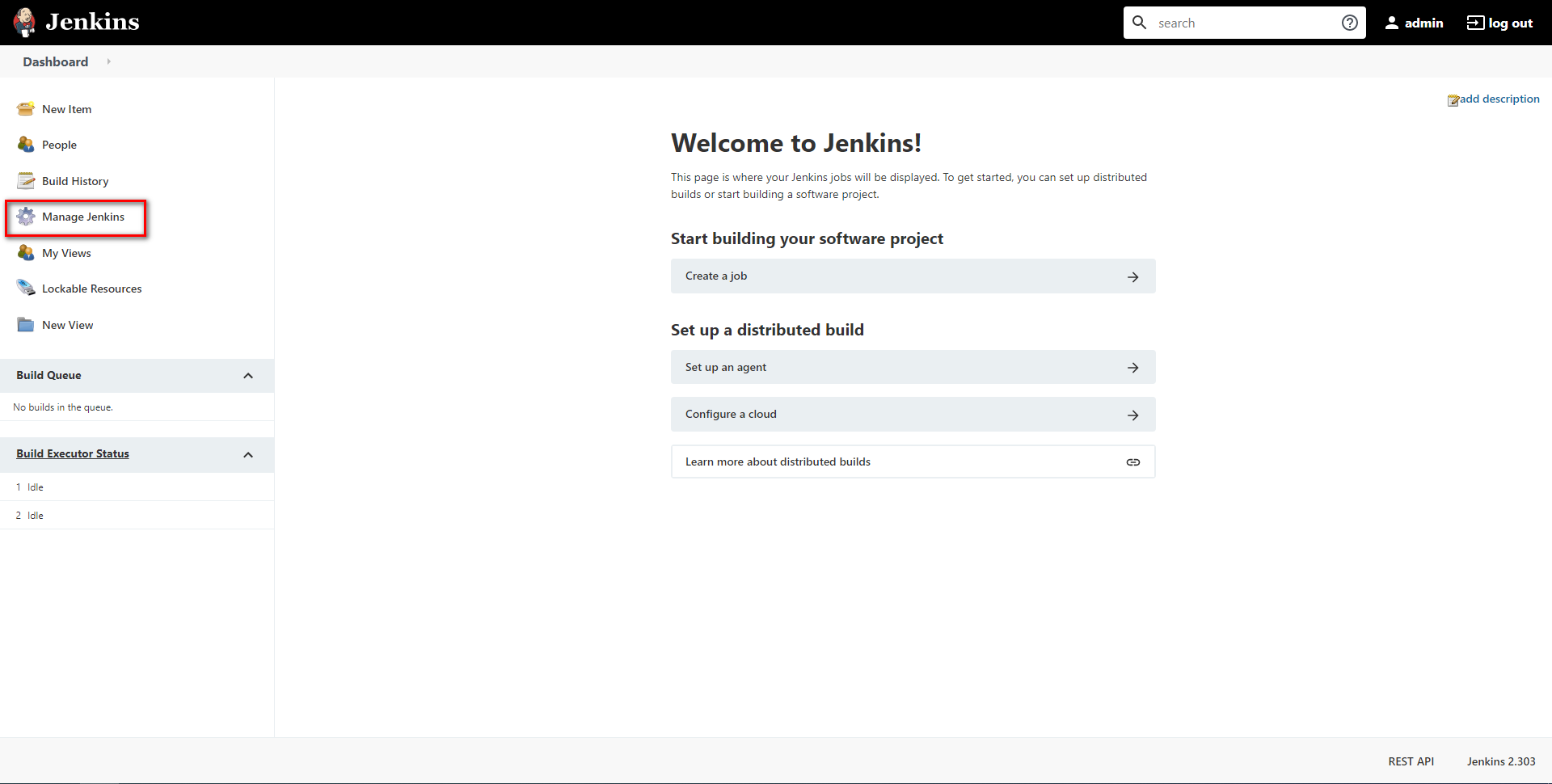


Jenkins基本配置完成以后，进入到Dashboard界面，如下图所示。

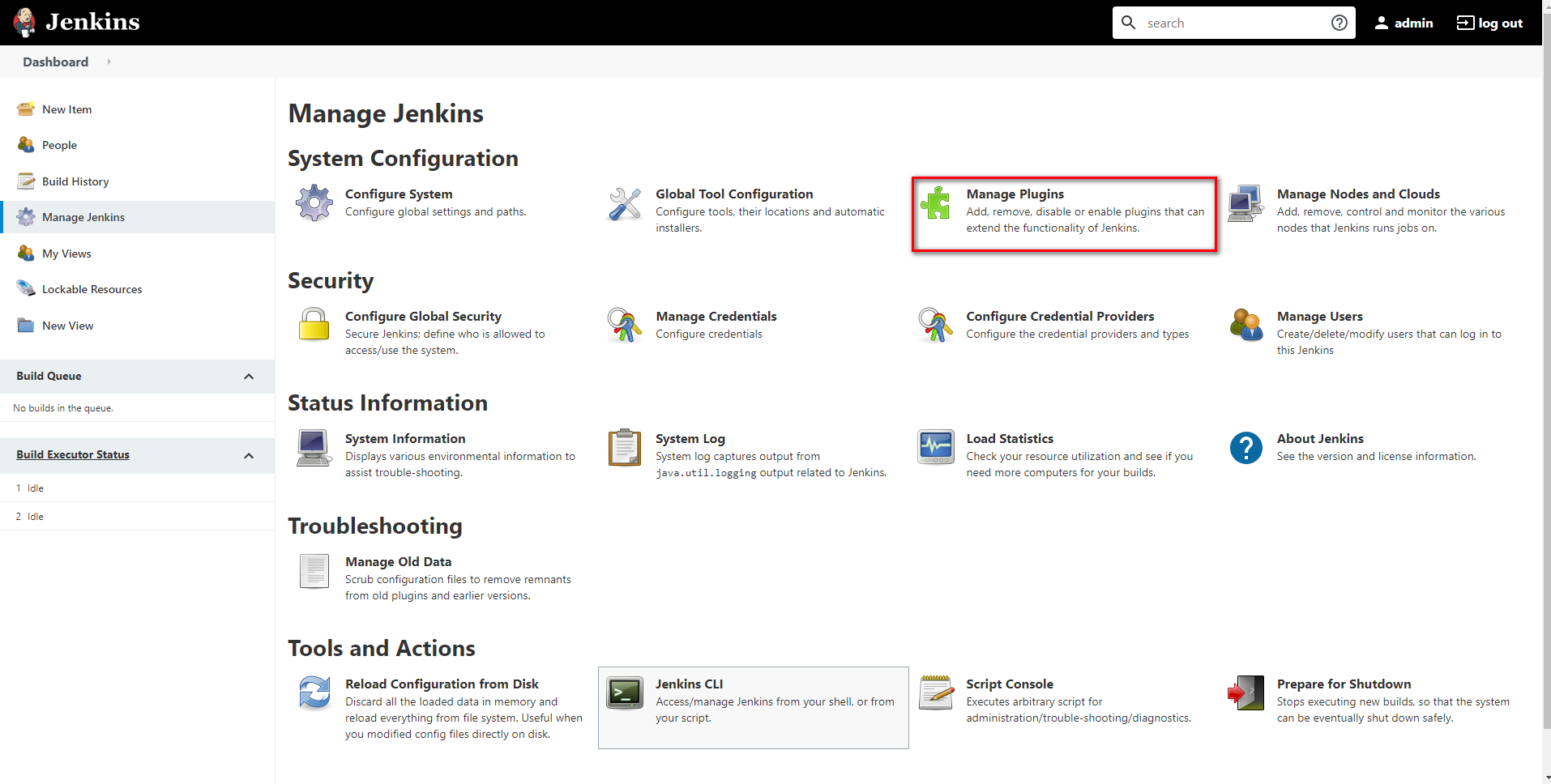


**安装HTML Publisher插件**

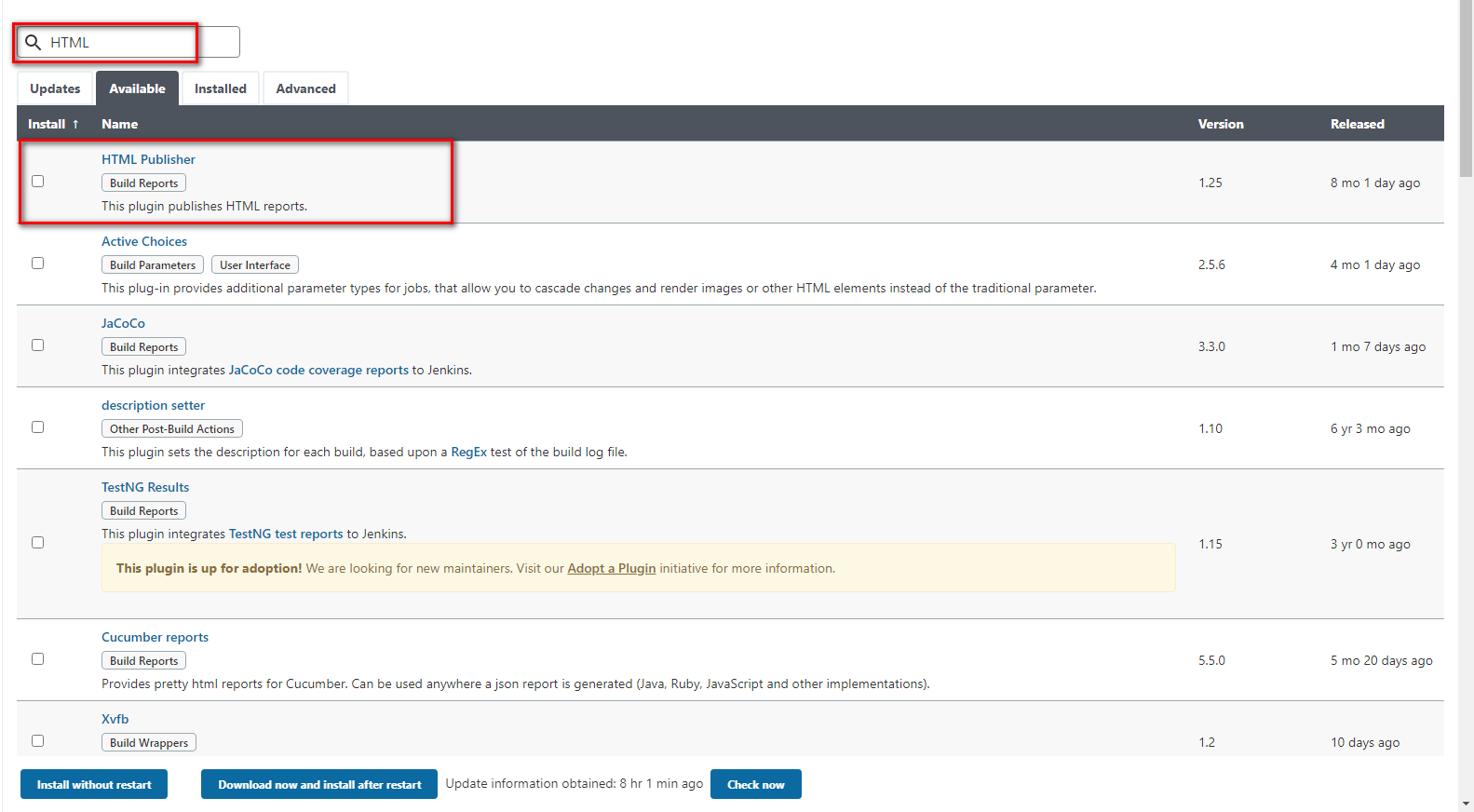
AutoTest Studio使用“HTML Publisher”插件在Jenkins中发布测试报告，因此我们需要安装HTML Publisher插件。在Jenkins的Dashboard界面中依次点击“Manage Jenkins”->“Manage Plugins”，进入插件管理模块。



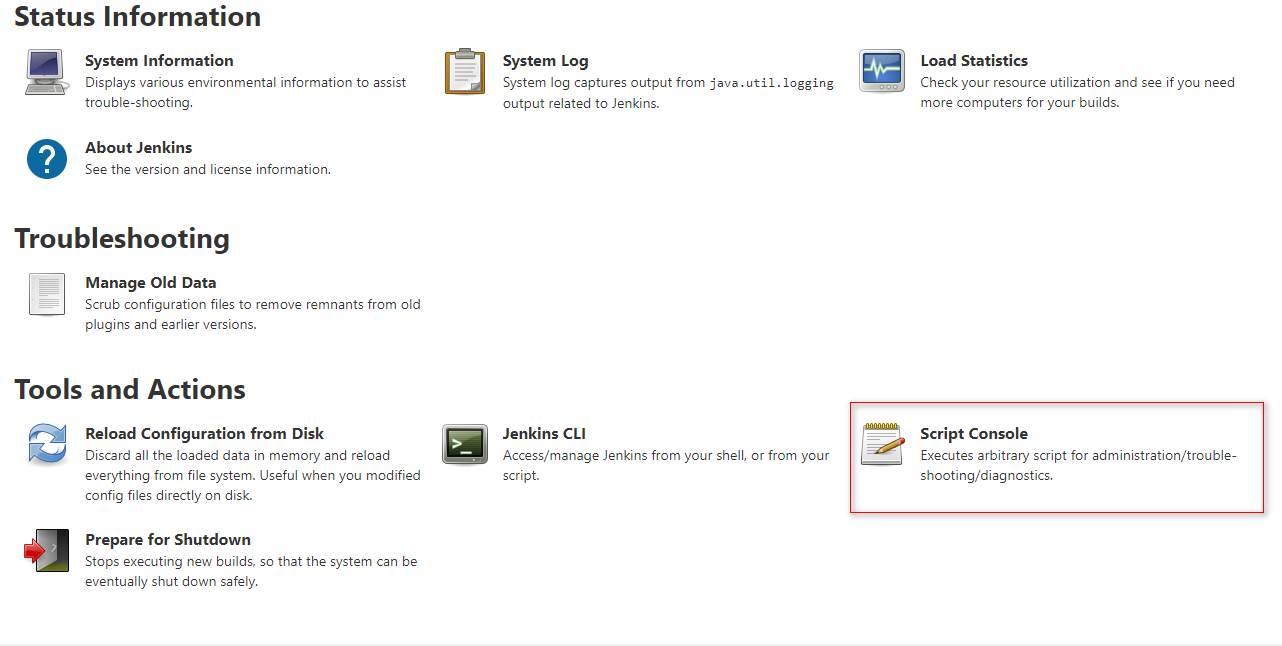
选择“Manage Plugins”



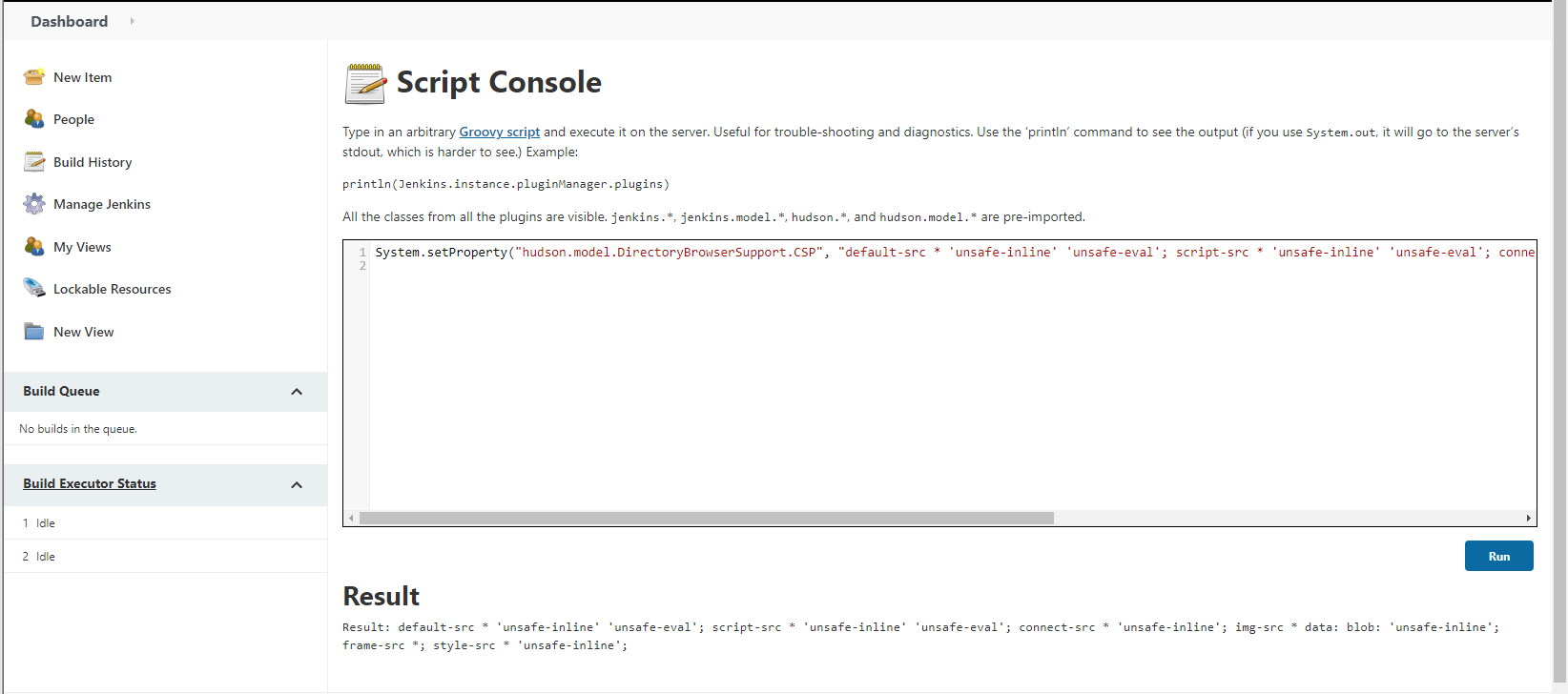
在搜索栏中输出“HTML”后，在搜索结果的列表中选中“HTML Publisher”插件，点击底部“Install without restart”按钮.



安装HTML Publisher插件完成以后，由于Jenkins出于安全考虑，禁止运行Html文件中js和css运行，因此我们还需要修改Jenkins中CSP安全机制，在Dashboard中依次点击“Manage Jenkins”->“Script console”，进入“Script console”。

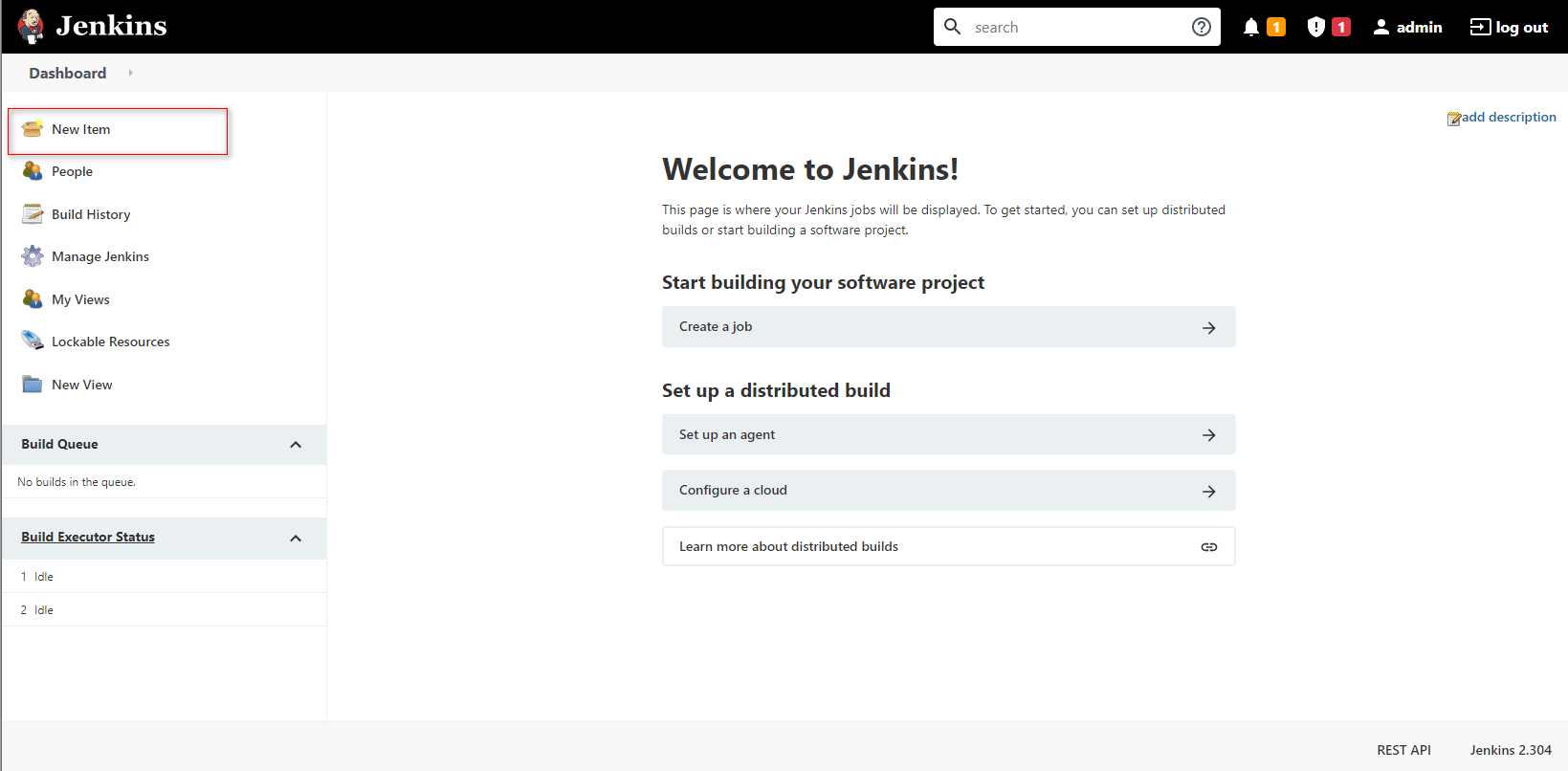


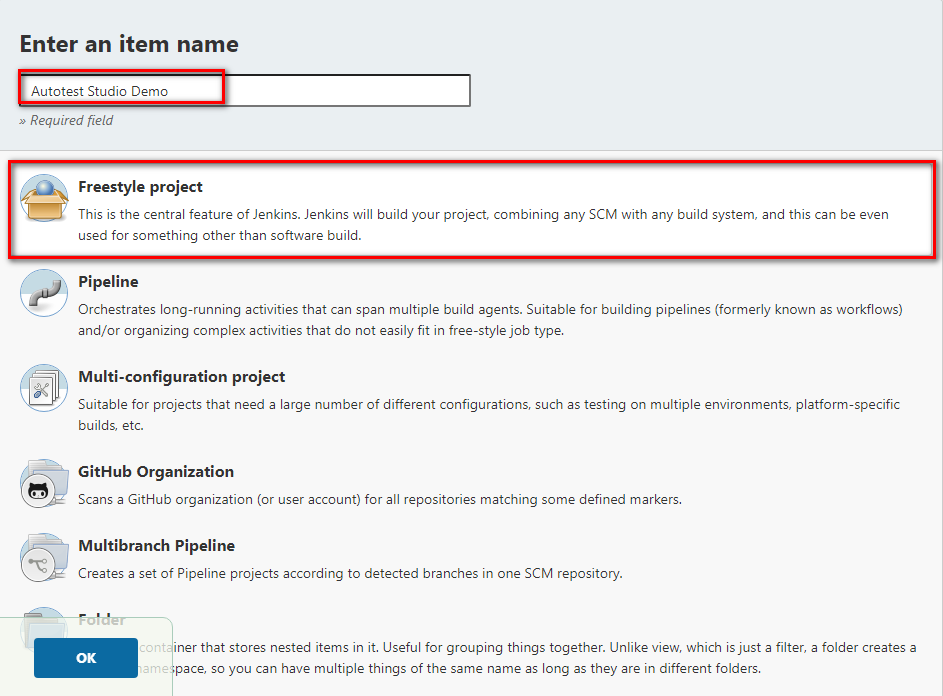
运行代码 *System.setProperty("hudson.model.DirectoryBrowserSupport.CSP", "default-src \* 'unsafe-inline' 'unsafe-eval'; script-src \* 'unsafe-inline' 'unsafe-eval'; connect-src \* 'unsafe-inline'; img-src \* data: blob: 'unsafe-inline'; frame-src \*; style-src \* 'unsafe-inline';")* ，运行结果如下图所示。

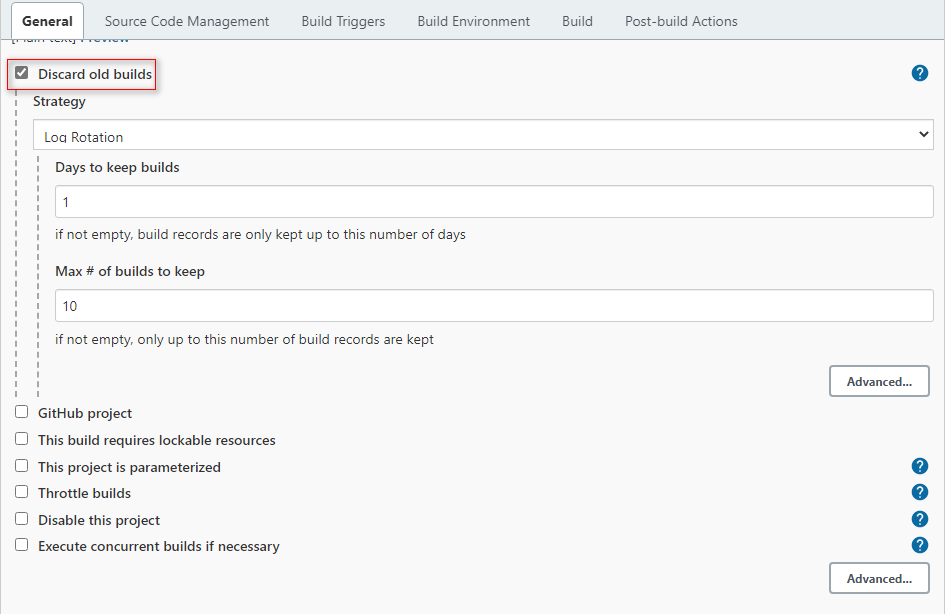


**新建一个Item**

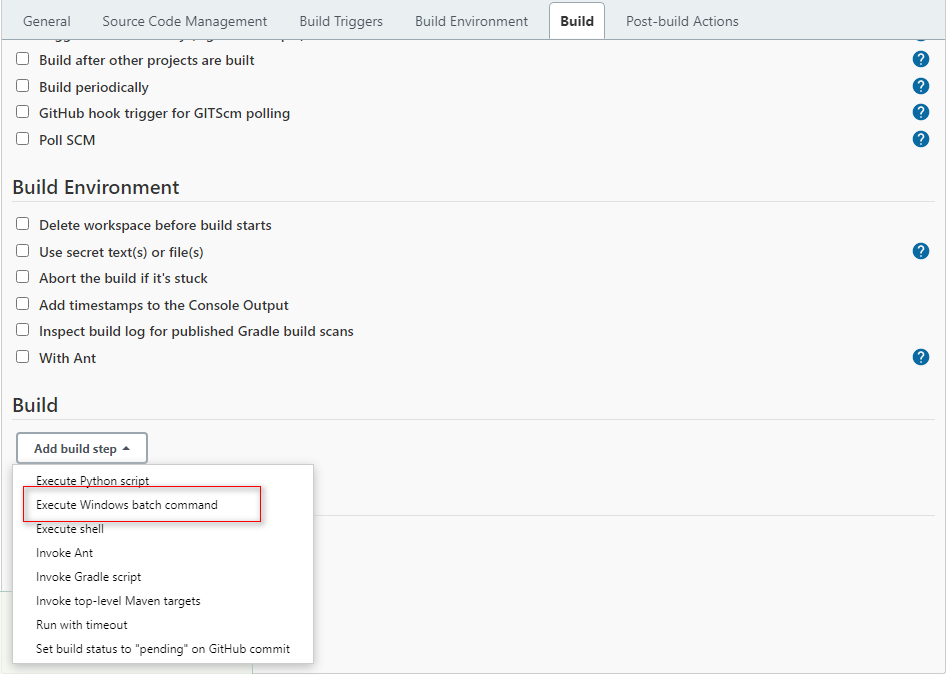
创建一个新的Item，名称为“AutoTest Stdio Demo”，风格为“Freestyle project”。







在Build中选择“Execute Windows batch command”添加AutoTest Studio调用命令。



调用命令*"C:\Program Files\AutoTest Studio\AutoTest Studio.exe" "C:\Users\Demouser\Documents\AutoTest Studio\Projects\ats\_example\ats\_example.tpro" -r -q* .

命令含义：

**"C:\Program Files\AutoTest Studio\AutoTest Studio.exe"**，AutoTest Studio.exe路径。

**"C:\Users\Demouser\Documents\AutoTest Studio\Projects\ats\_example\ats\_example.tpro"**，项目路径，在任务运行前需要在项目中选择要执行的脚本，AutoTest Studio会自动保存到项目文件中。

**-r**，启动后立即运行任务。必选参数。

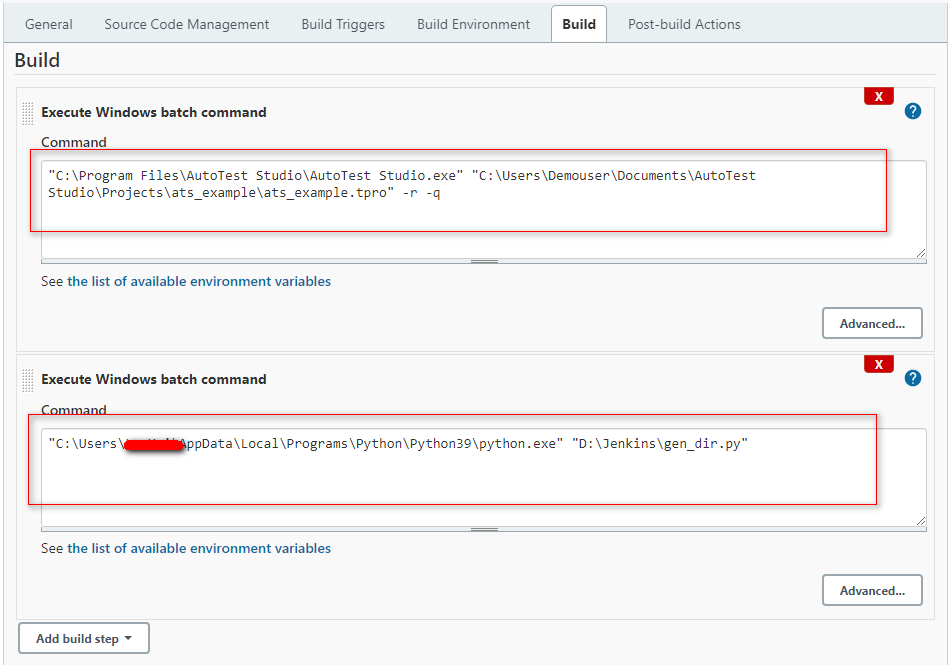
**-q**，任务运行完成后，退出AutoTest Studio程序。必选参数。

由于AutoTest Studio的报告和日志目录是动态产生的，因此我们需要通过一个脚本将任务输出文件“output.json”中日志目录解析出来，并映射到workspace中。

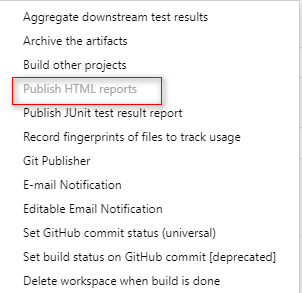
将以下python代码保存到一个文件中，如：“D:\Jenkins\jen\_dir.py”，但脚本位置没有限制。

|  |
| --- |
| import json import os   file=".\\output.json" target\_report\_dir="report\_logs" if not os.path.exists(file):  print("\"output.json\" does not exist.")  exit(-1)  fd=open(file,"r") log\_json=json.loads(fd.read()) fd.close() log\_dir=log\_json["LogDirectory"]  cmd='rmdir /s /q "{0}" & mklink /j "{0}" "{1}"'.format(target\_report\_dir,log\_dir) os.system(cmd) |

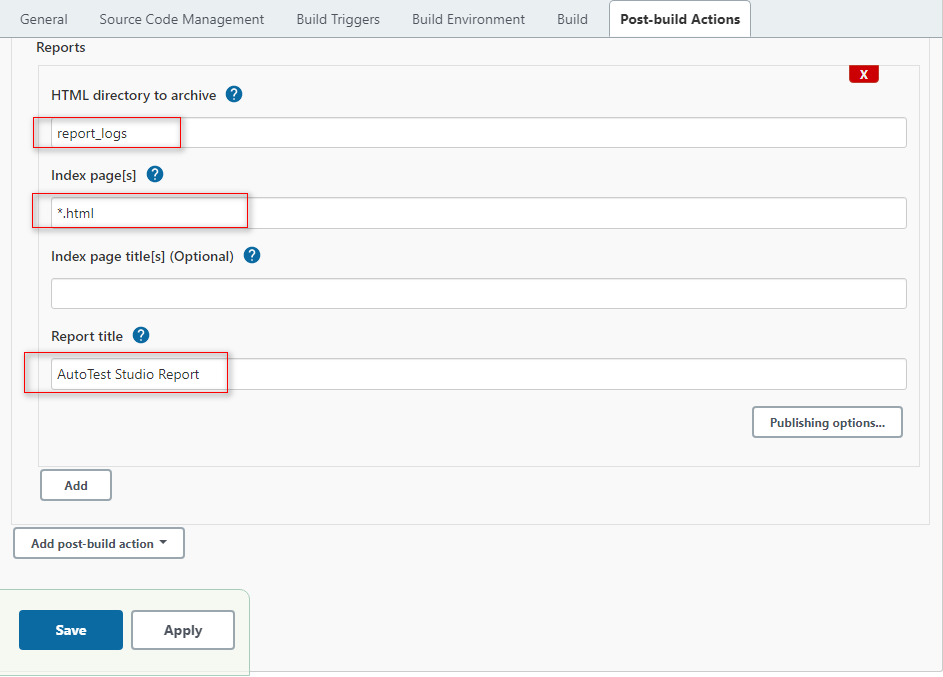
这两条构建命令如下图所示。



在“Post-build Actions”中通过“Publish HTML reposts”插件将AutoTest Studio的测试报告发到布到Jenkins中。



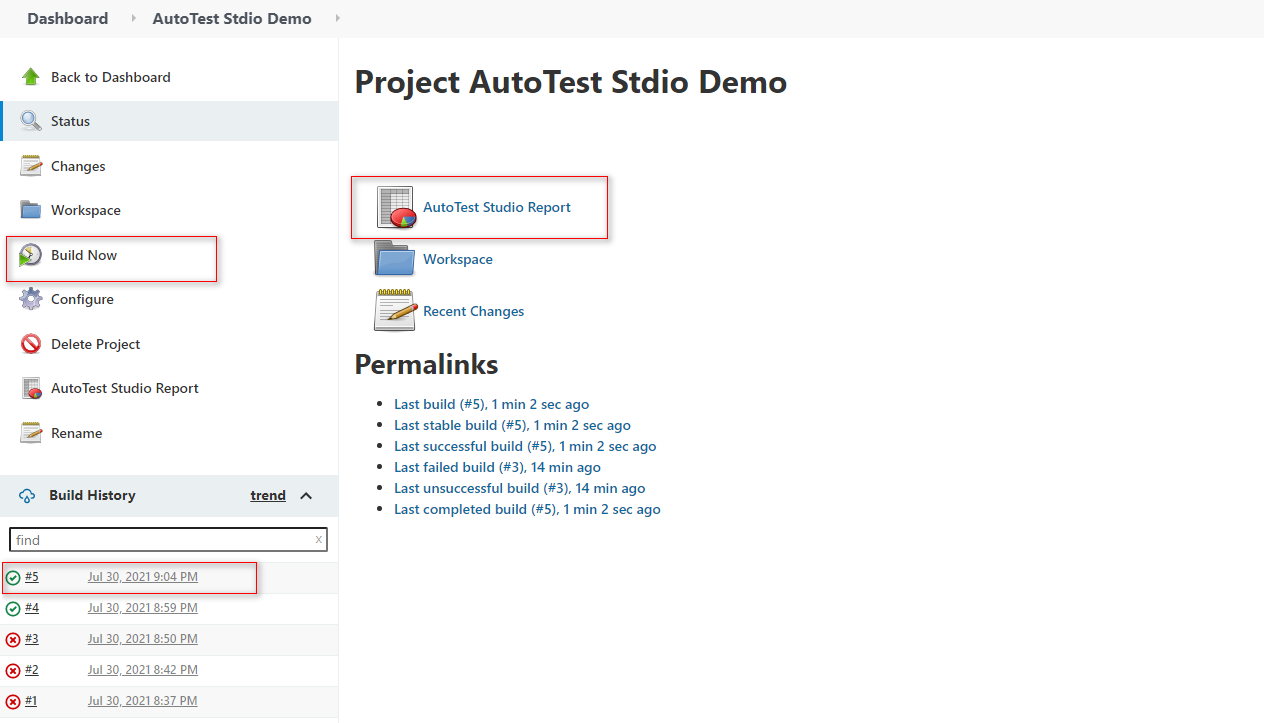
“Publish HTML reposts”插件配置参数如下，注意目录“report\_logs”必须上面脚本中命令保持一致。



配置完成，点击底部的“Save”按钮，保存配置。

**执行构建任务**

在Dashboard中点击“Bulid New”执行一个构建操作。



构建操作完成后，我们点击“AutoTest Studio Report”打开测试报告，可以看到如下图所示的内容，这个页面是由“Publish HTML report”插件抓取的，在页面的右上角点击“Zip”可以下载本次任务运行的测试报告和日志。

