目 录 (五十四期·上)

华山论剑，Web 性能测试工具谁与争锋.....01

机器学习与数据挖掘十大经典算法之逻辑回归算法.....13

一种基于 Tesseract 识别验证码实践.....20

简述实施自动化测试的过程.....28

开启 iOS Appium 自动化测试之门.....30

如何保证提测质量?.....40

Python 接口测试实战.....43

自动化数据稳定性方案.....54

测试 10 秒钟，分析 8 小时，性能评估只看跑分可不够.....59

如果您也想分享您的测试历经和学习心得，欢迎加入我们~(*^▽^*)

投稿邮箱：editor@51testing.com

华山论剑，Web 性能测试工具谁与争锋

◆ 作者：王 练

摘要： 互联网访问存在黄金 3 秒原则（3 秒网页没打开 57％的网友就会走），所以性能决定了网站的生死。如何进行 Web 性能测试，客观的评价网站的性能，并进行有效的优化具有决定性的意义。在众多 Web 性能测试工具中，LoadRunner 和 JMeter 是众人皆知的优良框架。事实上，还有很多优秀的工具在各个角度进行着 Web 性能测试的支持。这些软件或开源免费，扩展性强；或商业全面，提供强大的生态圈。有的用户界面功能强大，美观大方；有的短小精悍，上手迅速。

本文主要介绍 12 款 Web 性能测试的软件，最后也给出更多的，在有限篇幅无法覆盖的工具列表，为大家进行 Web 性能测试的学习、研究和项目实战提供参考。

1、前言

本文以金庸武侠小说的重大事件“华山论剑”，作为性能测试工具的出场介绍，对主要介绍的工具都赋予了一个小说中的人物或设定。排序并不是以武功高低进行，对工具的人物或设定对应，主要是某一方面的特征进行，也许并不完全准确。

实际上性能测试的工具非常多，单纯的 Web 性能测试工具也不胜枚举。本文限定在 Web 性能测试领域，重点介绍了 12 个工具，最后列出了目前作者能够看到的所有 Web 性能测试工具。

2、王重阳—LoadRunner

在 Web 性能测试领域，LoadRunner 永远是开山鼻祖式的存在。其他的性能测试工具、框架或多或少都与 LoadRunner 有相似之处。究竟是谁抄袭谁，没有必要深究，但从知名度、市场占有量来说，LoadRunner 都是 Number One。是性能测试领域的王重阳，毕竟是第一次华山论剑的冠军。

LoadRunner 最初是 Mercury 公司的产品，后被 HP 收购。完整的解决方案，易用的 IDE，丰富的报告。脚本编写提供多种捕捉模式， 支持高级的文本浏览和低级的 HTTP 浏览，并且支持图形化的树形结构和脚本浏览方式。支持 HTTP、RTSP、SIP、Socket、JAVA 自定义、媒体质量分析等。

使用基本上采取录制、修改（Virtual User Generator）、场景设计（Controller）、运行（Load Generator）、分析（Analysis），这样的流程。

LoadRunner 的劣势主要是商业收费软件，而且非常昂贵（社区版提供的并发量有限）。并且由于是传统的桌面软件，安装部署非常麻烦，安装包大概有 3-4G，网友反馈都有安装失败的情况。LoadRunner 是传统的一箩筐式软件，软件将能想到的都包括，没想到的我们想扩展也比较难。脚本语言是 C 语言，提供了丰富的库函数使用。

LoadRunner 还有问题就是持续集成比较麻烦，需要付出非常大的努力。

性能测试中非常关心并发的性能，LoadRunner 资源的限制主要是线程数量及内存大小。在 Windows 上每个虚拟用户会占到 1 Mb 内存。所以可以根据机器配置计算出单个测试机的并发上限。当然，LoadRunner 支持分布式加压。

总体来说，LoadRunner 是 Web 性能测试的开山鼻祖，其性能测试的协议模拟理念，脚本编写、场景设计、运行和分析的整体流程，也成为业界性能测试的标杆。建议花时间进行研究。

LoadRunner 目前最新版本是 12.60，可以下载试用版进行学习：https://www.microfocus.com/en-us/products/loadrunner-load-testing/download。

3、杨过—JMeter

说 JMeter 是性能测试界的杨过，并不是说杨过武功最好，而是他的主角光环。正如 JMeter 目前在 Web 性能测试领域的受人瞩目一样。

JMeter 基本属于目前 Web 性能测试领域的事实标准，普及范围非常广。可以说，进行 Web 性能测试的，没有不知道 JMeter 的，而且由于其开源免费，扩展性好，用的也非常多。作者所在的项目就是试用过 LoadRunner 和 JMeter 之后，选择了 JMeter。即便是没有使用 JMeter 的公司，也都经历过 JMeter 和其他框架比较的过程。

JMeter 是 Apache 组织开发的基于 Java 的压力测试工具。用于对软件做压力测试，

它最初被设计用于 Web 应用测试，但后来扩展到其他测试领域。JMeter 支持的协议非常多，Web - HTTP, HTTPS ,SOAP / REST,FTP,Database via JDBC,LDAP,Message-oriented middleware (MOM) via JMS,Mail - SMTP(S), POP3(S) and IMAP(S),Native commands or shell scripts,TCP 等等。并且不支持的协议也可以通过开源插件扩展，甚至可以自己编写插件（JMeter 提供了开发框架，只要实现特定接口即可）。

JMeter 的最大优势是开源免费，扩展能力强，安装简便，基本上是解压即可使用。JMeter 的扩展能力包括可以修改源码（使用很少），使用已有的开源插件（用的最多，而且能满足 90%的需求），还可以自己在 JMeter 框架下自行开发插件（某些特殊需求会用到），还有就是支持 BeanShell、JSR223 脚本。丰富的 IDE 功能，基本需求可以界面点击、填写就可以完成脚本编写。

简单的安装、丰富的功能、强大的扩展能力，让每一个接触过 JMeter 的使用者无法拒绝。

JMeter 自身的性能是硬伤，基于 JVM 的 Threads，采用了一个用户使用一个线程的方式，并发线程越多，越占用内存和 CPU。一旦并发线程过多，性能就急剧下降（可以通过分布式解决）。同时，JMeter 的脚本使用 XML，即使是 BeanShell 等脚本语言，也是嵌在 XML 文件里面的，不易维护和管理，协同工作有障碍。

总体来说，JMeter 是目前 Web 性能测试的事实标准，JMter 群众基础广泛，开源的生命力强大，持续集成容易。是值得一试的性能测试软件，强烈推荐。

JMeter 目前最新版本是 Apache JMeter 5.1.1，下载地址：http://jmeter.apache.org。

4、段智兴—RPT

南帝段智兴，出家后法号一灯法师，是老段家有名的段誉的爷爷。RPT 正如段皇爷一样，是名门出品。RPT 的全称是 IBM Rational Performance Tester，是 IBM 基于 Eclipse 平台及开源的测试及监控框架 Hyades，开发出来的性能测试解决方案。RPT 是一个为测试 Web 应用程序设计的软件，它的功能是在部署之前捕获并更正性能问题。

RPT 通过模拟并发用户的数量，产生一系列报告，这些报告被清晰的标志出来显示 Web 页面的性能、URL 和事务等信息，来帮助您在部署之前查明系统的瓶颈。高级的特性包括在每一个用户组的行为和使用方式层面的详细测试调度安排。RPT 同时还提供了一种自动化的"数据池"特性，它可以改变每一个模拟用户使用的测试数据设置。使用

一个集成了测试编辑器的浏览器窗口，您可以回顾在测试记录期间访问过的 Web 页面。除此之外，高级测试人员可以选择将自定义的 Java 代码插入到性能测试中，用来执行类似于高级数据分析和请求解析的操作。

熟悉自动化测试的人员应该听过另一款 IBM 出品的软件 RFT，是 IBM 对功能自动化测试提供的解决方案。事实上，IBM 对此给出非常丰富的软件套件，包括质量管理系统 RQM。这些软件的优势在于出于名门。IBM 出品，品质有保证。事实也是如此，RPT 支持 HTTP ，HTTPs， Citrix SAP SDK 等多种协议，支持脚本录制，IDE 界面完善、美观功能强大，同时类 Eclipse 的界面，让 Java 程序员有强烈的归属感。脚本语言使用 Java 编写，有非常大的发挥空间。支持分布式加压，先天支持与 IBM 其他系统集成的能力。

RPT 由于是传统的桌面软件，并且保罗万象，安装部署较为繁琐，同时并发性能较 LoadRunner 并无明显优势，并且持续集成需要付出很大的努力。商业软件，价格不菲。

总体来说，RPT 是名门 IBM 出品，质量、稳定性都有所保证。然而除非被绑架在其他 IBM 系统中，用的人还是少了些。没有 JMeter 的免费优势，又有 LoadRunner 的强烈挤占，是比较尴尬的存在。建议了解一下即可。

RPT 目前最新版本是 8.1，可以在官网上下载试用版:
https://www.ibm.com/developerworks/cn/downloads/r/rpt/。

5、黄药师—Locust

将 Locust 比作黄药师，主要是因为二者都是剑走偏锋的形象。黄药师，也称黄老邪，武功高强，但性格怪异。Locust 也是一样的，Locust 的原意是蝗虫，原作者之所以选择这个名字，估计也是听过这么一句俗语，“蝗虫过境，寸草不生”。光看这个名字，就明白 Locust 的初衷是多么的邪性。

Locust 工具生成的并发请求就跟一大群蝗虫一般，对我们的被测系统发起攻击，以此检测系统在高并发压力下是否能正常运转。在 Locust 测试框架中，测试场景是采用纯 Python 脚本进行描述的。对于最常见的 HTTP(S)协议的系统，Locust 采用 Python 的 requests 库作为客户端，使得脚本编写大大简化，富有表现力的同时且极具美感。而对于其它协议类型的系统，Locust 也提供了接口，只要我们能采用 Python 编写对应的请求客户端，就能方便地采用 Locust 实现压力测试。从这个角度来说，Locust 可以用于压测

任意类型的系统。

在模拟有效并发方面，Locust 的优势在于其摒弃了进程和线程，完全基于事件驱动，使用 gevent 提供的非阻塞 IO 和 coroutine 来实现网络层的并发请求，因此即使是单台压力机也能产生数千并发请求数；再加上对分布式运行的支持，理论上来说，Locust 能在使用较少压力机的前提下支持极高并发数的测试。

Locust 的最大优势也在于并发量。通过协程并发，编写简单易读的代码完成测试脚本，基于事件，同样配置下，单台负载机可模拟的负载数远超 JMeter 和 LoadRunner。开源免费软件，基于 Python 扩展，扩展能力强。默认只支持 HTTP 和 HTTPS，其他协议预留了接口，需要自己实现。Locust 的使用很简单也很方便，它提供了一个 Python LIB、一个命令行工具和一个 Web UI，用户通过自己写 code 来定义测试用例，通过运行 Locust CLI 来执行测试，通过 Web 界面查看测试结果。对于程序员（特别是 Python 程序员）来讲，Locust 另外一个优势就是，你不用去学不同工具为了定义测试用例而设计的不同的 DSL，直接写 code，通俗易懂。同时基于 PIP 安装，非常方便。

Locust 的缺点是默认的报告比较简单，只提供实时测试数据，错误统计，QPS、评价响应时间等几个简单实用的图表。不过在扩展性面前，这些都不是问题。

总体来说，Locust 是剑走偏锋的典范，是 Python 程序员的福音。支持并发加压，支持持续集成。建议进行性能测试时可以作为备选方案，特别是对并发性能有要求的，强烈推荐。

Locust 目前最新版本是 0.11.0（从版本号也可以看出其未来非常可期），可以在官网上下载使用（直接 PIP）：https://www.locust.io/。

6、周伯通—Gatling

周伯通也称老顽童，认为世间最重要的事情就是玩，就像小孩子一样。正如 Gatling 一样，非常好玩。玩 Gatling 的最大感觉就是简单，非常容易就可以安装和使用，让人一下就喜欢上这个简单的框架，正如喜欢老顽童的人非常多一样。

Gatling 是一款基于 Scala 开发的高性能服务器性能测试工具，它主要用于对服务器进行负载等测试，并分析和测量服务器的各种性能指标。目前仅支持 HTTP 协议（其他需要自己扩展），可以用来测试 Web 应用程序和 RESTful 服务。优良的性能，DSL 模式的脚本，轻量友好的报表系统，伴随大数据崛起的 Scala（Scala 支持 JVM 和 Net），活

跃的社区支持，开源化，这些都让前景充满希望。

Gatling 的并发性能非常可观，使用了 Akka Actors 和 Async IO，并发性更强。基于 JVM 的 Actors 模型，用更少的内存和 CPU。开源免费的特性也让 Gatling 拥有大批拥趸。Gatling 扩展能力强，基于一套开源的 Gatling DSL API，所以它的功能很容易扩展。同时脚本语言 Scala 也是基于 Java，可以任意调用 Java 的 API。70M 的大小，方便下载，容易安装，绝对的轻量化。持续集成非常容易，Jenkins 自带 Gatling，直接查看测试结果。

Gatling 的缺点是熟悉 Scala 的人相对较少。没有什么 IDE，脚本中的逻辑控制、参数化、检查点等都依赖于编程。脚本编写就是 Scala 编程，使用文本编辑器进行。报告也是提供一套轻量并且十分友好的 Html 报表系统，但种类不够丰富。还有一个致命伤就是不支持分布式。这点看来，Gatling 是在桃花岛监禁之前的周伯通，还没有练成双手互搏。

总体来说，Gatling 崇尚的就是简单，什么 IDE，什么多协议，只要能扩展都不是事，关键是让你的测试快速运转起来。事实上，这是一个大的趋势，不会编程的用不好 JMeter，会编程的更喜欢编写单纯的代码，而所谓的界面生成代码，往往是程序员嗤之以鼻的对象。建议作为了解的软件范围，等分布式完善了再进行深入研究，一般推荐。

Gatling 目前最新版本是 3.1.2，是 2019 年 5 月 15 发布的，可以在官网上下载使用：https://gatling.io/download/。

7、郭靖，黄蓉—Grinder 和 nGrinder

nGrinder 是基于 Grinder 开源项目，由 NHN 公司的开发团队进行了重新设计和完善的一个 Web 性能测试解决方案。所以，二者的关系和郭靖和黄蓉一样，属于“贤伉俪”。同时，Grinder 就像郭靖一样，稳扎稳打，提供最底层的支持；而 nGrinder 就像黄蓉一样，嘴皮子功夫了得，提供了非常易用的界面。

Grinder 是一个开源的 JVM 负载测试框架，它通过很多负载注射器来为分布式测试提供了便利。 支持用于执行测试脚本的 Jython 脚本引擎 HTTP 测试可通过 HTTP 代理进行管理。根据项目网站的说法，Grinder 的主要目标用户是“理解他们所测代码的人”，Grinder 不仅仅是带有一组相关响应时间的‘黑盒’测试。由于测试过程可以进行编码，而不是简单地脚本化，所以程序员能测试应用中内部的各个层次，而不仅仅是通

过用户界面测试响应时间。

nGrinder 是基于 Grinder 开源项目，由 NHN 公司的开发团队进行了重新设计和完善。nGrinder 是一款非常易用，有简洁友好的用户界面和 controller-agent 分布式结构的强大的压力测试工具。虽然是公司出品，但也是开源免费的软件。

nGrinder 测试基于 Python 测试脚本(Groovy 也可)，用户按照一定规范编写测试脚本，controller 会将脚本一集需要的资源分发到 agent，用 Jython 执行。并且在执行的过程中收集运行情况、相应时间、测试目标服务器的运行情况等。并且保存这些数据生成测试报告，以供查看。这款框架的一大特点就是非常的简单易用，安装也很容易，可以说是开箱即用。nGrinder 发布为 war 包，直接部署成 Web 服务，支持多用户使用，可扩展性好，可自定义插件。

二者都使用 Jython/Groovy 进行脚本编写，单节点支持 3000 并发、支持分布式、可监控被测服务器、可录制脚本、开源、平台化。支持多协议，如 HTTP Web servers, SOAP 和 REST Web services，还有客户端服务器 RMI、JMS、EJBs 等，还支持自定义协议。

总体来说，Grinder 是中规中矩的框架，nGrinder 对于 Grinder 来说也是换汤不换药。建议可以作为学习的对象，一般推荐。

Grinder 目前最新版本是 3.11，可以在如下链接下载使用:
http://grinder.sourceforge.net/download.html。

nGrinder 目前最新版本是 3.3，可以在如下链接下载使用:
https://sourceforge.net/projects/ngrinder/。

8、裘千仞—OpenSTA

裘千仞的武功其实是很高的，基本上能和王重阳一绝高下。然而由于道德属性的原因，知名度不高，同时喜欢他的人也不多。OpenSTA 也是这样的存在，它是否存在过，可能很多人都不一定知道，如今已经过气了，更不会为大众所知。

OpenSTA 是 Cyrano 公司开发的专用于 B/S 结构的、免费的性能测试工具。它的优点除了免费、源代码开放的优点外，还能对录制的测试脚本进行,按指定的语法进行编辑。测试工程师在录制完测试脚本后，只需要了解该脚本语言的特定语法知识，就可以对测试脚本进行编辑，以便于再次执行性能测试时获得所需要的参数，之后进行特定的

性能指标分析。OpenSTA 以最简单的方式让大家对性能测试的原理有较深的了解，其较为丰富的图形化测试结果大大提高了测试报告的可阅读性。

OpenSTA 是 Windows 上使用的软件，并且不支持监控 Linux 服务器。安装包在 5Mz 左右，安装较为容易。

OpenSTA 脚本语言是自定义的 SCL（Scipt Control Language），在开源大行其道的如今，自己定义一门语言，基本上属于作死的节奏。只有使用这个工具，这门语言才有生命力，这种投入太不划算。同时 OpenSTA 的 Results 结果报告的图表太简陋，功能太少。实际上是输出为 Excel，通过其论坛上大家贡献的宏进行计算，想法太理想。脚本的编写以录制为主，但支持到 IE6，修改由于 SCL 的限制，异常困难。并发性能方面，有报告称 3000 并发需要 1G 内存，优势不大。

总体来说，OpenSTA 属于过气的三线明星。从官网上看，2007 年之后就没有更新过，基本属于过气的软件。作为一个认识即可，强烈不推荐。

Locust 目前最新版本是 1.4.4（2007 年 10 月 19 提供），可以在官网上下载使用：http://www.opensta.org/download.html。

9、洪七公—WebLoad

洪七公可以说是家喻户晓的人物，武功高强，内力深厚，而且道德品质高，受人尊敬和喜爱。WebLoad 虽然知名度不像洪七公一样高，但其并发性能高，支持 JavaScript 和 Selenium 脚本，可以和降龙十八掌、打狗棒相提并论。

WebLoad 是 RadView 公司推出的一个性能测试和分析工具,它的最大特点是让 Web 应用程序开发者自动执行压力测试。WebLoad 通过模拟真实用户的操作,生成压力负载来测试 Web 的性能。支持界面点击、操作的压力测试，支持 Selenium 编写的脚本。这点和其他性能测试工具都是有差别的，是 WebLoad 在江湖占领一席之地的降龙十八掌和打狗棒。

并发性能方面 WebLoad 官网号称两倍于 LoadrRunner。在负载测试会话之后，WebLOAD Analytics 支持丰富的报告，可以从 80 多种不同的报告和图表中进行选择，从不同角度分析测试结果。支持与应用程序性能管理（APM）工具集成，可深入了解瓶颈背后的特定事件。目前支持三种大型 APM 工具--Dynatrace，AppDynamics 和 NewRelic。IDE 功能丰富，提供基于代理的记录器，可让您记录所有 Web 活动。而且录

制/播放能够识别网页资源（css，js，图像等）并自动加载它们。支持多种协议，包括 AJAX 在内的 REST/HTTP）、SOAP/XML 及其他可从脚本调用的协议如 FTP、SMTP 等。

WebLoad 提供了免费版本和商业版本，免费只支持 50 并发（其实就是试用版本）。支持持续集成，支持分布式。

WebLoad 的缺点是安装比较繁琐，和 LoadRunner 有的一拼。同时用户群体不大，资料较少。

总体来说，WebLoad 并发性能高，内力雄厚。而且支持点击式的加压和 Selenium 脚本。建议作为学习的软件，一般推荐。

WebLoad 目前最新版本是 2018 年 8 月 21 日发布的 11.2，可以在官网上下载使用：https://www.radview.com/Webload-licensing/。

10、欧阳锋—QALoad

欧阳锋是个亦正亦邪的人物，不能算是传统意义上的好人，但也做过一些好事。欧阳锋最出名就是其蛤蟆功。QALoad 的性能测试理念和 LoadRunner 基本一致的， 但最终获得的市场确是非常少，只能感慨既生瑜何生亮了。QA Load 有着自己的蛤蟆功，可以将 Flash 的 AMF 数据解析成可读的树形格式，并且允许对其中的数据进行参数化和修改。这在协议模拟的性能测试中独树一帜。

QA Load 是 Micro Focus 公司的一个产品，主要是用来做性能测试的，该工具可以用来模拟成百上千的用户并发执行关键业务而完成对应用程序的测试。因为这是个商业软件，所以在使用的便利性方面明显要比一些免费的工具好。我们可以利用该工具录制脚本，制定测试计划，模拟用户向 Server 发送 request，检测测试结果并生成丰富的测试报告。

QA Load 是商业软件，收费一般，从一些网站的评价看应该比 LoadRunner 便宜些。脚本使用 C++进行编写。提供功能丰富的 IDE 进行操作。

在并发性能方面，QA Load 单机在千级用户，竞争力一般。同时安装繁琐，不支持分布式，持续集成难。这些都是影响其普及的原因。

总体来说，QA Load 吃亏在于 LoadRunner 和其他框架太好用了，自身的蛤蟆功没

有被重视起来，既生瑜何生亮。建议作为了解的对象，不推荐项目使用。

QA Load 目前最新版本是 5.5，2009 年发布，可以在如下链接下载使用 https://www.microfocus.com/en-us/products/silk-performer/overview。

11、易筋经—WAST

WAST 只能算作一本秘笈，不能作为一个体系。

WAST 全称是 Web Application Stress Tool，它是一个比较简单易用的工具，不能进行参数设定，只适合模拟一些固定的网络地址连接的压力测试。

WAST 是由微软的网站测试人员所开发，专门用来进行实际网站压力测试的一套工具。透过这套功能强大的压力测试工具，可以使用少量的客户端计算机仿真大量用户上线对网站服务所可能造成的影响，在网站实际上线之前先对您所设计的网站进行如同真实环境下的测试，以找出系统潜在的问题，对系统进行进一步的调整、设置工作。脚本只能进行 HTTP 的测试，而且不能扩展。

总体来说，WAST 是微软私人订制的一个工具，目前微软已经不进行版本升级了。下载也只能各种百度获取，没有官网支持。作为了解即可，强烈不推荐。

12、屠龙刀—http_load

上面介绍的 WAST 功能简单，但总之还算是一个完整的软件。http_load 就是一个命令行的工具而已。但千万不能小看这个工具，小但是能力强大，不折不扣是 Web 性能测试界的屠龙刀。

http_load 是一款基于 Linux 平台的 Web 服务器性能测试工具，用于测试 Web 服务器的吞吐量与负载，Web 页面的性能。采用单一进程并行复用方式发包。由于只有一个进程，http_load 对于机器资源消耗较小，性能要求不高，但它的劣势就是最大并发量只能达到千的量级。在很多场景中，几千的量级也已经绰绰有余了，因此具体工具的选用还要看实际情况。

这个工具非常小，只有 79K，但对于基本的 HTTP 性能验证是足够了。

只缺点是只提供纯命令的使用，没有 IDE，提供的报告只是字符串输出的结果，给出重点的请求次数，并发量，传输量，运行时间，每秒的吞吐，RT 的信息，响应结果。默认只支持 GET 请求，POST 需要自己改源码。所以只能称为一个工具，不能算为

一个性能测试的框架。作为一个简单的 Web 性能验证脚本，推荐使用。想要丰富的验证、丰富的请求、丰富的报告，建议另请高明，或者作出惊人的修改。

总体来说，http_load 是简单验证性能的有力武器，如果要系统的进行性能测试，不建议使用。

http_load 目前最新版本是 http_load-09Mar2016.tar.gz，你没有看错，时间就是版本号，软件就是一个 tar 包。可以在如下链接下载使用 http://www.acme.com/software/http_load/。

13、倚天剑—Web Bench

Web Bench 和 http_load 一样，是一个命令行工具，同样功能强大，是 Web 性能测试的倚天剑。

Web Bench 是有名的网站压力测试工具，由 Lionbridge 公司开发。Web Bench 能测试处在相同硬件上，不同服务的性能以及不同硬件上同一个服务的运行状况。Web Bench 的标准测试可以向我们展示服务器的两项内容：每秒钟相应请求数和每秒钟传输数据量。Web Bench 不但能具有对准静态页面的测试能力，还能对动态页面（ASP,PHP,JAVA,CGI）进行测试的能力。还有就是他支持对含有 SSL 的安全网站例如电子商务网站进行静态或动态的性能测试。

Web Bench 的并发量惊人，采用多进程发包，最多支持 3 万并发量。脚本使用 C 语言编写，默认只支持 GET，POST 需要自行开发。

总体来说，Web Bench 和 http_load 类似（二者的官网非常像），都是简单验证性能的有力武器，如果要系统的进行性能测试，不建议使用。

Web Bench 目前最新版本是 1.5，是 2004 年 7 月 13 日发布。可以在如下链接下载使用 http://home.tiscali.cz/~cz210552/Webbench.html。

14、其他工具

还有很多其他性能测试工具，这里简单说明一下。

与 http_load、Web Bench 类似的开源小工具，如 Apache Bench（就是经常提到的 ab ）、Siege、wrk、LoadSim、Taurus、LoadGen、Tsung、Multi-Mechanize、 ，都是快速部署，快速验证 Web 性能的倚天剑、屠龙刀。

还有与 LoadRunner、JMeter 同级别的 Web 性能测试系统解决方案软件，如 GoReplay 、OctoPerf、Apica Loadtest、LoadView 、Agileload、Loadster、Load Complete、Sandstorm、Load2Test、Xceptance LoadTest、Blazemeter、LoadStorm、Neustar Web Performance、Load Impact、AppLoader、fwptt、Curl-Loader、StressTester、Avalanche、NeoLoad、QTest、WAPT、Visual Studio Test Professional、OpenLoad Tester、Web Polygraph、PureLoad、Load Tester by Web Performance、SilkPerformer，都是功能强大，支持协议众多的 Web 性能测试英雄好汉。

15、总结

所有的 Web 性能测试工具可以分为两类，一类是以 LoadRunner、JMeter 为例的全套解决方案，不但可以进行 HTTP 协议验证，还可以进行其他请求的验证（如数据库、文件），同时提供丰富的功能对脚本编写、调试、性能分析、报告输出进行支持。还有一类是以 http_load、Web Bench 为例的纯命令行工具，提供了简单的 HTTP 并发请求示例，更多的工作需要性能测试人员自行扩展。

项目性能测试中，推荐使用全套解决方案类的软件，可以进行系统的测试。命令行的工具，可以作为简单验证、问题定位的利器使用。

在众多解决方案中，并发能力方面 Locust 是绝对的王者。易用性方面各有千秋。开源免费和商业软件的选择，更是老生常谈，要看项目具体的情况。不过使用开源，进行扩展的方式越来越流行了。

如果进行学习、研究，推荐对 LoadRunner、JMeter、Locust、Gatling 和 nGrinder 进行深入考察。

如果是项目实战，需要根据项目实际情况进行考量。个人建议，目前阶段还是 JMeter 排第一（用的人太多了），之后是 Locust（并发能力强）和 nGrinder。Gatling 目前不支持分布式，单机性能堪忧。

最后需要说明的是，工具只是工具，最重要的还是用工具的人。不要把宝都压在工具上，需要更多的研究待测系统的软件架构、业务流程，设计最适用于待测系统的性能测试方案。

往期作品:《软件测试证书知多少？》

机器学习与数据挖掘十大经典算法之逻辑回归算法

◆作者：咖啡猫

题记：前一篇《机器学习与数据挖掘十大经典算法之 PageRank 算法》发表之后，有不少读者给我留言说好希望快点更新，奈何最近工作实在太忙，逻辑回归算法的介绍只能在这期电子杂志里和大伙儿见面了。

一、逻辑回归的简介

逻辑回归，英文名是 logistic regression。它是一种广泛使用的机器学习算法。尽管名字中有“回归”二字，但它其实是一种二元分类算法，也就是通过已知的自变量来预测数据是属于哪个类别的（也就是，判断数据的标签是 真或假，是或否等）。

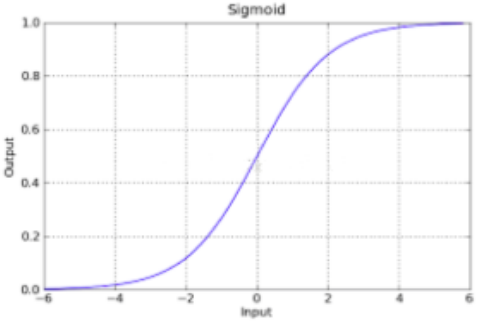
在现实生活中，也有很多二元分类问题。例如，判断一封邮件是否为垃圾邮件？判断一次金融交易是否存在欺诈嫌疑？判断肿瘤是否是良性的？

二、逻辑回归的数学思想

通过拟合一个逻辑函数/假设函数（logic function/hypothesis function）来预测一个事件发生的概率，既然预测的是概率值，那么输出结果必须在（0--1）之间。

一般线性回归的结果（即 y=h(Θ,x)的结果）是连续值，取值范围也是不确定的。我们需要找到一个数学公式，将连续值转换成（0--1）之间的值。

这个转换函数就是 sigmoid 函数：$g(z)=\frac{1}{1+e^{-z}}$，它的图像如下：



三、逻辑回归的算法步骤

初始化权重：即给定向量Θ的初始值，方便后续不断迭代更新。

加载数据：加载训练数据，方便拟合出假设函数

计算假设函数

$$h_\theta(x) = g(\theta^T x)$$
$$g(z) = \frac{1}{1+e^{-z}}$$

注：上面的公式表面自变量和标签之间的函数关系，下面的公式将标签取值范围缩小到 0 到 1 之间，以判断一个事件发生的概率。

计算单一样本的代价函数

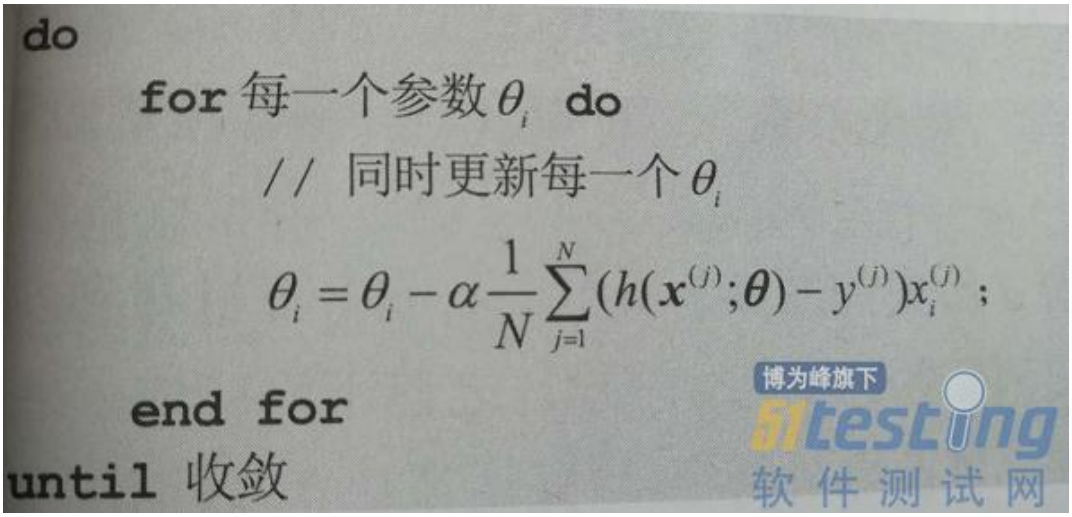
$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

注：对单一样本而言，它的错分类代价就是：当 Y 等于 1 时，h(x)的结果是 0；当 Y 等于 0 时，h（x）的结果是 1.

计算总体样本的损失函数：即 N 个样本的代价均值。

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \text{Cost}(h_\theta(x^{(i)}), y^{(i)})$$
$$= -\frac{1}{m}[\sum_{i=1}^{m} y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))]$$

使用梯度下降法来迭代更新参数值向量Θ，以保证损失函数最小



7.测试：用拟合好的假设函数，对测试数据集进行分类。

四、逻辑回归的 PYTHON 实现

```
    import os
import string
import sys
import math

class LogisticRegression:
    def __init__(self):
        self.__X = []  # 特征集合
        self.__Y = []  # 标签
        self.__theta = []  # 权重
        self.__LEARNING_RATE = 7  # 学习率
        self.__FEATURE_CNT = 1 + 2  # 特征数
        self.__load_training_data()  # 加载数据
        self.__SAMPLE_CNT = len(self.__Y)  # 样本数
        self.__feature_scaling()  # 特征缩放

        for idx in range(0, self.__FEATURE_CNT):
            self.__theta.append(0)

    def __load_training_data(self): # 加载训练数据，分离出特征和标签
        fp = open(r"E:\xdlv\testSet.txt", "r")
        for line in fp.readlines():
            (x1, x2, y) = line.strip('\r\n').split('\t')
            self.__X.append([1, float(x1), float(x2)])
            self.__Y.append(float(y))
        fp.close()

    def __feature_scaling(self): #为了帮助梯度下降算法收敛更快，进行特征缩放，使所有特征的取值范围具有相似的尺度。
        max_value = []
        min_value = []
        for fidx in range(0, self.__FEATURE_CNT):
            max_value.append(0)
            min_value.append(100)

        for idx in range(0, self.__SAMPLE_CNT):#依次遍历每一个样本的每一个特征值，找到每一个特征值的最大值和最小值，然后做特征变换。
            for fidx in range(1, self.__FEATURE_CNT):
                if max_value[fidx] < self.__X[idx][fidx]:
                    max_value[fidx] = self.__X[idx][fidx]
                if min_value[fidx] > self.__X[idx][fidx]:
```

```
                    min_value[fidx] = self.__X[idx][fidx]
        for idx in range(0, self.__SAMPLE_CNT):
            x = self.__X[idx]
            for fidx in range(1, self.__FEATURE_CNT):
                self.__X[idx][fidx] = (x[fidx] - min_value[fidx]) / (max_value[fidx] - min_value[fidx])

    def batch_learning_alogrithm(self):#机器学习流程：迭代到 10W 次，每次迭代中训练数据（更新参数集），知道 Loss 收敛到最小。
        last_loss = 0
        for itr in range(1, 100000):
            # 1、训练数据
            self.__training()
            loss = self.__get_loss()
            sys.stdout.write("After %s iteratorion loss = %lf\n" % (itr, loss))

            if math.fabs(loss - last_loss) <= 0.01:
                break;
            last_loss = loss

        sys.stdout.write("The coef of the logistic model :\n")
        for idx in range(0, self.__FEATURE_CNT):
            sys.stdout.write("theta[%d] = %lf\n" % (idx, self.__theta[idx]))

    def __training(self):
        # 初始化权重为[0 0 0]
        weight = []
        for idx in range(0, self.__FEATURE_CNT):
            weight.append(0)

        """计算 loss"""
        for idx in range(0, self.__SAMPLE_CNT):
            x = self.__X[idx]
            y = self.__Y[idx]
            h = self.__sigmoid(x)
            for fidx in range(0, self.__FEATURE_CNT):
                weight[fidx] += (h - y) * x[fidx]

        """更新权重"""
        for idx in range(0, self.__FEATURE_CNT):
            self.__theta[idx] -= self.__LEARNING_RATE * weight[idx] / self.__SAMPLE_CNT


    def __sigmoid(self, x):
        logit = 0
        for idx in range(0, self.__FEATURE_CNT):
            logit += self.__theta[idx] * x[idx]
```

```
            return 1.0 / (1.0 + math.exp(-logit))

        def __get_loss(self):
            loss = 0
            for idx in range(0, self.__SAMPLE_CNT):
                x = self.__X[idx]
                y = self.__Y[idx]
                h = self.__sigmoid(x)
                loss += y * math.log(h) + (1 - y) * math.log(1 - h)
            return loss

        def test(self):
            wrong_ans = 0
            for idx in range(0, self.__SAMPLE_CNT):
                x = self.__X[idx]
                y = self.__Y[idx]
                h = self.__sigmoid(x)
                check = 0
                if y > 0.5 and h < 0.5:
                    check = -1
                if y < 0.5 and h > 0.5:
                    check = -1
                sys.stdout.write("样本 %d : 真值 = %.2lf, 预测 = %.2lf check = %d\n" % (idx, y, h,
check))
                wrong_ans -= check
            print
            "错误 = %d" % wrong_ans


if __name__ == "__main__": #main 是主函数，也是程序入口。
    lr = LogisticRegression()
    lr.batch_learning_alogrithm()
    lr.test()
```

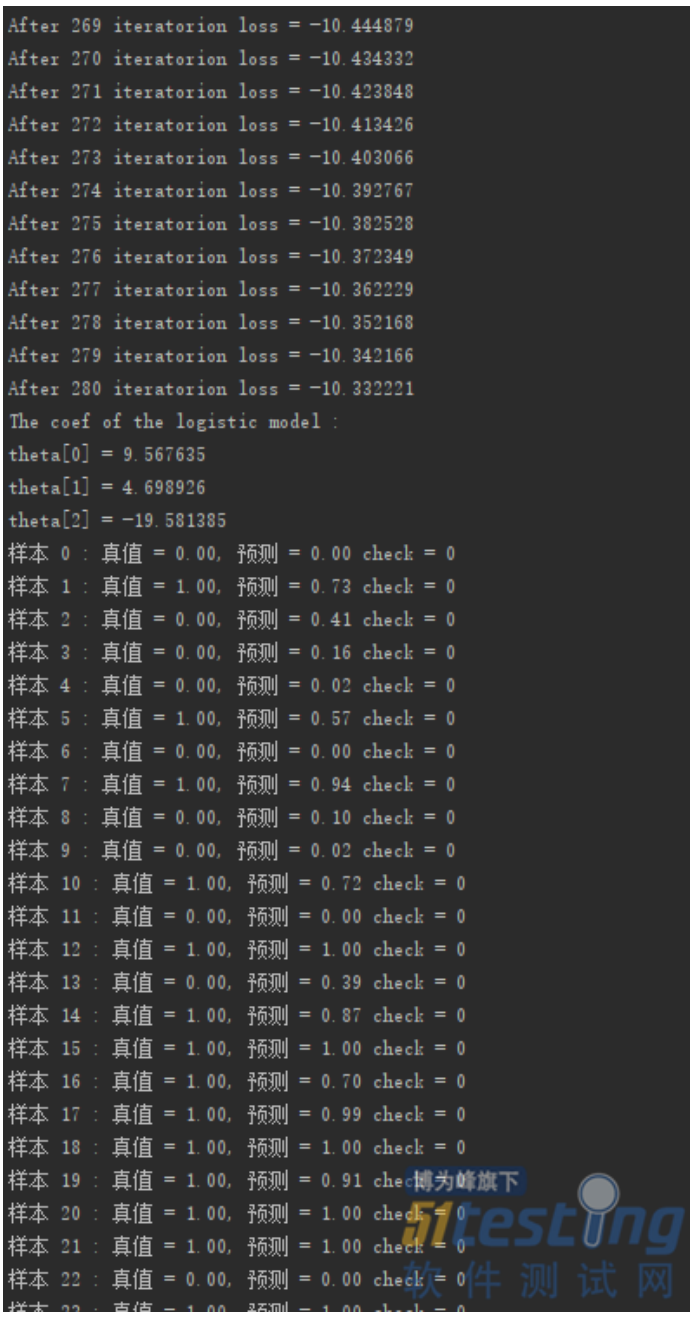
测试数据:

testSet - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
-0.017612       14.053064       0
-1.395634       4.662541        1
-0.752157       6.538620        0
-1.322371       7.152853        0
0.423363        11.054677       0
0.406704        7.067335        1
0.667394        12.741452       0
2.460150        6.866805        1
0.569411        9.548755        0
0.026632        10.427743       0
0.850433        6.920334        1
1.347183        13.175500       0
1.176813        3.167020        1
1.781871        9.097953        0
0.566606        5.749003        1
0.931635        1.589505        1
-.024205        6.151823        1
-.036453        2.690988        1
-.196949        0.444165        1
1.014459        5.754399        1
1.985298        3.230619        1
-.693453        -0.557540       1
-.576525        11.778922       0
-0.346811       -1.678730       1
-.124484        2.672471        1
1.217916        9.597015        0
-.733928        9.098687        0
-.642001        -1.618087       1
0.315985        3.523953        1
1.416614        9.619232        0
-.386323        3.989286        1
0.556921        8.294984        1
1.224863        11.587360       0
-.347803        -2.406051       1
1.196604        4.951851        1
0.275221        9.543647        0
0.470575        9.332488        0
-.889567        9.542662        0
-.527893        12.150579       0
-.185247        11.309318       0
-.445678        3.297303        1
1.042222        6.105155        1
```

运行结果:



后记:

整个逻辑回归的基础知识就介绍到这里了，我个人觉得代码实现并不是最大的困难，难点是数学公式的推导。学习算法的时候，要理解每一步数学公式的作用和含义，然后再用 Python 把数学公式实现出来就可以了。朋友们，你 GET 到了吗?

往期作品:《记一次难忘的腾讯面试经历》

一种基于 Tesseract 识别验证码实践

◆ 作者：付 伟

验证码是一种防止程序自动化的一个措施，是一种反自动化技术，可以防止恶意破解密码、刷票、论坛灌水，有效防止黑客对网站、应用程序以暴力破解方式进行不断的身份验证。实际上验证码是很多网站通行的方式，利用比较简易的方式实现了这个功能，同时验证码的样子也会尽量千奇百怪，让机器不能够识别。

金融领域目前大力推广自动化测试，注册，登录，找回密码，支付等交易过程中必不可少的会有验证码环节，验证码的存在使得很自动化测试工作止步。针对这一现状，本文以识别某系统验证码为例，通过环境搭建、验证码图像分析和预处理、Tesseract-OCR 识别识别验证码、Tesseract-OCR 语言包训练、训练的语言包识别验证码等过程介绍字符验证码识别的方法，对后续 Web 自动化测试和 OCR 识别都有一定的参考价值。

1、验证码获取

大多数验证码识别研究为了方便计算识别率基于 Captcha 生成大量的验证码，本文针对某系统验证码进行验证，通过利用 selenium 截取页面，定位验证码元素的位置，利用 PIL 进行处理，获取其中验证码部分。

验证码获取基于 python 通过 selenium+PIL 实现。Selenium 本身不是测试工具，是模拟浏览器操作的工具，支持全部主流的浏览器，同时支持主流的编程语言。 PIL 是 python 的第三方图像处理库，具备强大的图形处理功能，包含格式转换、对点的处理图像过滤、大小转换、图像旋转等。

具体安装 python、selenium、PIL 不在此展开，需要提醒的是用于 webdriver 的浏览器驱动需要与电脑上的浏览器版本相对应，否则无法驱动浏览器执行操作。

实现思路是:

1)登录页面按 F12 检查元素，找到验证码图片的元素 id

2)调用 chromedriver，通过 webdriver 定位工具定位验证码在页面中的上下左右边界

3)通过 Image 截取网页中验证码验证码图片并保存

主要实现代码如下:

```
"""导入所需包"""
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from PIL import Image
"""设置浏览器属性"""
option=Options()
option.add_argument('--no-sandbox')
option.add_argument('--disable-dev-shm-usage')
option.add_argument('--start-maximized')#最大化浏览器
option.add_argument('--window-size=1366,768')
option.add_argument('--disable-infobars')#禁用浏览器正在被自动化程序控制的提示
chrome = webdriver.Chrome(executable_path="chromedriver",options=option)
chrome.get(url)
"""定位验证码位置"""
loc=chrome.find_element_by_xpath('//*[@id="vcJpeg"]').location
size=chrome.find_element_by_xpath('//*[@id="vcJpeg"]').size
left=loc['x']
top=loc['y']
right=loc['x']+size['width']
buttom=loc['y']+size['height']
chrome.save_screenshot('webpage.png')#截屏保存
webpageimg=Image.open('webpage.png')
codeimg=webpageimg.crop((left, top, right, buttom))#截取验证码
codeimg.save('code.png')
```

通过上述代码获得类似以下的验证码:

2、验证码图像分析与预处理

初步观察验证码为 RGB 图像，验证码内容由阿拉伯数字和小写字母组成，长度为 4 个字符，字符间偶有重叠，背景噪声与验证码字体明显不同，验证码字符为相近颜色。至此想到对验证码图片像素进行统计分析，思路如下:

1) 遍历每个像素点的 RGB 值，统计每个颜色出现的次数;

2) 为体现像素间的差距对 RGB 进行加权处理，查看加权后的颜色分布;

3) 通过查看颜色分布设定去掉背景噪声的阈值;

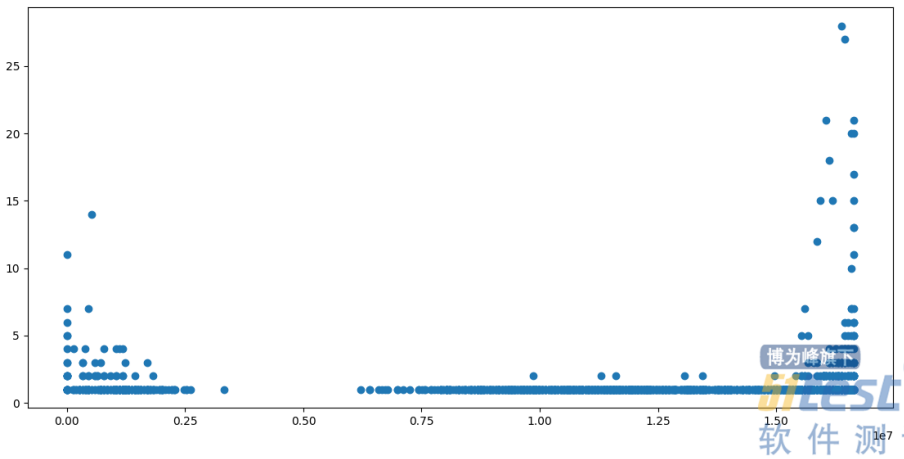
4) 将保留下的像素填充到与原图尺寸一样的白色背景的图片上;

5) 去除边框，最后对所得图像灰度化，二值化。

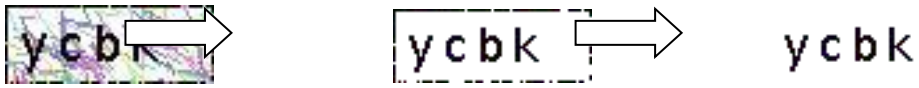
实现代码如下:

```
from PIL import Image
import matplotlib.pyplot as plt
import os
#字典转list
def dict2list(dic:dict):
    keys = dic.keys()
    vals = dic.values()
    lst = [(key, val) for key, val in zip(keys, vals)]
    return lst
code = Image.open(codeimg)#打开图片转成RGB格式
"""分析图像颜色分布"""
newImg = Image.new("RGB", img.size, (255,255,255,255))
w,h = code.size
a = {}
"""统计颜值出现的次数"""
for i in range(w):
    for j in range(h):
        r,g,b = code.getpixel((i,j))
        t = r*255*255 + g * 255 + b
        if t in a.keys():
            a[t] =  a[t] + 1
        else:
a[t]  = 1
l = sorted(dict2list(a), key=lambda x:x[1], reverse=True)
dic = {}
"""画图显示颜色分布"""
plt.figure(1, figsize=(20,5))
plt.scatter(dic.keys(), dic.values(), linewidth=1)
plt.show()
"""新增与原图一样大小的图片，用白色填充背景"""
code_denose = Image.new("RGB", code.size, (255,255,255,255))
for i in range(w):
    for j in range(h):
        if(i==0 or i== w or i==w-1 or j==0 or j==h or j==h-1):
            rmg.putpixel((i,j),(255,255,255))
        r,g,b = img.getpixel((i,j))
        t = r*255*255 + g * 255 + b
        if(t<5000000): #设定阈值为5000000
            code_denose.putpixel((i,j), (r,g,b))
        if(i==0 or i== w or i==w-1 or j==0 or j==h or j==h-1):#去除边框
            code_denose.putpixel((i,j),(255,255,255))
rmg.convert('LA').convert('1').save(code_denose)#灰度化、二值化
```

颜色分布如下，明显看出阈值为 5000000



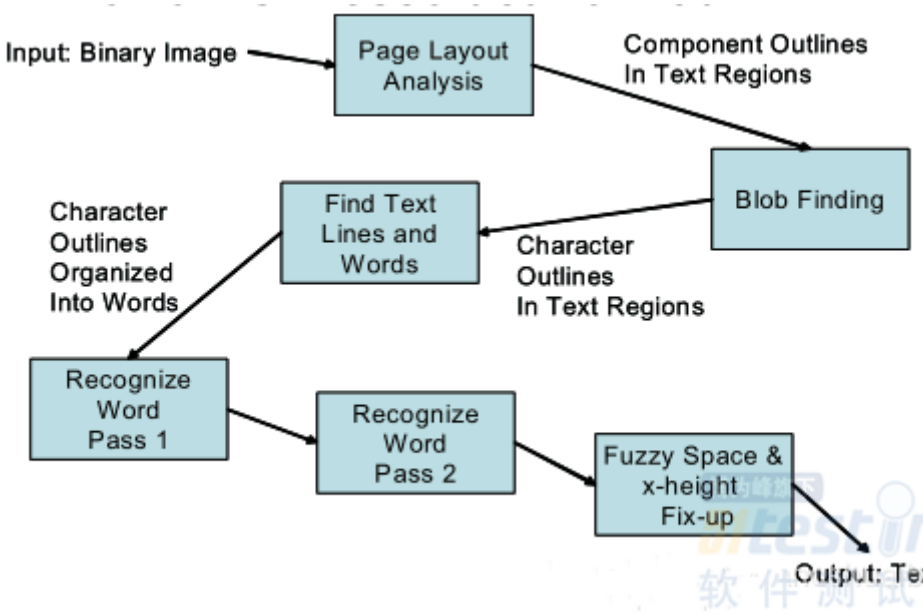
图像处理效果为:



3、Tesseract-OCR 识别识别验证码

OCR 即光学字符识别，是指通过电子设备扫描纸上的打印的字符，然后翻译成计算机文字的过程。也就是说通过输入图片，经过识别引擎，去识别图片上的文字。Tesseract 是一种适用于各种操作系统的光学字符识别引擎，本文通过该识别引擎对预处理后的验证码进行识别。

Tesseract 的识别步骤大致如下:



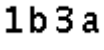
1) 连通区域分析，检测出字符区域区域以及子轮廓。在此阶段轮廓线集成为块区域。

2) 由字符轮廓和块区域得出文本行，以及通过空格识别出单词。固定字宽文本通过字符单元分割出单个字符，对比例文本通过一定的间隔和模糊间隔来分割。

3) 依次对每个单词进行分析，采用自适应分类器，分类器有学习能力，先分析且满足条件的单词也作为训练样本，所以后面的字符识别更准确。

本文首先对单个验证码图片进行识别，Windows 系统进入命令行模式，进入验证码图片所在的目录，运行 tesseract name.png textname（其中 name.png 要被识别的验证码图片文件名，textname 为识别后的验证码字符存放的文件名，默认格式为 TXT 文件）。随

机选取两个，识别结果如下：

验证码图片为  ，识别结果为 1b3a

验证码图片为 ，识别结果为 Inf 8

结果表明验证码中字符间有重叠的情况识别有误，为了统计识别率，手工对 1052 张验证码进行标记，此部分工作需手工完成，需消耗较大人工成本。下载 GitHub 上开发者贡献的语言包，执行以下语句：

pytesseract.image_to_string(code.png,lang='eng',config="-psm7")进行识别，识别结果如下：

```
zl8w : zl8w.png
Bus : zm5u.png
anix : znix.png
zn28 : znz8.png
2p4t : zp4t.png
apql : zpql.png
2pul : zpvi.png
2px8 : zpx8.png
arfx : zrfx.png
zrw6 : zrw6.png
zudb : zudb.png
av)5 : zvj5.png
2whi : zwki.png
zur4 : zwr4.png
2zal : zzal.png
Total pic num:
1052
Crrect Identify num
554
>>>
```

识别正确率为 554/1052=52.7%。

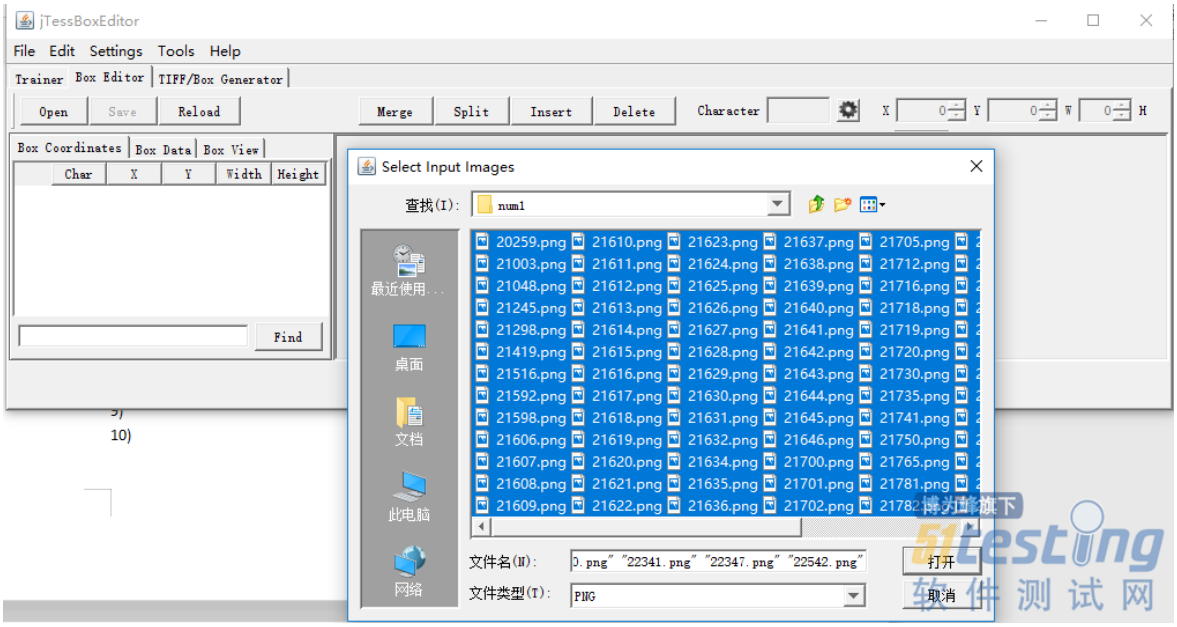
4、Tesseract-OCR 语言包训练

根据上述统计结果，识别正确率不高，因此需要针对验证码图片样本进行训练，提高识别率，通过训练，也可以形成自己的语言库，进而应用于后续自动化测试。

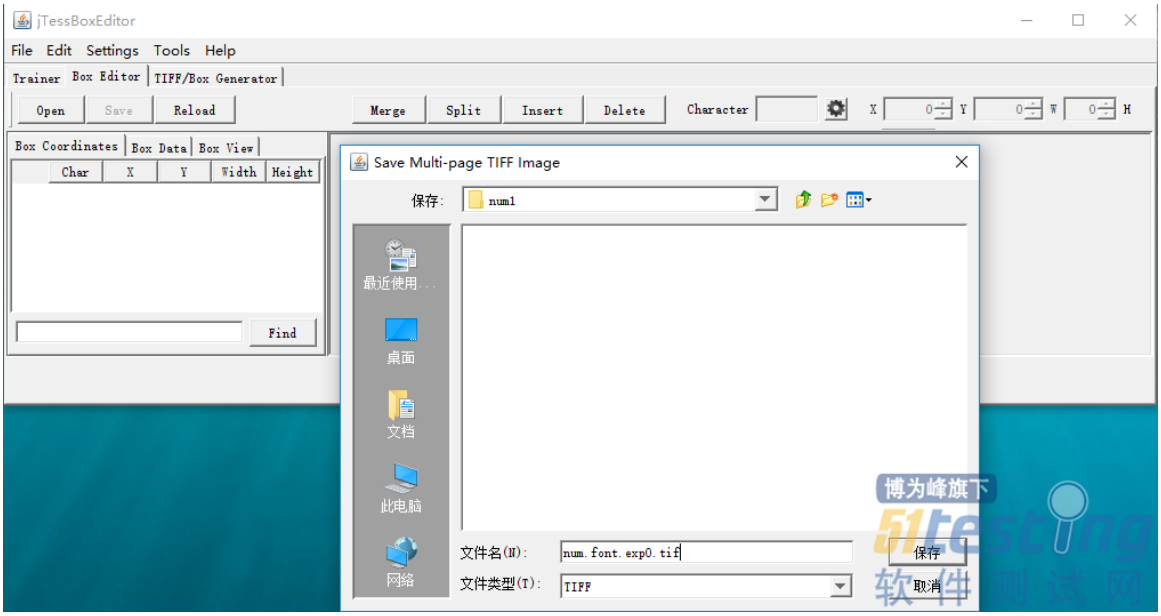
本文采用 jTessBoxEditor 工具训练验证码语言包，训练前需要安装好该软件，该软件为免安装版，解压即可使用。需要特别说明，这个工具是基于 java 虚拟机运行的，需要安装一个 java 虚拟机。本文选取 100 张经过处理的验证码图片进行训练，具体语言包训练过程及图示如下：

1) 进入解压后的 jTessBoxEditor 文件夹，双击里边的 “jTessBoxEditor.jar”，打开该工具；

2) 点击 tools-Merge TiFF 然后选中经过处理的验证码图片，命名为 num.font.exp0.tif 保存;



按住 shift 选中图片



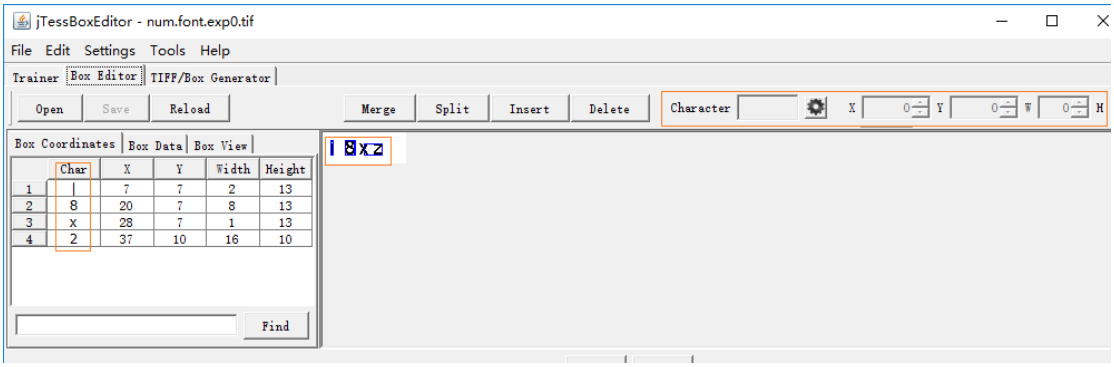
保存合并后的 tif 文件

3) 合并生成后，运行命令行窗口，通过 cd 命令进入 tif 文件所在的文件夹，运行以下命令生成 box 文件;

```
tesseract num.font.exp0.tif num.font.exp0 -psm 7 batch.nochop makebox
```

4) 在 jTessboxedit 中 Box Editor 选项卡中点击 open 按钮打开上一步生成的 tif 文件，若验证码图片中字符的位置和大小定位不准就调整 X、Y、W、H 的值，识

别出来的字符不对就修改字符，对所有验证码图片调整完成后点击 save 保存;



5) 生成训练的库文件，进入到 tif 和 box 所在的文件夹，依次执行以下命令;

tesseract.exe num.font.exp0.tif num.font.exp0 -psm 7 nobatch box.train #使用 tesseract 生成.tr 训练文件

unicharset_extractor.exe num.font.exp0.box #生成字符集文件

echo font 0 0 0 0 0 > font_properties.txt #生成 font_properties 文件

mftraining -F font_properties.txt -U unicharset -O num.unicharset num.font.exp0.tr #生成聚字符特征文件

cntraining.exe num.font.exp0.tr #生成字符正常化特征文件

rename normproto num.normproto #文件重命名

rename inttemp num.inttemp

rename pffmtable num.pffmtable

rename shapetable num.shapetable

combine_tessdata.exe num. #合并训练文件，生成 zwp.traineddata 文件

至此系统验证码语言包已训练完毕。将生成的“num.traineddata”语言包文件复制到 Tesseract-OCR 安装目录下的 tessdata 文件夹中，就可以使用训练生成的语言包进行验证码识别了。

5、训练的语言包识别验证码

执行以下语句对上述 1052 张验证码图片进行识别，识别效果如下，

pytesseract.image_to_string(code.png,lang='num',config="-psm7"):

```
zl8w : zl8w.png
zm5u : zm5u.png
znix : znix.png
znz8 : znz8.png
zp4t : zp4t.png
zpql : zpql.png
zpvi : zpvi.png
sz8 : zpx8.png
zrfx : zrfx.png
zrw6 : zrw6.png
zudb : zudb.png
zvj5 : zvj5.png
zwki : zwki.png
zwr4 : zwr4.png
zzal : zzal.png
Total pic num:
1052
Crrect Identify num
926
>>>
```

识别正确率为 926/1052=88%

6.总结与展望

通过验证码识别的实践，积累了相关的技术，也对相关工作有一定的借鉴作用:

1) 通过对比两次识别的正确率可以看出，针对性的训练特定情况下的样本形成特定的语言库，能够明显提高识别率，训练 100 张验证码图片正确率能够达到 88%，可以推断随着样本数量增加，正确率会继续提高。

2) 本文通过从验证码图像分析到训练语言包，以某系统验证码为例，比较完整的介绍了字符验证码识别过程，对其他系统的验证码识别也有一定的参考价值，对后续的自动化测试开展做了相关的技术准备。

3) Tesseract-OCR 支持多种语言，验证码识别只是它的一部分应用场景，其他的图片转换成计算机文字的相关领域研究、实践也可借鉴本文方法。

参考文献:

https://blog.csdn.net/zbj18314469395/article/details/79954153
https://blog.csdn.net/kangshuaibing/article/details/84333611
https://blog.csdn.net/chouzhou9701/article/details/82587833

简述实施自动化测试的过程

◆作者：祝德慧

摘要：稍具测试规模的项目团队皆想引进自动化测试，然而动手实现自动化测试的团队却不多，未能真正实施的原因多种多样，有扼杀在摇篮里的，有写了后弃之不用。那么是不是所有的业务都适合自动化测试呢？下面就介绍下自己在项目中怎样实施自动化测试。

我所在项目组将自动化测试应用于项目中已经有三年多，自动化脚本应用至今，按每月执行两次的频率，至今已经执行有七十二次有余。由于系统的 GUI 足够稳定，这期间发生了两次由于句柄（hwnd）引发的脚本维护，由于规律性极强，也只花费了十人日来维护所有的脚本，维护脚本的资源和时间成本较低。下面就简单记录下自动化测试实现和运用时的一些要点。

自动化测试引进实施的背景主要有两个方面客户投诉和系统改版。客户投诉的主要原因是系统的数据计算不稳定，发布的新版本对低版本的系统产生的数据计算错误。系统改版需要兼容已发布的系统的数据，保证改版后的数据计算与已发布的系统的数据一致。

自动化测试的实施，最重要的是自动化工具的选择。由于系统的 GUI 是图形界面，不能获取每个单元格的内容，TestComplete、QTP 等考虑过，写了一两个基本功能的脚本，验证后实施起来有困难，TestComplete 的通过图片的形式来进行结果的对比产生的误差比较大，运行的测试结果不稳定；放弃其它测试工具的原因就不一一描述了。最后使用的自动化测试工具是公司的开发人员基于系统框架开发的，提供了一些获取系统单元格和其他数据的函数，保证了自动化测试的顺利进行。

自动化测试工具确定之后，主要是测试脚本的培训和开发。由于自动化测试脚本语言和系统使用的二次开发语言相通，在二次开发人员的帮助下，测试人员很快掌握了测试脚本语言的使用方法。

接下来就是确定自动化测试的范围。如果系统能达到 100%的自动化测试，当然这是

最好的。项目的实际情况是有些功能的 UI 变化大和测试结果数据变化频繁，这些都不在自动化测试脚本开发的范围内。因此，我们最后确定的自动化测试的范围是版本迭代时验证系统的核心功能。

紧接着就开始自动化测试脚本的开发。开发第一个脚本是最费时间的，首先需要确定读取的配置文件格式及参数内容，参照数据的存放形式（Excel、Access、XMl 或其他文件格式），实现的功能脚本，测试结果的存放形式（html、txt、Excel 或其他文件格式）。开发脚本完成后，就是调试，在调试的过程中遇到问题时若三十分钟不能解决，则应向有经验丰富的同事请教，以免耽误脚本开发进度。第 1 个脚本开发完成后，接下来的开发工作可以在第 1 个脚本的基础上进行修改，并将常用的函数存在公共函数文件中，这样会大大提高开发效率。

自动化测试脚本的应用。每次版本迭代时，手工测试完成，需求变更等功能趋于稳定，这时运行自动化测试脚本的时机相对来说比较合理，这样保证了发布的系统的核心功能的准确性和稳定性。

最后要做的事就是自动化测试脚本的维护。系统 GUI 变化，自动化测试范围的增加，这些皆应在系统定版后安排进行，待下一次版本迭代时，用于验证系统功能的正确性。

经过这几年的自动化脚本用于项目的实战经验，我的个人看法是自动化测试并不能完全替代手工测试。每个系统都会发生需求变更，若系统的核心功能不变，则可以将不变的核心功能用自动化测试来替代，其他功能手工完成，这才是有效利用资源和时间的最佳方式，才能获得良好的 ROI。

■用自动化测试工具 Selenium 做电商平台，查看详情>>http://www.atstudy.com/course/1325

开启 iOS Appium 自动化测试之门

◆ 作者：枫 叶

Appium 介绍

Appium 是一个开源、跨平台的测试框架，可以用来测试原生及混合的移动端应用。Appium 支持 iOS、Android 及 FirefoxOS 平台。Appium 使用 WebDriver 的 json wire 协议，来驱动 Apple 系统的 UIAutomation 库、Android 系统的 UIAutomator 框架。Appium 对 iOS 系统的支持得益于 Dan Cuellar’ s 对于 iOS 自动化的研究。Appium 也集成了 Selendroid，来支持老 Android 版本。

Appium 进行自动化测试的两个好处

(1) Appium 在不同平台中使用了标准的自动化 APIs，所以在跨平台时，不需要重新编译或者修改自己的应用。这里，跨平台的意思是指可以在不同的系统上用相同的方式编写测试脚本，而不是指用于 Android 系统的测试脚本可以完全不用修改的应用于 iOS 系统上。事实上，Android 和 IOS 应用几乎需要独立的进行编写。

(2)Appium 支持 Selenium WebDriver 支持的所有语言，如 java、Object-C、JavaScript、PHP、Python、Ruby、C#、Clojure，或者 Perl 语言，更可以使用 Selenium WebDriver 的 Api。Appium 支持任何一种测试框架。如果只使用 Apple 的 UIAutomation，我们只能用 javascript 来编写测试用例，而且只能用 Instruction 来运行测试用例。同样，如果只使用 Google 的 UIAutomation，我们就只能用 java 来编写测试用例。

Appium 的系统需求

Android 自动化测试可以在 Windows、Mac、Linux 上进行，需要安装 Android SDK、Node 等工具。而 iOS 的自动化由于需要 Xcode 的支持，只能在 Mac 上运行，需要安装 Xcode、Node 等工具。此外，由于 Appium ios 自动化的底层使用的是 UI

Automation，因此在使用 Appium 之前必须搭建 iOS 开发环境。

一、搭建 Appium 环境

目前 Appium 测试 iOS 设备，要求 Mac 操作系统的最低版本是 mac OS 版本 10.7，本机使用的开发环境是 Xcode 9.4.1, Mac x 10.13.3。由于时间有限下文截图可能不太清晰，请谅解。

1) 安装 brew

在终端输入命令 /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

2) 安装 libimobiledevice

brew install libimobiledevice --HEAD

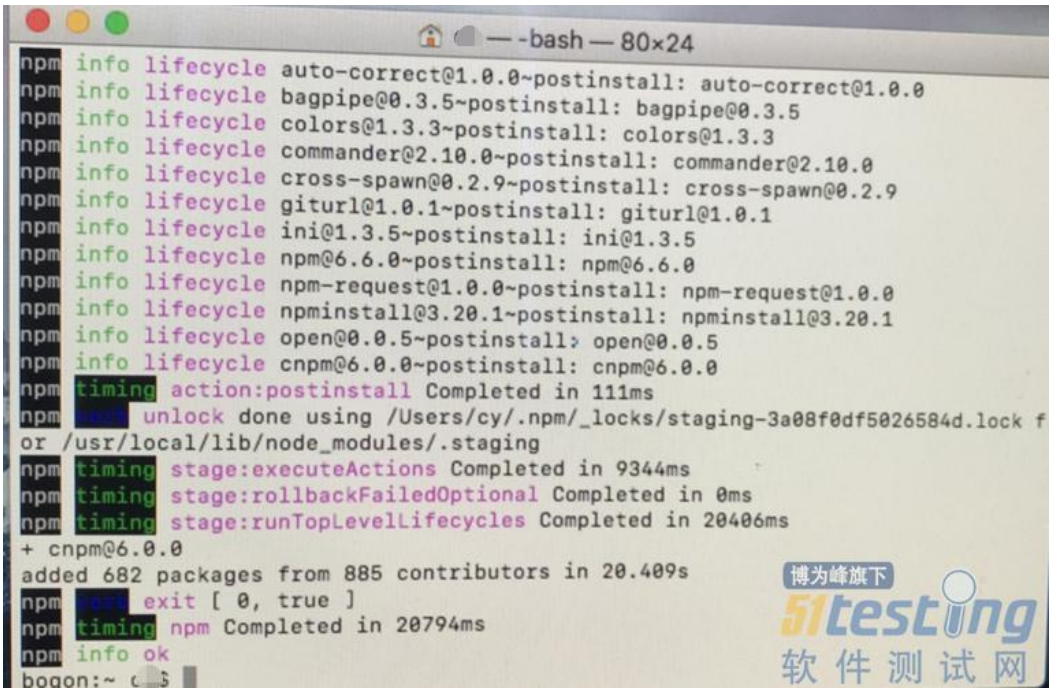
3) 安装 carthage

brew install carthage

4) 安装 node.js , https://nodejs.org/en/download/，下载.pkg 文件安装

5) 安装 cnpm https://npm.taobao.org/

npm install -g cnpm --registry=https://registry.npm.taobao.org



6) 安装 ios-deploy

sudo cnpm install -g ios – deploy

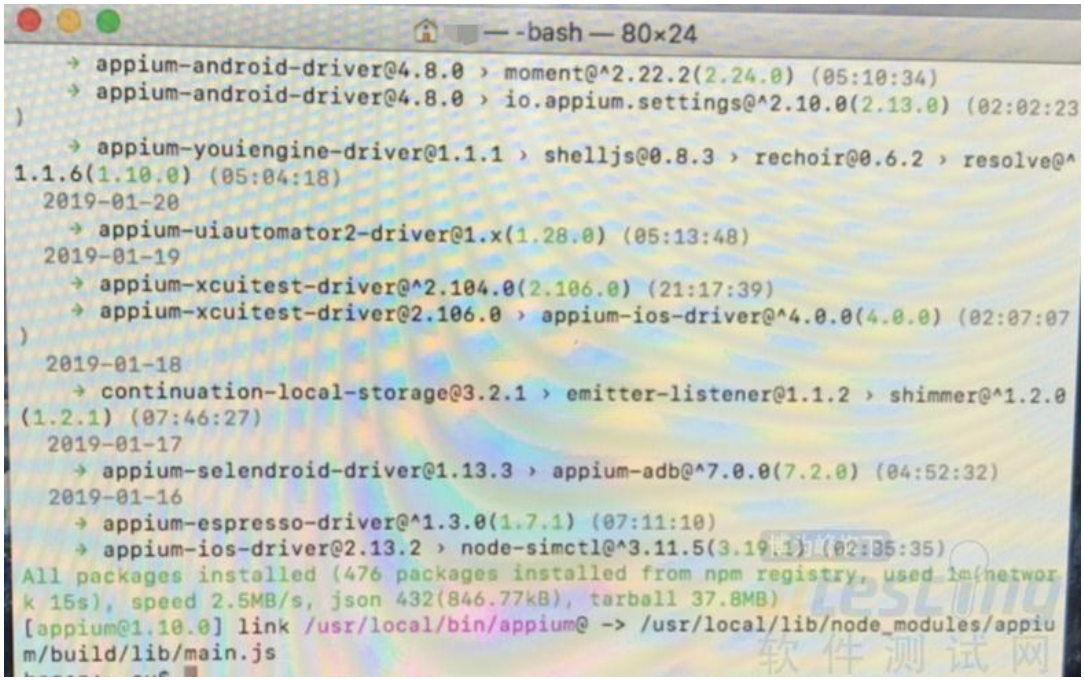
7) 安装 xcpretty

gem install xcpretty

安装的版本为 xcpretty-0.3.0

8) 安装 Appium1.10.0

sudo cnpm install -g Appium@1.6.3



9）安装 Appium-xcuitest-driver 依赖

（1）安装 WebDriverAgent

首先要搭建 WebDriverAgent 编译环境，首先需要安装如下的软件:

- Homebrew
- carthage
- python
- node.js
- Xcode8.0+（IOS9.3，Xcode8.0+才能正常编译）

安装 Homebrew

/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

安装 WebDriverAgent

使用 git clone 命令将 WebDriverAgent 项目克隆到本地

git clone https://github.com/facebook/WebDriverAgent

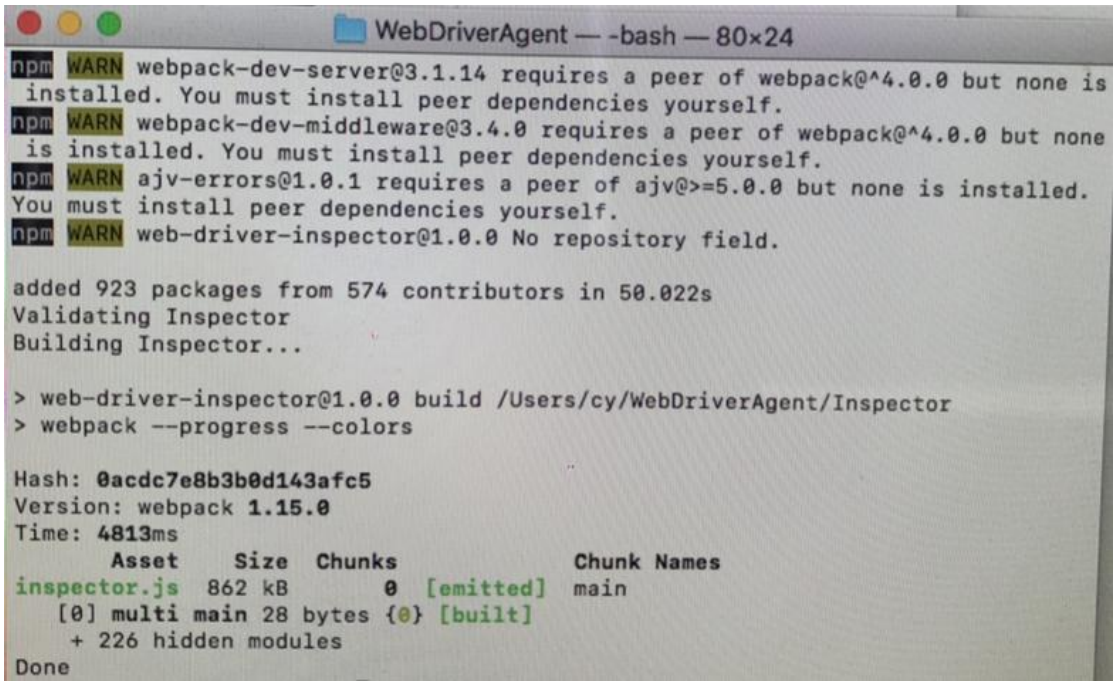
（2）安装 Appium-xcuitest-driver 依赖，进入 WebDriverAgent 安装目录，运行 bootstrap

cd /usr/local/lib/node_modules/Appium/node_modules/Appium-xcuitest-driver/WebDriverAgent

mkdir -p Resources/WebDriverAgent.bundle //执行脚本

sh ./Scripts/bootstrap.sh

如果出现报错，则关闭终端再打开，执行



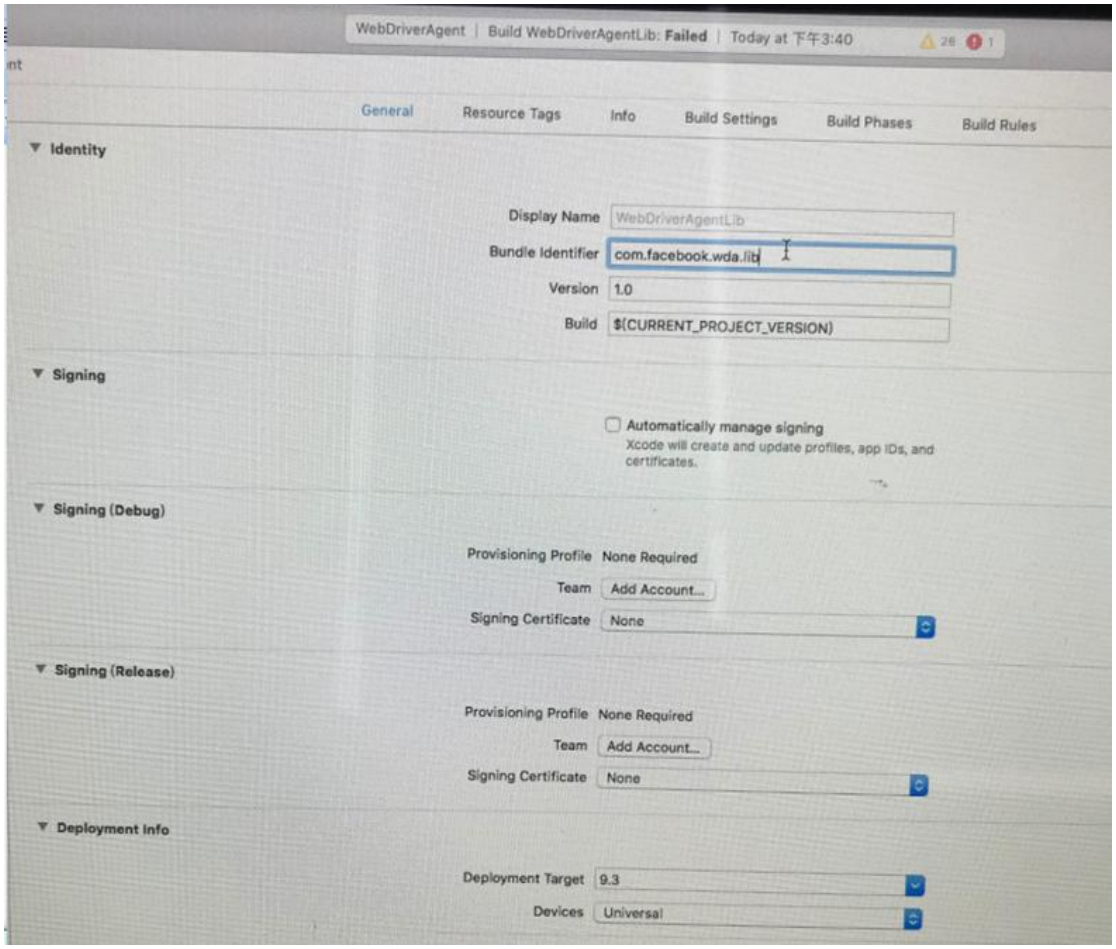
9) 下载 WebDriverAgent-master

（1）Xcode 打开 WebDriverAgent.xcodeproj，修改配置:

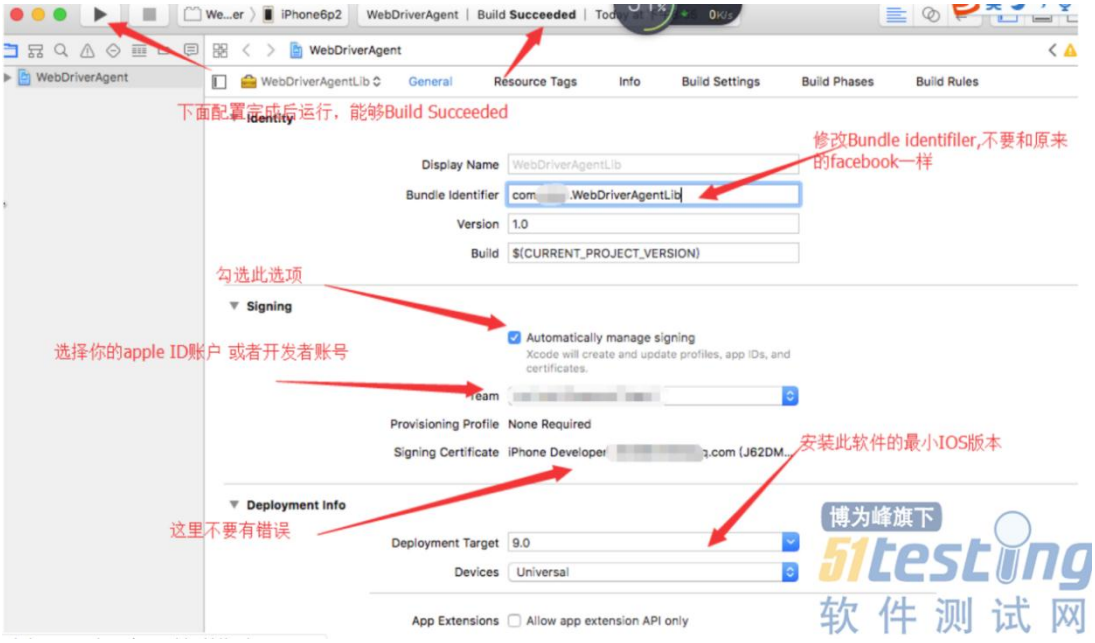
选择菜单 files->open，路径

/usr/local/lib/node_modules/Appium/node_modules/Appium-xcuitest-driver/WebDriverAgent

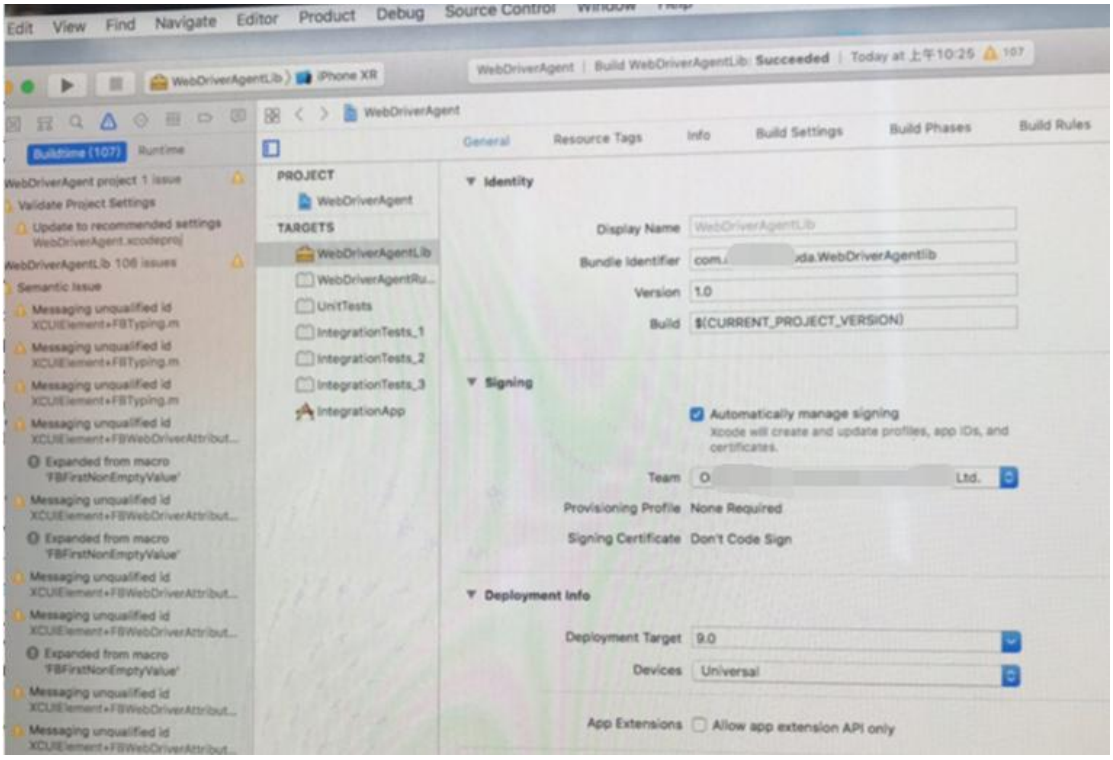
（2）双击 WebDriverAgentLib，设置后进行编译



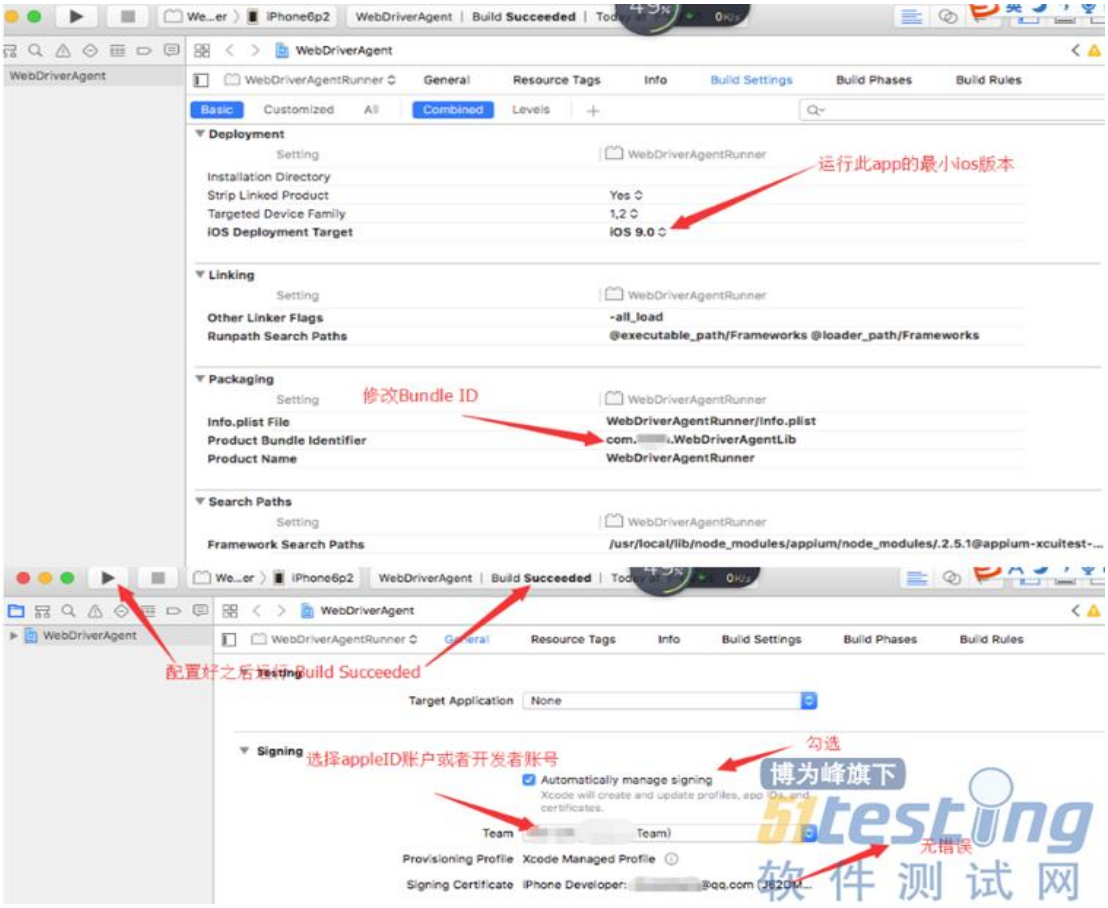
按如下进行修改



Bundle ID 改为 com.ming.wda.WebDriverAgentLib

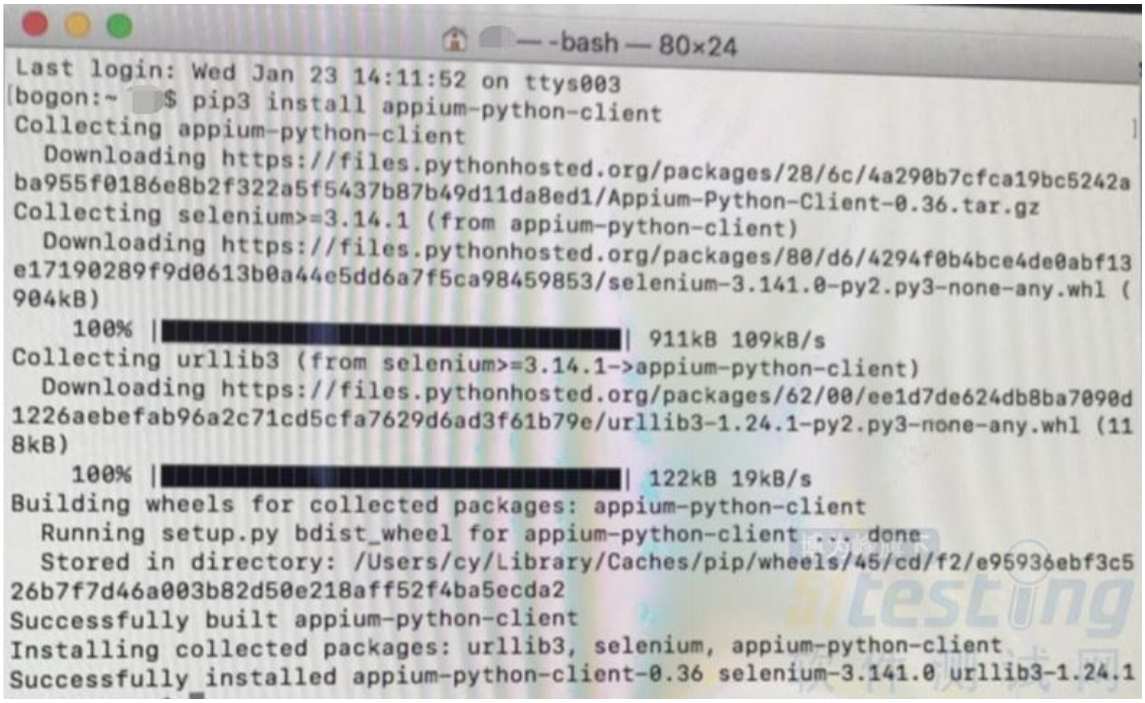


修改配置 WebDriverAgentRunner 后编译



10）真机的 udid 通过 iTunes—摘要，点击序列号，出现 UDID，右键拷贝即可。

安装 Appium-python-client



二、运行与测试

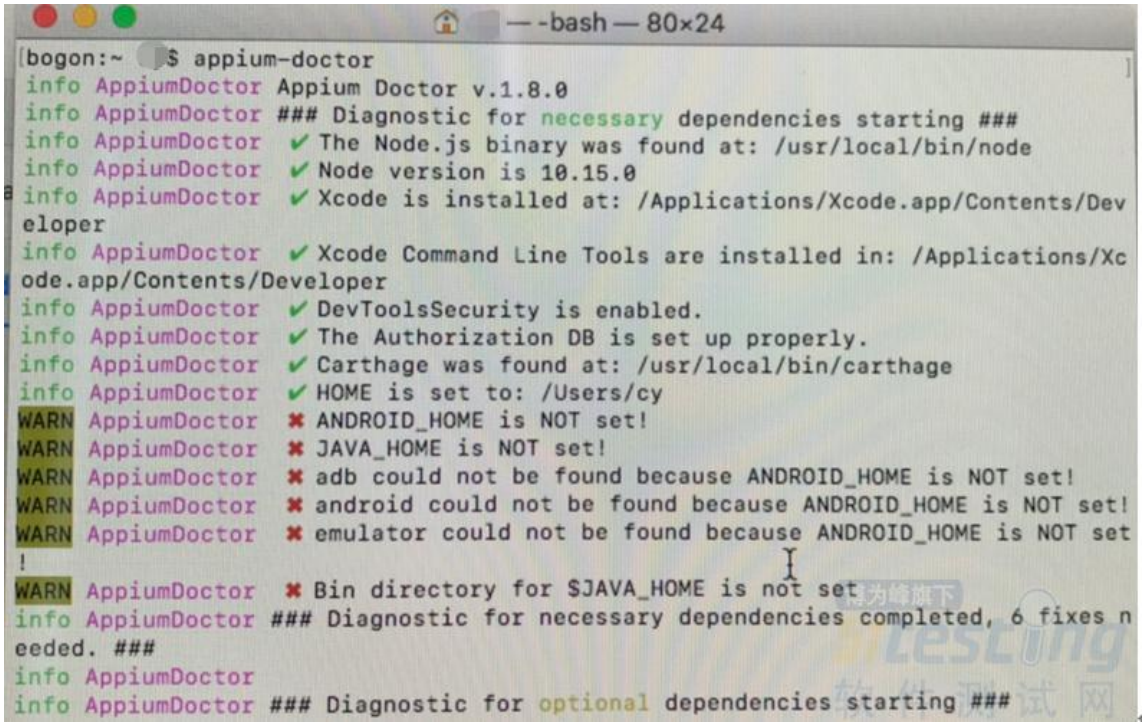
(1) Xcode 菜单栏选择目标设备，Scheme 选择 WebDriverAgentRunner，最后运行 Product -> Test。一切正常的话，手机上会出现一个无图标的 WebDriverAgent 应用，启动之后，马上又返回到桌面。这是正常的。

此时控制台界面可以看到设备的 IP。如果看不到的话，使用这种方法打开 view-debug area-activate console

出现上图，表示成功了

(2) 安装 Appium-doctor

确定所有依赖是否安装成功，可通过 Appium-doctor 验证，首先安装 Appium-doctor（sudo npm install -g Appium-doctor），然后在终端运行 Appium-doctor，如下图，都是打勾状态就证明环境正常



(3) brew install – HEAD libimobiledevice

(4) 运行 Appium

终端执行 Appium – p 4723

三、启动 APP

第一步，Xcode 打开 WebDriverAgentRunner，scheme 选择它，菜单 Project->Test，build 成功后在手机里装上 WebDriverAgent；第二步，手机连接 Mac 电脑；第三步，启动 Appium desk，start desired session;

就可以自动测试 App 了。

往期作品:《安卓 Appium 自动化测试实践》

如何保证提测质量?

◆作者：多则惑少则明

摘要：提测质量经常由于开发人员经验不足、项目太复杂、开发周期紧张、开发人员自测不足等原因，导致提测后要么主流程不通、要么实现与需求不一致，给后续 QA 同学测试带来很大"麻烦"。抛开每个业务的客观原因不谈，有没有什么事情可以在提测之前做，来预防提测质量差的事情发生呢？下面就自己一些经验和感悟，聊聊自己的一些体会。

一、背景

要说 QA 同学经常比较苦恼的一件事，莫过于提测质量了。因为提测质量的好坏，大大影响了测试质量、以及需要的测试时间、风险控制等。测试人员对项目提测质量的终极目标是，让提测质量直接等于上线质量，这样 QA 可以将最多的精力用于验证是否有更深层次的问题。虽然这只是理想，但实际项目中，提测质量不仅没有达到提测要求，甚至影响阻塞了项目的后续流程。举个例子：业务 A 提测质量极差，主流程都不通。这个时候，QA 通常的做法是执行提测打回流程，让开发同学继续自测；硬着头皮不打回，边催促开发同学修复 bug,边继续测试。

但无论 QA 同学怎么做，提测质量差已经成为事实了，这是任何团队成员都不想看到的结果。这个时候，对于上线，通常会考虑几种做法:

需求必须按时上线。这个时候，QA 同学只好加班加点，或压缩测试工时，来确保上线

需求可以 delay 上线。这时，QA 同学需要重新协调各方时间，甚至遇到和其他需求冲突的情况，需要协调其他需求测试的时间了。

但无论如何，提测质量差已经成为了事实，上述亡羊补牢的做法都属于事后补救了。那有什么可以事前做的吗，可以提前预防类似的提测质量差的情况发生呢？下面就自己在实际项目中的经验，说说自己的一点理解。

二、什么影响了提测质量?

结合以往项目经验，开发提测时的提测质量同时受多个因素的影响:

1、项目的规模。这里可以理解为，项目的规模越大，通常合作的人越多、需要考虑的逻辑越多/越复杂；项目的规模越小，相比之下，开发人员需要考虑的逻辑也比较容易把控。因而，规模较大的项目的提测质量，通常比小规模项目的提测质量要差一些。

2、开发同学的自身项目经验。日常的项目跟进中，对于类似的一些需求，通常经验丰富的开发同学的提测质量会好很多。

3、涉及业务的复杂程度。如果一个业务涉及的模块较多，关联的第三方业务较多，那么无疑这个业务修改时，出问题的概率也会越大。

4、开发周期的长短。开发周期通常需要技术团队来评估，但如果为了赶项目，而压缩开发周期时，通常会给提测质量留下隐患。

上面的因素会直接或间接影响提测质量，但由于这些因素属于客观因素，每个团队、项目都会有所不同，且不太受人为因素的影响，因而这些因素不是本文讨论的重点。那有哪些事情是可以提前做的，来确保提测质量呢?

三、确保提测质量该如何做?

确保提测质量的目标是在提测时，确保项目达到一定的质量水平。因而，抛开那些影响提测质量的客观因素外，我们需要做些事情，来确保提测时的质量水平，让提测质量尽可能逼近上线质量

下面是在实际的项目中，曾经使用过的一些方法，共大家参考:

1、健康的项目流程。这里的项目流程，不仅仅是提测流程，也包括需求评审、测试用例评审、技术评审、风险分析等等，确保项目有任何实现风险、提测风险时，能快速周知到项目成员，方便大家一起齐心协力解决，而不是一个问题的 block，直接阻塞的项目的顺利进行。

2、合理的需求评审。这里的需求评审的目标，是力求项目各个角色成员对需求的理解一致、对需求的风险点、影响范围等理解无歧义。避免技术实现与需求不符的情况发生。

详细的用例评审。这里的用例评审，是为了确保场景覆盖，以及再次确认项目各个角色成员对各个需求点理解一致

3、技术评审。QA 同学与开发同学一起技术评审，可以提前预防/发现技术设计、技术实现方面的问题

4、静态代码扫描。确保代码的实现规范符合团队要求，保证代码的长期可维护性

适时的人工 Code Review。这里可以针对代码实现逻辑、规范做具体检查，提前发现问题

5、明确的冒烟用例。冒烟用例是提测前，需要开发同学必须自测通过的用例。冒烟用例最好自动化进行验收，避免因为开发同学执行的问题，导致测试执行不彻底

6、公认的提测流水线。提测时，可以将静态代码扫描、自动化冒烟测试、自动化主流程回归测试进行集成，让开发同学提测前进行自检。

7、QA 提前介入测试。QA 提前测试本质上已经进行测试了，但由于尚未正式/全面进行测试阶段，因而一定程度上，也可以确保提测质量

当然了，上面的方法，在实际的项目中，可以结合项目的特点，选择其中的几种来重点推进。在推进的过程中，可以通过相关数据，来说明提测质量的提升情况。具体参考的质量指标，可以根据项目需要，和团队成员一起共同制定。

四、写在最后

测试人员对项目质量推动时，就是让提测质量逐渐靠近上线质量，这是最为理想的状态了。但由于客观因素的存在，如开发同学自身经营、项目规模大小等，项目的提测质量往往不尽如人意。如果团队成员、QA 同学不采取任何预防，听之任之，那么提测质量就完全不能提前控制了。由于测试阶段属于项目的后期阶段了，那么不达标的提测质量，势必会严重干扰测试、上线步调了。

因此，提测质量直接影响了测试质量，进而也间接影响了上线质量。提测质量的优劣，抛开不可控的问题不说，需要团队成员共同努力，来尽可能高的提升提测质量。

往期作品:《测试人员不得不小心那些职场套路》

Python 接口测试实战

◆ 作者：风清扬

引言：年初参与到一个后台系统开发的项目中，里面涉及了很多接口，我做为项目组测试人员，需要对这些接口进行测试，一开始使用 postman 工具测试，很是方便。但随着接口数量的增加，不光要执行手动点击测试，而且，一旦接口参数变动，都重新更改接口参数，次数多了，使得测试效率严重下降。

后来我将目光转向了自动化测试，考虑到项目组对接口质量要求很高，需要快速开发。最终选定 python 作为脚本开发语言，使用其自带的 requests 和 urllib 模块进行接口请求，使用优化后的 unittest 测试框架编写测试接口函数，测试结果选用 HTMLTestRunner 框架予以展示，并使用 python 的 ssl 模块支持 https 协议的验证。接下来，本文就章节详细地介绍这些模块，并给出各个模块完整的测试代码。

1、接口请求

python 特别是 python 3.x 中的 urllib 和 requests 模块，是用来请求 url 的两个主要模块。这两个模块中，如果仅仅是支持 http 协议的 url 请求，推荐使用 requests 模块。为什么这么说呢？因为爱因斯坦说过一句话：简洁就是美。requests 模块对 urllib 模块又做了一层封装，使用更加方便。该模块支持 GET, POST, PUT, DELETE 等请求方法。请求返回信息包含状态码和消息体，状态码用三位数字表示，消息体可用字符串，二进制或 json 等格式表示。下面用一个例子来介绍一下 requests 模块的使用。代码如下：

```
import requests
def get_method(url, para, headers):
  try:
        req = requests.get(url=url, params=para, headers=headers)
  except Exception as e:
        print(e)
      else:
           if req.status_code == "200":
```

```
            return req
        else:
            print("Requests Failed.")

if __name__=='__main__':
  url = "http://www.google.com"
  req = get_method(url=url, para=None, headers=None)
  print(req.status_code)
    print(req.text)
```

输出为:

```
200
<!DOCTYPE html>
<!--STATUS OK--><html> <head><meta...（省略）
```

上述程序输出状态码为 200，表明请求成功，返回消息体为网页内容。这里我仅对 requests 模块中的 get 请求方法做了封装，其它方法（如 post，put，delete 等）的封装类似。当让你也可以不用封装，直接使用 requests.methodName 来直接调用该方法。这里提醒一句，在实际的接口测试中，headers 和 data 都是有值的，要确保这些值的填写正确，大部分请求下的请求失败或返回结果错误，基本上都是由于这些值的缺失或错误造成的。更多关于 requests 模块的介绍，请参考官方文档。

2、测试框架优化

unittest 是 python 中进行单元测试使用广泛的框架，其与 java 中的单元测试框架 junit 类似。该框架使用简单，需要编写以 test 开头的函数，选择 unittest 框架运行测试函数，测试结果在终端显示。这里举一个简单的例子:

```
import unittest

class ApiTestSample(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass

    def jiafa(self, input01, input02):
        result = input01 + input02
        return result

    def test_jiafa(self):
```

```
            testResult = self.jiafa(input01=4, input02=5)
            self.assertEqual(testResult, 9)

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

简单解释下这段代码，首先我们创建一个类 ApiTestSample，这个类继承自 unittest.TestCase 类。然后在这个类中写了 jiafa 函数，它有两个参数 input01，input02，返回 input01 与 input02 相加的和。接着在 test_jiafa 方法中，我们对刚才 jiafa 函数进行了和值校验。通过给 jiafa 输入两个值，获取其函数返回值，并与真实值做相等判断，以此实现函数单元测试。这里用到了 unittest 中断言值相等的 assertEqual(m, n)函数，上述代码运行结果如下:

```
Ran 1 test in 0.000s
OK
```

以上是 unittest 框架最基本的单元测试应用，但是这个框架有个缺陷，就是不能自己传入参数。对于接口来说，往往需要传入很多参数，并且这每个参数又有很多取值，如果不对原先的 unittest 框架做改变，不仅无法用来进行接口测试，而且一个个结合参数取值去写测试代码，工作量极其庞大，也没有实现测试数据与脚本没有分离。基于此，我们对该框架做出一下两点优化。

- 扩展 unittest.TestCase 类，支持自定义参数输入;
- 测试数据与测试脚本分离，测试数据存储在文件和数据库中，以增强测试脚本复用性;

以下是对 unittest.TestCase 类的扩展，使其支持参数化把参数加进去。下面是具体的代码实现过程:

```
class ExtendTestCaseParams(unittest.TestCase):

    #扩展 unittest.TestCase 类，使其支持自定义参数输入

    def __init__(self, method_name='runTest', canshu=None):
        super(ExtendTestCaseParams, self).__init__(method_name)
        self.canshu = canshu

  #静态参数化方法
    @staticmethod
    def parametrize(testcase_klass, default_name=None, canshu=None):
```

```
        """ Create a suite containing all tests taken from the given
            subclass, passing them the parameter 'canshu'
        """
        test_loader = unittest.TestLoader()
        testcase_names = test_loader.getTestCaseNames(testcase_klass)
        suite = unittest.TestSuite()
        if default_name != None:
            for casename in testcase_names:
                if casename == defName:
                    suite.addTest(testcase_klass(casename, canshu=canshu))
        else:
            for casename in testcase_names:
                suite.addTest(testcase_klass(casename, canshu=canshu))
        return suite
```

这里，canshu 就是优化后加的自定义参数，参数类型可以是元组或列表。下面使用这个参数化类来改写之前的代码。

```
class ApiTestSample(ExtendTestCaseParams):

    def setUp(self):
        pass
    def tearDown(self):
        pass

    def jiafa(self, input01, input02):
        result = input01 + input02
        return result

    def test_jiafa(self):

        input_01 = self.param[0]
        input_02 = self.param[1]
        expectedResult = self.param[2]
        result = self.sub(input_01, input_02)
        print(result)
        self.assertEqual(result, expectedResult)

if __name__=='__main__':
    testData = [
                (10, 9, 19),
                (12, 13, 25),
                (12, 10, 22),
                (2, 4, 6)
               ]
```

```
        suite = unittest.TestSuite()
        for i in testData:
            suite.addTest(ExtendTestCaseParams.parametrize(ApiTestSample, 'test_jiafa', canshu=i))
        runner = unittest.TextTestRunner()
        runner.run(suite)
```

执行结果如下:

```
....
## 19
25
Ran 4 tests in 0.000s
22
6
OK
```

通过对 unittest 框架优化，我们实现了 unittest 框架的参数化，这样就可以用于接口测试了。虽然我们实现了参数化，但是测试结果的展示不够直观，这个时候需要一个可视化页面来直接显示测试结果。所幸的是，python 中有专门展示测试结果的框架: HTMLTestRunner。该框架可以将测试结果转换为 HTML 页面，并且该框架可以和 unittest 框架完美的结合起来。接下来我们讲述一下 HTMLTestRunner 框架的使用。

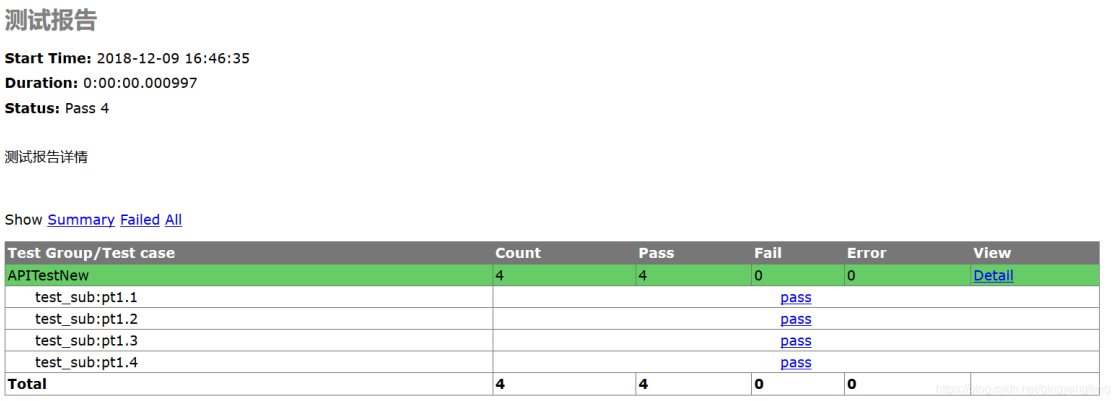
3、测试结果可视化

HTMLTestRunner 框架可用来生成可视化测试报告，并能很好的与 unittest 框架结合使用，接下来我们以一段代码来展示一下 HTMLTestRunner 的使用。

```
if __name__=='__main__':
  from HTMLTestRunner import HTMLTestRunner
  testData = [
                    (10, 9, 19),
                    (12, 13, 25),
                    (12, 10, 22),
                    (2, 4, 6)
                   ]
  suite = unittest.TestSuite()
  for i in testData:
     suite.addTest(ExtendTestCaseParams.parametrize(ApiTestSample, 'test_jiafa', canshu=i))
  currentTime = time.strftime("%Y-%m-%d %H_%M_%S")
  result_path = './test_results'
  if not os.path.exists(path):
       os.makedirs(path)
```

```
report_path = result_path + '/' + currentTime + "_report.html"
reportTitle = '测试报告'
desc = u'测试报告详情'
    with open(report_path, 'wd') as f:
        runner = HTMLTestRunner(stream=f, title=reportTitle, description=desc)
        runner.run(suite)
```

测试结果如下:



测试报告

Start Time: 2018-12-09 16:46:35
Duration: 0:00:00.000997
Status: Pass 4

测试报告详情

Show Summary Failed All

Test Group/Test case	Count	Pass	Fail	Error	View
APITestNew	4	4	0	0	Detail
test_sub:pt1.1			pass		
test_sub:pt1.2			pass		
test_sub:pt1.3			pass		
test_sub:pt1.4			pass		
Total	**4**	**4**	**0**	**0**	

下面详细讲解一下 html 报告的生成代码:

runner = HTMLTestRunner(stream=fp, title=reportTitle, description=desc)

HTMLTestRunner 中的 stream 表示输入流，这里我们将文件描述符传递给 stream，title 参数表示要输出的测试报告主题名称，description 参数是对测试报告的描述。在使用 HTMLTestRunner 时，有几点需要注意:

- HTMLTestRunner 模块非 Python 自带库，需要到 HTMLTestRunner 的官网下载该安装包;
- 官网的 HTMLTestRunner 模块仅支持 Python 2.x 版本，如果要在 Python 3.x 中，需要修改部分代码，修改的代码部分请自行上网搜索;

如果需要生成 xml 格式，只需将上面代码中的

```
runner = HTMLTestRunner(stream=fp, title=reportTitle, description=desc)
runner.run(suite)
```

修改为如下代码

```
import xmlrunner
```

```
runner = xmlrunner.XMLTestRunner(output='report')
runner.run(suite)
```

4、接口测试分类

通过前面 3 节的讲解，大家对接口请求，测试框架和测试结果可视化方面有了深入的了解。有了前面的基础，对于接下来理解和编写接口测试会有很大帮助。这里我们先来讲解一下接口测试与单元测试的区别。单元测试只针对函数进行多组参数测试，包括正常和异常参数组合。而接口测试是针对某一接口进行多组参数测试。实际接口测试中，我们又将接口测试分为两种:

- 单接口测试;
- 多接口测试。

对于单接口测试，只需针对单个接口测试，测试数据根据接口文档中的参数规则来设计测试用例；对多接口测试，首先要确保接口之间调用逻辑正确，然后再根据接口文档中的参数规则来设计用例进行测试。下面我就根据这两种不同情况的接口测试，用实际项目代码展示一下。

4.1 单接口测试

```
class TestApiSample(ExtendTestCaseParams):
    def setUp(self):
        pass
    def tearDown(self):
        pass

    def register(self, ip, name, desc):
        url = 'http://%s/api/v1/reg' % ip
        headers = {"Content-Type": "application/x-www-form-urlencoded"}
        para = {"app_name": name, "description": desc}
        req = self.Post(url, para, headers )
        return req

    def test_register(self):

        for index, value in enumerate(self.param):
            print('Test Token {0} parameter is {1}'.format(index, value) )
        self.ip = self.param[1]
        self.name = self.param[2]
        self.desc = self.param[3]
```

```
            self.expectedValue = self.param[4]
            req = self.grant_register(self.ip, self.name, self.desc)
            self.assertIn(req.status_code, self.expectedValue, msg="Test Failed.")

    if __name__=='__main__':
        import random
        import string
        ip = '172.36.17.108'
        testData = [
            (1, ip, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '', 200),
            (2, ip, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '', 200),
            (3, ip, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '',   200)
        ]
        suite = unittest.TestSuite()
        for i in testData:
            suite.addTest(ExtendTestCaseParams.parametrize(TestApiSample, 'test_register', canshu=i))
        currentTime = time.strftime("%Y-%m-%d %H_%M_%S")
        path = './results'
        if not os.path.exists(path):
            os.makedirs(path)

        report_path = path + '/' + currentTime + "_report.html"
        reportTitle = '接口测试报告'
        desc = u'接口测试报告详情'
        with open(report_path, 'wd') as f:
            runner = HTMLTestRunner(stream=f, title=reportTitle, description=desc)
            runner.run(suite)
```

上述代码中的 register()为注册接口函数，test_register()为测试注册接口函数，testData 为测试数据，这里没有完全做到测试脚本与测试数据分离。为了实现测试数据与测试脚本分离，可以将 testData 列表单独写在文本文件或者数据库中，运行测试脚本时再去加载这些数据，就能实现测试脚本与测试数据的分离。

4.2 多接口测试

```
class TestApiSample(ExtendTestCaseParams):
    def setUp(self):
        pass
    def tearDown(self):
        pass

    def register(self, ip, name, desc):
        url = 'https://%s/api/v1/reg' % ip
        headers = {"Content-Type": "application/x-www-form-urlencoded"}
```

```
            para = {"app_name": name, "description": desc}
            req = self.Post(url, para, headers )
            return req

        def oauth2_basic(self, ip, name, desc):
            apps = self.register(ip, name, desc)
            apps = apps.json()
            url = 'http://%s/api/v1/basic' % ip
            data = {"client_id":apps['appId'], "client_secret":apps['appKey']}
            headers = None
            req = requests.post(url, data, headers)
            basic = str(req.content, encoding='utf-8')
            return apps, basic, req

        def test_oauth2_basic(self):

            count = 0
            for i in self.param:
                count += 1
            self.ip = self.param[1]
            self.name = self.param[2]
            self.desc = self.param[3]
            self.expected = self.param[4]
            apps, basic, req = self.oauth2_basic(self.ip, self.name, self.desc)
            self.assertIn(req.status_code, self.expected, msg="Grant Failed.")

    if __name__=='__main__':
        import random
        import string
        ipAddr = '172.36.17.108'
        testData = [
            (1, ipAddr, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '', 200),
            (2, ipAddr, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '', 200),
            (3, ipAddr, ''.join(random.sample(string.ascii_letters + string.digits, 7)), '',    200)
        ]
        suite = unittest.TestSuite()
        for i in testData:
            suite.addTest(ExtendTestCaseParams.parametrize(TestApiSample, 'test_oauth2_basic',
    canshu=i))
        currentTime = time.strftime("%Y-%m-%d %H_%M_%S")
        path = '../Results'
        if not os.path.exists(path):
            os.makedirs(path)

        report_path = path + '/' + currentTime + "_report.html"
```

```
    reportTitle = '接口测试报告'
    desc = u'接口测试报告详情'
    with open(report_path, 'wd') as f:
        runner = HTMLTestRunner(stream=f, title=reportTitle, description=desc)
        runner.run(suite)
```

上述代码中，我们对两个接口进行了函数封装，两个接口之间有依赖关系，oauth2_basic()函数在请求之前必须先去请求 register()函数获取数据。对于这种多接口测试，且接口之间存在互相调用的情况，最好是在调用该接口前时，将互相之间有依赖的接口封装进该接口中，保证接口调用逻辑一致。其次再针对该接口的其它参数设计测试用例去测试该接口。

5、https 协议请求

前面我们提及的接口测试，仅是关于请求 http 协议的。然而，http 协议在传输过程中并不安全，通过该协议传输内容容易被截取，由此人们提出了 https 协议。该协议在原先的 http 协议之外，对传输过程中的内容进行了加密处理，这样就能确保信息在传输过程中的安全。目前很多公司的访问 url 都已转换到 https 协议。因此在接口测试中也要考虑到对 https 协议访问的支持。目前对于 https 协议访问的处理有以下几种方案。

- 第一种，对于一般网站访问，无法获得支持 https 协议的证书信息，因此只能选择忽略 ssl 校验;
- 第二种，对于外部网络访问公司内容网络和内容来说，除了要经过防火墙外，访问具体业务要经过负载均衡器。而负载均衡器一般要求支持 https 协议，这个时候就需要使用 Python 中的 ssl 模块对证书进行校验;

关于忽略访问 https 协议的证书校验，这里忽略不表。重点讲解 https 协议证书的校验。在 Python 中，提供了 ssl 模块，用于对 https 协议证书的认证。这里以一段代码来展示该模块的应用。

```
import ssl
cont = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
cont.check_hostname = False
cont.load_cert_chain(certfile=public_key, keyfile=private_key)
cont.verify_mode = 2
cont.load_verify_locations(ca_key)
```

上述代码中先生成 ssl 上下文对象 cont，接下来用这个上下文对象 cont 依次进行域名校验、证书导入、验证模式选择及 CA 证书验证。cont.checkhostname 用于域名校

验，值为 True 表示进行主机名校验，值为 False 表示不进行主机名校验。cont.loadcertchain(certfile=publickey, keyfile=privatekey)，certfile 表示导入公钥证书，keyfile 表示导入私钥证书。一般情况下，Python 支持的 certfile 证书文件后缀为.crt，keyfile 证书文件后缀为.pem。cont.verifymode 为验证模式，值为 0 表示不做证书校验，值为 1 表示代表可选，值为 2 表示做证书校验。cont.loadverifylocations(ca_key)表示导入 CA 证书。一般的证书校验都要经过上述这几个步骤。此时 ssl 证书的基本配置已完成。接下来就需要在发送 https 请求时加入证书验证环节，示例代码如下:

```
req = request.Request(url=url, data=para, headers=headers, method='GET')
response = request.urlopen(req, context=self.context)
```

整个完整的 ssl 证书验证代码如下:

```
if __name__=='__main__':
  from urllib import parse, request
  import ssl
  context = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
      context.check_hostname = False
      context.load_cert_chain(certfile=pub_key_cert_file, keyfile=pri_key_pem_file)
      context.verify_mode = 2
      context.load_verify_locations(ca_file)
      req = request.Request(url=url, data=para, headers=headers, method='GET')
  response = request.urlopen(req, context=self.context)
```

上述代码中，我们选择了 python 中 urllib 模块做接口请求，是因为在多次对比了 reuests 模块和 urllib 对 https 证书验证的支持之后，发现 urllib 模块能够很好地支持 ssl 证书校验。更多有关 python 中 ssl 模块的信息，请参考 ssl 官方文档

6、总结

回顾整个项目经过，应该说是是被现实问题逼着进步，从一开始的走捷径使用 API 集成工具来测试接口，到后来使用自动化测试脚本实现接口测试，再到最后增加对 https 协议的支持。这一路走来，带着遇到问题解决问题地思路，我的测试技能得到很大提升。总结这几个月的项目经历就一句话：遇到问题，解决问题，能力才会得到快速提升，与大家共勉。

■进阶必备 Python 全栈测试开发，挑战高薪的系统内容>>http://testing51.mikecrm.com/r2WUAya

自动化数据稳定性方案

◆ 作者：多则惑少则明

摘要：自动化测试用例执行时经常遇到因为测试数据变更，导致用例执行失败的问题。这时候就需要对自动化数据本身投入人力/时间来维护了。那有没有什么方法可以尽可能减少自动化数据维护的成本了，尽量让自动化数据长期处于稳定状态呢？下面就自己一些经验和感悟，聊聊自己的一些体会。

一、问题背景

自动化测试用例执行时经常遇到因为测试数据变更，导致用例执行失败的问题。**具体的测试数据导致自动化执行失败的场景，大致可以分为如下几类:**

1、测试数据，由于系统的定时任务，动态变动等原因，不满足自动化的执行需求了

2、测试数据由于是公用数据，其他测试人员/开发同学修改了/删除了对应的数据;

3、不同测试环境中的测试数据不共享，导致在测试环境 A 中跑通的 case，在测试环境 B 中跑不通了;

上述无论哪种情况，跑自动化 case 的结果都是失败了，进而执行 case 的同学不得不费时费力去分析为什么失败，进而继续”修复“case，再将 case 执行起来。最后需要在下面的循环中不断维护 case: 执行 case——>case 失败——>排查 case——>修复 case。面对这样一个循环，自己在项目中的体会颇深，特别是项目频繁变动时期，往往还需要维护自动化数据，确保自动化 case 的运行通过。一方面此时维护自动化数据的人力投入很大，另一方面也容易分散精力，最终并没有达到预期的投入产出比。

那么，有没有什么方法，能够减少在上述不断维护 case 中的人力、时间的耗费呢？毕竟，如果大量时间&精力都花在了自动化 case 维护上，甚至投入比手动测试还要大，

那么我们还有什么必要来维护自动化 case 呢？下面就自己在自动化数据维护方面的一些实践经验，说说自己在这反面的一些心得体会。

二、自动化测试数据维护需要考虑什么？

自动化测试数据维护的根本目标是尽可能确保每次自动化 case 运行的成功，尽可能高的测试覆盖率。为了达到这个目标，就要求自动化数据在每次自动化运行时的稳定性，即自动化数据在数据的量级上、单条数据的内容上都应该是尽可能稳定。

当然了，自动化数据的维护粒度也需要考虑自动化 case 的检查粒度。因而，自动化数据的维护，需要考虑以下几个方面:

1、自动化数据的数据量/数量级。这里的数据量/数量级，具体应该准备多少合适，其实没有一个硬性标准，总的原则是：自动化数据的数据量/数量级，至少需要满足自动化 case 检查的需要。当然了，如果测试环境、数据库、人力资源扛得住的情况下，多多益善。

举个例子，一个自动化 case 检查了排序功能，假如第一次运行时有 10 条数据(当然了，多了更好)，运行也通过了，那么可以认为排序功能没有问题。但假如第二次运行时，由于某种原因，只有 1 条数据了，这种情况下，即使自动化 case 运行通过了，由于数据量太少，也不能确定排序功能没有问题。

2、自动化数据满足测试场景的要求。为了确保自动化尽可能高的测试覆盖率，测试数据需要尽可能满足多种测试场景的需求。

举个例子，假如某个功能点的完整测试需要 10 种测试场景，那么测试数据至少要准备这 10 种测试场景需要的输入数据。当然了，假如你只有 5 种测试场景需要的数据，自动化 case 是可以运行通过的，只是你的覆盖率不会太高而已，甚至可能由于遗漏测试场景导致问题遗留到了线上。

3、尽量避免自动化数据的”动态变化“。这里的动态变化分为两种:

1）人为操作造成的变化。比如，为了某种调试/测试需要，删除/修改了某些自动化 case 用到的数据

2）系统实现造成的变化。比如，系统存在定时任务，会定期刷新某些自动化 case 用到的数据

为了保持尽可能高的测试覆盖率，那么我们需要做到前面的 1、2 条；另一方面，为了尽可能保证自动化 case 能够重复执行，我们需要尽可能避免自动化数据的”动态变化“，即确保自动化测试数据的稳定性。下面就如何保证自动化测试数据的稳定性，来谈一谈自己在这方面使用过的一些方法。

三、自动化测试数据稳定性方案

自动化测试数据维护，为了在长期自动化运行中保持其稳定，一般可以考虑如下原则:

1、与其他测试数据解耦合。满足需要的前提下，将自动化数据与其他测试数据解耦合，形成自动化专用的测试数据，即可以有专用的自动化测试数据库。

2、采用独立的数据集。满足需要的前提下，虽然只有一个测试数据库，可以将自动化数据与其他测试数据在业务上隔离开。比如，自动化数据采用账号 A 操作，其他测试采用账号 B 来操作

3、约定自动化数据范围。满足需要的前提下，当一批数据打标后，与其他同学约定：打标的数据为测试数据，不可随意增删改。

以上方法是从自动化测试数据能够长期稳定、尽可能免收”篡改“的角度，来维护数据的。但具体到每个业务中，可以参考以下的方法，并结合自身的业务特点，综合考虑、权衡利弊后，形成自己的一套自动化数据维护方案。

方法 1: 前置数据构造

这种方法的出发点是：既然自己之前使用的自动化数据可能被程序更改、或人为操作，导致自动化数据无法为自动化 case 反复使用。那自动化数据就不再反复使用了，而是每次运行自动化 case 前，先把其需要的测试数据”造“出来，接着再运行自动化 case。在实际项目中，自动化整体执行大致分为如下步骤:

1）先自动化进行数据构造(比如，调用接口、SQL 等方法插入所需数据)

2）运行自动化 case

3）自动清理数据(比如，调用接口、SQL 等方法删除所需数据)

这样这些自动化数据被程序更改、或人为操作的可能性就大大减少了。当然了，凡事没有绝对。这种前置数据构造的方法虽然将自动化测试数据被“篡改”的可能性大大

降低了，但仍然可能由于系统本身的一些实现(比如：定时任务自动刷数据状态等)、或人为操作等，导致数据被“造”出来后，自动化 case 执行前被“篡改”了。

这种方法的优点显而易见了，但其缺点也不得不说，而且在某些数据库较多、且表关联十分复杂的项目中，此种方法的弊端就更多了。举一个自己身边的经历：当时有一个项目 A，需要自动化。某几个接口涉及的表大概有十几个，而且要根据一张表的数据，去造另一些表的数据。而表的结构又频繁的变动，这种模式下维护起来的自动化数据由于耗时很大，自动化的维护很滞后了（往往是上线后才维护起来的），进而其发挥的作用也大打折扣了。

当然了，某些场景下，这种前置数据构造方法新建处理的数据是不适用的。比如，存在定时任务的场景，或存在消息机制、与第三方交互，并且第三方有回调的场景、有跨端/跨业务的场景。特别是当数据构造属于第三方的业务时，你就不得不”求助于“第三方的同学了。

方法 2：数据监控

这种方法的出发点是：既然自己之前使用的自动化数据可能被程序更改、或人为操作，那么就实时监控自动化 case 用到的数据，一旦发现这些数据不符合要求了，立刻告警出来、或自动对数据进行维护。在实际项目中，整体执行大致分为如下步骤：

1）利用现有数据/创造某些自动化需要的数据

2）利用监控，检查自动化数据是否符合要求

3）一旦发现数据不符合要求，但发出告警/自动修复数据

这种方法的优点不言而喻，不用像方法 1：前置数据构造 一样，需要 100%知道数据应该怎么构造出来，而这本身就是需要耗时耗力的。但正是因为不用费心费力频繁造数据，因而这种数据被“篡改”的可能性就大大提高了。

这种方法也有一定的局限性，不能适用哪些需要“实时”数据的场景。比如，一个功能点是查询今天的订单，类似这种场景下，自动化测试数据就不能使用之前的了，而需要重新构造了。

方法 3：数据/接口 mock

这种方法，与方法 1，方法 2 的最大不同点在于：方法 1/方法 2 是维护本系统/业务

的自动化数据常用的方法，而方法 3 则是维护与第三方/其他业务/其他系统交互的数据的方法。

这种方法的出发点是：既然要测试的业务是 A，为了避免其他业务(比如业务 B)与业务 A 交互数据的变化，也为了简化数据的维护，故而直接把与业务 A 交互的业务的响应接口/数据 mock 掉。这样可以确保业务 A 每次都会处理相同的输入数据，一方面可以专注于业务 A 的测试，另一方面，也排除了其他业务数据变动、环境不稳定等给测试带来的干扰。**在实际项目中，整体执行大致分为如下步骤:**

1）对相关业务进行 mock

2）执行自动化 case

这种数据/接口 mock 的方法，属于假定其他业务响应数据不变的场景，这种方法带来益处的同时，恰恰也会带来一定程度的风险。比如，mock 的数据/接口不合格，导致最终没有检查出问题、甚至将 mock 数据带上线。

四、总结

为了让自动化测试数据，更加稳定地服务于自动化 case 的运行，需要综合考虑各种维护方法的优劣。当然了，方法 1~3 也可能交叉、结合使用。但无论使用哪种方法，自动化测试数据都需要更好地服务于自动化 case，更准确地说是服务于测试效率的提升。

自动化测试数据的维护，除了要考虑上面说的数据量、测试场景、数据稳定性之外，当然还要考虑具体业务本身的一些客观因素，比如：测试环境的耦合程度、自动化数据与手工集成测试数据的耦合程度、项目本身的迭代频繁程度等等。举一个从 0 到 1 时期的初创产品的例子，疯狂加功能的产品，由于产品本身不适合大规模自动化，那么相应的自动化测试数据的维护也要考虑”精兵简政“了，以确保自动化的投入与产出。

无论黑猫白猫，能快速抓住老鼠就是好猫。类比于测试领域，无论是自动化执行，还是手工执行，能以尽可能快地速度测试、尽可能高地覆盖场景，这才是测试效率提升的本质。围绕测试效率提升这个角度来考虑自动化测试数据的维护，想必大家会有更多的心得和体会，而且不会让自己限于为了维护而维护的恶性循环之中了。

往期作品:《什么是测试惯例？如何打破测试惯例？》

测试 10 秒钟，分析 8 小时，性能评估只看跑分可不够

◆ 作者：于无声处

写在前面：大约 4 年前在中科院软件所，为评估国产兆芯 CPU 性能，我接触到一些 benchmark，涉及 CPU 指标的就有 ubench、sysbench、c-ray、SPEC 和 unixbench 等。

工具在手，不要动脑，跑起来就行。后来情况变了，每个月都有数个系统性能调优的 kpi 压在身上，你不光要知道这些性能怎么测，还得分析瓶颈在哪，最后还要不断实验去提升性能。

能力有限，很多就不会去关注，像 sysbench 也是个老掉牙的 benchmark 了，诞生到现在有 15 年历史，一个工具走到现在，还有它使用的场景，那肯定有它的价值。

正巧最近又需要验证虚拟化下 CPU 的性能，仍然利用 sysbench 去测试，这次既然测了，就不单单只看跑分，看看数据背后的意义。

一、sysbench 测试 CPU 的过程

使用 benchmark 前，不妨先看下 readme，sysbench 在 CPU 上是这样自我定义的：

`cpu`: a simple CPU benchmark

划重点：simple。工具设计之初，就不是为 CPU 考虑的，它最常使用场景基本在 mysql 等数据库评测上。所以自评为 simple benchmark，必然有原因，这个后面详说。

回归正题，sysbench 在大多数 Linux 下建议 yum 或 apt-get 安装，万一需要手动编译，需要注意 2 点：

- 如果你的 sysbench 是运行在 64 位 CPU 上，请先确保编译的是 64 位二进制文件；
- 万一你的环境是 32 位 CPU，要知道在 32 位 CPU 上进行 64 位操作数的运算，

需要使用多条 32 位的汇编指令来模拟其行为，随之带来的性能开销势必影响到结果。

（1）一般测试方法

事先声明，我用的是 sysbench1.0.17 版本，自从 1.0.8 版本后，一改以往使用计算达到 10000 内素数所用时间为测试结果，增加了 events per second 为输出报告。

测试方法大同小异，sysbench + 参数 + 测试项 + 命令，如下图:

```
[root@host-192-168-33-17 ~]# sysbench --help
Usage:
  sysbench [options]... [testname] [command]
```

以 CPU 为例（测试在 4 线程下 CPU 的运算速度）:

```
[root@host-192-168-33-17 ~]# sysbench --threads=4 cpu run
```

在本例测试中，只是采用最小化参数：--threads，其他常用参数还包括:

参数名	含义
--cpu-max-prime	素数生成数量的上限，默认是 10000
--time	运行时长，sysbench 会在 time 参数内循环执行素数计算
--threads	线程数，一个关键参数，sysbench 会根据设置的值，启动对应的线程数，可以同时进行素数计算
--report-interval	间隔时间，支持周期性输出性能数据

根据实际需要，可选择多种参数组合，本文只采用线程作为参数，测试结果如下所示:

```
[root@host-192-168-33-17 ~]# sysbench --threads=4 cpu run
sysbench 1.0.17 (using system LuaJIT 2.0.4)
Running the test with following options:
Number of threads: 4
Initializing random number generator from current time
Prime numbers limit: 10000
Initializing worker threads...
Threads started!
CPU speed:
    events per second:   549.36
General statistics:
    total time:                          10.0022s
    total number of events:              5496
Latency (ms):
         min:                                    1.73
         avg:                                    7.27
         max:                                   56.80
         95th percentile:                       34.95
         sum:                                39934.21
Threads fairness:
    events (avg/stddev):           1374.0000/1.73
    execution time (avg/stddev):   9.9836/0.01
```

测试输出清晰，数据一目了然，benchmark 轻度用户重点关注 2 个数据就好，一是 Prime numbers limit，二是 events per second。

以上面为例，最终结果就是计算 10000 个素数，4 线程平均每秒完成 event 个数为 549.36，数值越大代表性能越好。

如果只简单衡量云主机 CPU 性能，以上 2 个结果足够了，明白了在什么样的基数下测试，并且知道每秒完成的 events，可以获得 1 个基准数据。

（2）sysbench 的实现方式

工具 run 了起来，数据也拿到了，但是离马放南山还早。这时候你还不知道数据准确与否，如果是准确的，是不是你想要的最优结果。

在评估数据之前，先了解下 sysbench 是怎么去测试 CPU 的，做到知己知彼后，再

去看测试结果，说不定会有更多体会，特别是把测试中观察到的性能问题跟系统原理关联起来。

sysbench 作为一个多用途开源 benchmark，按照框架设计惯例，其 CPU、thread、io、memory 和数据库也遵循着松耦合的设计，CPU 的源码有独立的模块。

- **简约不简单的 CPU module**

采用农村包围城市，先从 CPU 测试源码入手，源码见/src/tests/cpu/sb_cpu.c

```
19  #ifdef HAVE_CONFIG_H
20  # include "config.h"
21  #endif
22  #ifdef _WIN32
23  # include "sb_win.h"
24  #endif
25
26  #ifdef HAVE_MATH_H
27  # include <math.h>
28  #endif
29
30  #include <inttypes.h>
31
32  #include "sysbench.h"
33
```

图 1：sb_cpu.c 头文件声明

你不熟悉 sysbench 的代码结构也没关系，仅从 sb_cpu.c 中简单几行头文件声明，就可以得到很多信息，抛开前面 2 个 config.h、sb_win.h 不谈（测试环境在 Linux 上），真正用到的就 2 个类库路径里面的头文件：<math.h>、<inntypes.h>和 1 个内部实现函数声明的"sysbench.h"。

math.h	数学函数库,一些数学计算都包含在内
inntypes.h	提供整数输入的各种转换宏
sysbench.h	sysbench 头文件，一般存储类型的定义，函数的声明

重点关注"sysbench.h"，但具体实现还要看 sysbench.c，暂且不表，继续看/src/tests/cpu/sb_cpu.c

```
33
34  /* CPU test arguments */
35  static sb_arg_t cpu_args[] =
36  {
37    SB_OPT("cpu-max-prime", "upper limit for primes generator", "10000", INT),
38
39    SB_OPT_END                                          sysbench给出了默认上限值
40  };
41
```

图 2：初始化的 CPU 测试参数

图 2 中，定义了一个 cpu 默认参数，如果不特别指定，cpu-max-prime 默认以 10000 作为上限。这里工具采用 static 函数声明的方式，直接引用/src/sb_options.h 中的 sb_arg_t 结构体，简单声明测试设置的 name、desc、value 和 type，在以后的 sysbench.c 会多次调用，如图 3 所示：

```
54  /* Test option definition */
55  typedef struct
56  {
57    const char          *name;
58    const char          *desc;
59    const char          *value;
60    sb_arg_type_t       type;
61    sb_opt_validate_t   *validate;
62  } sb_arg_t;
63
```

图 3：sb_arg_t 结构体

沿着/src/tests/cpu/sb_cpu.c 代码继续看，是对 CPU 测试的函数声明和静态变量定义，在 cpu_test 中通过变量可看到 CPU 性能测试流程，如下图 4 中：

```
41
42  /* CPU test operations */
43  static int cpu_init(void);
44  static void cpu_print_mode(void);
45  static sb_event_t cpu_next_event(int thread_id);
46  static int cpu_execute_event(sb_event_t *, int);
47  static void cpu_report_cumulative(sb_stat_t *);
48  static int cpu_done(void);
49
50  static sb_test_t cpu_test =
51  {
52    .sname = "cpu",                          测试用例名称
53    .lname = "CPU performance test",
54    .ops = {
55      .init = cpu_init,                      测试前的初始化步骤
56      .print_mode = cpu_print_mode,
57      .next_event = cpu_next_event,
58      .execute_event = cpu_execute_event,    核心测试环节，具体执行步骤
59      .report_cumulative = cpu_report_cumulative,   阶段性输出结果
60      .done = cpu_done                       测试完成后的清理步骤，本例中只是退出
61    },
62    .args = cpu_args                         需要的Cpu参数说明
63  };
64
```

图 4：cpu_test 静态变量的定义

图 4 中，基于 sb_test_t 的结构体变量 cpu_test，ops 下所有变量在 sb_test_t 中引用了 sb_operations_t。cpu_test 完全依据 sb_test_t 结构体中包含的数据项，从名称设置开始，到初始化操作，再到定义核心 event 函数执行、报告输出，最后到结束后的动作。短短 10 几行，就把 CPU 测试场景描述清楚了，剩下要做的只是代码依次执行罢了。

```
65  /* Upper limit for primes */
66  static unsigned int     max_prime;
67
68  int register_test_cpu(sb_list_t * tests)
69  {
70    SB_LIST_ADD_TAIL(&cpu_test.listitem, tests);
71
72    return 0;
73  }
74
75  int cpu_init(void)
76  {                    通过sb_get_value_int()去获取终端对cpu-max-prime的传值
77    int prime_option= sb_get_value_int("cpu-max-prime");
78    if (prime_option <= 0)
79    {                    这里有个prime数值判断
80      log_text(LOG_FATAL, "Invalid value of cpu-max-prime: %d.", prime_option);
81      return 1;
82    }
83    max_prime= (unsigned int)prime_option;
84
85    return 0;
86  }
87
```

图 5：cpu-max-prime 取值判断

图 5 中，cpu-max-prime 在/src/tests/cpu/sb_cpu.c 代码中有单独实现，其他类似 time、event、threads 的实现全都放在 sysbench.c 中。

sb_get_value_int()函数在 sb_options.c 中已完成定义，很多参数都要 sb_get_value_int 去获取。

```
100  int cpu_execute_event(sb_event_t *r, int thread_id)
101  {
102    unsigned long long c;
103    unsigned long long l;
104    double t;
105    unsigned long long n=0;
106
107    (void)thread_id; /* unused */
108    (void)r; /* unused */
109
110    /* So far we're using very simple test prime number tests in 64bit */
111
112    for(c=3; c < max_prime; c++)
113    {
114      t = sqrt((double)c);
115      for(l = 2; l <= t; l++)
116        if (c % l == 0)
117          break;
118      if (l > t )
119        n++;
120    }
121
122    return 0;
123  }
124
```

测试CPU所采用的算法实现，素数原理众所周知的，因此实现代码很清晰

图 6：CPU 测试的主要执行环节

cpu_execute_event 是 sysbench 在 CPU 性能评估最主要环节，之前代码多次提到 prime 字眼，知道是素数，通过百度也知道它的计算原理，但在 cpu_execute_event 函数中详细定义了具体计算方式。在这种模式下，每个请求都由素数计算组成，由 cpu-max-primes 选项指定的值为上限，所有的计算使用 64 位整数执行。

```
131  /* Print cumulative stats. */
132
133  void cpu_report_cumulative(sb_stat_t *stat)
134  {
135    log_text(LOG_NOTICE, "CPU speed:");
136    log_text(LOG_NOTICE, "    events per second: %8.2f",
137             stat->events / stat->time_interval);
138
139    sb_report_cumulative(stat);
140  }
141
142
```

通过sb_report_cumulative()对各种数据处理生成report

图 7：CPU 报告实现

图 7 中，cpu_report_cumulative 是最后的实现环节，采集、统计及打印是一连串的处理过程，如果要实时打印 report，要停止运行计时器。具体代码实现在 sysbench.c 中 sb_report_cumulative()，这是个复杂的处理流程。

- 管中窥豹看框架

sysbench 测试 CPU 的代码没那么复杂，素数原理摆在明面，算法固定，自身实现代码比较清晰，单看素数计算的部分，任何语言都能实现，自己写个脚本都能直接用，难点主要在 events 和 cumulative 处理上，这也是基准工具的魅力所在。

前文对 CPU 模块有了粗略分析，这里我用一张图将整个流程贯穿起来，如下图所示：

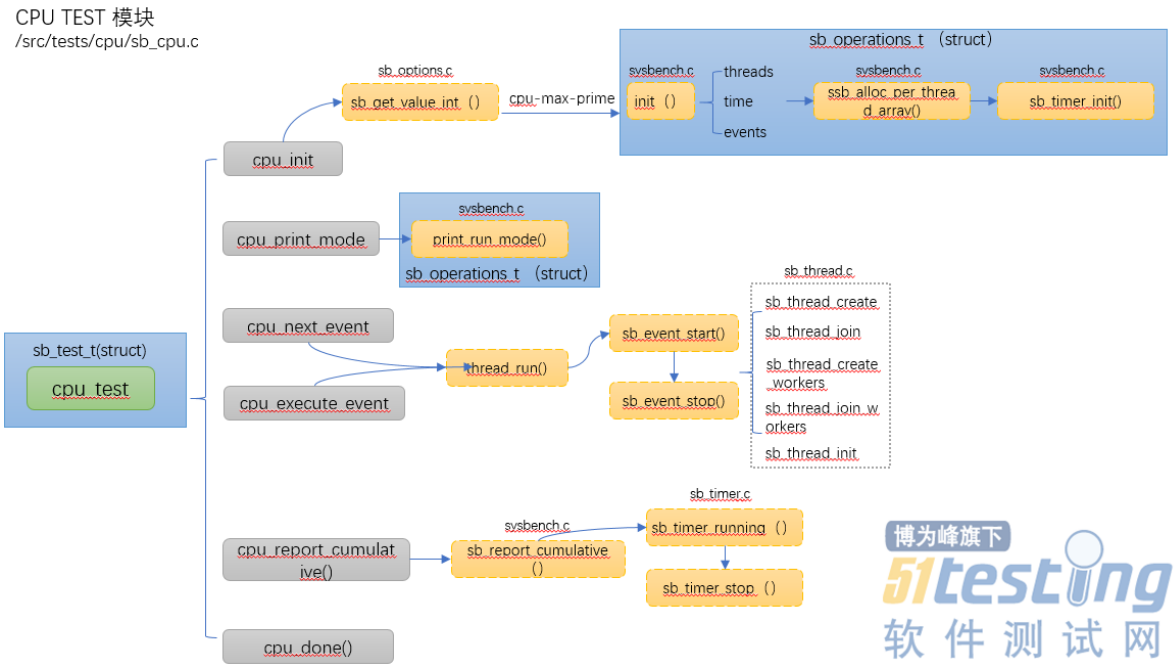


图 8：CPU TEST 关系图

图 8 中，除了能看到代码之间的调用链，有一点值得注意，就是几个 struct：sb_test_t、sb_operations_t，前文提过，基于 sb_test_t 的结构体变量 cpu_test，ops 下所有变量在 sb_test_t 中引用了 sb_operations_t。

阿里云性能专家西邪在他的《sysbench 的框架实现介绍》文章中提到一段话：

> *sysbench 是一个总体框架，它用来操作各个测性能的计算，那各个部门只需要做的一件事情是声明需要的实现。只要理解三个 struct 就可以了*

文中所提到的三个 struct，就是指 sysbench.h 中的 sb_test_t、sb_operations_t 和 sb_builtin_cmds_t，见下图 9：

```
/* Test commands structure definitions */

typedef struct
{
  sb_builtin_cmd_func_t *help;  /* print help */
  sb_builtin_cmd_func_t *prepare; /* prepare for the test */
  sb_builtin_cmd_func_t *run;   /* run the test */
  sb_builtin_cmd_func_t *cleanup; /* cleanup the test database, files, etc. */
} sb_builtin_cmds_t;

/* Test operations structure definition */

typedef struct
{
  sb_op_init            *init;            /* initialization function */
  sb_op_prepare         *prepare;         /* called after timers start,  but
                                             before thread execution */
  sb_op_thread_init     *thread_init;     /* thread initialization
                                             (called when each thread starts) */
  sb_op_print_mode      *print_mode;      /* print mode function */
  sb_op_next_event      *next_event;      /* event generation function */
  sb_op_execute_event   *execute_event;   /* event execution function */
  sb_op_report          *report_intermediate; /* intermediate reports handler */
  sb_op_report          *report_cumulative;   /* cumulative reports handler */
  sb_op_thread_run      *thread_run;      /* main thread loop */
  sb_op_thread_done     *thread_done;     /* thread finalize function */
  sb_op_cleanup         *cleanup;         /* called after exit from thread,
                                             but before timers stop */
  sb_op_done            *done;            /* finalize function */
} sb_operations_t;

/* Test structure definition */

typedef struct sb_test
{
  const char        *sname;
  const char        *lname;
  sb_operations_t   ops;
  sb_builtin_cmds_t builtin_cmds;
  sb_arg_t          *args;

  sb_list_item_t    listitem;
} sb_test_t;
```

图 9: sysbench case 的主要结构体

从 CPU 入手看 sysbench 的实现，3 个 struct 定框架，实现了从测试场景结构到操作结构，再到命令实现结构。

写到这里，对 sysbench 的框架在本文只是粗略了解，有些地方并未深入，比如 thread 的设计。

二、计算素数对 CPU 性能的参考价值

一直觉得数据必须要产生价值，价值就是对外能够给用户带来实际意义，比如帮助用户虚拟机选型，对内能够推动研发不断优化提升用户体验等。

有一段时间，因工作需要，我验证了很多操作系统的 CPU 性能，跑了很多次

sysbench，最快只要 10 秒钟出结果，立等可取。但是跑分不是比大小的游戏，测的越多越迷茫，甚至都怀疑基准测试是不是认真的，进而对数据打上问号，这样得出的数据有没有参考价值？

sysbench 去评估 cpu 性能，打个比方就像田径赛事，对 cpu 的运算测试，包括素数计算，整数，还是浮点数，好比 100 米、110 米栏等比赛，赛道是测试对象，而裁判就是 benchmark，都是看谁跑的快。

（1）为啥要使用计算素数的方式

在前文第一章节讲 sysbench readme 时说过，评估 CPU 性能，sysbench 只是一个 “a simple CPU benchmark"，再结合/src/tests/cpu/sb_cpu.c 源码，可清晰看到它就是通过执行 64 位整数相加、整除等操作来获取素数，直到满足设定最大值，这个过程用来评估 CPU 运算速度。

所谓素数就是质数，一个大于 1 的自然数，除了 1 和它自身外，不能被其他自然数整除的数。

为啥要采用素数相加计算来判断 cpu 性能？这个可追溯不到历史，也许当年工具开发者看了电视节目，节目正好讲什么人机大战、数学猜想等，诸如此类，who care？

（2）素数运算衡量的是 CPU 什么能力？

在讲 CPU 源码时，提到/src/tests/cpu/sb_cpu.c 中 cpu_execute_event 函数，如下源码所示：

```
for(c=3; c < max_prime; c++)
  {
    t = sqrt((double)c);
    for(l = 2; l <= t; l++)
      if (c % l == 0)
        break;
    if (l > t )
      n++;
  }
```

利用计算最大素数来衡量 CPU 运算速度，在代码中主要使用到的是整数计算（加法），cpu 整数能力就是 cpu 执行加法指令的能力，cpu 一秒能执行多少条加法指令，整

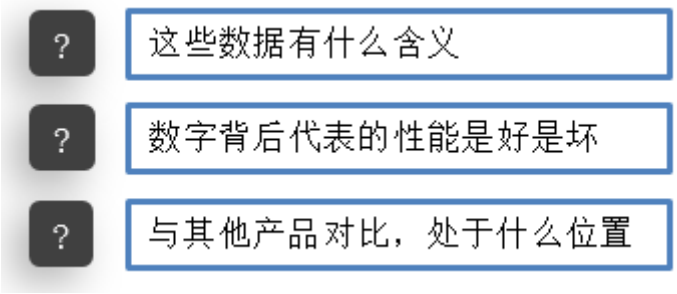
数运算是 cpu 最基本能力。

通过 sysbench 得到的 CPU 事务数，能一定程度上反映 CPU 整数运算性能，但不完整（只涉及加法），相加是衡量 CPU 整数能力，但是相除就是浮点运算能力（但涉及不多），实际上日常大多数应用（包括 sysbench）都是整数计算为主的程序(会包含少量浮点指令)。

总结一下，利用 sysbench 计算素数获取 CPU 运算速度，比较接近实际应用 CPU 场景（不包括 GPU），具有代表性。如果想独立评估 CPU 整数或浮点运算能力，建议尝试别的工具。

三、如何评估测试数据

到了现在，sysbench 测试方法熟悉了，结果也有了，把数据填进 excel 中呈交上去，来杯咖啡可以悠闲的等待下班开黑打野了。不过还没开心多久，同事就来问你:



你不得不重新打开电脑，运行工具，试图从数字中找到其中的奥秘，可是很快就发现除了从 sysbench 获取一组孤零零数字外，空荡荡的脑子里好像啥也没有留下。是不是白算了？并没有，CPU 不会白算，每一步都算数，测试结果只是空洞的一串数字，还需要人为赋予价值。

（1）磨刀不误砍柴工

无论是基准测试，还是性能测试，在执行过程中有一个很容易做到，但很多人都忽略的现象：不会做笔记。不是不会做，是压根不记录过程中任何输出，没有这些记录，会给后续跟踪性能问题带来麻烦，产生问题原因从手指间溜走，时间浪费不说，还容易带给你负能量。

我个人使用 OneNote 收集过程， 文字+图片任意组合，方便日后追踪，如果遭遇环境重装、时间不允许等客观因素不至于老鼠拉龟，无从下手了。

（2）给测试定基调

我的同事老王说每次发布新版，都要对 CPU 和内存运算能力重新验证，好像没意义。

狭义上的“意义”是啥？意义=价值，就是投入时间后所得到的回报。说到价值，其实在第 2 章节中分析素数运算衡量 CPU 什么能力时已给出了答案，《学霸的黑科技系统》一书在某个章节中也曾提到“梅森素数”，你说梅森素数有什么用？好像也举不出什么实际应用的例子，通常它都是被用作考验计算机性能，IntelSkylake 芯片也曾由此发现 bug。

不能指望每次基准测试都能发现 bug，那这个性能评估还要不要做了？在回答问题之前，先要明白基准测试的目的。

基准测试（benchmarking）是一种测量和评估软件性能指标的活动。你可以在某个时候通过基准测试建立一个已知的性能水平（称为基准线），当系统的软硬件环境发生变化之后再进行一次基准测试以确定那些变化对性能的影响。

51testing 软件测试网

总结一下：

- 基准测试一般发生在环境变更、版本迭代、配置或参数改变的前后；
- 当为软件或系统创建性能基准后，这个数据可作为参照，用来判断任意项变更给系统带来的影响；
- 了解系统优化前后的性能提升/下降指标，获取系统整体性能趋势，及早识别系统性能风险

这么看来，我们的工作还是有做的必要性，但不妨定 3 个小目标：

1. 没有基准目标值，那就分析选出基准线
2. 有基准参照，对比变更为系统带来的影响
3. 确认有影响的地方，分析存在的原因

（3）从 sysbench 中读懂数据

sysbench 数据模板是固定的，测试结束会将数据直接嵌套进去再呈现到终端。拿到数据先做 2 个确认：

→确认 1：Number of threads 和你发起测试参数中一致

→确认 2：Prime 默认为 10000，除非你发起测试参数中有特别指定值

步骤正确，参数没错，我们从上到下依次看下 sysbench 输出的测试数据。

测试结果中最重要的就是 CPU speed，即：所有线程每秒完成的 events 数，这个数值越大代表性能越好，如下图 10：

```
CPU speed:
    events per second:   549.36
General statistics:
    total time:                          10.0022s
    total number of events:              5496
```

图 10：cpu speed 结果

每秒完成的 events 数可以理解成跑分，在手机领域每当手机厂商发行新手机产品，各种发布会上毫不例外的都会附上跑分。但是单看跑分还不够，还要重视延时。

```
Latency (ms):
         min:                                    1.73
         avg:                                    7.27
         max:                                   56.80
         95th percentile:                       34.95
         sum:                                39934.21
```

图 11：Latency 结果

上图 11 中， sysbench 给出了 Latency 的结果，包括 min、avg、max 和 95%时延，类似 jmeter 输出，这个数值越小代表性能越好。在分布式存储（如 ceph）领域，很多指标都将延时作为重要参考，但在计算能力上，对 CPU 运算延时重视度并不够，由于 CPU 和内存、硬盘不公平的进化规律，很多场景下 CPU 可能会消耗在 IO 等待中。

（4）不同工具齐上阵，寻找蛛丝马迹

众所周知在 Linux 上，应用是不能直接访问底层硬件，当进程需要访问硬件时，必须由用户态模式切换至内核态（暂不考虑 KVM 虚拟化，在虚拟世界还要借助 intel 硬件虚拟化辅助），再通过系统调用访问硬件。so，在 Linux 中有一个放之四海皆准的应用负载模式，如下图 12 所示：

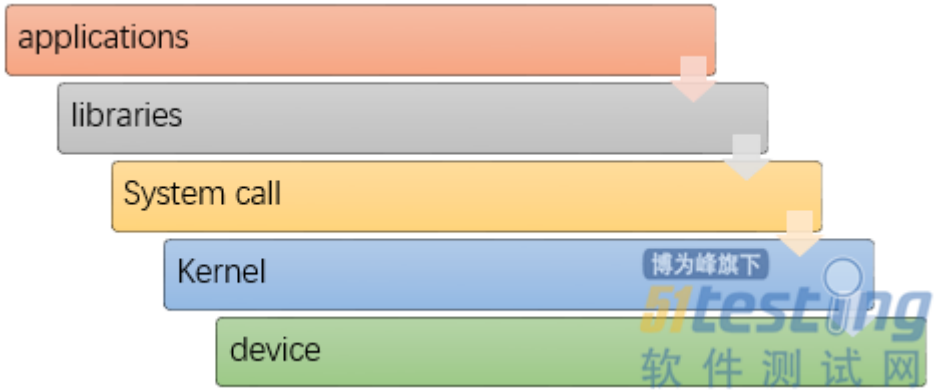


图 12：linux 应用负载模式

这里把 sysbench 当成一个普通应用，和其他同样运行在虚拟机 linux 系统上的程序没什么两样，当它 run 起来后，会发生什么？

任何应用进程，哪怕复杂度很低，也是依赖 Linux 库来执行操作，sysbench 测试 CPU 中一定需调用 os 库函数，通过 ltrace 可以跟踪进程调用库函数的情况，如下图 13 所示：

```
[root@host-          ~]# ltrace -c -S -p 15849
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 52.72  135.831651        2071     65563 clock_gettime
 17.51   45.120095        2064     21854 log
 17.34   44.674237        2044     21855 floor
 12.40   31.957013         736     43404 futex
  0.01    0.016365          97       167 free
  0.00    0.009981         433        23 snprintf
  0.00    0.009896         430        23 __vsnprintf_chk
  0.00    0.002437         135        18 __printf_chk
  0.00    0.001776          98        18 pthread_mutex_lock
  0.00    0.001775          98        18 strcmp
  0.00    0.001745          96        18 pthread_mutex_unlock
  0.00    0.001714          95        18 strncpy
  0.00    0.000787         787         1 exit_group
  0.00    0.000537         134         4 oldfstat
  0.00    0.000455         113         4 umask
  0.00    0.000423          23        18 write
  0.00    0.000393         131         3 pthread_join
  0.00    0.000346         346         1 munmap
  0.00    0.000179         179         1 exp
  0.00    0.000139          69         2 alarm
  0.00    0.000118         118         1 memset
  0.00    0.000115         115         1 pthread_rwlock_wrlock
  0.00    0.000108         108         1 pthread_rwlock_unlock
  0.00    0.000101         101         1 pthread_rwlock_destroy
  0.00    0.000100         100         1 pthread_mutex_destroy
------ ----------- ----------- --------- --------------------
100.00  257.632486                153018 total
```

图 13: ltrace 库函数调用情况

库函数调用开销是一个可能存在性能瓶颈的地方，这就要看进程在需要的库中花费的时间开销了，先从调用哪些库函数入手，看看他们分别做了什么。

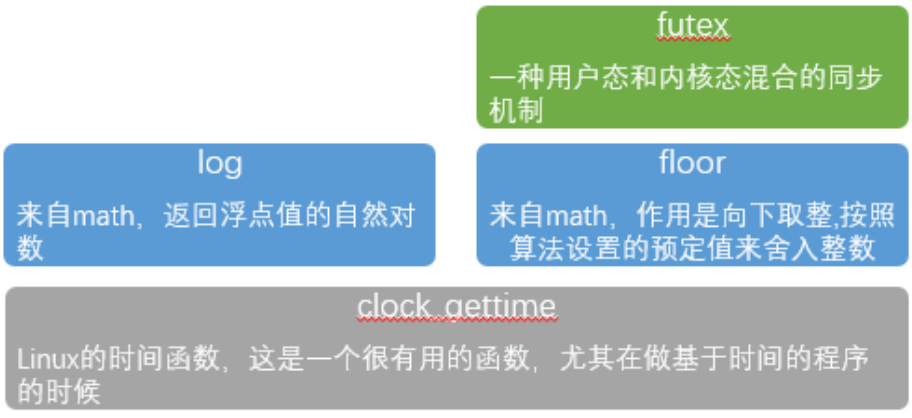


图 14: sysbench 库函数调用矩阵

- clock_gettime 贯穿整个 sysbench 过程，在库函数调用中占了半数，调用了 65563 次，开销时间随着 sysbench 的--time 参数改变而改变。有一点值得注意的是，sysbench 进程已结束，但 clock_gettime 仍持续长达 10 秒+调用；

- log、floor 函数调用一直维持在 17%左右，这是评估 CPU 运算所使用的算法核心函数，从时间开销来看，此处并未发生明显瓶颈；
- futex 是对内核的调用，主要就是管理进程挂起时的等待队列以及锁的睡眠与唤醒操作（即 futex_wait、futex_wake）

ok，通过 ltrace 定位到进程具体调用库函数，并且也知道哪些函数调用次数最多，下一步就是要在代码中找出 clock_gettime 在哪里被调用。我们利用 sysbench 执行测试，自然要在 sysbench 源码中去寻找。下图 15 是 clock_gettime 在代码中的调用位置，只是简单列出，不包含调用链。

代码位置	作用
sysbench-1.0\configure.ac	库函数检查，这个还延伸出不同系统架构的检测
sysencbh-1.0\src\sb_timer.h	封装各种*gettime*函数，提供给SB_GETTIME()、gettimeofday（）

图 15：sysbench 代码中 clock_gettime 调用位置

同样 log、floor 在 sysbench 代码也有相应调用关系，在本文不再详述。

至此，毋庸置疑的是 clock_gettime 在 sysbench 调用过于频繁，这是跟 sysbench 的架构设计有关系，如果通过修改源码让其少调用库函数的次数，或者尝试修改库的源代码，是不是能够再提高性能？

醒醒吧，性能真有那么简单就下结论？首先修改库源代码并不实际，牵一发而动全身；其次库函数调用频繁并不代表此处遭遇瓶颈，需要和资源开销综合观测。

1）CPU 使用率和平均负载

CPU 使用率人人都会查，这是首先能想到 CPU 监控指标，下图 16 是 sysbench 执行中 CPU 实时监控情况，可以看到 4 线程同时在跑，平均分配到 4 个 vCPU 上，都满负荷运行，过去 1 分钟的平均负载达到 4.06，并且持续在这负载线上。

结论 1： 平均负载最理想的情况是等于 CPU 个数，4.06 代表系统并未发生过载现象

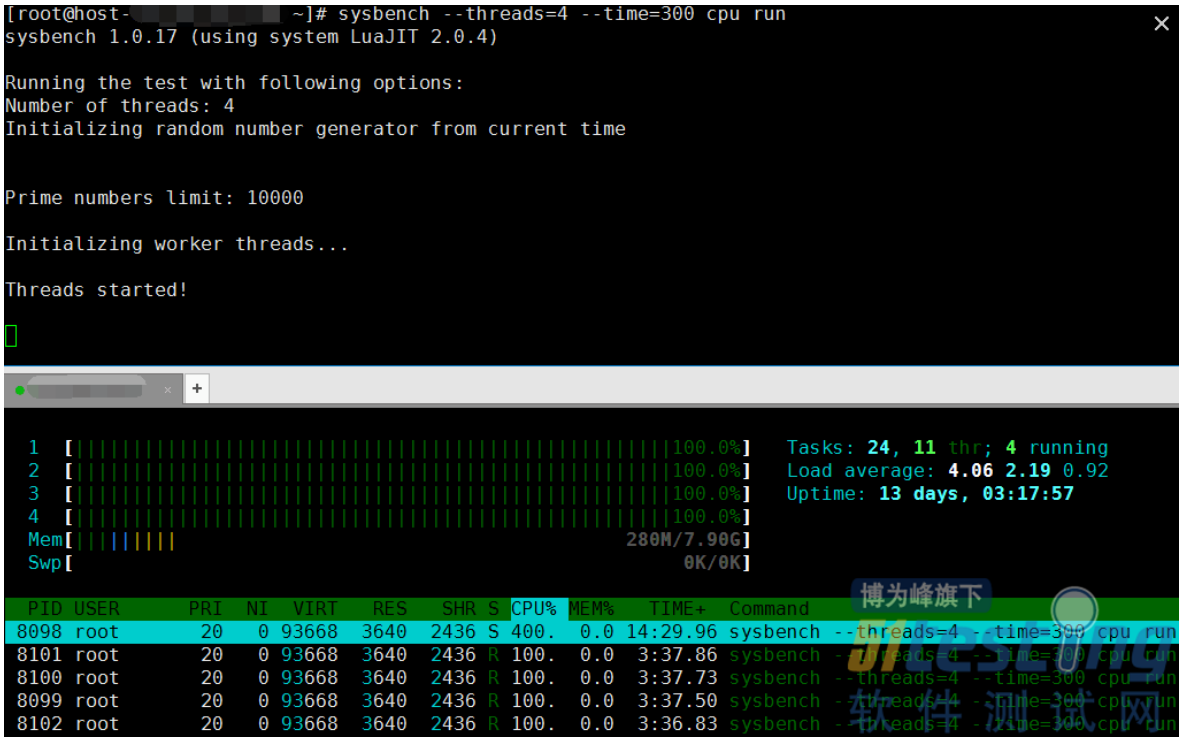


图 16: CPU 使用率

2）用户/内核消耗 CPU 的时间，有无 io 等待

下图 17 是实时 top 监控，us100%，内核无 CPU 消耗，至始至终 CPU 没有出现 iowait 状态，内存也无开销，这是一个 CPU 基准测试工具该有的表现，只针对 CPU 检测，不牵连无辜。

结论 2: CPU 占用时间片正常，无等待 I/O

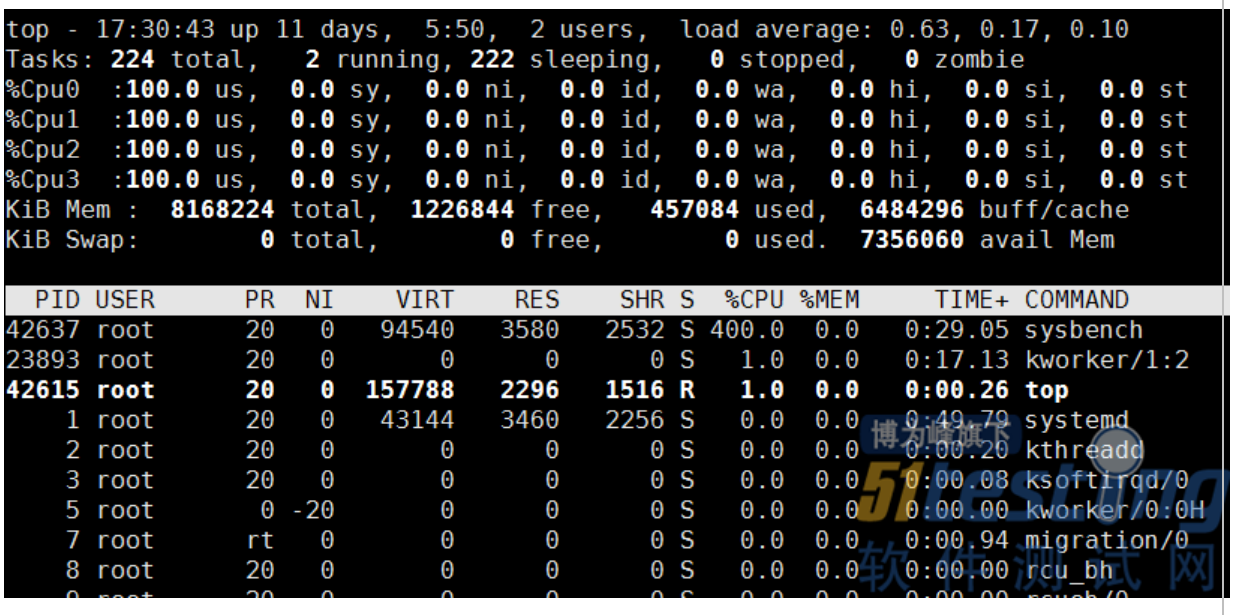


图 17: us/sy/wa 的监控

3）系统、线程上下文切换

图 18 是 vmstat 实时监控，看到随着 sysbench 的运行，in 次数增加明显，上下文切换短时增加，后趋于减小到正常水准。

结论 3: sysbench 运行中，中断次数明显增加，但不能确定是软中断，还是硬中断；系统上下文切换无异常

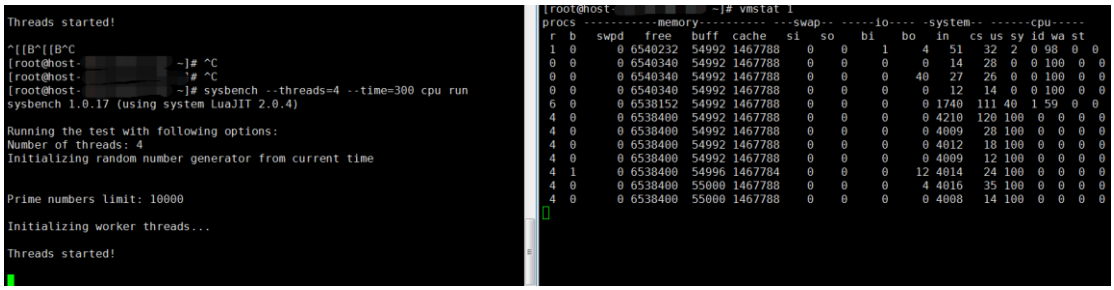


图 18: 系统上下文切换/中断次数

图 19 是利用 pidstat 监测到的线程开销，可看到 4 个 sysbench 线程均存在非自愿的上下文切换，次数维持在 5-20 之间，上下文切换必然产生性能开销，但如果超过 1 万或者切换次数出现很大增长，就可能出现性能问题。

结论 4: sysbench 在多线程运行下会存在非自愿的上下文切换，在单线程下则不会发生。整体上下文切换稳定在个位数，不会对性能结果造成明显影响。

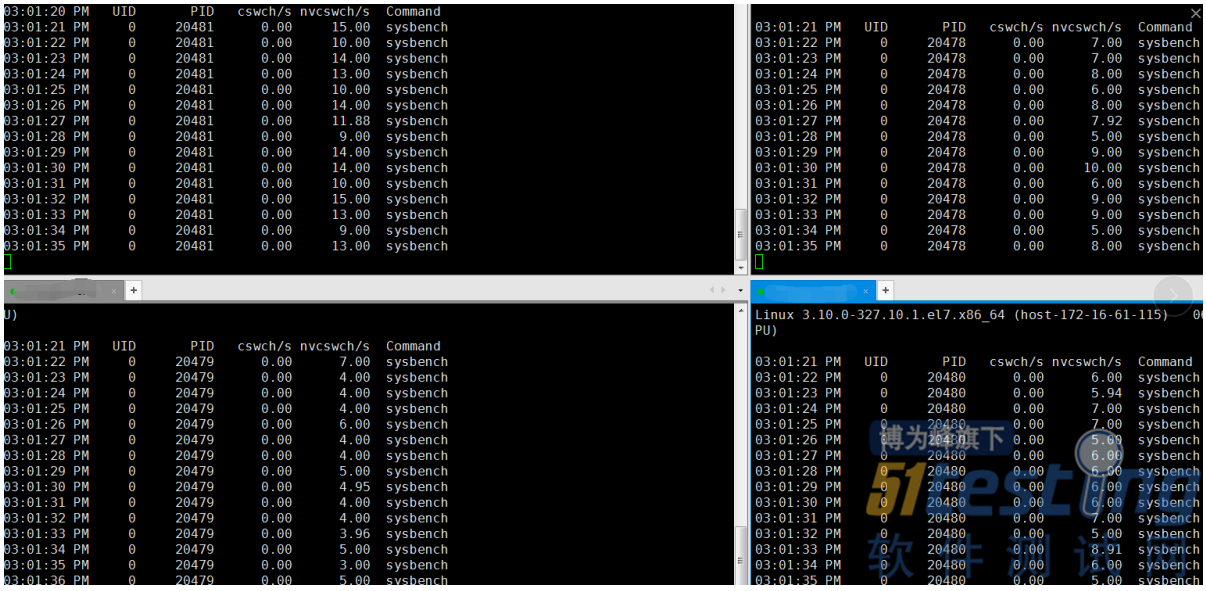


图 19: 线程上下文切换

4）软硬件中断

之前 vmstat 监控到中断次数明显增加的情形，但无法判断中断类型，用 mpstat 继续排查后，发现 4 个 vcpu 并没有明显的中断发生，如下图 20 中%irq、%soft 所示:

```
[root@host          ~]# mpstat -P ALL 1
Linux 3.10.0-327.10.1.el7.x86_64 (host-          )   06/04/2019      _x86_64_        (4 CPU)

04:12:19 PM  CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
04:12:20 PM  all  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
04:12:20 PM    0  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
04:12:20 PM    1  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
04:12:20 PM    2  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
04:12:20 PM    3  100.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

图 20: mpstat 监控 CPU 中断

先排除掉 sysbench 运行时性能瓶颈导致的中断增加，再监控下硬中断/proc/interrupts 信息，找到中断发生的地方。图 21 中，通过测试前后对比，找到硬中断产生的位置，都是由 LOC 产生的 Local timer interrupts，LOC 是 LInxu 使用的驱动，用户无法访问和配置，这些都属于时间中断。

结论 5: Local timer interrupts 的发生，与频繁调用 clock_gettime 有关

```
[root@host-          ~]# cat /proc/interrupts
           CPU0       CPU1       CPU2       CPU3
  0:        122          0          0          0   IO-APIC-edge      timer
  1:         10          0          0          0   IO-APIC-edge      i8042
  6:          3          0          0          0   IO-APIC-edge      floppy
  8:          0          0          0          0   IO-APIC-edge      rtc0
  9:          0          0          0          0   IO-APIC-fasteoi   acpi
 10:         29          0          0          0   IO-APIC-fasteoi   ehci_hcd:usb1, uh
 11:          2          0          0       7951   IO-APIC-fasteoi   uhci_hcd:usb3, uh
 12:        144          0          0          0   IO-APIC-edge      i8042
 14:          0          0          0          0   IO-APIC-edge      ata_piix
 15:     155643          0          0          0   IO-APIC-edge      ata_piix
 24:          0          0          0          0   PCI-MSI-edge      virtio0-config
 25:          7          0      30778          0   PCI-MSI-edge      virtio0-input.0
 26:          1          0          0          0   PCI-MSI-edge      virtio0-output.0
 27:          0          0          0          0   PCI-MSI-edge      virtio1-config
 28:         44      28928          0          0   PCI-MSI-edge      virtio1-virtqueue
 29:          0          0          0          0   PCI-MSI-edge      virtio2-config
 30:       2783      32665          0          0   PCI-MSI-edge      virtio2-req.0
NMI:          0          0          0          0   Non-maskable interrupts
LOC:    3735003    3581052    3478206    3409604   Local timer interrupts
SPU:          0          0          0          0   Spurious interrupts
PMI:          0          0          0          0   Performance monitoring interrupts
IWI:      30310      43711      33913      22453   IRQ work interrupts
```

图 21: interrupts 实时信息

（5）学会对比

从资源视角转了一圈，依次分析了 load average、上下文切换、CPU 使用、io 等待和软硬件中断等，除了 clock_gettime 现象，其他并没发现明显性能瓶颈的位置，掉转船头，再回到 sysbench 调用库函数问题上。

ltrace 是跟踪系统函数调用好工具，不过精准力度欠缺了些，下面使用 perf 看一看 sysbench 在运行中的热点函数是哪些。

在图 22 中，perf top 实时看到当前系统上所有函数执行情况，不过什么鬼，在虚拟机中 perf 居然未能定位本地符号表对应的 symbol 和地址对应关系，有些符号不能解析为函数名，只能用地址表示，这谁看的懂。

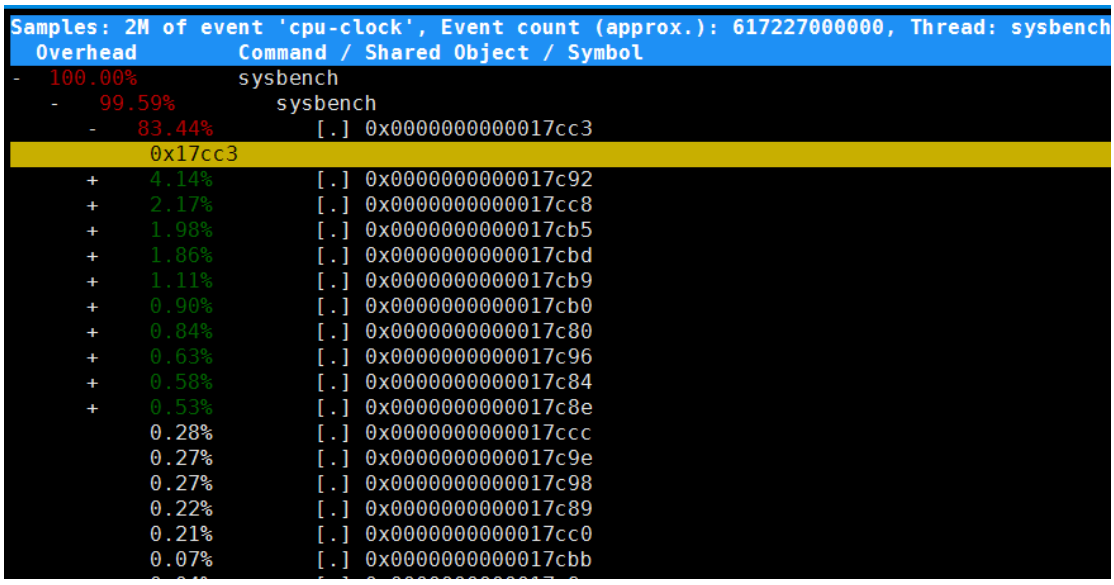


图 22: 虚拟机上 perf 追踪各级函数执行情况

不能解析函数名，猜测是 KVM 虚拟化问题，找一个物理机用同样测试参数试下，可以看到 sysbench 100%执行的是哪个函数，如图 23 所示:

```
Samples: 56K of event 'cycles:ppp', Event count (approx.): 220529796052432
Overhead  Shared Object     Symbol
 100.00%  sysbench          [.] cpu_execute_event
   0.00%  [kernel]          [k] __raw_v4_lookup
   0.00%  [unknown]         [k] 0x0000000000419f23
   0.00%  [kernel]          [k] update_wall_time
   0.00%  [kernel]          [k] native_write_msr_safe
   0.00%  [kernel]          [.] irq_return
   0.00%  [kernel]          [k] _raw_qspin_lock
   0.00%  [kernel]          [k] retint_userspace_restore_args
   0.00%  [kernel]          [k] ipt_do_table
   0.00%  libm-2.17.so      [.] __ieee754_log_avx
   0.00%  [kernel]          [k] apic_timer_interrupt
   0.00%  [kernel]          [k] _raw_spin_lock_irqsave
   0.00%  sysbench          [.] sb_histogram_update
   0.00%  [kernel]          [k] hrtimer_interrupt
   0.00%  [kernel]          [k] rcu_process_callbacks
   0.00%  [kernel]          [k] read_tsc
   0.00%  [kernel]          [k] irq_enter
   0.00%  [kernel]          [k] __do_softirq
   0.00%  [kernel]          [k] native_queued_spin_lock_slowpath
   0.00%  [kernel]          [k] perf_event_task_tick
   0.00%  [kernel]          [k] __x86_indirect_thunk_rax
   0.00%  [kernel]          [k] __hrtimer_run_queues
```

图 23：物理机上 perf 实时追踪

cpu_execute_event 是 sysbench 评估最主要实现，在前文代码解读中已详细讲到。为了得到更细粒度分析，还可执行 annotate，在 C 和汇编混合显示下获取热点函数下 cpu 指令，如图 24 所示：

```
cpu_execute_event  /usr/local/bin/sysbench
  1.28  20:   add    $0x1,%rbx
  0.10        cmp    %rbp,%rbx
            ↓ jae    88
  1.06        cvtsi2 %rbx,%xmm0
  4.74        sqrtsd %xmm0,%xmm2
  1.01        ucomis %xmm2,%xmm2
  0.24      ↓ jp     91
  0.00  38:   ucomis %xmm3,%xmm2
  0.52      ↑ jb     20
              test   $0x1,%bl
            ↑ je     20
  0.02        mov    $0x2,%ecx
            ↓ jmp    68
              nop
  1.64  50:   cvtsi2 %rcx,%xmm1
  0.81  55:   ucomis %xmm1,%xmm2
  9.29      ↑ jb     20
  0.24        xor    %edx,%edx
  3.33        mov    %rbx,%rax
 72.08        div    %rcx
  1.09        test   %rdx,%rdx
  0.06      ↑ je     20
  0.46  68:   add    $0x1,%rcx
  2.04      ↑ jns    50
              mov    %rcx,%rsi
              mov    %rcx,%rdx
              shr    %rsi
              and    $0x1,%edx
              or     %rdx,%rsi
              cvtsi2 %rsi,%xmm1
```

图 24: annotate 获取更详细的热点指令

100%时间开销在执行 cpu_execute_event 操作上，分析运行代码，其中 72.08%的时间占用%rcx 一项指令上，这属于 64 位汇编参数传递的知识范围。

我曾经遇到一个奇怪现象，在 64 位 linux 系统上，素数运算速度反而低于 32 位，通过查找指令延迟表发现 64 位整数除法指令在某国产 CPU 上的执行时间远长于 32 位执行时间。这就解释了 64 位操作数编译成 64 位可执行程序为什么会发生性能异常。

后来我在 cenots 7.2 64 位系统上做过对比实验，确定相同的处理器下，64 位系统素数运算速度是高于 32 位的，CPU64 位指令集可以运行 64 位数据指令，一次可以提取 64 位数据，比 32 位提升一倍。

在 64 位系统中，增加了比 32 位系统多的寄存器。当 sysbench 调用函数时，传递参数会发生改变。一般参数都放在寄存器中传递，回过头看下 sysbench 在参数传递的代码，可以看到一个函数在调用时，前四个整型值依次传给寄存器 rbx、rax、rcx 和 rdx。

因为涉及知识盲区，这里不好评价是否存在问题，底层约定的通用参数传递方式，硬件层是无法改变，sysbench 代码倒是可以变动，但效果未知。

如果从主动寻找性能问题无果，可以试试被动对比。所谓被动，是通过对比方式找到可能存在的问题。

1）不同物理 CPU 的性能表现

手上正好有 2 套不同 CPU 物理环境，试下物理 cpu 主频、核心进程的不同对性能的影响程度。

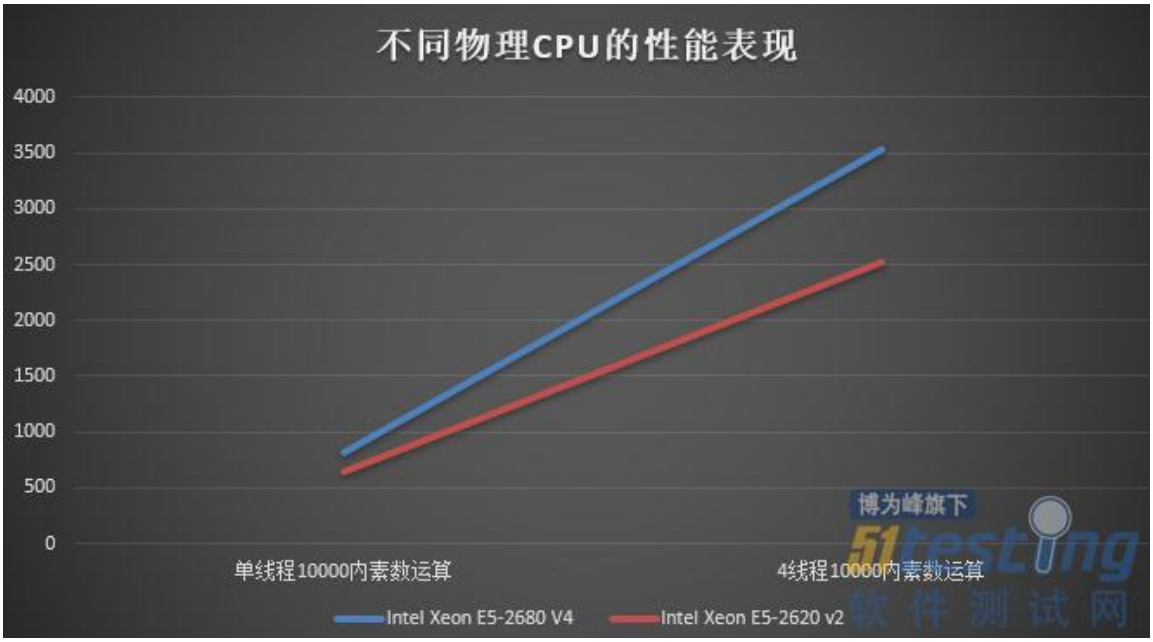


图 25：不同物理 CPU 下的性能对比

从图 25 折线图看到，在 Intel Xeon E5-2680 V4 上执行 sysbench 基准测试得到的数据优于 Intel Xeon E5-2620 v2。

如果站在普通用户的角度看，这个结果差强人意，性能领先幅度也就 45%左右，要知道从价格上看 Intel Xeon E5-2680 V4 可是甩 E5-2620 v2 几条街的。

不过考虑到 sysbench 只是对 CPU 的部分指标考量，并不能代表全部，慎重起见，需要综合 CPU 的架构、主频、L3 缓存、线程数和总线规格一起分析。

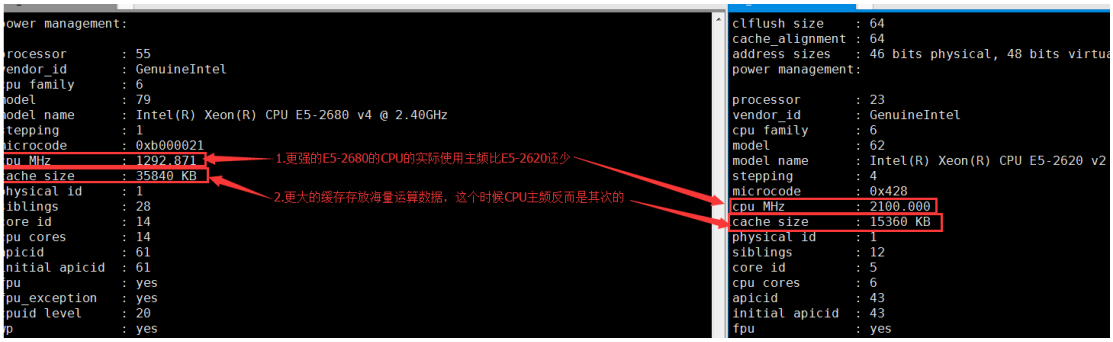


图 26：E5-2680/E5-2620 对比

上图 26 是 E5-2680 V4 和 E5-2620 V2 对比，E5-2680 单个线程实际使用主频虽只有 1292MHZ，但实际性能胜过频率更高的 E5-2620。

怎么回事，难道 E5-2680 装上了涡轮增压？1.2T 的发动机爆发出超过 2.0L 自然吸

气的动力？这里引出新的问题，对于普通云计算使用者或企业上云用户，开通一个虚拟机，是关心高主频，还是看多核心？

小朋友才做选择，让我选，我两个都要，高主频多核心都是王道。不过现实啪啪啪打脸，你不得不考虑成本，贵的东西自然好，但浪费资源也没必要。所以你需要先明确在虚拟机上要部署什么应用、什么使用场景、处理负载多大、是否计算密集型等，还有种特殊情况，有的应用或业务对 CPU 指令集有要求，比如有的软件要求 AVX 指令集支持，这样 IvyBridge 架构的 CPU 就不能选择了，必须选择 Broadwell 以后的 CPU。

除了主频，CPU 的性能还和 CPU 的架构、工作频率、Cache 大小、支持指令系统等有重要的影响，单一看 CPU 主频不准确。

现代计算或标准型虚拟机大多数使用场景可能是多核多线程并行处理，需要 CPU 更快更大的缓存来暂存海量数据，这个时候 CPU 频率反而其次。

总之一句话，CPU 性能先看架构，再看主频，然后对比核心线程，最后再看缓存。

2）不同线程数的性能对比

我曾经看过一些利用 sysbench 测试 CPU 的错误例子，操作者会把测试参数中的--threads 不断加大，甚至远超过虚拟机 CPU 总核数。

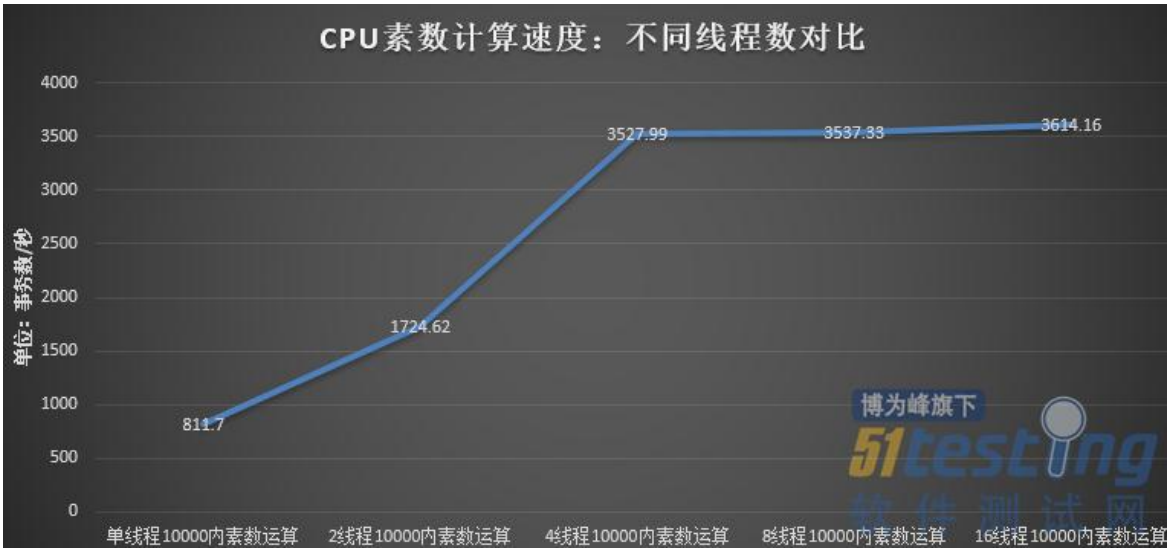


图 27：threads 超过 CPU 线程数后的表现

多核 CPU 在多线程运用下，CPU 的运算能力线性提高，如上图 27 所示，但当 thread 超过 cpu 核数后，CPU 运算速度不会再增加。

通常在实际使用场景中，如果是单核 CPU 运行多线程技术，执行多线程运算是没有效果的，而当云主机需要执行其他 IO 操作的任务，比如读取文件、网络通讯等，多线程技术才能发挥作用，因此多核 CPU 在多线程运用下，CPU 运算能力会明显提高。就好比你就两条腿怎么跑的过四条腿，你又不是美国队长。

threads 不断增加，远超过虚拟机 CPU 总核数后，性能不会提升，只会增加这个 CPU 平均负载，如下图 28，load average 可以看到，在 4 核 CPU 虚拟机中，运行 8 个线程 sysbench，导致 sysbench 线程互相争抢 CPU 使用权，负载达到 8 以上，造成严重负载，影响操作系统运行速度。

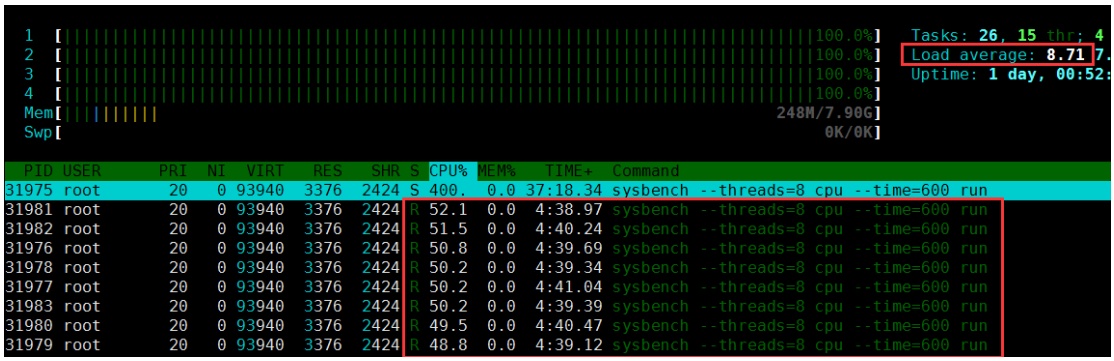


图 28: 超线程运行导致平均负载增加

3）和公开数据对比

很多时候在调查性能过程中，利用别人的经验，参考别人的数据不安全是坏事，自己闭门造车也要不得，当然这个过程需要慎重进行。

基于 PM 测试的全球 CPU 天梯榜，该排名是基于数千个 PerformanceTest 基准测试得出的评分，相对公正，如下图 29 所示:

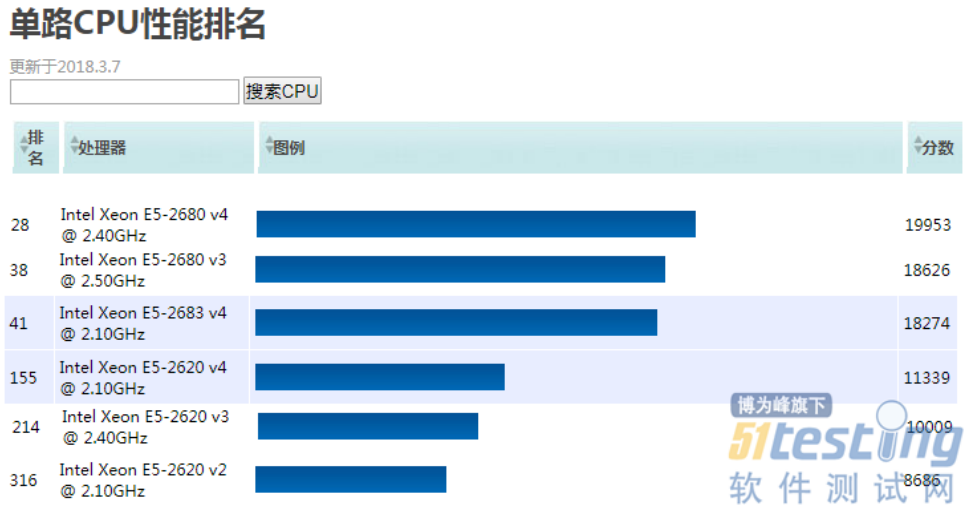


图 29：CPU 天梯榜

除此之外，推荐 1 个基准测试网站：openbenchmarking.org，这是 1 个测试结果数据的公共和私有存储网站，用于共享结果和有效比较多个测试结果集的有效协作。在这个网站有来自全球各地主动推送上来的测试数据。

4）虚拟机的 cpu 性能损耗

虚拟机 cpu 性能相比宿主机硬件的损耗程度，也是一个用户关心的：我们创建的 KVM 虚拟机能实现“零损耗”么？

这是一个比较“庞大”的话题，无论使用哪种 flavor（规格）创建虚拟机，它的 vCPU 在宿主机上物理 CPU 不同核之间调度。在虚拟机上，执行 sysbench，命令请求下去后，先经过 vCPU 到虚拟机 os，再调度到物理机 CPU 上，vCPU 在物理机系统上就是 1 个线程调用，有调度就会产生开销，有了开销必然带来性能上的影响。

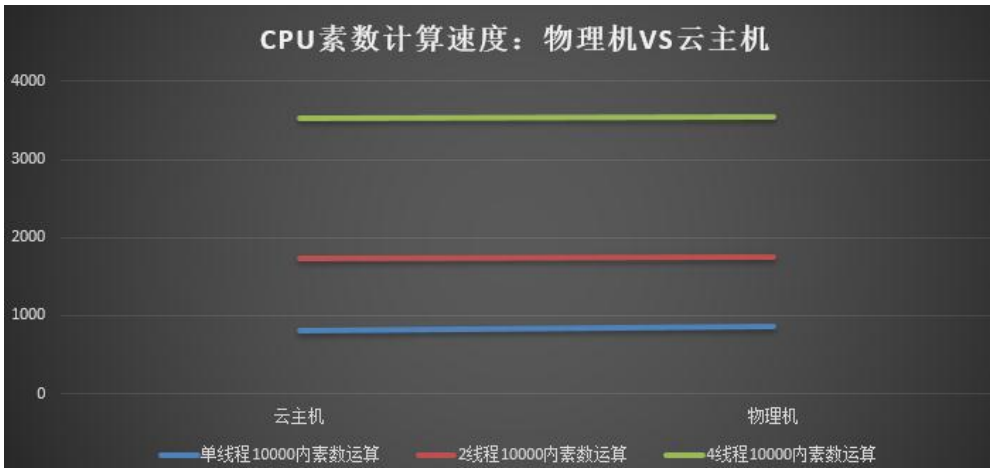


图 30：物理 CPU、云主机 vcpu 性能对比

参考上图 30 中的数据，在同等硬件环境、相同工具下，虚拟机 cpu 素数运算性能与物理节点的 CPU 性能几乎“一致”，但这里的“一致”，是有前提的，需要保证在执行测试过程中不会受到其他虚拟机干扰，因为通常虚拟机 vCPU 是随机共享，vCPU 的共享方式会根据节点宿主机上其他主机的负载，将 vCPU 调度到不同核心上，当然也可能在同一核心上。

不过这样对比并不公平，过于理想化。因为一个宿主机资源不可能只提供给 1 台虚拟机使用，绝大多数的情况下，单个计算节点上同时有多台虚拟机存活，他们互相争抢 CPU 的使用权，另外，宿主机所在 Linux 系统还可能会将内存交换、软中断等进程调度

到虚拟机正在使用的物理核上，这些因素叠加必然会导致虚拟机相对于宿主机 CPU 性能产生抖动（性能抖动又是另外一个话题了）。

四、总结

测试 10 秒钟，分析 8 小时，一个 sysbench 测试 CPU 的过程，看起来不起眼，但很多地方仍然存在疑惑，还需再挖掘。

从工具执行原理，到操作系统内核影响，涉及编译器，再到 cpu 指令集，还有 vcpu 或 qemu-kvm 在宿主机的进程调度带来的性能开销。各个环节，每一个单独拿出来都是长篇大论。

就目前已知的性能问题发现或瓶颈看，可以尝试四个方向的性能调优方案:

（1）CPU 独占

这是目前各大云计算厂商常用的方法，前文也提到同一个宿主机节点的 cpu 资源被多个虚拟机争抢，且因为 cpu 的抖动，增加 cpu 切换，导致虚拟机计算能力的不稳定。如果使云主机机独占物理 CPU，虚拟机的 vCPU 能够固定绑定到宿主机的指定 CPU 上，在整个运行期间，避免 CPU 浮动，减少 CPU 切换开销，能够一定程度上提高虚拟机 CPU 计算性能。

（2）编译器优化

不久前华为发布最新产品 P30，这款手机因为各种段子火了，另一个火的就是他家的方舟编译器，号称可以实现 Android 性能革命。

不要小看编译器，它是应用程序和 CPU 硬件（指令集）之间的桥梁。GCC 是目前广泛应用的 Linux 系统的默认编译器，在很多 Linux 桌面发行版、服务器版上都能看到它的身影，但是在各硬件平台上却并不是性能最优的编译器。

这里有 2 个方向:

- 取消 GCC 自动编译，采用人工编译，开启 GCC 全部优化项
- 直接用 intel 自家开发的 ICC 取代 GCC，Intel 基于 x86 架构 CPU 开发出 ICC，能够最大程度发挥 x86 架构 CPU 的真实性能

（3）bios 调优，启用睿频

曾经有过启用睿频大幅提升存储 io 性能的经历，那时对 io 请求进程做了频率上限导致性能被压制。开启睿频，可以让 CPU 根据实际运行程序的需求，动态增加运行频率用来提高处理器的性能。

（4）sysbench 库函数 clock_gettime 调用

sysbench 对 clock_gettime 调用频繁，这虽然与它本身设计有关系，但如果通过修改源码让其调用更少库函数的次数，也是未来的调优方向。

《51 测试天地》(五十四) 下篇 精彩预览

- “软件测试已死”了吗
- 测试环境管理：测试环境该填坑了
- 5 种常见的测试用例设计方法
- 为 Web 元素创建最终 XPath 的 20 种最佳方法
- Docker 领路，走进压力测试的现代化
- 深入研究可持续性的测试自动化
- 压测必经之路，解读 JMeter 分布式
- 词根字典用户测试
- 互联网时代测试女巫的自动化开篇

马上阅读