目 录

Contents

2013 年 07 月

基于用户行为的生成自动化测试用例方案.....01

【淘测试】大型商品中心测试策略.....04

数据库 diff 的脚本实现与应用.....15

【淘测试】文本分类算法评测.....23

高容量自动化测试概述.....29

测试 Python 和 C#代码.....37

自动化测试之脚本维护.....49

报表类模块测试一些注意点读.....54

软件质量评估模型与应用系列.....57

手机软件兼容性测试.....63

做好软件测试提升医疗软件服务质量.....67

基于用户行为的自动化测试用例生成方案

作者：咖喱

摘要

测试自动化，是测试领域的一个重要方向。目前，各个方向的自动化也发挥着不同程度的节约人力的作用。然而，在拥有了一个好的自动化框架的基础上，频繁的业务改动的情况下，编写、维护自动化 Case 本身还是一个机械劳动，尤其是前端页面自动化。对测试人员本身的素质提高也好，项目自动化投入产出比也好，都是一个瓶颈。本文提出了，自动生成测试用例的方案，配合自动执行测试用例框架，能够更加彻底地解决人肉回归或者人肉编写自动化 Case 问题，降低研发成本。

关键词

测试用例自动生成前端自动化

背景

前端自动化一直是争议不断的问题。主要矛盾集中在变化太大，维护成本太高，在实施过程中投入产出比太大。因此，本文针对这个问题，提出测试用例的自动生成方案，希望能给读者带来帮助。

正文

测试自动化，是测试领域的一个重要方向。现在，各种方向的测试，都涌现出各种自动化框架搭建方案：前端自动化，接口测试自动化等。自动化的目的，当然是将劳动力，从繁杂，冗余的重复回归劳动中抽取出来。然而，在拥有了一个好的自动化框架的基础上，频繁的业务改动的情况下，编写、维护自动化 Case 本身还是一个机械劳动，对测试人员本身的素质提高也好，项目自动化投入产出比也好，都是一个瓶颈。因此，在有了一个好的自动化框架之后，想办法把编写自动化 case 的过程也自动化掉，也就是测试用例自动生成，才能彻底把测试人员从模块回归工作中剥离出来，是一个很有价值的工作。

参与了不少模块级别的测试，有前端 Web，后端接口，server 类，自动化测试用例维护成本最大的就是前端 Web 的测试用例。由于前端页面迭代进行，形式变化多样，导致 Case 需要调整的可能性很大。因此，本文针对前端 Web 进行测试用例自动生成的方案设计。

本方案有个假设：

1、上线的版本的新功能都是经过人肉测试的。

2、上线的版本的基本功能都是没问题的。

基于数据驱动的测试用例，测试输入和预期输出是不可缺少的两部分。前端 Web 的测试输入就是用户行为、操作，预期输出分三类：跳转、DB 字段检查、文件存储。而前端 Web 自动化框架中，一般仅对跳转进行检查，DB 校验和文件存储在服务器端，检查较为麻烦，可以用异常监控来作简单的校验，这里不进行详细描述。

测试用例自动生成方案有四大块工作需要进行：

第一：已经具备针对该项目的 Web 自动化执行框架

比如：基于 selenium 的自动化框架（控件层-页面层-Case 层），（原子场景-场景-Case 层）等，根据各自项目特点已经能顺畅使用的框架。

第二：在 RD 的代码中，Server 层加入特定标识的 LOG

如：在新建 Plan 的 Server 层，在 Plan 接口中，执行代码前，打上 LOG，LOG 内容含：[4Case]logId，userid，action:addPlan，argslist，timestemp。在异常分支中打上预期失败的 LOG，LOG 内容含：[4Case]logId，userid，expect:fail，timestemp。

在正常分支代码中打上预期失败的 LOG，LOG 内容含：[4Case]logId，userid，expect:pass，nextUrl:XXX，timestemp。其中，logid 在用户登录时创建，同一个用户登录登出多次，logid 不同。

第三：建库

将 LOG 和基于框架的 Case 代码，用数据库表的形式一一对应起来。表结构：

```
No.   LOG   CaseCode
No：1
LOG：{…..}{….} action:addPlan argslist:{……}
CaseCode:   Scene.AddPlan(…..)
No：2
LOG: {….}{….} expect:pass,nextUrl:XXX
CaseCode:   Scene.CheckJump(….)
```

接着举新建 Plan 的例子，RD 代码中，由于执行了 plan-server 这个接口，必然有两条日志，一条 action 类日志，一条 expect 日志。第一条 action 类日志，在数据库里匹配为 No1，第二条日志匹配为 No2。那么，自动化的 Case 将应该是：

```
Scene.AddPlan(….)
Scene.CheckJump(….)
```

第四：测试用例自动生成的框架

即将 LOG 转化成 Case 的框架。下面将详细描述该框架的内容。该框架有个

前提条件，就是前面的假设：上个版本的新功能是经过人肉测试，并且是没问题的。如果有问题，将会影响 LOG 中 expect 的准确性。

框架步骤如下：

1、读取一段时间的线上日志，筛选出含[4Case]的日志，按时间 timestemp 重新排序。

2、将所有日志按 logId,userid 分类，放入 arraylist CASE 中，一个 logid+userid 标识一个用户的一次系列操作行为。

3、循环读取 arraylist CASE 中的日志，一次处理一个日志。

4、读取一次系列操作行为，里面含 action 和 except 标示日志多个。按时间次序挨个去数据库里匹配 LOG，读取对应的 CODE 字段，拼成可执行的测试用例代码。

5、将测试用例代码放到测试用例模板中，重命名文件，如：Case1.java

```
Case1.java
import org.testng.Assert;
import org.testng.annotations.BeforeTest;
import org.testng.annotations.Test;
import scene.MaterialScene;
import util.UrlMgr;
@Test
public void Case1(){
Scene.AddPlan(….)
Scene.CheckJump(….)
}
```

其中，绿色纹底的代码是框架自动填写的，其余是用模板定义的。

基于上述四项工作，可以利用线上用户行为自动生成出能在自动化框架跑的 Case。

大型商品中心测试策略

作者：韩锣

解决场景：

1.大规模数据中心

2.依赖应用较多（上百个）

效果：

1.将近 9000 多个脚本每日回归成功率 高于 99.5%，发布前脚本成功率均保持 100%

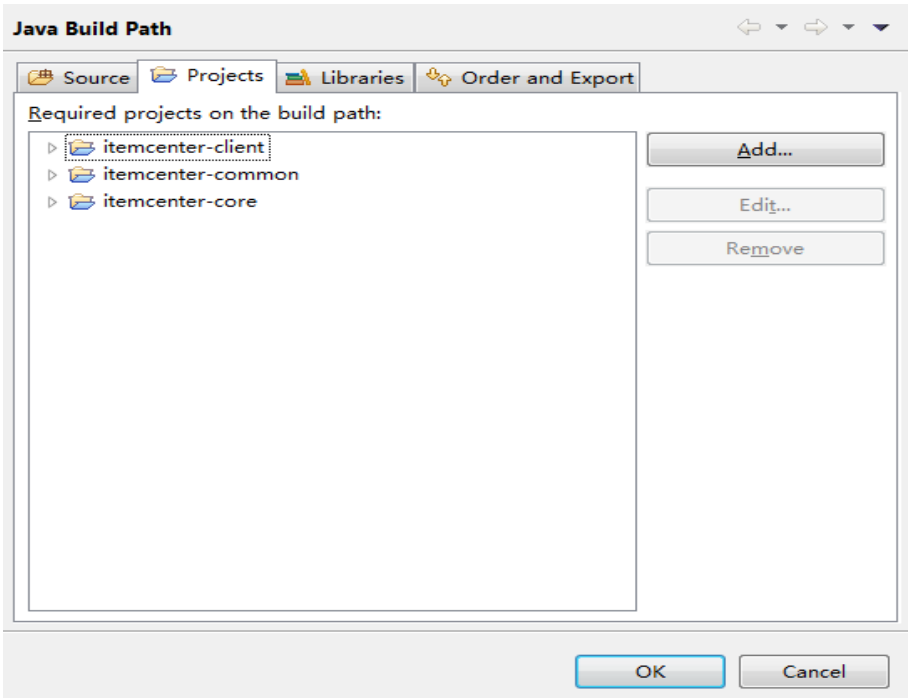
2.互相依赖的系统能够及时回归，监控陷阱的脚本单独独立

测试开发工程结构：

工程依赖与 jar 包依赖的选择

商品中心选择工程依赖

当然也可以选择 jar 包依赖，将 build path 的开发工程直接移除，一般情况下建议测试工程直接依赖开发工程，好处是：



1.开发工程修改后，测试工程只需要直接更新即可

2.可以通过引用的方式直接引用开发的配置文件（后续会讲到），防止测试工程也需要维护一份开发工程的文件，导致不一样（很多测试工程直接把开发工程的配置文件拷贝过来，坏处就是如果开发工程修改了配置文件，测试工程也需要再重新拷贝一下，经常这样挂一片代码）

当然测试工程直接依赖开发工程坏处则是：

1.当本工程提供的功能是通过 client 包提供出去时，则测试工程测试的是未打包的代码，打包的代码并没有测试。

2. 如果开发工程不稳定,老是报错或者编译错误,则测试工程受影响很大,并且大部分时候测试是无能无力的,只能找开发帮忙。

POM 的配置

<maven 插件>

<开发工程依赖>

<测试依赖>

<第三方工具依赖>

这个按照各自工程配置即可，不过有个 maven 插件可以配置，注意一下：

```
<plugin>
<groupId >org.apache.maven.plugins </groupId >
<artifactId >maven-surefire- plugin</ artifactId>
<configuration >
<argLine>-XX:PermSize=128m-XX:MaxPermSize=128m</argLine>
<includes >
<include >**/qatest/service/**/*Test.java </include >
<include >**/qatest/utilCase/**/*Test.java </include >
<include >**/qatest/project/**/*Test.java </include >
<include >**/qatest/business/**/*Test.java </include >
<include >**/qatest/dao/**/*Test.java </include >
</includes >
</configuration >
</plugin >
可以通过 includes 指定需要跑的测试类，如果不指定则全部跑
```

测试文件结构

良好的测试文件结构非常有助于后续工程的维护

测试文件结构分为测试代码和测试资源文件，

测试代码：

base：基类

mock：mock 类

qatest:测试类

business:

dao:

project

search

utilCase

utilTest

serialize:

service.impl:

util:

测试资源：

db: 数据库资源

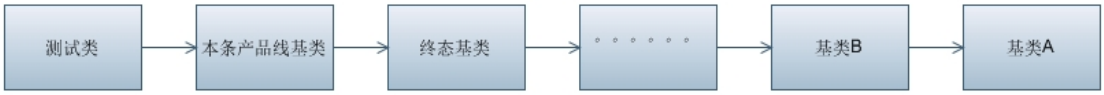
image: 其他资源

mock：mock 资源

service: 加载的 service 资源

公告资源 xml：集中加载类

类的继承关系结构



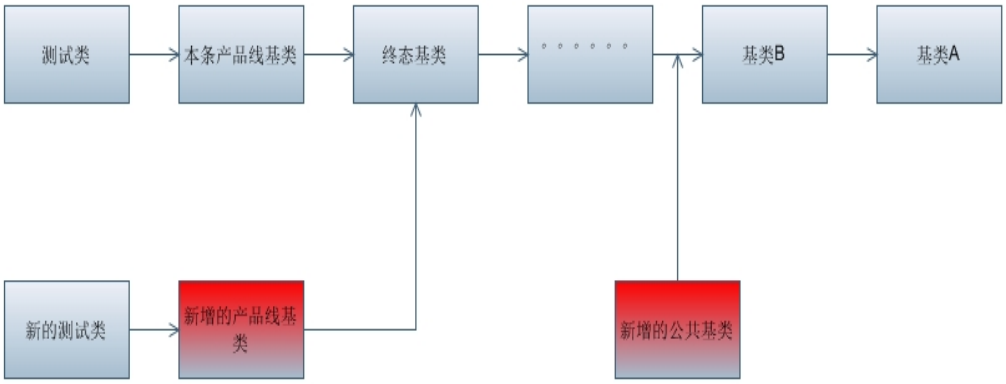
类的继承关系为测试类继承本条产品线基类，再继承终态基类再继承，如基类 C、基类 B、最后继承基类 A

其中基类 A：一般指的是所使用的测试框架所需要继承的类

基类 B、基类 C 等：指的是按照不同功能划分的基类，但是具有通用性，大部分测试类都可能用到它的公有方法

终态基类指的是最终可能要被测试类继承的基类

本条产品线基类指的是本次新增加的功能，但是其他大部分的测试类不会用到他的公有方法



该结构的好处是当测试的数据中心出现新的公共基类时（它的公共方法其他的新增测试类可能都需要），则扩展新比简单，只需要在终态基类和测试类 A 直接再添加一个基类即可

而当新增加的产品线基类不具有公共性时，则只需要使新的测试基类直接继承此新增的产品线基类，然后该新增的产品线基类直接继承终态基类即可以

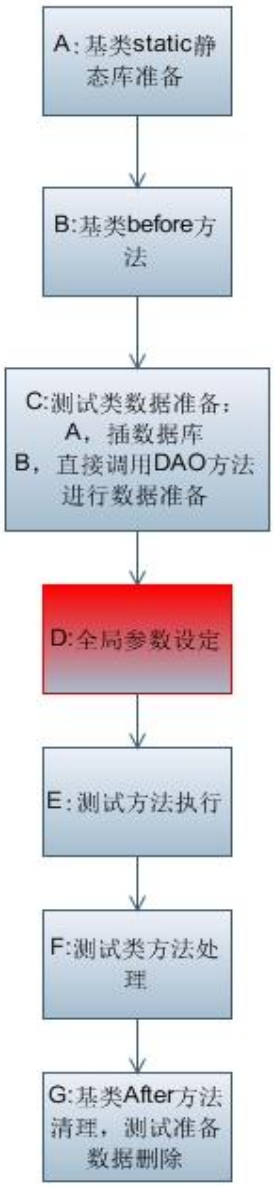
保持良好的基类结构非常重要，当后续出现大规模的用例时，良好的基类结构可以避免很多重复性的公共方法的编写，同时防止由于用例的急速扩张而使后面失控，当开发工程一个方法变动时，测试由于公共方法东一块西一块而需要进行大规模更改

需要补充说明的是根据经验，测试用例的执行时间并不会因为继承的类特别多而速度显著下降，同时也要提醒，我们需要避免把所有的方法都放到基类，并不是说不好，而是因为基类的方法太多则后面不方便自己查找，所以我们还需要把一些方法放到 Util 中。

测试类的生命周期

简单一点的测试类的生命周期如上所示：

A，首先是基类 static 静态库准备（这个看开发工程而定，许多可扩展性较好的开发工程都需要动态的传入一些可配置的参数），这个步骤视开发工程而定



B，基类 before 方法准备，一些需要的 before 方法，不建议在基类 before 方法中进行类似于数据准备等工作，因为并不是所有的测试类都需要数据准备，所以当测试用例一旦多起来时会带来不少脏数据或者无用数据， 也不建议在基类的 before 方法中进行类似于数据清理等工作，因为在测试用例执行之前的数据清

理一般是全量清理，比如需要将一张表的数据删除等，性能太差，当测试用例数量上来时，每次全量回归所需要的视觉较长。

C，测试类的数据准备有两种方法

1）直接插数据库

2）调用开发 DAO 的方法插入数据

方法 1 的优点是比较方便，想准备什么数据都能准备什么数据，但是缺点也比较明显，比如业务需求为发布一个商品，该动作需要操作多张表格 ，则插入数据准备的成本较高。

方法 2 的优点是准备的数据比较真实，为直接调用接口插入的数据，缺点则是插入的数据依赖开发所提供的接口参数，并不是所有的数据都能插入。

一般如果业务比较简单建议用第一种方式准备数据，业务比较复杂则建议用第二种方法准备数据

D，在插入数据的过程中，可以将一些全局参数设定好。本持续集成使用的是调用开发 DAO 插入数据，比如在基类中插入一个商品时，在插入方法结束后，会将商品 ID 设定为全局变量，方面后面测试用例使用

E，测试方法执行

F，需要处理的测试方法

G，数据删除操作，在基类的 After 方法中需要执行测试数据删除

如下代码所示（均是伪代码）：

A ：基类 static 静态库准备

```
public abstract class BaseCase extends ITestSpringContextBaseCase {
@BeforeClass
static public void initSwichStatus() {
// 设置开关的默认值
CommonSwitchor.isCallMicCheckCspuId = true;
}
}
```

B:基类 before 方法

```
public abstract class AdminServiceBaseCase extends BaseCase {
protected Long ItemID= 0L;
@Before
public void before() {
```

```
// 清理本地缓存数据
txtFileManager.cleanTFS();
}
// 基类 After 方法清理数据
@After
public void cleanItemDataAfter() {
cleanItemDataBySql(ItemID);
}
}
// 设定全局参数
protected Map<Long, Date> creatItem() {
// 准备数据
ResultDO result =adminServiceClient.insertDataIntoDB()
// 设置全局变量
ItemID = result.getItemID();
return itemDateMap;
}
}
```

C:测试类数据准备：

1，插数据库

2，直接调用 DAO 方法进行数据准备

```
public class Test01 extends AdminServiceBaseCase {
@Test
public void test_01_01_test() throws IcException {
// 数据准备
creatItem();
// 测试执行
ResultDO result = adminServiceClient.xiaoerDownShelfItem();
// 测试类方法处理
ResultAssert.assertTrue(result);
}
}
```

D:全局参数设定

```
// 设定全局参数
protected Map<Long, Date> creatItem() {
// 准备数据
ResultDO result =adminServiceClient.insertDataIntoDB()
// 设置全局变量
ItemID = result.getItemID();
return itemDateMap;
}
```

E:测试方法执行

```
// 测试执行
ResultDO result = adminServiceClient.xiaoerDownShelfItem();
```

F:测试类方法处理

```
// 测试类方法处理
ResultAssert.assertTrue(result);
```

G:基类 After 方法清理，测试准备数据删除

```
// 基类 After 方法清理数据
@After
public void cleanItemDataAfter() {
cleanItemDataBySql(ItemID);
}
}
```

依赖系统的 mock

大型数据中心会依赖许许多多的系统，当依赖的系统出现问题时，则测试用例会失败，9000 多的测试用例，一旦每天失败超过 100 个以上，则维护成本会非常巨大，甚至需要专门安排一到两个人去维护，这样的脚本是没有意义的，因为：

1.脚本不稳定导致浪费了人力，需要额外安排不少人去排查

2.不利于持续集成，当开发提交代码后，不能马上发现到底是自己加入的代码导致脚本挂了还是其他系统挂了导致脚本挂了

Mock 系统的选择

需要 mock 的

1.不稳定的

Mock 系统的选择首要元素是稳定，如果这个系统很稳定，那么完全没有必要 mock，比如这个系统一年可能就改一次，而且平时环境都没问题。当然这种随着依赖的越来越多而变得都不稳定，比如 A 每天有 10 分钟不稳定，B 也有 10 分钟，那么一百个系统就疯了。这个时候就要考虑 mock 了。

2.数据特别难制造的

如果你准备这个应用的数据需要花费大量的视觉，那么你需要考虑 mock 他，比如你可能需要插 10 多张表，否则会报错，但是你仅仅需要几个字段，复杂系统经常出现的问题。

不需要 mock 的

1.与 session 等浏览器身份识别有关的

根据经验，涉及到 session、cookies 等 mock 特别痛苦，因为一般情况下系统很多后面需要用到 session、cookies 里面的数据，而且要求是真是数据，此时 mock 成本会很高

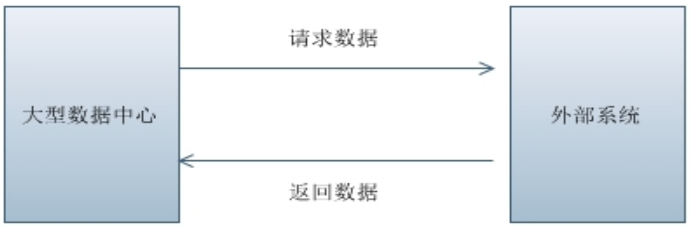
2.稳定的

如果依赖系统不多且其他系统比较稳定，不建议进行 mock，有种费力不讨好的感觉

因此大型数据中心，尤其是还在持续开发，比如每周有发布的数据中心需要 mock 外部系统，以提高自身的可用性。

mock 的场景比较多，本次主要介绍系统间进行数据推送的 mock 方式

下面通过一个地址的推送举例说明下，比如通过外部系统推送一个地址列表



给大型数据中心，调用外部系统提供的 client 包，

开发 bean

```
public class DefaultAreaBO extends AbstractBO implements AreaBO {
```

```
private List<AreaDO> areaDOList;
/**
* 取得数据库 DBC lg_area 表中的全部的地址数据
*/
public List<AreaDO> getAreaList() {
//读数据库等操作　伪代码
areaDOListTmp = ReadDatabase();
areaDOList = areaDOListTmp;
return areaDOList;
}
}
```

测试 bean

```
public class AreaBOMock implements AreaBO {
private List<AreaDO> areaDOList = new ArrayList<AreaDO>();
@Override
public List<AreaDO> getAreaList() {
areaDOList　= 测试准备数据;
return areaDOList;
}
}
```

我们最终会使用外部系统 DefaultAreaBO 这个类来获取地址，mock 方法如下：

1.在 spring 中加载

```
<!-- 开发 bean -->
<bean id=" area"　class="com.taobao.common.category.biz.impl.DefaultArea BO">
<!-- 测试 bean 区域信息 BO -->
<bean id ="area" class= "com.taobao.item.mock.millau.AreaBOMock" init-method= "init" />
```

通过 spring 的加载顺序，先加载开发的 bean，后加载测试的 bean，则最终 area 加载的为测试的 bean，所以代码最终执行 getAreaList()时不会调用开发的：

```
public List<AreaDO> getAreaList() {
//读数据库等操作　伪代码
```

```
areaDOListTmp = ReadDatabase();
areaDOList = areaDOListTmp;
return areaDOList;
}
```

而是最终会调用测试的

```
public List<AreaDO> getAreaList() {
//读数据库等操作　伪代码
areaDOListTmp = ReadDatabase();
areaDOList = areaDOListTmp;
return areaDOList;
}
```

通过上诉方法很好的解决了推送数据的 mock 工作

数据库 Diff 的实现与应用

作者：ga_li

【摘要】

在测试过程中，数据库的变化是 Case 检查环节一个很重要的检查点。目前常用的直接查询数据库的方式有两个弊端：1、编写 case 检查点工作量大。2、检查字段不全导致漏测。本文提出数据库前后状态 Diff 的方式进行结果检查，并实现了 diff 脚本接口，最后给出调用方式案例，希望能给大家带来启发。

【关键词】

数据库检查，Case 结果检查，diff，数据库 diff

【正文】

一、背景与预期：

测试人员编写 Case 的过程中（不管是人肉测试还是自动化测试），数据库的变化值是一个很重要的检查点。目前最常用的检查方式就是直接查询数据库中相关字段的值，一般通过直接写 sql 或者调用一些封装好的查库接口。这种方式有两个弊端：第一、写 Case 时工作量较大，由于每个被测对象或者说测试用例要校验的字段不同，需要编写不同数据库 Check 接口。第二、容易漏测，这种方式仅能保证测试人员关心的字段，若发生被测对象新增、删除、修改了一些不该变化的地方，这种方式是肯定发现不了的。

本文提出一个数据库 Diff 脚本，该脚本实现了将 Case 执行前后，数据库所有变化，以数组 Array 的形式返回给调用者。测试人员通过调用该脚本，能够有效提高编写测试 Case 的速度，并且能避免上述类型的漏测。

Diff 命令在大家看来并不陌生，diff fileA fileB >fileC，linux 命令中，就是将两个文件的不同点表示出来。数据库 diff，linux 系统并没有相关命令，本文受原 Diff 命令启发，编写了该脚本 DataBaseDiff，希望能给读者带来帮助。

二、解决方案：

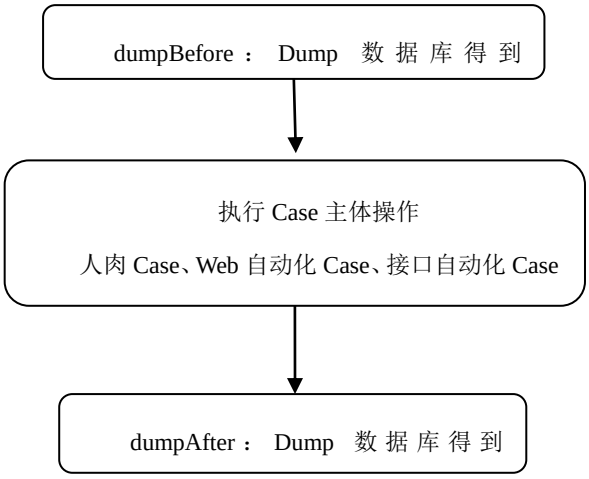
DataBaseDiff 提供三个外部调用接口，dumpBefore（），dumpAfter（），DBD（）。它们的功能分别是：

dumpBefore：在 Case 执行之前，调用该接口，以特定格式，dump 出数据库数据，并保存到文件 Before.sql 中。

dumpAfter：在 Case 执行之后，调用该接口，以同样的格式，dump 出数据库，保存在文件 After.sql 中。

DBD：核心功能，将上述两个 sql 进行文件 diff，得到 diffFileC 后，通过 Before.sql 和 diffFileC 两个文件进行分析，得出修改的结果，以数组 Array 的方式输出。

整体流程图如下述图表 1。



图表 1 数据库 Diff 脚本应用框架

下面对这三个接口进行详细描述：

1、dumpBefore 接口：

定义该接口，主要是为了限定 dump 后 sql 文件的格式问题。若让外部调用程序自己直接调用 mysqldump 命令产生 sql 文件，由于每个程序员习惯不同，喜欢的 mysqldump 的参数也不尽相同，产生的文件格式不同，会影响 DBD 对 dump 后文件的解析，本脚本使用下面的命令来 dump 文件：

```
./mysqldump -uroot -p123456 'DataBaseA' --no-create-info --complete-insert=true >./Before.sql
```

Dump 出来的 Before.sql 格式为：

```
1  LOCK TABLES `AlipayUser ` WRITE;
2  /*!40000 ALTER TABLE ` AlipayUser ` DISABLE KEYS */;
3  INSERT INTO `AlipayUser` (`id`, `name`, `type`, ` time`) VALUES
4  (1，ga_li，1，'2012-12-03 14:43:23'),
5  (2，myPepleo，1，'2012-12-03 14:48:07'),
6  (3，hello，1，'2012-12-03 15:10:26'),
```

2、dumpAfter 接口：

同 dumpBefore，格式一致。两个 sql 通过文件 diff 后，得到的 diffFileC 文件如下：

```
4，6c4
< (10，'2013-05-14 12:13:50'，NULL，test)，
< (10，'2013-05-14 12:13:50'，NULL，abc)，
< (10，'2013-05-14 12:13:50'，NULL，selenium);
---
> (10，'2013-05-14 12:13:50'，NULL，delete)
7a12
< (10，'2013-05-14 12:13:50'，NULL，test)，
< (10，'2013-05-14 12:13:50'，NULL，aa)，
30d
>(10，NULL，test)，
```

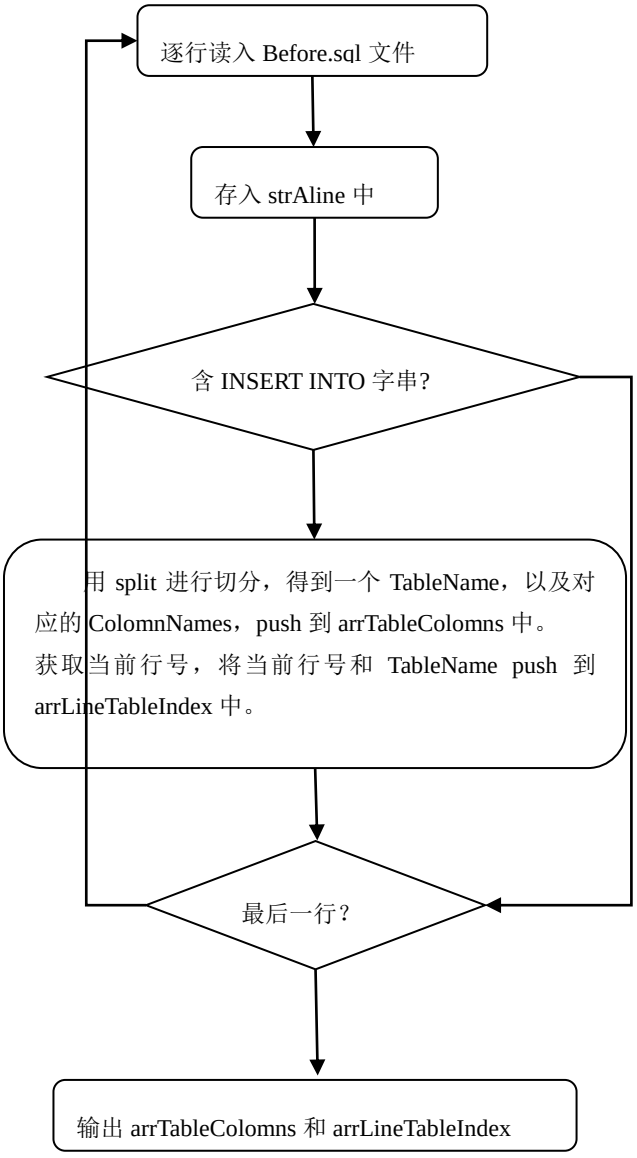
3、DBD 接口：

DBD 接口是主要的处理流程，针对上述 Before.sql 和 diffFileC 文件的格式进行解析，得出数据库变化的结果，输出 Array 文件。处理步骤如下：

i 通过分析 Before.sql 文件，得到 1.所有 Table 的属性 TableName，ColomunNames，命名为 arrTableColomns。2.Before.sql 文件的行号和 Table 的对应关系，命名为 arrLineTableIndex。

第一个 arrTableColomns 数组，是为了 1.后面校验值和 Colomns 属性个数是否匹配。2.输出修改内容的时候，可以知道修改的是哪个 Colomn 对应的内容。

第二个 arrLineTableIndex，是因为在 diff 文件中，并不能知道某行增加的内容属于哪个表的，所以，只能通过行号，在 before.sql 文件中定位对应的 Table。具体实现方法如下图表 2：



图表 2

得到的 arrTableColomns 和 arrLineTableIndex 格式如下所示：

```
arrTableColomns：{‘TableA’：{‘id’，’name’，’type’，’time’}，
‘TableB’：{‘id’，’chargename’，’signal’}}
arrLineTableIndex：{
TableA，
TableA，
TableA，
TableB}
```

ii 通过分析 diffFileC 文件，得到增加的行、删除的行以及修改的字段，分别放入输出数组 arrInsert，arrDelete，arrModify 中，这三个数组也是最终外部接口得到的结果。

下面分析下 diffFileC 文件的特点，文件 diff 结果包含三种，参考上文给出的 diffFileC，其中：

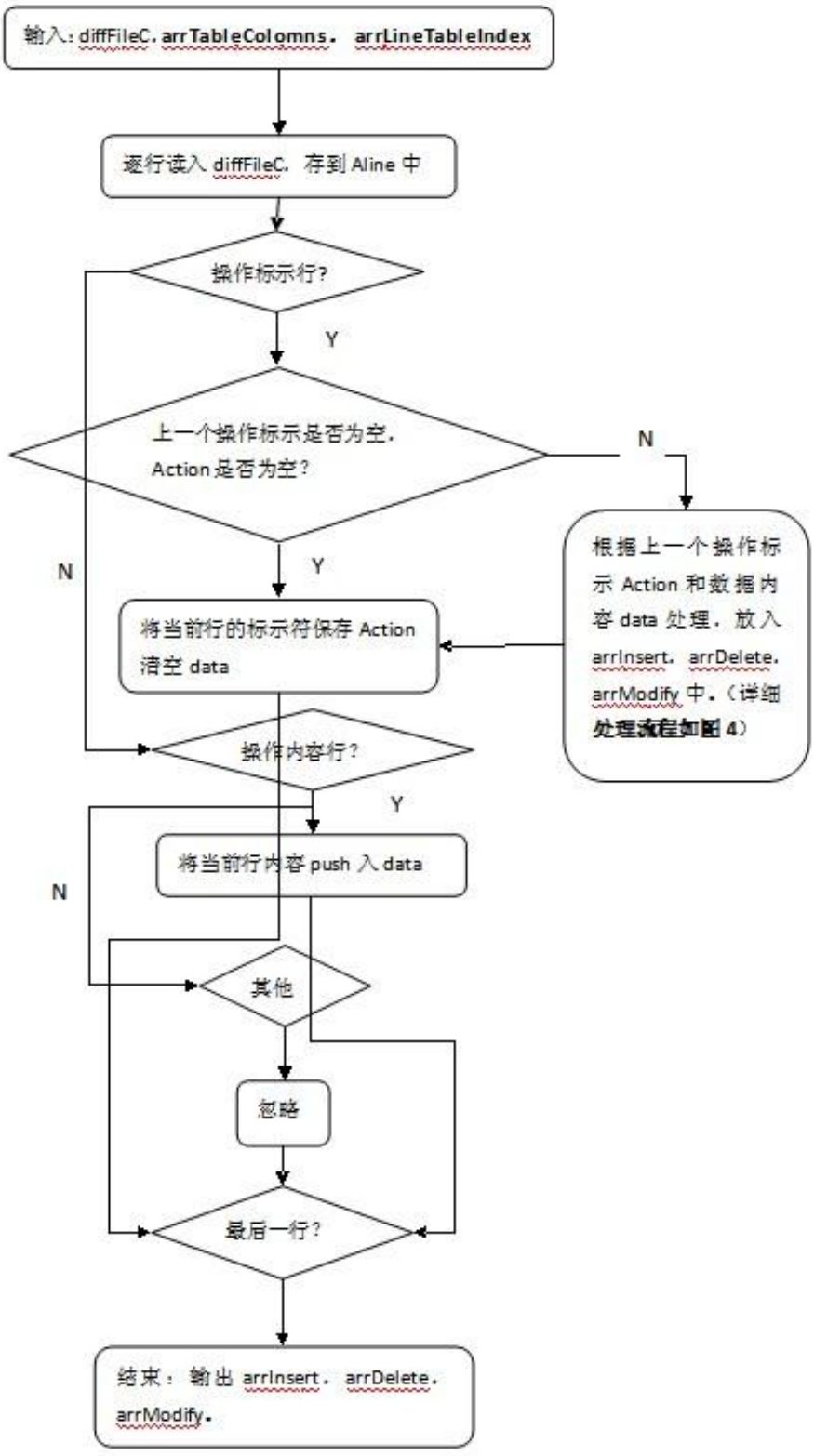
操作标示行。如”12a23”，表示原第 12 行后面，增加了一行，字母 a 代表 add。此类操作标示符一共有三个，分别为：a（add），d（delete），c（change），我们可以通过操作标示行知道数据库进行了哪种变化。字母 a 和 d，能够确定是在数据库中增加或者删除了某行。然而，c 有两种情况，可能是数据库中进行了某个字段的修改，也可能是某个表中有增加、删除、修改的操作，导致 dump 出数据不在原来的位置，因此需要单独处理。

操作内容行。如”>XXXXX”，表示增加了符号>后面的内容。”<XXXX”，表示删除了符号<后面的内容。

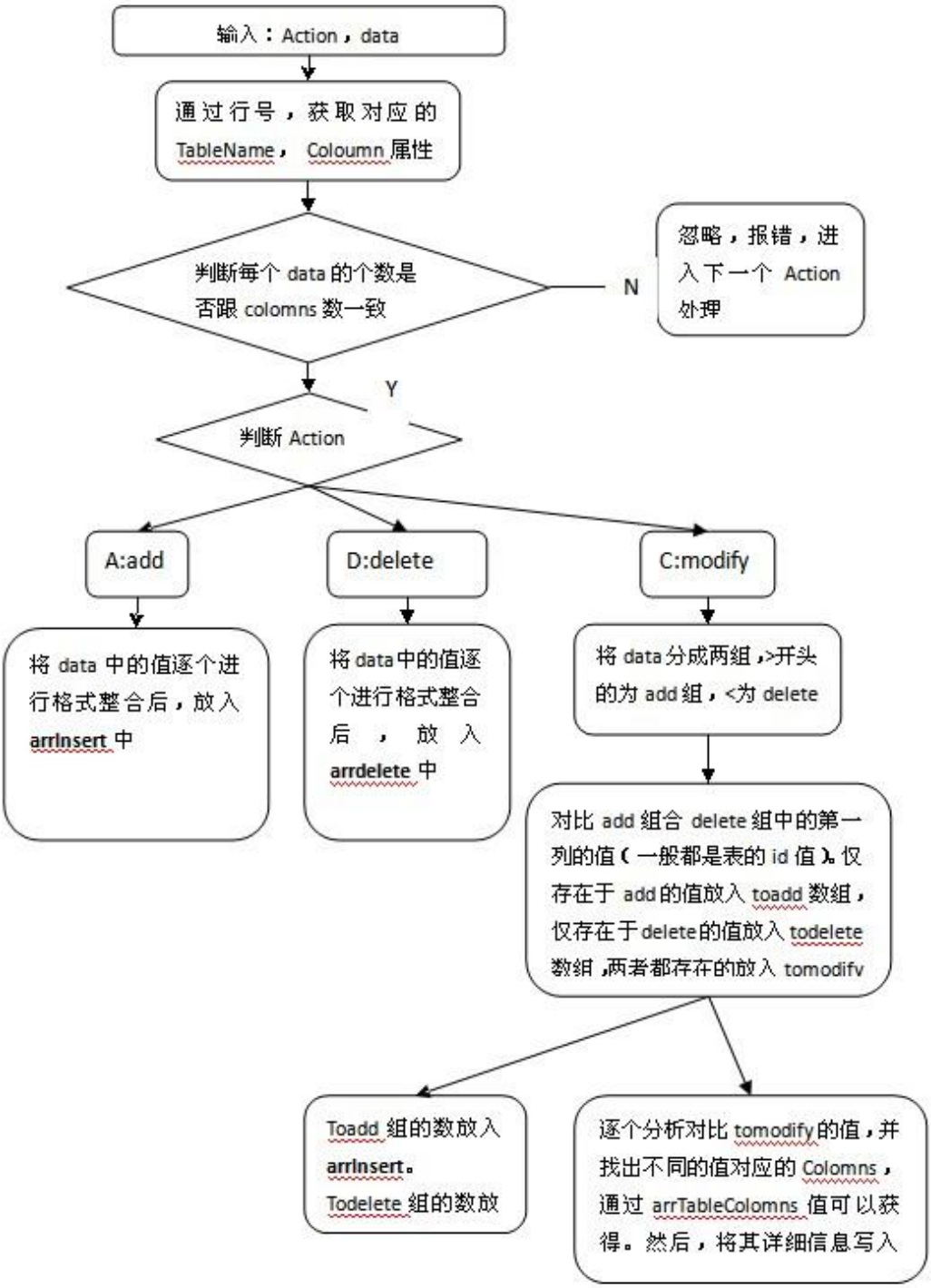
信息行。如”------XXXX”，这些是跟处理无关的信息行。

基于上述对文件 diffFileC 的格式理解，以及 i 步骤中 table 信息的输入，ii 中处

理流程如下图：



图表 2 步骤 ii 处理流程



图表 4 对 Action，Data 的一次处理（局部）

最后，得到的结果形式如下：

```
arrInsert:{‘TableA’:{1:{“1”，”ga_li”，”2013.2.2”}
2:{“2”，“xiaoming”，“2014.2.2”}
}
‘TableB’:{1:{“123”，“18282”，“aaa”}，
}
}
arrDelete:{‘TableA’:{KeyIndex:“Colomn1”
KeyValue:{“1”，”2”}
}
‘TableB’:{KeyIndex:“Colomn1”
```

三、使用效果及应用方法

该脚本已投入使用，得到较好的评价。一方面，节约了 Case 处理查询数据库的时间。另一方面，返回值仅需简单处理就能完成 Case 校验。

下面列举下在 Web 自动化测试、接口测试、人肉测试，该脚本的使用案例。

Web 自动化测试 Case 中使用该脚本：

```
WebSeleniumCaseAlipay()
{   dumpBefore();
runAplipayAction();
dumpAfter();
result=DBD();
checkModifyData(result.arrModify);
AssertNull(result.arrInsert);
AssertNull(result.arrDelete);}
```

在接口测试中，使用该脚本：

```
TestInterfacePay()
{   dumpBefore();
runPayInterface();
dumpAfter();
result=DBD();
checkInsertData(result.arrInsert);
checkModifyData(result.arrInsert);
AssertNull(result.arrDelete);}
```

人肉测试中，脚本能发挥的作用。使用者可以自己做一个 web 界面，上面

有两个按钮，执行前数据库存档、查看数据库变化。它们分别对应 dumpBefore（），dumpAfter&DBD（）。在人肉测试中，一般在 rd 自己自测调试时，可以观察对数据库的影响。用起来比自己去查数据库方便，也不容易漏掉。

希望该脚本能给大家带来方便，欢迎一起探讨。

文本分类模型的介绍与评测

作者：周中晟

信息技术、Internet 的发展，使人类从信息匮乏步入了信息过载的时代，在这个时代中，无论是信息的拥有者还是信息的消费者都遇到了很大的挑战，面对各种海量信息的复杂性和非结构性，如何高效、便捷地认知自己拥有的海量数据对象，是信息的拥有者一直在尝试和思考解决的问题。随着技术的发展人们对数据的认知已经从过去数据报表展示，回归到挖掘数据背后所反应的本质。

本文通过介绍数据挖掘应用中的文本分类模型的处理方法和过程，介绍了目前在我们的项目中使用的对文本分类的评测的方式方法，评测指标，以及评测方法的推广等相关问题。由于该领域评测方式的相关资料比较有限，所以在此抛砖引玉，希望有更多的人一起来将该领域的评测方法进行完善，也希望该领域的专家们能提出宝贵的意见，对理解错误之处加以斧正。

在本文分类领域常见的方法有：分类和聚类两种。分类和聚类不仅是文本分类处理中的常用手段，更是人工智能中比较常见的两种方法。

在文本分类当中，分类和聚类都是将对象归类的一个过程，但它们分别有自己擅长的领域。分类(classification):是事先定义好类别，类别数不变，是一种已知类别情况，对不同文本进行分类的处理过程，通过找出描述并区分数据类或概念的特征规则或模型(或函数)，以便能够使用模型预测类标记未知的对象类。具体的处理方法既可以通过特征，规则等方式将不同的句子归类到不同的类别，也可以通过分类器将一条语句归类到不同的类别当中。它们都需要人工的介入，属于监督学习的范畴。唯一的区别是：通过特征词和规则等方式来进行分类往往是由于缺少用于分类训练的语料而“不得已为之”的处理方式；而通过分类器进行分类，丰富的语料数据作为模型训练的“原料”是处理的基础，需要由人工标注的分类训练语料训练得到，这种语料既可以来自以往结构化后的后台处理历史，也可以来自日常有意识的人工分类积累的结果。分类技术在数据挖掘中是一项重要任务,目前商业上应用最多。分类的目的是学会一个分类函数或分类模型(也常常称作分类器),该模型能把数据库中的数据项映射到给定类别中的某一个类中。

相对分类而言聚类则没有事先预定的类别，类别数不定的场景。聚类不需要人工标注和预先训练分类器，类别在聚类过程中自动生成。主要处理过程是把对象向量化，通过数学建模计算各向量与质心的距离，不断的聚集迭代替换质心，直到收敛的过程。

分类适合类别或分类体系已经确定的场合，比如根据文章特征分类文章在门

户中的类别；聚类则适合不存在分类体系、类别数不确定的场合，一般作为某些应用的前端或无人工参与分类的场合，比如搜索引擎结果后聚类(元搜索)等。

而具体到文本分类中，真正分类的过程仅仅是一小个步骤而已，它还需要做分词的处理，停词的提取，特征词的提取，无效句子的识别，关键句的抽取，近义词的替换，文本内容向量化，语料库的建立和维护等步骤。那说了那么多，那如何评测分类出来的结果好坏呢？下面举一个场景来说明我们在项目中使用的和计划实施的评测方法：

客户每天产生成千上万的反馈信息，通过编写模型算法将客户的反馈自动分类为 A、B、C、D 中业务类型。那么目标就是构造一个分类模型，将出现的文本分类成 A、B、C、D 四个类和其他（不可识别或不属于四类之外的其他）类别。

很显然在这个场景中不适合使用聚类算法，聚类算法只是将模型认为相似的类放到一起，对任意一条文本进行归类都需要将所有的文本放到一起聚一遍，这相当影响文本分类的效率，而且分类模型并不能告诉模型使用者放在一起的类到底是什么类，并且聚类模型会将所有出现的文本都分别放在各自相似的类中，对于一些噪音数据或者本来不属于任何类的数据来说这种分类是不合理的，因此这个场景中仅适合使用分类算法。假设开发工程师们针对这个场景开发了一个分类模型，但模型的好坏不得而知，好坏的程度更加无法量化（模型的处理一定是从粗到细，不断调优，不断迭代更新的过程），那如何评测这个模型的优劣，怎么驱动开发区做模型的优化工作，还是直接发布使用这个模型呢？

我们认为不论是使用何种分类算法，都是把属于某个类别的文本分到相应的类型，因此我们抽象了下边十一个指标来衡量和评价分类器的好坏：

分词器的正确性。分词器的正确分词对识别反馈反应的意图学习有很大的帮助。

停词的处理是否合理，恰当。停词又叫停止词，是由英文单词：stop word 翻译过来的，停词常为冠词、介词、副词或连词。通常表示不能反应一个句子的关键意思的一类词。

特征词的选取是否正确，有无遗漏。特征词与停词相反，它是一个句子的核心和骨架，特征词反馈的是一个句子的关键含义。但是有时特征词与停词会因为不同的业务场景而相互转化。

覆盖率（Coverage）

覆盖率是对于一个已知分类结果的待分类集合的分类覆盖情况，对未知分类的文本的发掘能力。覆盖率的定义方式是分类结果集中分类出的类型的个数比上

整个集合中属于待分类类型结果的个数，比值就是分类模型的覆盖率。假设待分类的文本集合为 U，属于 A、B、C、D 四种模型的集合为$N_{v\in V}$，分类结果中属于四个类型的结果集为$R_{v\in V}$，不属于四种模型的集合为 N。那么模型的覆盖率可以通过以下的公式计算得到：

$$Coverage=\frac{|R_{v\in V}|}{|U-N_{v\in V}|}$$

分类召回率（Recall）

召回率是在分类结果中正确分类的结果在所有正确结果中所占的比值。可以用分类结果集和正确的分类结果集$T_{v\in V}$的的交集与正确的分类结果集$S_{v\in V}$的比值来计算，模型的召回率可以通过以下的公式计算得到：

$$Recall=\frac{|R_{v\in V}\cap S_{v\in V}|}{|S_{v\in V}|}$$

分类准确率（Precision）

准确率是在分类结果中正确分类的结果在所有分类结果中所占的比值。可以用分类结果集和正确的分类结果集的交集与正确的分类结果集$R_{v\in V}$的比值来计算，模型的准确率可以通过以下的公式计算得到：

$$Precision=\frac{|R_{v\in V}\cap S_{v\in V}|}{|R_{v\in V}|}$$

无效语句和噪音数据的处理逻辑是否正确。无效语句或噪音数据的处理会对分类器的覆盖率，召回率和准确率都会有所影响。

计算时间复杂度

计算复杂度主要影响的是生成计算结果的速度，可以通过单个文本和批量文本进行分类所耗费的时间进行度量。

模型的可维护性和可扩展性

可维护性对于描述模型的维护成本有很大的帮助，通常配置文件组织越合理，配置项越简洁可维护性越强，可描述性越强，维护人员对维护越容易理解，模型越容易优化和迭代；可扩展性是描述一个模型的复用度的衡量指标，对于其他业务的处理是可以直接通过修改简单的配置复用还是需要重复的开发一套完全不同的算法框架，这是衡量一个模型是解决一个问题还是解决一类问题的重要指标。

模型的兼容性

兼容性主要是指模型是仅仅适合离单机处理还是可以同时支持分布式处理，这种能力对于海量数据处理有很大的帮助。

模型是否有自我学习能力

这个能力有助于不断的改变模型分类的准确率和召回率，具备这个能力意味着模型有自我提高的能力随着时间的积累可以由不准到准，由准到更加精准。

假设我们有 10 万属于 A、B、C、D 四种业务类型的反馈，我们先拿出 8 万已知分类的反馈来提供给分类器进行学习训练，另外 2 万的进行分类器的各项指标的检查和测评，除此之外，我们还会加入一些与四种分类毫无关系的其他业务类型的反馈和一些无效的语句，一同来检查模型分类的效果。之所以要有一定数据量的数据来评测，是因为我们希望这个评测结果有一定的统计学意义。

按照上述的评测方法，我们会作如下处理：

对分类结果中反馈的分词作相应的抽查，以检查对于一些特定含义的词，是否做了正确的分类，比如：旺铺、天天快递、天天特价，淘宝，直通车等分词有无被模型按照通用的分词逻辑分词成：旺 铺、天天 快递、天天 特价、淘 宝、直通 车。以确保分类器的处理是与我们希望的处理逻辑相符。着重抽查的是分类错误和无法分类的部分的反馈的分词情况。

对停词和特征词做相应的抽查，以检查停词和特征词在特定的业务场景中划分是否合理，比如：在分类属于旺铺还是直通车问题的分类场景中类似：旺铺如何申请，直通车怎样报名等，客户反馈里的“如何”和“怎样”这类疑问词就应该被定义为停词。但在分类是属于旺铺类和直通车类的具体什么问题的场景中，“如何”和“怎样”就是一个特征词，他们表示的是客户不懂怎样参与这个具体问题。所以停词和特征词的处理的不好也会直接影响到分类的结果。同样也会着重的查看分类错误和无法分类部分的反馈的分词情况。

对覆盖率、召回率和准确率进行计算。这部分可以量化模型目前的状态和提供是否符合发布要求的具体量化指标。也可以很好的为迭代过程中判断模型优化效果的提供可量化的指标。

抽查对无法分类的数据存在的问题，主要是分词问题，停词问题还是特征词的问题，这对于改进分类器的分类能力有很大的帮助。

抽查对无效数据的检查，检查无效反馈是否真的属于无效反馈检查无效反馈的处理逻辑是否合理，优化模型的去噪能力。

计算的复杂度，主要是检查模型分类的速度问题，是否在业务允许的时间范围内完成了问题的分类，以判断分类器的具体应用场景应该使用在线分类还是离线分类算法。

检查模型的可维护性，可扩展性和兼容性和自我学习能力等，为模型的改进提供更合理的方向和建议。

以上是我们对文本分类模型评测的具体指标和指导的测试维度。在实施的过程中，还将会进一步的把相关的评测指标自动化，另一个很重要的工作是将需要人工维护的东西平台化和半人工化，通过日常的积累以及工作的化整为零逐步的减少人工的干预和投入成本，积累分类模型的原料“语料库”，并将测试和评测过的结论直接反馈到分类器的学习语料处理中，使测试与开发形成闭环，使分类模型能够不断的自我迭代，自我优化。

除此之外，文本分类中可量化部分（准确率，召回率，覆盖率）的计算依据是根据已知样本，检查模型对已知样本的识别能力，其计算方式、评测手段完全可以通过文本分类评测的计算方式推广到类似差评师，易流失等相关模型的评测中，因此文本分类模型的评测平台的建立解决的不是一种，而是一类业务场景的评测问题。

高容量自动化测试概述

作者：钟彩红

摘要

本文描述了高容量自动化测试（HiVAT）的基本概念以及有关该技术的十二个例子。这些技术一个共同的思路是自动化测试的生成，执行和评估多次随机测试。单一的测试往往是力量微弱的，但是集中到一块儿，就可以揭露单一的测试会忽略的问题。运用最简单的技术能达到高覆盖率：通过设计一定的特殊的用例发掘程序的弱点。其他的技术通过长序列测试将逐渐暴露程序的问题（如内存泄露，堆栈溢出，内存崩溃 ）或者是令人意想不到的时间事件。在我的经验中，这些长序列测试往往在嵌入式系统中发现了严重的问题，尤其是与一些处理器产生交互的系统中。我们把更多的软件嵌入到消费系统上，包括会造成性命攸关的危险的系统，我相信这些测试将变得至关重要。

关键字

高容量自动化测试 功能等价测试 恶意数据流测试

正文

这篇文章只是一篇粗略的草稿，作为佛罗里达理工学院（Florida institute of Technology）HiVAT 技术课程的介绍。它在课堂上给出了一个整体的结构。It is intentionally under-referenced。在课堂上学生的任务之一是通过高度脱节的文献、地图研究论文、从业者文档以及会议演示深入挖掘这儿列举的技术。通过帮助文档，或者直接相关的自动化支持技术和自动化策略/结果的调查文档的 add 部分，学生们可以汲取 HiVAT 技术的相关知识。若我们在课堂上取得进步，我会用在学术上更加完整的草稿来替代这篇文章。

考虑到自动化回归测试，我们会多次重复利用一个回归测试——也许在每一个版本中都会运行。但是这些测试真正做到自动化了吗？计算机可以执行测试并对结果做一些简单的评估，但是测试人员设计了整个测试，是测试人员编写了计算机能执行的测试代码，是测试人员给出测试输入数据，通过把数据直接写入测试代码或者在输入文件中指定特定的参数，是测试人员提供了程序用来评估测试结果的预期结果。而且如果有迹象表明出现问题，是由测试人员来检测结果，排疑解难，手工编写测试报告（程序崩溃的情况下）或者重新设计测试。所有由测试人员完成的工作都是手工的。

我们可以注意到，无论测试是在系统级别运行，使用如 QuickTest Pro 的工具，或者在单元级别运行，使用如 Junit 的工具，还是在混合级别进行，使用如 FIT 这样的工具，人的努力和计算机工作相辅相成，起了重要的作用。

所以，自动化回归测试实际上是具有自动化指示的手工测试。

而每一次手工测试都被自动化了。当你执行手工测试时，你可能键入输入值并查看输出结果，但是从输入值的接收到这些数据处理后结果的显示之间发生的一切都是在程序的控制下由计算机来完成——这就是自动化。

“手工”和“自动化”之间的区别是一个度的问题，而不是原则的问题。对一些测试而言，自动化部分比较多，而其他的自动化部分相对较少，不存在“全部”自动化测试或者“零”自动化测试。

专注于输入的高容量测试

高容量参数变化

我们可以通过将一些人的工作转化为计算机的任务来使自动化回归测试的自动化程度更高。例如，我们可以想象测试一个接收两个输入值（first，second）并返回两个值之和（sum）的方法。一个典型的回归测试包括两个特定的值 first，second 和预期结果，即两个值之和。但是假设我们使用提供随机数据的测试数据生成器来取代特定的测试数据，赋值给 first 和 second，计算预期总和的值 sum。用这种方式我们可以生成上十亿个测试，每一次测试都会有些不同，而人的工作量却几乎没有增加。

这是高容量测试最简单的例子之一。人提供算法，而测试工具运用算法创建测试数据，运行测试过程，并评估许多次随机测试产生的结果。

在这个例子中，那些测试用例几乎是相同的（相当相似）。为什么同时运行所有的这些测试会干扰呢？传统的回答是“互不干扰”的。域测试是一种广泛使用的软件测试技术。域测试的意义在于帮助测试人员选择一组数量较小的输入值代表最广泛的可能性从而将测试冗余最小化。通常情况下，这种做法效果不错。偶尔，程序使用小部分特定的值进行测试会有一些问题。例如，程序可能在处理一些特定的值时被优化，也可能容易出现一些太小难以注意到的计算错误，这些错误偶尔是可见的。

大数据量组合测试

与其考虑测试接收几个输入值的单一功能，不如想象测试一些更加复杂的功能。程序对若干输入值进行处理，也许有一系列功能，然后报告结果。同样，传统的测试方法旨在将冗余最小化。当我们进行组合测试时（若干个变量放在一起

测试），那么测试的数量是巨大的。若变量是相互独立的，我们可以通过组合测试（成对测试或者 all-triples（三次））将组合测试用例数量最小化。若变量是相关的，我们可以使用因果图技术来取代。但是如果我们认为程序只有当用小部分特定的值的组合进行测试时才会出现问题（当然我们不知道这些特定的组合），那么我们必须生成输入值，计算每个组合的预期结果来测试更大数量的值的组合。

对于生成输入值有以下几种不同的策略。

穷尽取样。这种测试方法测试了所有的组合情况。但是数量相当大而不可取。

随机取样。生成随机值作为输入，直到测试了某些庞大的数据。

优化取样。使用一个以某种方式优化组合集的算法。举一个相当简单的例子，如果你即将测试一百万个组合，那么你可以将组合的空间划分成一百万个同样大小的，不重叠的子集，或者你也可以使用给下一个组合分配值的排序算法，每生成一个测试用例都与之前的测试有一定的“距离”（根据某些距离函数）。

输入模糊

至今为止，我已经介绍了既生成测试输入值又生成预期结果的测试。预期结果的作用是作为准则，用来判断程序是否通过本次测试。测试准则是不完整的，但是对于自动化测试大有裨益。虽然我们不能检测到所有的故障，但我们可以检测到一定类型的错误（如计算错误）。“模糊”是指那些输入值千变万化而没有标准的高容量自动化测试的组合。这些测试一直运行直到程序崩溃或以其他不太可能的方式发生故障。

恶意数据流测试

Alan Jorgensen 曾经在标准格式（如.PDF）的文件中，用其他字符串来替换文件中的一个字符串来损坏文件，而测试出许多类型的安全故障。Jorgensen 在工作时，新的字符串一般来说长度远远大于文件中的字符串，而且在语法上也不一致。然后，Jorgensen 在被测应用程序中打开损坏文件，程序可能将其作为损坏文件拒绝打开，也可能正常打开。若程序正常打开该文件，Jorgensen 会要求程序以某种方式使用该文件从而发现异常的行为。在一些案例中，Jorgensen 把程序不会正确执行的可执行代码嵌入系统中，以发现故障，识别损坏文件。Jorgensen 寻找在“探测”眼皮底下悄悄溜走的这种损坏问题的确是一项自动化活动，但是他在探索过程中的分析并不是。

存在若干种已发布的文件模糊的变量（如输入文件的内容或格式的变量），适合不同类型的风险分析。

高容量测试，利用准则（Oracle） 的可用性

一个准则为高容量自动化测试奠定了强大的基础。一切要求是为被测程序产生输入数据，驱动程序处理这些输入数据。在得到结果的情况下，你可以运用这些准则来检查程序是否通过测试。没有任何准则是完美的：因为即使输出结果与准则相匹配，程序也有可能没有通过测试。虽然依靠一项准则，你不可能学会所有，有可能是有趣的，但是通过运行大量用来充分检查若干类型的错误而设计的测试，奠定了良好的基础，并且让你有机会找到其他类型的错误（比如由内存泄露引起的崩溃），你之前并没有特意设计测试用例针对这些错误。至于其他方法，你可以根据需要设计一些简单而合适的测试用例，使程序不能通过测试。

在一项准则的基础上进行高容量测试，你不能做到完全测试程序（所有可能的风险），但是这样可以达到手工不能完成的一定的覆盖率。

功能等价测试

功能等价测试开始于一项指导——一个与被测功能以同样方式运行的参考功能。在给定参考功能的情况下，你可以向被测功能提供输入值，获得结果，并检查参考功能是否输出同样的结果。无论你想测试多少个输入值，都可以如愿以偿。也许会生成大量随机输入值（如果你有充分的可用时间）或者测试了每一个可能的输入值。

在我的程序员-测试课程的期终考试中，我说明了函数等价测试。学生们必须回答的问题是 Open Office Calc 的计算方式是否与 Excel 相同。我们采纳了一条质量标准：如果 Calc 与 Excel 计算方式相同，即使 Excel 犯了一些计算错误，那么也足够好了。

回答这个问题，学生应该做以下几步：

1、单独测试若干个功能

2、通过把加减乘除中的几项法则结合起来同时测试几种功能

3、做好这些，学生们应在 Calc 中挑选出若干个单一的功能

4、然后测试每个功能，向 Calc 中输入随机值，并向相应的 Excel 功能中输入同样的值

5、然后生成包含几项功能的随机法则，输入随机数据，比较结果

如果你测试了足够多的输入值，那么 Calc 的结果和 Excel 总是一致的（允许少量的误差）。我们能合理的总结出 Calc 中的计算功能与 Excel 中的计算功能是等同的。

约束检查

我们利用约束准则来检查不合法的值或者不存在的关系。

例如，美国的邮政编码 必须是五位或者九位。如果你正在测试一个读取邮政编码的程序，你可以检查程序处理的每一个编码。如果程序接受任何非数字字符或者字符数量错误（把它看做邮政编码），那么一定是 bug。如果你能够找到一个方法驱动程序使其读取大量的邮政编码，那么就为大容量自动化测试奠定了良好的基础。

逆（倒置）操作

想象一下，将一个已经从高到低排好序的列表，重新从低到高排序，再还原（从高到低）。如果你可以为程序提供足够多的列表（大小，值的多样性丰富），你可以最终得出结论：排序正确或者排序错误。任何可以倒置的操作，都能设计高容量测试系列来应对。

基于状态模型的测试（SMBT）

如果你获得了一个状态模型（包括确定程序接受一个值后如何流转的方法和确定程序是否正确流转的方法），你可以向程序输入任意的一系列值并检查结果。

我认为 SMBT 的最普遍的方法是设计一组确定的测试用例，典型的做法是设计一组能够达到一个特定的水平的覆盖率的测试用例。这种做法的目的是完成从一种可能的状态向其它每一种能够到达的下一个状态的转换。你可以随机选取状态和输入值，在任意长的一段时间内执行这一连串操作，从而将这次测试数据转变为高容量序列。Ben Simo 提醒我们这种情况必须得到监控因为程序被锁定进入紧凑循环，永远不会达到其他的状态或者完成状态的转换。但是，如果你有编写测试执行系统来检测这种情况，你可以强制程序退出循环并进入尚未测试的状态。

基于诊断的测试

我曾经在一家设计电话系统的公司工作（Telenova）。我们的电话为客户提供了一个菜单驱动接口，包括 108 种声音特征和 110 种数据特征。想象一下，运行系统测试，利用 100 台电话机相互打电话，每台电话机都排队等候，电话从一台机子打向另一台机子，然后在外线召开会议。我们永远都不能为这种类型的系统测试设计一个完整的状态模型，因为太复杂了。

于是，Telenova 的员工（包括程序员，测试人员和硬件工程师）设计了一种模拟器，能够驱动电话机以特定的值或者随机值从一种状态转换为另一种状态。他们会把探针写入代码中来检测系统是否以意外的方式运作。探针像一则警告命令，如果程序触发了探针，系统会弹出一个错误而不是终止程序。探针可以检查出变量是否有意外值，一组变量是否有相互关联的意外值，程序是否以意外的顺

序经历一系列状态的转换。

因为这些探针检查的是系统的内部状态，可以检查系统的任何方面，所以我们称之为诊断。

执行这类测试需要大量合作。程序员把探针写入代码中，测试人员对测试日志进行第一轮评估，并深入分析问题，找到在日志中出现的事件的简单重现条件。程序员和测试人员齐心协力修复 bug，变更探针，指定下一个测试组。

虽然执行过程是富有挑战性的，这类测试也揭露了大量引人思考的有趣的问题，包括我们采用传统方法很难发现但是有可能在这个领域引起严重错误的问题。这正是使我坚信高容量自动化测试的价值所在。

高容量测试,利用已有的测试或工具的可用性

有时，采用高容量自动化技术的最佳理由是采用这种技术相对来说更简单。也就是说，大部分工作已完成或者可以用来工作的昂贵工具已经到位。

长序列回归测试

例如，我和 Pat Macgee 曾经写到一家知名公司，这家公司重新设置了用于办公自动化固件的大量回归测试集。遵从公司意愿，在此不透露名字。称之为 Mentsville。

当你做长序列回归测试时，是以对系统当前版本运行回归测试开始。接着，选择程序通过的测试。以随机的顺序运行这些测试直到程序出现问题（或者你已经运行这些测试有一段足够长的时间了）。最初的回归测试用例是被设计来揭示功能问题的，但是当你一次执行一个测试用例时，我们只利用我们知道程序会通过的测试，从而绕过了那些功能问题。我们发现的 BUG 来自于长序列测试。例如，程序可以运行 1000 次测试的序列，其中三十次测试和第一次是相同的（Test 1），但是直到执行第三十次测试时程序才会出现问题。为什么在第 30 次测试执行时出现问题而不是在之前呢？

有时，问题是由堆栈或者内存中的坏数据不断堆积产生的。

有时，问题与时序有关。例如，偶尔，处理器长时间处于忙碌的状态，被用来执行测试时还没准备好。有时，固件会等待存储于已更新的存储器中一个位置，但是在这种不寻常的序列下，处理器和进程都没有完成相关的计算操作。

有时，问题出自于内存空间不足（内存泄露），一些内存泄露是不易察觉的，需要一个特定序列的事件而不是对单一函数的简单调用。

这些问题和我们在 Telenova 发现的问题类型颇为相似（整晚运行一系列支持诊断的测试）。

分析并解决故障是一个挑战，因为我们很难明确潜在的故障真正什么时候发

生。事情可能发生在测试的初期，不会立即引起失败，但是会逐渐造成内存破坏，直到几小时后，系统崩溃。那么最初的结果需要用来重现故障。为了使排疑解难更容易，我们开始在测试之间运行诊断，检查内存的状态，一项任务执行了多长时间，或者处理器是否仍处于忙碌的状态，超过 1000 种其他的可进行的系统检查。我们只在测试之间运行一些诊断（每一次诊断都会改变系统的状态，我们认为，运行太多的诊断会导致状态改变的次数太多，而让我们不能对连续运行若干回归测试产生的效果有一个精确的描述）。但是那些少量的测试能够给我们大量的信息。并且如果我们需要更多信息，我们可以重复运行相同的第 1000 个测试序列，但是在测试之间运行不同的诊断。

高容量协议测试

一条协议说明了两个程序之间或者两个系统之间的对话规则。例如，一个电子商务网站，在顾客购买商品使用 Visa 卡（信用卡的一种）时，会和 Visa 进行交互，开购买账单。协议具体说明了站点可以向 Visa 发送哪些命令（消息），他们的结构应该是怎样的，发送的消息中应该出现什么类型的数据，可能的响应的地点和内容。一个协议测试包括向远程系统发送命令并评估结果（发送一系列相关命令并评估一系列收到的响应）。

当前受欢迎的系统，如 VISA，大量程序都希望测试其是否配合该系统正常运行（这些程序是否能正确实施协议，远程系统是如何真正响应的）。运行这些类型的测试的工具可能已经能够获得，或者部分可以获得。如果能够得到足够的工具，那么很容易对测试拓展，能生成生成随机的长序列命令发送到远程系统并处理系统针对每一个发送命令的程序的响应。

增加负载的功能测试

20 世纪 70 年代人们普遍认为（common lore）当系统在负载下运行时，系统行为异常，功能上异常。当系统进入繁忙的状态时，系统在“正常”负载下可以正确运行的任务都会执行错误。

Alberto Savoia 在 2000 年 STAR 会议上的演说中描述了关于负载测试的自己的经历。Segue（一家自动化工具开发公司）对此建立了一项服务。根据我对结果的理解，系统可能在 50%的时间内保持忙碌的状态（忙碌，但没有达到饱和的状态）但是仍然不能正确运行一些回归测试。这类测试的价值体现在它可以揭示系统的功能缺点。例如，假定一个运行在低内存上的系统更大程度上依赖于虚拟内存那么它的行动也会放慢脚步。假定一个运行在系统上的程序以一种使它在竞争条件下很脆弱的方式把任务分配给处理器。当系统在正常负载下运行时，处理器 1 上任务 1 先于处理器 2 任务 2 完成，但是在一些高负载情况下，处理器 1

会受到限制，并且处理工作没有处理器 2 那么快。如果程序假定任务 1 总是在任务 2 之前完成，那么这个假设在高负载条件下是不能通过的。

在 IT 领域，像这样的 bug 是难以重现的。除非你找到了系统的时间（timing）的规律，否则你永远不能在实验室重现这样的问题。它们就成为了谜一样的故障，而一些客户会打电话投诉的故障。

如果你正打算给程序做负载测试，并且已经准备了回归测试套件，那么通过并行运行合理负荷的负载测试，我们很容易把它转化为长序列回归测试。准则包括运行每一次回归测试产生的结果（让你知道程序是否通过测试）加上崩溃，明显的长时间延时，或者由诊断产生的警告（如果你添加了那些诊断）。

参考文献：An Overview of High Volume Automated Testing

测试 Python 和 C#代码

作者：于 芳 （译）

Monkey 补丁修补和映像仅仅是一些测试复杂系统的方法。

在这个测试复杂系统三部曲系列的第一部分，我覆盖了深入测试的实际角度（方面）并且展示了怎样测试代码的困难块，例如用户界面，网络连接和异步代码。第二部分，讨论一些我曾经成功测试过的 C++复杂系统。

在本文的最后安装部分，讨论类似的复杂系统测试方法像 Python（和 Swig C++/Python 绑定）和 C++，.NET 高容量，高可得性的 Web 服务。

Python

Python 是强输入和动态输入的语言。它也是一门演绎的语言（即时编译）。这意味着打字错误会在运行时显现，而不是编译时。举个例子，如果你传递一个 INT（整型）值给期望得到字符型值的函数，你的程序将仍然可以运行，也许还会成功运行到最后如果你的代码从未遇到不好的调用:

```
defsayHello(name):
print'Hello, '+name
defsayHelloToMyLittleFriend():
sayHello(5)
sayHello('Jebediah')
```

输出：

Hello，Jebediah

这个 sayHelloToMyLittleFriend()函数包含的不好的调用 sayHello(5)从未被召唤，所以这个程序运行正常。因此，你是不是应当在用静态输入的语言写程序时做点不同的事情？不！如果你没有发现那个不好的调用，就代表你没有测试代码路径。

如果没有测试代码路径，你就不能说出来这个程序是不是坏了。而且这根本没有关系它是否是因为打字错误或者一些其他的错误坏了。（顺便说下，在静态输入语言中，有一个很类似的问题突然出现很多。令人畏惧的 NULL/null/nill/None （空值/或无值）或者其他任何你想调用的。如果你传递一个 NULL 空指针给试图解引用它的 C++函数，你会引起一个崩溃：这实际上比向一个期待字符值的函数传递 Int 值更常见。）

用 Python，如果你真的是妄想狂者，你可以测试自变量的类型。这对经常使

用的库也许有用，还有对那些你想提供给用户一个好的错误信息而不是只是因为一个打字错误的崩溃。你可以用一个 try-except 块把有问题的代码包围起来，捕捉 TypeError，或者你可以用 instance of() 函数来动态决定自变量的类型（但是这第二个选项十分脆弱）。下面是程序用了 try-except 的样子：

```
defsayHello(name):
try:
print'Hello, '+name
exceptTypeError:
print'Oh, no!', name,'is not a string'

defsayHelloToMyLittleFriend():
sayHello(5)

sayHelloToMyLittleFriend()
```

输出：

Oh, no! 5 is not a string

Monkey 补丁

测试 Python 代码（和其他动态语言）是件愉快的事情。没有必要去为你的代码设计可测试性，或者去创建特殊的界面，或者跟旧的代码打架。用 Python，你可以到达任何对象的内部，替换它的成员以你喜欢的任何东西。你甚至可以替换函数。再没有比这个更好的了。这个历史悠久的惯例被称作 monkey patching。即使在你不用依赖性注入，monkey patching 也可以工作，因为你可以在任何时候替换掉那些依赖。在以下的代码例子中，类 A 实例化了它自己的 Dependency 依赖：

```
classDependency(object):
def__init__(self):
self._state='complex state'

defbar(self):
print'Doing something expansive here...'
print'state: ',self._state
classA(object):
def__init__(self):
self._b=Dependency()
deffoo(self):
self._b.bar()
a=A()
a.foo()
```

输出：

Doing something expansive here... state: complex state

如果你想测试 A，但是不想让它唤醒真的扩展的 Dependency.bar()方法，你有两个选择：

1、你可以替换掉 A 的_b 成员，替换为一个有 bar()方法的模拟对象

2、你可以替换掉那个 Dependency 的 bar()方法

如果你想模拟整个 dependency 依赖，方法 1 有用，经常和多个方法和内部状态一起使用。如果你想使用实际的 dependency 依赖对象用其逻辑和初始化的状态，方法 2 有用，但是只是替换一些方法。

Method 1:

```
classMockDependency(object):
def__init__(self):
"""No state"""
defbar(self):
print'Doing something cheap here...'
a=A()
a._b=MockDependency()
a.foo()
```

输出：

```
Doing something cheap here...
```

Method 2:

```
defmock_bar(self):
print'Doing something cheap here...'
print'state: ',self._state
a=A()
Dependency.bar=mock_bar
a.foo()
```

输出：

```
Doing something cheap here... state: complex state
```

你最好聪明点，限制使用你的 monkey patching 来测试和应付其他的特殊情况。很多人对 monkey patching 的强大很兴奋，把它用在不管愿不愿意的地方，结果导致了缺陷多多和不能维护的代码。

Python 测试框架

Python 有多种测试框架。xUnit-类似的 unittest 补丁自从 Python 2.1 以后成为 Python 标准库的一部分，但是从来都没有像其他测试框架那样强大，和用户友好直到 Python 2.7 的出现。我很喜欢 nose。用 nose，你不必去写那些从一个基础类衍生的测试类，但是相反可以写一些简单的测试函数，nose 会发现并且运行它们。在 Python 2.7, 标准的单元测试补丁包从第三方测试框架中获取很多可利用的特性，从而变成一个更强大的容器（尽管 nose 也已经演化，有 nose2）。

There is a lot of material available around the Web on unittest, so I'll just quickly list the main features of the Python 2.7 (and Python 3.2) unittest package:

在 unittest 网页上有很多资料，所以我只是快速地列下 Python2.7 （和 Python3.2）unittest 补丁包的主要特性：

- assert 函数的大量丰富的使用
- 测试发现物
- 测试组织
- 汇报测试结果
- 命令行界面
- 在运行中测试的良好的粒性控制

Python 还有一个有趣的模块叫 doctest，它允许写入测试桩嵌入到你的

docstrings 中。我不用 doctest 因为我喜欢在测试程序的测试代码下分离代码。我的目标也在更复杂的测试上，这个在 doc string 里比较难（更不要去提它会将代码弄乱）。

Python 扩展

像任何一个自重的动态语言，Python （CPython）有一个扩展和嵌入机制。它意味着你可以用本地的 C 库扩展 Python，甚至将 Python 嵌入到一个本地程序中。其他的以特定的运行时环境为目标的 Python 执行，如 JVM 上的 Jython 和.NET 上的 IronPython，提供了类似的便利。这里，我将只讨论 CPython。

首先--Python 很慢。它当然是，与本地的语言向比较来说，但是它与其他动态语言向比较也慢。我不会深挖个案（超灵活，超动态，GIL，多指针 问题）。数年来做过各种努力来改善 Python 的性能。最著名的是 PyPy 项目。在 CPython 生态系统里，推荐的扩展 Python 与本地代码的方法是用 Cython，一个 Python 类似的语言，它让你大部分用 Python 写扩展程序到 Python。如果你需要与现有的 C/C++代码交互，还有其他几种工具可用，例如 Boost.Python 和 Py++。

处理语言绑定

我的语言绑定经验大部分都与使用 Swig 有关。Swig 是一个老项目，它可以为几乎任何编程语言创建 C/C++绑定。我用它给 Python，Java 和 C#的 NuPIC 2.0 API 创建过语言绑定。我也必须通过它的界面代码为 NuPIC 1.0 Python 绑定来做 Debug。测试这些代码没有乐趣。都是配齐数据类型,匹配调用惯例,和调整错误处理机制的工作。这里我可以给出的最好的建议是，保持绑定的代码尽可能的瘦。不要屈服于加一块句法的糖的诱惑者或试图遵循目标语言的成语习惯来做绑定代码。用一个有层的方法--绑定本身，不管是手写的代码还是自动产生的代码，应当紧紧匹配本地代码。后来，一旦你回到了你的目标语言世界里在它上面写一个包或者写一个句法糖纸。你可以使用包层来写测试用例。

自从起始版本 Beta2，我已经遵循.NET 框架和 C#语言的演化规则。我在家写了很多程序，并且发表了几篇关于框架和语言的文章，但是直到最近，我从来没有这个机会去一个职业的能力来使用它门。这些天，我为 Roblox 工作，在这里我用 C#语言和.NET 框架写了可高度扩展和可高度分布的服务，我享受在那里工作的每一分钟。C#语言是一个很棒的编程语言。这个服务是以一个更好的 JavA 开始的，在每次修订中，添加了更多更好的东西（基类，匿名函数，属性，LINQ，异步支持 ），有很棒的平衡。由于一些原因（伟大的语言设计者？），C#的力量和富余从未给我那种其他语言给我的吹嘘的感觉。当你把自己置身于.NET 框架和 Visual Studio 时，你获得了一个超级棒的开发和运行环境。

Visual Studio 提供了一个很好的单元测试框架，用它你能配置属性。它允许你在自己的解决方案中运行任何测试子集。我的走向技术是将鼠标的游标放在一个我想运行的测试上，然后点击 CTRL+R+T 以调动那个测试。在过去，我常常写些小的平台的程序来练习代码的不同部分。现在再也不了。

与此同时也有很多别的很棒的测试框架（有商业性的也有免费的），我会在接下来的例子中使用内建的 Visual Studio 支持，但是这些概念不关联与任何特定的测试框架。

映像

.NET 框架为每个对象保存了出色的大数据，并且允许映像。你可以取任何对象，即使不咬知道它的类型（所有的对象都是从 object 类衍生而来），在运行时，发现她的字段，属性，方法等等。你也可以访问这些字段和属性，然后激活方法。映像最重要的一件事是他允许你忽略访问修改器。你可以访问私有状态和激活私有方法，访问内部类型，还有初始化私有类和内嵌类。下面的例子展示了类 A 和私有数据成员_x 和私有方法 DoubleIt():

```
    classA
{
privateint_x;
    publicA(intx = 5)
{
_x = x;
}
    privatevoidDoubleIt()
{
_x *= 2;
}
}
```

类 A 碰巧是一个内嵌类。这个很重要，因为当访问内嵌的类时，你要加个+让他们的名字合格。如果你不确定你要测试的类的确切类型，你可以使用集合的 GetTypes()方法来列举它包含的所有类型。这里是一个使用原始映像的类 A 的单元测试：

```
    [TestMethod]
publicvoidRawReflectionTest()
{
var assembly = GetType().Assembly;
    // Get all the types in the assembly
var types = assembly.GetTypes();
    var typeName ="ReflectionDemo.ReflectionTest+A";
var t = assembly.GetType(typeName);
var instance = Activator.CreateInstance(t, 4);
    // Get the value of X
var filed = t.GetField("_x", BindingFlags.NonPublic | BindingFlags.Instance);
var value = filed.GetValue(instance);
Assert.AreEqual(4, value);
    // Invoke DoubleIt
var method = t.GetMethod("DoubleIt", BindingFlags.NonPublic |
BindingFlags.Instance);
method.Invoke(instance,newobject[] {});
    // Verify result
value = filed.GetValue(instance);
Assert.AreEqual(8, value);
}
```

很显然，这个程序十分冗繁，所以我创建了一个小的用途类叫做 ReflectedObject ，它可以更直接了当点达到效果。下面是用 ReflectedObject 类的同样的测试方法：

```
    [TestMethod]
publicvoidReflectedObjectTest()
{
var assembly = GetType().Assembly;
var typeName ="ReflectionDemo.ReflectionTest+A";
var instance =newReflectedObject(assembly, typeName, 4);
    var value = instance.GetField("_x");
Assert.AreEqual(4, value);
    instance.InvokeMethod("DoubleIt");
    value = instance.GetField("_x");
Assert.AreEqual(8, value);
}
```

如果你在使用.NET 4.x 系列框架，使用 DynamicObject 类是可能测试的更深入的，然后写一个子类，允许你直接访问私有字段，属性和方法。

我最近在花精力在将 RabbitMQ 整合进 Roblox 的技术堆栈。RabbitMQ 是一个很棒的消息队列，有着内建的聚类支持。这个 RabbitMQ C#客户提供完整特性的 API 与 RabbitMQ 服务器对话。不幸的是，它并不具有群集意识。所以如果你想跟群集对话，处理像节点走上走下，临时性的断网以及恢复的情形，你需要些很多额外的代码。我就是做了这个，我想测试它。

因为 RabbitMQ 过去曾是被用来做我们系统关键部件的新的第三方软件，我想彻底地测试它的集成功能。那个涉及到一个对有着三个节点的本地群集的多个测试（都在我的本地机器上运行），以及在远程 RabbitMQ 群集上运行的同样的测试。这些测试涉及到用不同方法搞垮群集，重建群集，和配置群集，然后再做其上做压力测试。启动和配置一台远程 RabbitMQ 群集需要很多步骤，每个步骤正常的话不到 1 秒钟。但是，偶尔，一次可能要用 30 秒。下面是一个典型的配置一个远程 RabbitMQ 群集要的必不可少的步骤：

- 关掉群集中的每一个节点
- 重置每一个节点的连续的元数据
- 以彼此独立的模式启动每个节点
- 将这些节点聚集在一起
- 在每个节点上启动该应用程序
- 配置虚拟主机，交换机，队列和绑定

我创建了一个 Python 程序叫做 Elmer ，它是使用 Fabric 跟集群远程焦糊。

由于 RabbitMQ 管理各集群的元数据的方式有些不同，在集群的每个节点执行的时候，你必须等每个步骤完成才能进行下一个步骤的执行；而检查每个步骤执行的结果需要从语法上分析 Shell 命令的控制台输出（呸！真令人讨厌！）。将它与特定节点问题和网络上的一些小问题放到一起，你得到一个时间变化很大的流程。在我的测试中，除了优雅的关闭和重启整个集群，我常常强烈地想杀掉或者重启一个节点。

从操作的角度来看，这不是个问题。启动一个集群，或者替换一个节点，这类事件不常发生而且也说得过去如果这样只花费几分钟。但是对于开发者来说，这就是另一回事了，他们要在每个改动过后运行十几二十个集群测试。另外个复杂的事情是有些用例需要测试不需要回应的节点，这些节点可以导致中断问题（所以这是真的没有回应还是只是回应慢？）。在遭受跑每个测试时都因为要等远程的集群响应耽搁很长时间，跑了很多测试之后，我用下面的方法结束：

1 Elmer（那个 Python/Fabric 集群远程控制程序）暴露了流程的每个步骤

2 一个叫做 Runner 的类可以启动 Python 脚本和 Fabric 命令并且捕捉输出结果

3 一个叫做 RabbitMQ 的类使用那个 Runner 类来控制集群

4 一个叫做 Wait 的类可以动态地等待一个任意的操作到完成

关键在 Wait 类。这个 Wait 类有一个静态方法叫做 Wait.For()，它允许你在一个特定时间段内等待任意一个操作到完成，否则就暂停。如果这个操作完成很快，你将不用等待这个时间到过期，而 Wait 会很快解放出来。如果该操作没有及时完成，Wait.For()会在时间过期后返回。Wait.For()接受一个时间段（或者是一个时间段，Wait.For()里的 TimeSpan 或者是数毫秒），有个函数会返回 BOOL。它还有一个变量成员 Nap，默认值是 50 毫秒。当调用 Wait.For()方法时，它循环地调用你的函数直到它返回 True 真或者直到时间段过期（在调用之间不警戒）。如果该函数返回 True 及真时，那么 Wait.For()会返回 True 真；但是如果是时间段过期，就会返回 False 假。

```
publicclassWait
{
publicstaticTimeSpan Nap = TimeSpan.FromMilliseconds(50);
publicstaticboolFor(TimeSpan duration, Func<bool> func)
{
var end = DateTime.Now + duration;
if(end <= DateTime.Now)
{
returnfalse;
}
while(DateTime.Now < end)
{
if(func.Invoke())
{
returntrue;
}
Thread.Sleep(Nap);
}
returnfalse;
}
publicstaticboolFor(intduration, Func<bool> func)
{
returnFor(TimeSpan.FromMilliseconds(duration), func);
}
}
```

现在你可以有效率地等那些运行时间变化很大的过程到完成了。这里是我如何用 Wait.For()方法来检测一个 RabbitMQ 节点是否停止的：

```
privateboolIsRabbitStopped()
{
var ok = Wait.For(TimeSpan.FromSeconds(10), () =>
{
var s = rmq("status", displayOutput:false);
return!s.Contains("{mnesia,") && !s.Contains("{rabbit,");
});returnok;
}
```

我调用 Wait.For()方法给了它一个10秒钟的时间段，这样做是因为我不想每次检查一个节点是否停掉时都被阻塞掉（因为这种事情一直都在发生）。我传过去的那个匿名函数用 Status 命令调用了 the rmq()方法。the rmq()方法在远程集群上运行了 Status 命令，然后返回命令行输出作为文本。下面是当 Rabbit 在运行时的输出：

```
Status of node rabbit@GIGI ... 节点 rabbit@GIGI 的状态。。
[{pid,8420},
{running_applications,
[{rabbitmq_management,"RabbitMQ Management Console","2.8.2"},
{xmerl,"XML parser","1.3"},
{rabbitmq_management_agent,"RabbitMQ Management Agent","2.8.2"},
{amqp_client,"RabbitMQ AMQP Client","2.8.2"},
{rabbit,"RabbitMQ","2.8.2"},
{os_mon,"CPO  CXC 138 46","2.2.8"},
{sasl,"SASL  CXC 138 11","2.2"},
{rabbitmq_mochiweb,"RabbitMQ Mochiweb Embedding","2.8.2"},
{webmachine,"webmachine","1.7.0-rmq2.8.2-hg"},
{mochiweb,"MochiMedia Web Server","1.3-rmq2.8.2-git"},
{inets,"INETS  CXC 138 49","5.8"},
{mnesia,"MNESIA  CXC 138 12","4.6"},
{stdlib,"ERTS  CXC 138 10","1.18"},
{kernel,"ERTS  CXC 138 10","2.15"}]},
{os,{win32,nt}},
{erlang_version,"Erlang R15B (erts-5.9) [smp:8:8] [async-threads:30]\n"},
{memory,
```

```
[{total,19703792},
{processes,6181847},
{processes_used,6181832},
{system,13521945},
{atom,495069},
{atom_used,485064},
{binary,81216},
{code,9611946},
{ets,628852}]},
{vm_memory_high_watermark,0.10147532588839969},
{vm_memory_limit,858993459},
{disk_free_limit,8465047552},
{disk_free,15061905408},
{file_descriptors,
[{total_limit,924},{total_used,4},{sockets_limit,829},{sockets_used,2}]},
{processes,[{limit,1048576},{used,181}]},
{run_queue,0},
{uptime,62072}]
...done.
```

这个函数是在确认 mnesia 组件和 Rabbit 组件不在输出结果中出现。注意如果该节点仍然出现，那么这个函数将返回一个 False 即假值，Wait.For()会继续执行很多次。Wait.For()减少了我的测试对偶尔在响应时间上的峰值敏感度（在平常时候，我可以等待更久如果没有减慢测试时间），而且降低了整个测试集的运行时间，实现了运行时间从数分钟降低到数秒钟。

结论

总结这一系列的文章展示了丰富的设计原则和对付难测系统的测试技巧。重要的代码总是会有缺陷，但是深入的测试保证了可以降低未被发现的问题。

【测试管理】自动化测试之脚本维护

作者：王海兰

摘要

本篇文章主要面向采用 QTP 进行自动化测试的读者朋友们，其中主要阐述了对自动化测试脚本的维护与管理的一些经验，同时本篇文章也作为 QTP 初学者宝典的提高篇，希望能给自动化测试的各位同行朋友们带来帮助，在您的测试工作中助您一臂之力!

关键词

软件测试；自动化测试；QTP 脚本维护；

引文

自动化测试的必要性就不再多做阐述,使用 QTP 毋庸置疑是为，降低测试成本,提高测试效率。或许刚刚入门的朋友都会觉得 QTP 使用很简单，基本上不用培训就能直接上手操作。但是其实不然，长期做自动化的朋友们或许就能明白 QTP 有一难点，那就是脚本维护。

一个庞大的测试项目必然包含了大量复杂的测试脚本，如果说对于脚本的管理没有一个好的制度，那么随着程序的频繁更新，脚本的反复执行，这些脚本必将步履维艰！

比如说兼容问题，如果项目组提出新需求,要求我们的产品支持最新版本的 IE。而我们当初录制的 IE 是旧版的，那么在新版的 IE 上执行这些脚本时可能就困难重重，甚至是寸步难行。

再比如，按照业务的操作流程，我们在测试时都会遵循添加再删除的方式去执行系统交易，但是由于更新上来的程序在处理删除时出错，于是，出现了添加成功，删除失败的尴尬局面。我们都知道,在每次回归测试时,都要执行自动化脚本，所以在这种删除失败的情况下，再次运行脚本就出现了同样的数据添加两次的情况。因此，毫无疑问，这种操作必将以失败而告终，此时，只有一种方法去解决这个问题了，那就是我们手动去删除已经添加成功的数据。大家可以想象，在成百上千的测试案例中，如果出项大量这样的脚本我们要怎么去维护呢。不要说去发现问题，我们的主要任务估计就会被迫转向调试脚本了。

第三种情况，前两个案例讲的是程序更新引起的，是不可避免的。现在我们来探讨下，有时无心之举也可能会带来不小的麻烦。

一般一个系统，都有其自身的特点，比如说要在这个系统上进行业务操作可能会有一些前提条件，例如，我们的系统在做交易之前有个系统开关，必须保证

其是 ON，只有这个开关是 ON 的情况，才允许用户在该系统下进行业务操作。如果说在某次跑脚本时，这个开关并不是 ON 状态，等到第二天来查看执行结果的时候，岂不是傻眼了?

以上的三个案例充分说明了测试脚本的不稳定性和难以维护的特性。

场景设想:

系统程序更新后，领导要求利用夜间进行所有脚本的回归测试，并且第二天统计出测试结果。但是基于上述三个案例，我们知道要想在一轮循环后得出合理的结果并非易事，可能各种各样或是主观或者客观原因影响了脚本的正常执行。那么要怎样才能在最短的时间统计出脚本执行的合理结果呢?这就是一个关于如何高效管理脚本的问题。

因此，根据我的测试脚本管理经验，方案如下：

我们称之为：脚本维护四大法则！

法则 1—面面俱到

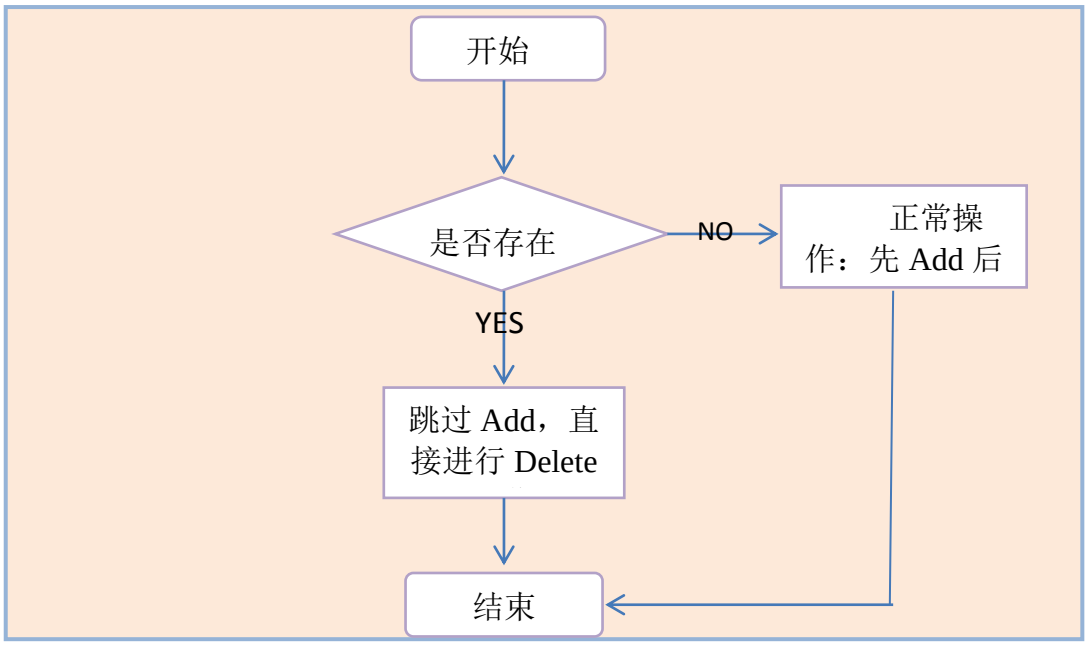
面面俱到说的是脚本要做到全面无漏洞，即脚本的严谨性。

在录制测试案例时，必须要考虑到脚本的周密性。比如，对于添加删除(Add-Delete，我们简称为 AD 类型)类脚本，针对 AD 类脚本，个人建议如下录制更有利于脚本维护，达到事半功倍的效果。

首先，做 Add 操作时，要先判断，当前系统中是否存在该数据，如果存在跳过 Add 操作，直接进行 Delete。 反之 ，如果系统中不存在该笔数据，则进行正常逻辑的操作。

再比如日期类操作，如果系统要求我们选择的日期不得早于当前日期，那么针对这类问题，我们一律选择比较长远的日期，这样便可避免在后期回归测试时由于日期而引起的问题。

面面俱到的要旨是考虑未来可能发生的问题，将麻烦扼杀在摇篮中，使脚本坚如磐石，无懈可击！



法则 2 —未雨绸缪

何为未雨绸缪，即在下雨前就做好防雨准备，那么这跟维护脚本有何关系呢？

很显然，此处意指在执行脚本前要做的准备工作，必要的准备工作会大大提高我们的测试效率，所谓磨刀不误砍柴工。我们可以从系统需求和工具需求这两方面去着手准备。

系统需求：我们在上面提到，一个系统正常运行通常会有些前置条件，比如，系统开关为 ON，服务器状态为 Stat，服务器日期不能早于当天等等。这些都是在执行脚本之前需要确定的因素。根据个人经验，这些操作一般都有对应的 SQL 语句，那么把这类行为的 SQL 放到一个 AutoTest．sql 文件中，在执行脚本前拿过来跑一遍即可。

工具需求：在本文中，我们默认的工具就是指 QTP。对于工具，基本上我们不会去更改其设置，一般要做的就是确认工作，一是 run mode 下的 delay time，该时间不能太短，以免脚本执行太快导致由于不能及时找到对象而引起脚本报错。二是重新启动一下测试工具，该步骤不是必须的，但是以防因为工具长期执行产生的一些不必要的麻烦，该类问题在测试过程中也偶有发生，尤其是盗版的工具。

法则 3 ―分门别类

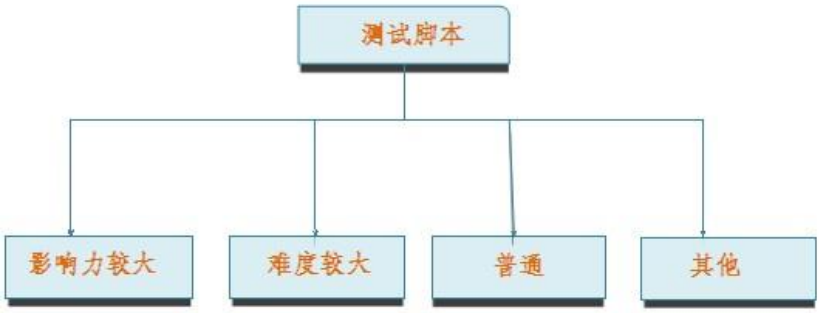
有这么一类测试案例：只要它执行失败了，那么它下面的所有脚本都执行不了。对于这种情况，做自动化测试的朋友应该不会陌生吧。

我们还以系统前置条件来举例，系统要求开关状态是 ON，有这样一个测试案例，在执行的时候会去更改这个开关，但是在更改的过程中操作失败了，导致开关的后期状态一直是 OFF，So，问题产生了，接下来的所有脚本一旦检测到当前系统是 OFF，就会报错，于是乎，接踵而来的是华丽丽的 Failed 这个敏感的不受人待见的执行结果。

另外有些覆盖面比较广且测试点比较复杂的测试脚本，也可以单独放在一个 folder 下，等确定测试环境基本稳定的情况下，再去执行这些难度较高的脚本，这样做同样也可以减少测试难度。

总而言之，分门别类的宗旨就是说把特殊的脚本归为一类，放在一个固定的 folder 下，然后单独去执行这些测试脚本，从而来降低风险提高测试效率。比如我们上述提到的影响面大的，难度高的这类测试案例。

法则 4 ―留有余地



一起来思考这样的一个问题：自动化测试的目标是什么？是为了发现更多的 bug，还是为了确保测试点没有受到新程序的影响？

如果从一个手动测试者的角度考虑，答案无疑是为了发现更多的 BUG，然

后上报 BUG，解决 BUG 这一流程，当然这也是我们测试人员必备的测试素质。但是作为一个自动化测试脚本维护与增强者，我们的目标不是为了发现意外的更多的 bug，我们的目标很明确，就是保证已经存在的测试案例是正确的，确保没有受到新环境的影响。

我们举例来说明，假设现在有 100 个测试案例，这些案例很特殊，他们的测试点包罗万象，不仅包含了原本要求的点，还增加了大量的扩充。在一次环境更新后，我们进行了回归测试，结果由于每个测试案例里包含的测试点过多，导致 100 个案例全部失败！(一种不太可能发生的情况)

问题出现了：

失败率如此高的脚本要怎么继续维护下去？对于每个失败的脚本要花费多大的精力才能让其 status 从 Failed 变为 Passed 呢？

如果把这些脚本按照一对一原则，即一个测试点对应一个测试案例，那么按照 100 个测试案例中，假设原来每个案例包含两个测试点，那么一对一之后，就是 200 个案例，之前的每个案例中都出现了一个问题，那么在 200 个案例中，失败率为 50%。如此，失败率就由原来的 100%降低到 50%了。这样有什么好处：一来提高测试人员的信心；二来，脚本的这种一对一方式也更能高效的管理脚本。

究竟什么是一对一原则，要怎么去把握这个原则呢，简而言之，就是在完成测试点的前提下，不去多做任何操作，让可能产生的并且与本测试案例无关的问题留给手动测试吧！

案例：我们现在有个交易是检查用户的年龄是否只能显示为数字。如果是一套完善的脚本会这样操作，添加一个用户，在添加用户的页面上完成测试点的操作，然后再去删掉该用户。但是，此处，我们不提倡这样的流程，我们采用最简单的测试方式，即，选中一笔系统中永远存在的用户，进入到添加页面后进行测试点的检测，得到测试的预期结果后，退出该页面。也就是说在整个操作中，我们不会涉及到提交操作，这样就减少了大量的额外检查点，这样的脚本既简单也容易维护。

采用最单纯的操作来完成测试点就是我们留有余地的宗旨。

参考文献

[1] 余杰、赵旭斌. 精通 QTP：自动化测试技术领航[M]，第 1 版. 人民邮电出版社，51 软件测试网组编，2012 年 1 月 1 日

[2] 蔡为东. 赢在测试:中国软件测试先行者之道[M]. 电子工业出版社, 2010 年 1 月

报表类模块测试一些注意点

作者：吴治宾

进行报表类测试也有一段时间了，结合自己的经验以及查阅过的一些资料，简单总结下报表类模块的测试点以及注意点，这里从测试数据构造和测试执行两点来展开。

报表类模块测试数据的构造，也就是源数据，主要有两个来源，一个是本测试平台业务产生的数据，另一个是第三方数据库过来的数据。针对不同的源数据来源，测试重点也应有所区分。

1> 本测试平台业务产生的数据，这种情况的报表测试相对来说较容易（注意是相对，主要还是看具体项目），测试重点在报表的准确性上，即真实产生的业务数据与报表统计的数据是否一致。需要注意的是真实的业务操作是否真的产生了数据，这个点是我们很容易忽略的测试点，比如日志系统，进行某一个操作平台是否记录了这个操作进而产生相关的日志，实际测试中我们更多的是进行大量数据产生的测试准备，忽略了数据产生的正确性（比如数据准备时是否有应该记录的数据没有记录，也就是写数据是否正确），这个需要测试负责人在测试前期测试任务划分时需要明确的，是专门人员来测试还是每个模块负责人负责。

2> 第三方数据库过来的数据，该类型报表测试，应该把更多的注意力放在源数据与被测平台数据对比上，如报表展示出来的数据是直接对第三方数据库通过算法计算的还是测试平台先把第三方数据库中的源数据转换写入被测平台的数据库，然后再对本平台数据库数据通过规则进行统计展示。测试过的多个项目发现，开发人员自测时常忽略源数据的重要性，他们自测时习惯性在测试平台上数据库直接插入一些数据进而查看报表展示功能，而忽略了源数据到被测平台数据转换是否正确这个过程，作为系统测试人员我们要保证源数据到被测平台再到页面报表展示这整个过程的正确性。

源数据的准备是测试执行的基础，如果源数据就不正确且不可控，那么报表功能就失去了意义，如何能有效的进行完备的各种数据类型的准备，首先需要我们了解业务流程，从数据产生到报表展示整个流程，了解报表的统计规则；其次就是平时的积累，做好测试数据的备份和维护。源数据的产生，我建议最好采用业务产生的数据，不管是第三方数据库还是被测平台，最好不要使用数据库直接插入方式，具体缘由读者自己思考。

测试执行，当然首先要按照测试用例来走，但是我们知道测试所发现的缺陷如果有 40%是由执行测试用例发现，那么这份测试用例文档就相当优秀了。这里简单说下实际测试时一些注意点，这些注意点可能并没有在你的测试用例文档中。

1> 数据可控，接触过测试的人都知道，我们测试时多用自己的测试资源，很少共用，主要是怕测试时相互影响，导致测试数据不稳定性，测试的前提是你有能力去控制你的数据。

2> 数据的实时性，这个主要是针对从第三方数据库过来的数据，一般的项目都会有个采集器，定期去采集数据，这时就应该注意这个“定期“的设置测试，从第三方数据库产生数据到被测平台统计展示的时间，比如报警类和资源状态类这两种数据的采集时间就应该有区分，报警类对实时性要求可能要高点，“定期”的设置应该短些（由于定期采集时间可设置，要保证我们测试时的设置与实际用户场景保持一致）。

3> 查询条件，一般的报表展示都会有设置查询条件框，按某种统计方式展示的选择设置：

a> 可输入的查询框，可以当做一般控件测试，测试查询结果与查询条件是否一致，特别注意的是通配符的测试，如% _等。

b> 可手动增加的查询选择项，如资源类型，可以在其它模块更改增加类型，测试报表时就应该注意展示轴，如果展示轴是根据类型全部展示，当类型较多类型名称较长时，展示轴的显示就会出现问题。当然类型较多时，查询选项框展示的名称、速度以及 GUI 也是一个检测点。

c> 默认查询，一般的报表查询展示都会有默认查询项，检测默认查询项设置是否合理，是否有必选项，不选择时是否有提示信息；是否有前置条件，如报表查询展示时一般先选择某个区域或者某个部门，如果不选直接查询，是否会出现 js 错误等等。

d> 关联查询设置项，设置一个查询项后，另一个查询项是否有相应变化，如开始时间是否可以设置大于当前时间，开始时间与结束时间的控件设置测试等

e> 查询条件测试用例设计应覆盖所有统计展示方式，保证测试的充分性。

4> 统计报表展示的准确性，这个是报表测试的重中之重，如何对比？除了要了解上面提到的业务流程外，还需要开发人员的协助，最好让开发人

员提供查询 SQL 语句以及统计算法、各个报表字段之间的关联等等，通过 SQL 语句以及手工的方式对比测试。

5> 测试数据的有效期，任何数据都不太可能无限制保存，如操作日志最长保留时间一年，这个数据的有效期也应该注意。

6> 测试数据变动：

a> 增加，这个没什么好说的。

b> 修改，修改已产生数据的某个字段，如人员名称，之前产生的数据记录，在展示页面是否有相应变化。

c> 删除，如果报表以各个人员来统计，删除某个人员，跟这个人员相关的数据是否全部清除（这个看具体业务，有些是做逻辑删除，页面不统计展示，但是数据库会保留，以便以后查看对照）

d> 权限（权限放这可能不太合适），设置某个用户对某部分资源没有查看权限，用该用户登陆查看相关资源，报表展示页面是否加以控制，是否对没有权限的资源不做报表统计处理，再次配置查看权限后是否能够正常统计。

7> 导出功能，一般的报表模块都有导出功能，查看导出文档名称，sheet 页名称以及内容；这里要注意包含小数的字段显示，excel 的四舍五入和被测平台的四舍五入规则可能不一致。

8> 注意报表展示各字段单位及小数位数控制。

9> 注意大数据量和空数据时报表展示。

10> 注意不同报表间的关系，对照测试。

11> GUI 展示是否合理易理解，报表格式和颜色使用是否统一。

以上是根据近期进行报表模块测试时总结的一些注意点，希望对大家有所帮助。

软件质量评估模型与应用系列（1）

------Gompertz 模型评估与应用

作者：Ghump

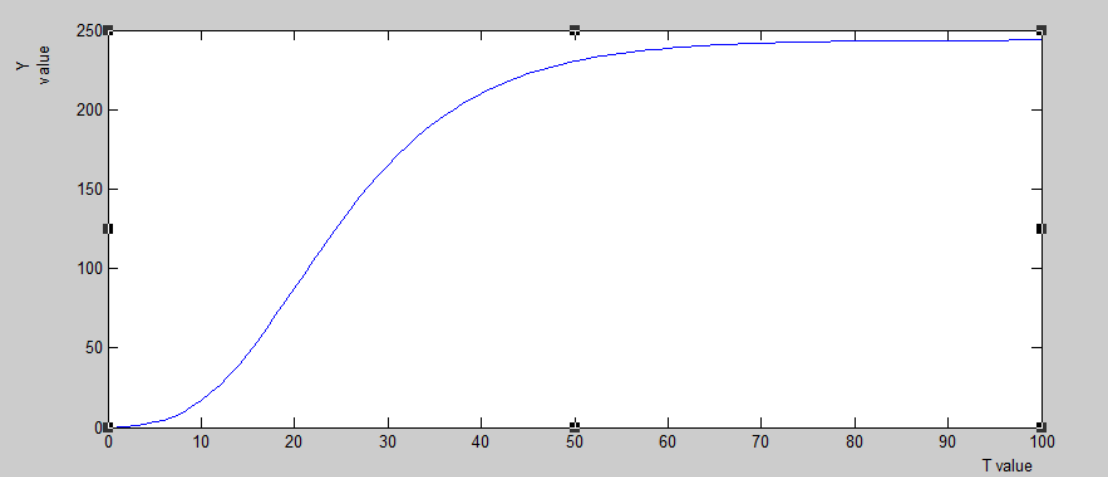
在软件开发过程中,质量、进度和成本是需要重点关注的三大要素。在当下竞争激烈的 IT 环境下,如何做到质量、进度和成本的平衡,是领导者们面临的一大挑战。软件和其他商品一样,如果没有达到一定的质量标准就交付给客户，遗留的问题过多且严重的话,会直接影响用户的满意度，同时也会影响后期为修复缺陷所付出的设计开发等环节的投入成本，从这两个意义上讲，质量都是必须的，但是如果过度的关注质量，在质量控制上投入远超过预期的过多的时间和成本，会直接影响公司的整体成本，造成的直接后果就是虽然软件产品质量很好也很让客户满意但是公司赔钱了甚至破产了。不充分的测试是对用户不负责任，过度的测试是对公司不负责任，确实是比较难以权衡的选择， 作为软件项目的开发和测试负责人，项目经理、开发经理和测试经理们经常面临着这种权衡，本文的目的之一就是帮助这些软件项目决策者们解决一个主要问题：对正在进行测试或发布的软件进行质量成熟度评估，以确定其心中的最大疑问：测试的差不多了吗？还要测多久才能发布？是不是该停止了？现在停止会不会还有风险？风险大概有多大？

对他们而言,如何掌握这个尺度,就需要科学的定量评估工具,帮助其做出正确的判断。本文首先推出的是 Gompertz 可靠性模型,在利用已有的测试数据的基础上,对后期的测试过程进行分析和预测,对软件产品质量进行定量评估,对是否结束测试任务给出判断依据。

Gompertz 模型简介

1968 年 E.P Virene 提出了累计增长的 Gompertz（戈珀兹）预测模型，Gompertz 模型在各个行业都得到了普遍的应用，是一个简单易行的预测模型，被人们广泛的用于各种数据的累计预测上，例如用来预测汽车数量的增长、预测人口的增长、预测油田的产量等等。

Gompertz 模型曲线的典型特点是初期增长缓慢、中期快速增长后期逐渐趋于极度缓慢增长甚至不增长，是一个典型的 S 型曲线，如图一。



图一 Gompertz 曲线

在软件测试领域，很多测试工程师在实际测试中会发现：初期由于环境不熟悉且只做功能测试等原因，初始阶段发现的缺陷会比较少，发现缺陷的增长速度相对缓慢；随着测试人员逐渐进入状态和熟悉环境以后，发现缺陷的速度会迅速加快；测试进入后期阶段以后软件缺陷的隐藏加深，测试难度加大，加上软件中隐藏的缺陷是有限的这个特性也限制了缺陷的无限增长，所以后期测试阶段中缺陷发现效率会进入一个非常缓慢甚至不增长的阶段。软件测试过程中的这种缺陷变化趋势以及增长速度就是一个非常典型的 S 型曲线，满足 Gompertz 模型增长模型的应用条件。

Gompertz 模型可以用来处理测试缺陷数据,进而评估软件的缺陷增长情况。该模型的基本形式为:

$$Y = ab^{c^T} \qquad (1)$$

公式（1）中,

Y 表示系统在测试时间或阶段 T 的缺陷数量 ;

a 表示当测试时间或阶段 T 趋于无穷大时 Y 的极值;

ab 表示系统测试初始, 即 T = 0 时 Y 的初值;

c 表示缺陷的增长速度。

a, b, c 三个模型系数，随着 Y、T 样本数据的不同而各有不同，可以通过非线性回归等方法得到 a、b、c 的数据，当前的很多统计工具软件如 SPSS、SAS 等都可以算出来，也可以用 MATLAB 等数学计算软件来进行拟合求解。确定了 a、b、c 的数据以后，就可以用模型公式来进行预测了。

Gompertz 模型的意义

就软件质量评估领域，Gompertz 模型可以在两个方面进行应用：

当前软件测试质量的评估

即判断当前测试质量处于什么阶段，预计可以在什么质量状况下结束，例如对 Gompertz 模型定义一个客户可以接受的测试结束准则：被测对象的缺陷移除率超过 f，如果缺陷的已发现比例（或者移除率）超过了预先设定的标准，就可以决定发布软件或者版本。

缺陷移除率计算公式如下：

$$f = \frac{Y}{a} \qquad (2)$$

公式（2）中，

f 表示缺陷移除率（已发现缺陷比例）；

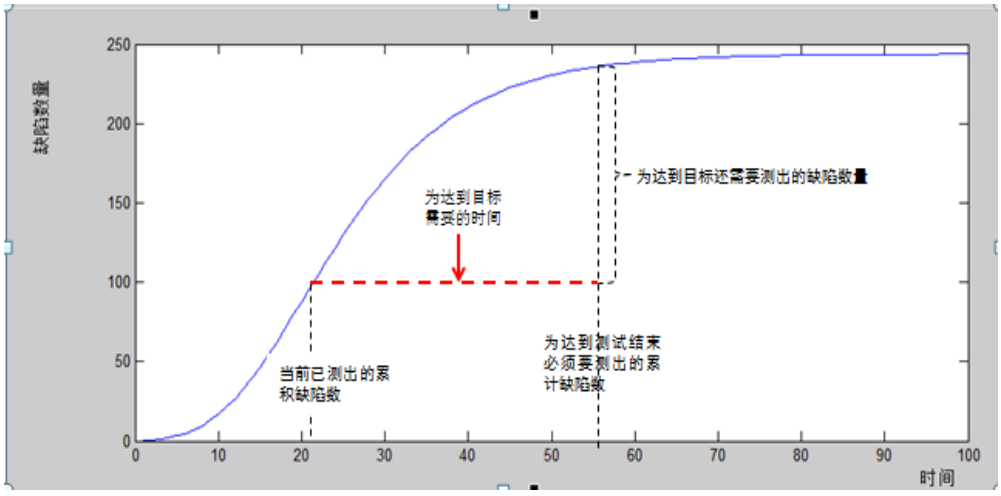
Y 为当前已经发现的累计缺陷数量;

a 为理论上预测到的极限总缺陷数量，即公式（1）中的 a。

如果测试经理在测试计划或者其他相关的质量计划控制中,规定了一个关于软件缺陷数量的成熟度发布标准（例如认为 98%的缺陷被发现以后可以发布软件，或者允许保留 2%的遗漏缺陷等指标），那么在 GompertZ 模型下，如果实际的 f>=指定目标值，就可以按照预期的计划控制发布当前软件系统。

缺陷增长情况预测

对于不满足测试可以结束的场景,可以在当前测试状况的基础上,估计能够达到测试结束要求的时间。参考图三，使用 Gompertz 模型,在得到当前已测出的累计缺陷数、为达到测试结束目标必须要测出的累计缺陷数的情况下，分别代入公式（1），就能得到 2 个不同的时间，二者的时间差就是为达到测试结束还需要的测试时间投入。



图三 缺陷增长情况预测

Gompertz 模型应用实例

为验证 Gompertz 模型，笔者特地选取了本人所在公司某产品在系统测试阶段的缺陷数据，如表一。

表一 测试阶段累计缺陷数量表

测试第 n 周(n=1~8)	每周发现缺陷数量	累计缺陷数量
1	19	19
2	26	45
3	37	82
4	81	163
5	40	203
6	24	227
7	7	234
8	7	241

上述数据的 Gompertz 模型采用 matlab 软件可以进行计算和曲线拟合，matlab 程序代码如下：

```
function f=Gompertz(r,tdata)
%UNTITLED Summary of this function goes here
%     Detailed explanation goes here
f=r(1)*r(2).^(r(3).^tdata);
end
T=[1:1:8]
Y=[19,45,82,163,203,227,234,241]
r0=[238,0.5,0]
r=lsqcurvefit ('Gompertz',r0,T,Y)
```

拟合结果得到 Gompert 模型的三个系数结果如下：

r=2.528005471931133e+02　　7.761166905655105e-04　　0.509277728686215

对于公式（1），其中 a=r(1)= 2.528005471931133e+02，b=r(2)= 7.761166905655105e-04，c=r(3)= 0.509277728686215，所以本例应用的 Gompertz 模型公式为：

$$Y = 252.8005471931133 * 0.0007761166905655105^{0.509277728686215^{T}} \quad (5)$$

得到的缺陷和时间点阵图以及拟合后的 Gompert 模型曲线如图四，可以看到

当前第 8 周的累计缺陷 241 已经比较接近理论极限值 252.8。

1）假设测试发布标准为缺陷移除率（缺陷发现比例）f=95%。

当前缺陷移除率=241/252.8*100%=95.33%，是大于 95%的，所以软件可以发布。

2)假设测试发布标准为缺陷移除率(缺陷发现比例)f=98%,那么当前 95.33%的缺陷移除率并不足以保证发布。

为达到 98%的缺陷移除率，需要的累计缺陷数量为 Ym,代入公式（2）得到

98%=Ym/2.528005471931133e+02, 得出 Ym=247.7445（意味着还需要再测出 6.7 个缺陷才能发布）

Ym=247.7445 代入公式(5)得到

$$247.7445 = 252.8005 * 0.00077611669056551 05^{0.509277728686215^T}$$

得到 T=8.70024103848744（周），一周按照 5 个工作日计算的话，还需要投入 3.5 个工作日左右的时间。

这样产品测试经理就预算出为达到计划的测试发布标准，当前还大概需要投入 3.5 个工作日。

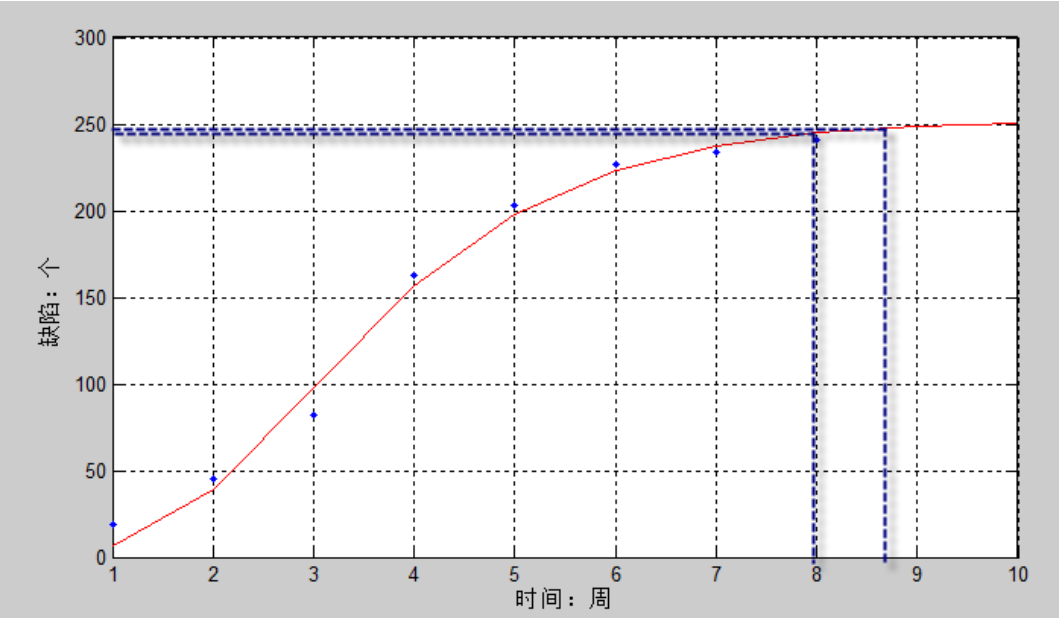


图 4 缺陷-时间点关系以及 Gompert 拟合曲线图

Gompertz 模型的局限和约束

Gompertz 模型的使用也有一定的局限性。

1)测对象的一致性。即在被统计的时间范围内,测试对象不应该有大的变动，

例如突然加入大量的新需求等这种突发状况就是不合适的，必须是针对规划好的软件开发目标，规划好的需求和任务的在测试的不同阶段都是被测试的对象上。

2）测试投入的一致性。在同一统计时间内，测试的组织架构和人力投入应该是均匀的，不能出现测试投入的极度不均匀性现象，否则曲线拟合结果的可靠性就非常低。

3）测试的轮次必须是多轮。只有一轮测试的数据是不可靠的，只有一轮的测试中,受测试策略以及人为安排等因素的影响，很可能产生截然相反的两种缺陷增长趋势,导致 Gompertz 拟合曲线的可信程度过低,进行决策的风险就较大，在多轮测试的场景下，这种风险会显著降低。

为了提高 Gompertz 模型评估的准确性,测试策略制定应遵循：

1）尽可能保证测试效率，在每一轮测试结束后建议进行测试分析，,结合一定的缺陷分析方法,如 ODC 方法等来补充测试用例，以提高缺陷发现的效率，保证每轮测试甚至每工作日的最大缺陷发现效率，提高样本数据的可靠性；

2）尽可能进行比较全面的覆盖测试,不要出现每轮测试的覆盖程度不一致的现象，尽可能的通过自动化测试等手段提高测试效率并保证合理的测试覆盖率

在其他细节上，需要考虑缺陷的权重影响，用加权缺陷值去替代简单缺陷数量；还需要考虑样本数据的取样尽量平滑，如果以天为单位的数据曲线不够平滑，考虑用多天或者以周为单位取样；另外如果拟合的过程中出现计算不收敛的情况，那就意味着模型不适用，必须要考虑样本数据的准确性是否合适或者换其他方式进行评估。

结语

Gompertz 模型是可以应用特定场景下的测试质量评估的，应用的条件相对比较严格，对有过科学严谨规划的研发型产品测试，对有多轮测试且测试投入相对比较均匀的测试团队来说是合适的，应用 Gompertz 模型时候必须要遵循测试效率最大化、测试对象一致、测试投入均匀等原则来保证测试样本数据的可靠性，同时 Gompertz 模型有助于指导上述场景的测试策略的制定，提高测试管理人员对当前测试进度的掌控，促进对开展高效测试的的思考。

手机软件兼容性测试

作者　王晓芹

摘要：目前智能手机种类繁多，功能各异，这就使得同一款手机软件需要在多种手机软硬件平台上运行。要保证手机软件运行正常，就需要对手机进行兼容性测试。本文在手机软件兼容性问题产生原因分析的基础上，介绍了手机软件兼容性测试的内容，并简要介绍了两种手机软件兼容性测试工具。

关键词：智能手机，手机软件，兼容性测试

1. 引言

智能手机（Smartphone）同个人电脑一样，具有独立的操作系统，可以由用户自行安装软件、游戏等第三方服务商提供的程序，通过此类程序不断对手机功能进行扩充，并可以通过移动通讯网络实现无线网络接入。随着计算机技术及通信技术的不断成熟及创新，智能手机市场不断朝更加智能化方向发展，智能手机占据了高端手机市场，在手机市场中所占的比重越来越大，功能机正在被智能手机逐步替代。而智能手机品牌的日趋繁多，使得其硬件与所配置的软件也不尽相同，手机软件正面临着兼容性的突出问题。

2. 手机软件兼容性问题产生的原因

造成手机软件兼容性从技术上来理解是指该手机软件能否与周围的软件与硬件正确地交互和共享信息。对于手机软件来说，它需要处理多种来源的信息，而这些信息的表达格式可能是各不相同的，这种 “表达”语言的不同，是造成手机软件不兼容问题最直接的原因。

商业因素可以说是造成手机软件不兼容的最根本的驱动力，正因为商业因素的影响，才出现了各种硬件及相关软件，最终导致手机软件兼容性问题。从商业的角度来讲，手机软件兼容性问题产生的原因主要有：

（1）手机用户需求不同。目前已经有越来越多的用户开始使用智能手机，每个用户对手机功能的需求不尽相同，而且用户也越来越重视用户体验，因此开发的手机基础既要满足用户的不同需求，又要满足同一功能在不同手机上具有相同的用户体验。

（2）移动设备种类繁多。目前市场上的各种移动终端设备越来越多，如手机、平板电脑等。

（3）生产厂商繁多。目前手机的生产厂商越来越多，如三星、诺基亚、苹

果、摩托罗拉等，各个生产厂商竞争激烈，在软硬件设计方面也不尽相同，这也使同一款手机软件与不同硬件之间产生了不兼容性。

（4）手机软件数量巨大。目前在智能手机上使用的软件越来越多，许多传统软件都有在智能手机上使用的需求，因此导致现在智能手机软件的开发者和开发软件数量也越来越多。虽然这一因素并不是真正导致兼容性问题产生的原因，但是却使得手机软件兼容性问题变得更加突出和迫切。

3. 手机软件兼容性测试的内容

手机软件兼容性测试的内容与传统软件的兼容性测试大致相同，包括硬件兼容性测试、软件兼容性测试及数据兼容性测试，具体说来，手机软件兼容性测试的内容主要包括以下几方面：

（1） 与手机操作系统之间的兼容性。有些软件需要在不同操作系统平台上重新编译才可运行；有些软件需要重新开发或改动较大，才能在不同平台运行，而理想的软件应该具有平台无关性。目前应用在手机上的主流操作系统有 Palm OS、Symbian（塞班）、Android（安卓）、iOS（苹果）、Black Berry（黑莓）OS 6.0、Windows Phone 8 等。

对操作系统的兼容性测试，需要考虑安装测试、卸载测试、功能遍历测试、运行稳定性测试、界面显示测试、流畅度测试、分辨率兼容性测试等内容。其中运行稳定性测试是指在软件功能抽查过程中未发现致命死机现象；界面显示测试主要包括界面输入（输入域内是否可以完成正常的输入）、界面显示无变形、界面文字无乱字符、排版显示是否正常、普通文本显示是否正常、图片显示是否正常、图片缩放是否正常、图片链接是否正常打开、表格显示是否正常、文字链接能否正常打开等；分辨率测试是为了保证页面版式在不同的分辨率模式下能正常显示、字体符合要求而进行的测试。不同型号的手机具备不同的分辨率，必须针对需求规格说明书中建议的分辨率进行专门的测试。对于需求规格说明书中规定的分辨率，必须保证测试通过，而对于需求规格说明书中没有规定分辨率的项目，测试应该在完成主流分辨率的兼容性测试的前提下，尽可能进行一些非主流分辨率的兼容性测试，在一定程度上保证支持大部分分辨率模式。

另外，对操作系统的兼容性测试还包括在同一操作系统的不同版本上进行兼容性测试。如 Android 系统的不同版本（Android1.0-Android4.0）、苹果系统 iOS 的不同版本等。

（2） 与其它应用软件之间的兼容性测试。手机软件运行需要哪些应用软件支持，判断与其他常用软件一起使用，是否会造成其他软件运行错误或本身不

能正确实现其功能。常用的手机软件有：聊天软件（qq、飞信等）、阅读器、导航、地图、浏览器、社交网站客户端等。其中由于浏览器应用的广泛性，软件对浏览器的兼容性往往成为用户使用中所关注的重点。对浏览器兼容性测试，需要网络应用程序在不同的浏览器上进行测试，以保证不论使用何种浏览器查看 web 应用程序，用户都有相同的视觉体验；在功能方面，应用程序通过不同的浏览器必须以相同的方式表现和回应。目前比较热门的手机浏览器有冲浪浏览器、掌上百度、UC 浏览器、Go 浏览器、Opera、YoYo 手机浏览器、MP 手机浏览器、Skyfire、Windows 手机浏览器、Safari、星际手机浏览器、MagicMaster、海豚浏览器、欧朋浏览器等。

（3） 与手机硬件之间的兼容性。对于不同厂家生产的同一操作系统的手机和同一手机的不同型号需要进行兼容性测试。目前市场上有许多不同的手机生产厂商，如小米、HTC、三星、苹果等。

（4） 对于移动信息化系统中所涉及的手机客户端软件，往往需要连接数据库，这种情况下，需要对不同类型的数据库进行兼容性测试。常用的数据库如 MySql、Oracle、SQL Server 等。当软件升级后可能定义新的数据格式或文件格式，这就涉及到对原有格式的支持及更新，原有用户记录应能继承，在新格式下依然可用，同时还应考虑转换过程中数据的完整性与正确性。

4. 手机软件兼容性测试工具

4.1 CTS

CTS 英文为 Compatibility Test Suite，意为兼容性测试，是一个专门用于 Android 的兼容性测试工具。只有通过 CTS 测试的设备才有可能获得 Android 的商标和享受 Android Market 的权限；Android 的 CTS 的目的与意义是使用户在 Android 系统中有更好的用户体验，并且展示 Android 应用的优越性，使得 Android 开发者更容易编写高质量的 Andorid 程序。

CTS 是一个免费的，商用级的测试套件，CTS 运行于台式机直接连接的设备或仿真器上，并执行测试用例。它的目的是揭示早期的不兼容问题，并确保该软件在整个开发过程中一直兼容。

CTS 是一个自动测试工具，它包括两个主要的软件组件：（1）CTS 测试工具运行在台式机器上并管理测试执行；（2）单个测试用例在连接的移动设备或模拟器上执行。测试用例是用 Java 编写的并作为 JUnit 测试运行，并包装成 Android .apk 文件运行在实际的设备上。

4.2 Mobi Ready

Mobi Ready 是爱尔兰 DOTMOBILE 公司所开发的一款在线测试工具，其主要

功能是帮助用户验证某个站点或页面是否适合在手持设备上显示。很多时候，我们会发现一些页面内容不可访问，也就是说能用电脑所访问的一些网站却无法支持手机访问，而 Mobi Ready 测试工具正是为解决这一问题而开发的。

打开在线测试页面 http://ready.mobi/launch.jsp?locale=en_EN，只需要在指定的输入框中输入要测试的页面 URL，提交后就可以返回测试报告，这种情况下用户使用的是默认配置。另外，可以通过 More Options 按钮得到自定义测试配置界面，目前配置界面中提供了 6 种选项，即 Choose a device；SonyEricssonK750i；Nokia6680；SAMSUNG-SGH-T809； Motorola RAZR；W3C mobileOk DDC。其中第一个选项即为工具的默认配置；1~4 选项分别为手机类型，第 5 个选项表示按照 W3C 组织所定义的缺省提交上下文进行配置。

做好软件测试，提升医疗软件服务质量

作者：沈月

摘　要

随着医疗软件实现的越来越多样化，软件服务商之间的商业利益直接体现在了所提供的软件质量上，软件质量的高低直接决定着服务商的口碑和信誉。一套成熟的可以上线使用的医疗软件形成的过程中，软件测试，作为质量把关的关键要素，贯穿在其软件生命周期的各个环节。在对医疗软件服务质量不断提出新要求的今天，软件测试也发挥着举足轻重的作用。

关键词

软件测试 质量 医疗 电子病历

背景

根据不完全统计，我国医疗卫生领域医疗软件的生产供应商约有 500 家，其中，医院信息系统生产供应商 300 多家，而这些生产供应商根据规模来分，大中型已经占据了 75%。

软件行业正处于持续稳定发展的势头，未来，软件供应商之间的竞争只会更加残酷，而供应商在扩大业务范围提高商业竞争力的同时，医疗软件的质量总是无时不刻不作为警钟被敲醒，不仅是作为企业树立口碑，建立良好形象的关键要素，同时，医疗软件因其服务人员的特殊性，数据存储的正确性和安全性，更是对我们提出了巨大的挑战。

在软件行业乃至更多的行业中，无论何种类型的产品，质量往往决定着公司的命脉。而在软件研发的生命周期中，我们多数重视的是软件的开发过程，而忽略了软件测试。在国内软件发展初期，软件测试普遍地地位低、发展慢，一直得不到重视，使得我们的软件质量缺陷不可避免。

软件测试：使用人工或者自动手段来运行或测试某个系统的过程，其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别。

医疗软件在各大医院中的实行受到国家相关部门的监督和规范，其质量要求在某些方面不亚于银行软件业务要求，如涉及到重大医患、死亡病例数据等一系列特殊的医疗业务，尤其引起我们对质量把关的重视。其特殊性也对从事其软件研发的人员提出了更为严格的要求。

软件测试与质量

美国质量管理专家戴明博士首先提出 PDCA 循环，又称戴明环，它是全面质量管理所遵循的科学程序。全面质量管理活动的全部过程，就是质量计划的制

定和组织实现的过程，这个过程就是按照 PDCA 循环，周而复始地运转。

P：计划，即制定计划，包括制定方针和目标。

D：执行，即执行计划，具体运作，实现计划内容。

C：检查，即检查结果。找出与计划不符的问题，总结执行的效果。

A：行动，即采取行动。对检查结果采取适当的修正行动使之符合计划中提出的目标。

图（1）

在软件过程领域中，戴明环同样适用，软件测试与“检查（C）”密切相关，测试人员检查并确定软件是否能达到用户期望的要求，将不符合项汇报给开发人员进行修正，修正后再有测试人员进行验证，以这样一个循环的过程，来提高用户满意度。

因而，软件测试与软件质量是密不可分的，充分做好软件项目的测试工作，在交付使用前做到尽可能发现更多隐藏的缺陷，从而提高用户满意度，是提高软件质量的重要措施之一。

医疗软件测试的过程

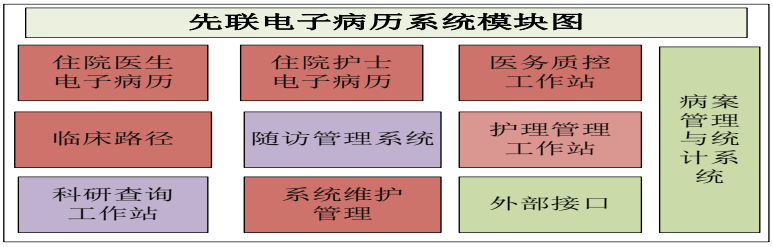
3.1 业务要求

不同于其他有形产品或是一般服务软件，医疗软件的业务需求是建立在国家制定的一系列规范标准基础上，具有独特性。以卫生部下发的《电子病历系统功能规范（试行）》为例，文件中明确了医疗机构电子病历所应具备的功能，涵盖了电子病历的基础功能，包括用户授权与认证、数据存储管理、数据字典等。这就要求我们的测试团队在进入到医疗软件服务的领域中来时，首先要对卫生部下发的文档进行解读，确保我们的产品符合卫生部最基本的要求，除此之外，涉及到更细节的文档如电子病历书写规范等，更是对我们过程提出了更为具体的测试思路。

测试人员除了对卫生部下发的规范文档要具备一定的认识以外，还要熟悉本公司签订的技术需求合同，一般的，软件供应商会安排专业的需求人员在技术合

同的指导下进行数据采集与分析，制定需求用例说明书，测试人员根据此拟定相应的测试方案、测试用例等相关文档。

以我们公司主要的产品—电子病历 EMR 为例，主要涉及到以下十大模块的设计。如图（2）



图（2）系统模块图

其中，贯穿整个系统的一条主流程业务线：病人入院—住院诊疗护理工作—出院或转院。测试人员必须要掌握这样一条主线，才能更有效地展开测试。

其次，电子病历业务中涉及到最多的当属表单，表单的共性一般是具有增、删、改、查的功能，其他功能一般以卫生部或地方卫生机构的政策或标准为前提根据表单作用进行定制化研发。大多数表单形式与内容都是由卫生部统一的，如图（3）卫生部下发的关于临床路径的表单，表单中规定了该临床路径所要进行的诊疗相关工作内容，页面需要展示的字段名称等。

图（3）临床路径表单

表单的测试在整个 EMR 系统测试过程中应该是属于比较繁琐和重复性的一项工作，往往的，我们的测试人员只会去关注需要我来测哪些功能，怎么测，而忽略了对每张表单的理解，其实，每张表单都是相关卫生部门根据医院中涉及到的具体业务来制定的，对医院运转形式、病人住院过程有一个清晰的思路和认识，就很容易理解每张表单的具体作用，例如，临床路径表单是针对针对一组特定诊断或操作，如针对某个 ICD 码对应的各种疾病或某种手术等，进行统一规范的治疗手段而制定的表单，避免了同一疾病在不同地区、不同医院，不同的治疗组或者不同医师个人间出现不同的治疗方案，提高了准确性、预后等等的可评估性。在理解了表单分别不同的作用后，在测试时才不会觉得自己总是在重复着同样的工作。另外，表单具体内容的准确性测试应该以电子病历的书写规范标准作为参考。

综上，好的软件离不开测试团队对业务的掌握程度，如果测试人员拥有高超的测试技术和测试方法而对业务需求却一知半解，是交不出优秀的产品出来的，医疗软件行业中有其特殊的业务需求，且医院作为一个公共的卫生场所，在必要

急性 ST 段抬高心肌梗死临床路径表单

适用对象：**第一诊断为**急性 ST 段抬高心肌梗死（STEMI）（ICD10：I21.0- I21.3）
患者姓名：_____性别：____年龄：____门诊号：_____住院号：_____
发病时间：___年__月__日__时__分　　到达急诊科时间：___年__月__日__时__分
溶栓开始时间：___年__月__日__时__分　PCI 开始时间：___年__月__日__时__分
住院日期：___年__月__日　　出院日期：___年__月___日，
标准住院日 10-14 天　　实际住院日：___ 天

时间	到达急诊科（31—90 分钟）	住院第 1 天（进入 CCU24h 内）
主要诊疗工作	□ 询问病史与体格检查 □ 建立静脉通道 □ 心电和血压监测 □ 描记并评价“18 导联”心电图 □ 开始急救和常规治疗	□ 急请心血管内科二线医师会诊（5 分钟内到达）：复核诊断、组织急救治疗 □ 迅速评估“溶栓治疗”或“直接 PCI 治疗”的适应证和禁忌证 □ 确定再灌注治疗方案 □ 对拟行“直接 PCI”者，尽快术前准备（药物、实验室检查、交待病情、签署知情同意书、通知术者和导管室、运送准备等） □ 对拟行“溶栓治疗”者，立即准备、签署知情同意书并尽早实施
重点医嘱	□ 描记“18 导联”心电图 □ 卧床、禁活动 □ 吸氧 □ 重症监护（持续心电、血压和血氧饱和度监测等） □ 开始急性心肌梗死急救和“常规治疗”	□ 急性心肌梗死护理常规 □ 特级护理、卧床、禁食 □ 镇静止痛 □ 静脉滴注硝酸甘油 □ 尽快准备和开始急诊“溶栓”治疗 □ 从速准备和开始急诊 PCI 治疗 □ 实验室检查（溶栓或急诊 PCI 前必查项目） □ 建立静脉通道 □ 血清心肌酶学和损伤标志物测定（不必等结果）
主要护理工作	□ 建立静脉通道 □ 给予吸氧 □ 实施重症监护、做好除颤准备 □ 配合急救治疗（静脉口服给药等） □ 静脉抽血准备 □ 完成护理记录 □ 指导家属完成急诊挂号、交费和办理“入院手续”等工作	□ 急性心肌梗死护理常规 □ 完成护理记录 □ 特级护理 □ 观察并记录溶栓治疗过程中的病情变化及救治过程 □ 配合监护和急救治疗 □ 配合急诊 PCI 术前准备 □ 做好急诊 PCI 患者转运准备
病情变异记录	□无　□有，原因： 1. 2.	□无 □有，原因： 1. 2.
护士签名	白班 / 小夜班 / 大夜班	白班 / 小夜班 / 大夜班
医师签名		

注：适用于 STEMI 发病<12 小时者，择期 PCI 患者不适用本流程。

的时候，测试人员可以做实地观察，了解身边医院的业务流程，掌握周边医院信

息化工作的展开方式，观察或走访用户（医院工作人员）使用软件的习惯以及反馈。这些看似不起眼的数据搜集工作，往往会对我们在做测试工作时产生巨大影响。

3.2 技术要求

3.2.1 功能测试

功能测试就是对照产品的功能清单进行验证，逐项执行测试用例，检查产品是否达到用户要求的功能。功能测试又称为黑盒测试，即不考虑产品的内部程序及实现方式。

在做功能测试之前，需要测试人员做好以下准备工作：

（1）测试计划：测试计划一般由测试负责人制定，根据项目周期协调人员、安排工作量等。

（2）测试方案：在测试过程中可省略，一般来说，测试方案是根据产品技术合同中的需求来制定的较为详细的初稿，我所在的公司，会由专门的产品规划部门来进行这项工作。产出需求用例规格说明书。

（3）测试用例：测试用例是作为软件测试人员必须会书写制定的文档，一条测试用例应至少包括用例编号、标题、优先级、描述、预置条件、预期结果、实际结果。功能测试的用例编写方法有：等价类划分法；边界值分析方法；因果图方法；错误推测方法；判定表驱动分析方法；正交实验方法等。

（4）文档评审，即测试执行前所有产出文档的评审，包括需求规格说明书和测试用例。需求规格说明书也是由公司的需求人员解读后整理的，所以不能100%肯定的说，与医疗机构签订的技术合同完全相符，需要项目组的所有成员进行评审，如有质疑，可进一步与医疗机构进行确认。而用例则需要内部项目组成员达成一致意见，防止存在遗漏问题。

EMR 住院系统的功能测试，以临床路径为例，其业务需求主要由后台定制、前台应用和统计报表三大模块组成。后台定制又分为 7 块功能，每个功能对应到不同的表单。在医疗软件中几乎没有独立于业务流程外的表单，每张表单都是嵌套在业务流中的，对业务操作有直观的了解后，便能迅速地将业务拆分成对表单的具体操作。运用用例设计方法进行用例设计，如新增或修改表单中某个数据字段时，一方面要考虑输入正常值和非正常值（等价类划分法），还要考虑最大值和最小值（边界值法）等。

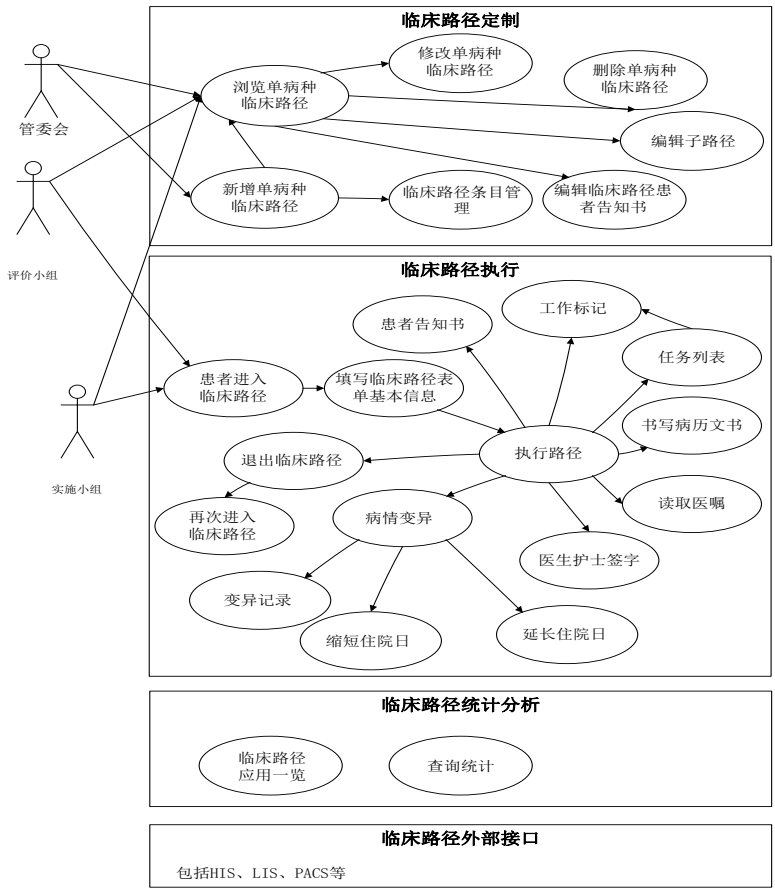
图（4）临床路径实现流程

除此之外，软件的功能测试正常情况下会经历 2~3 轮的迭代，任何一款软件几乎都不可能一次性开发后就可以测试全部通过。如果是全新的系统进行开发，可能会采用敏捷开发模式，中间进行的测试迭代次数会更多。

3.2.2 性能测试

我们知道，一家三甲级别的医疗机构，每天的接诊数量是庞大的，医护人员对系统的使用频率也是频繁的，甚至会要求我们的系统必须 24 小时不停地运转使用。我国对卫生信息化的普及使得中大型医疗机构对软件系统的依赖性越来越高，也正在取代纸质书写的做法。对软件供应商来说，仅仅完成功能需求，显然是不够的。而软件的承载能力究竟有多少？是否能满足日益增加的数据量？软件能在正常状态下维持多久？这些都是需要我们去思考的问题。

性能测试作为验证上述问题的手段之一被提出来，以 LoadRunner 测试 EMR 为例，在测试前，确保系统功能测试中的问题已修改完毕，同时还需要做好测试



服务器的规划，如表（1）：

服务器	IP	硬件配置	软件及版本	备注

WEB 应用服务器	10.0.1.152	CPU： Intel Xeon(R) E5645 @ 2.4GHz 内存：2G	windows server 2003 R2 X64 tomcat6	http://10.0.1.152:8082/emr
DB 应用服务器	10.0.1.130	CPU： Intel Xeon® E5645 @ 2.4GHz 内存：2G	windows server 2008 R2X64 Sql Server2005	jdbc:db2://10.0.1.130:50000/emr_TEST
客户端	192.168.100.127	CPU： Intel Core E7400 @ 1.8GHz 内存：2G	WindowsXP Professional SP3 IE6 Loadrunner11	可以与 10.0.1.XXX 网段互连

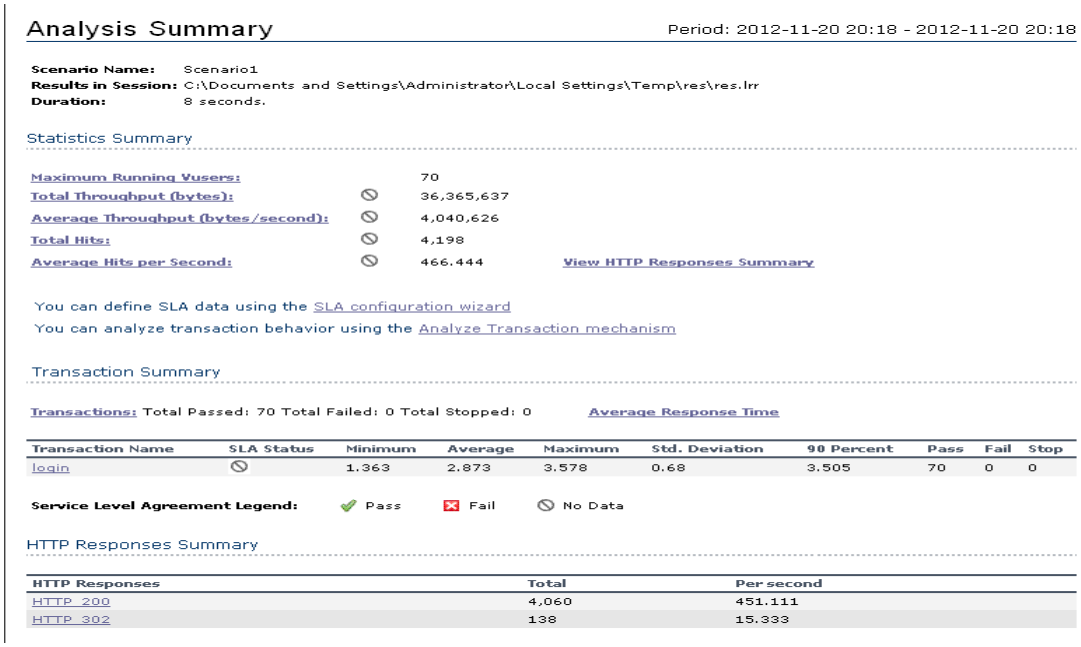
表（1）服务器规划

其次，需要通过各种方式搜集相关医疗机构的数据（如工作人员总数、平均每天的在线数量、平均接诊数量、频繁操作的表单列表、统计报表的使用情况等与性能有关的数据），从而制定性能测试场景。如表（2）是简单测试并发 70 个用户数登录同时登录系统的响应情况

用例序号	场景名	场景描述	测试结果			
			实际并发响应时间（秒）			实际人通过率
			最大值	最小值	平均值	
TC01	场景一（登录）	70 人输入账号密码，并发 70 人点击登录，查看系统响应时间				

表（2）场景设计

场景设计完毕后，开始执行场景，进行性能分析如图（5），如需要发现性能问题的瓶颈，可通过不断增加用户数量来实现。



Analysis Summary

Period: 2012-11-20 20:18 - 2012-11-20 20:18

Scenario Name: Scenario1
Results in Session: C:\Documents and Settings\Administrator\Local Settings\Temp\res\res.lrr
Duration: 8 seconds.

Statistics Summary

Maximum Running Vusers:	70
Total Throughput (bytes):	36,365,637
Average Throughput (bytes/second):	4,040,626
Total Hits:	4,198
Average Hits per Second:	466.444

View HTTP Responses Summary

You can define SLA data using the SLA configuration wizard
You can analyze transaction behavior using the Analyze Transaction mechanism

Transaction Summary

Transactions: Total Passed: 70 Total Failed: 0 Total Stopped: 0 Average Response Time

Transaction Name	SLA Status	Minimum	Average	Maximum	Std. Deviation	90 Percent	Pass	Fail	Stop
login		1.363	2.873	3.578	0.68	3.505	70	0	0

Service Level Agreement Legend: Pass Fail No Data

HTTP Responses Summary

HTTP Responses	Total	Per second
HTTP_200	4,060	451.111
HTTP_302	138	15.333

图（5）性能分析概要

3.3 测试与维护

提到软件，便离不开维护，且医疗软件在一开始具有版本可移植性，为大多数供应商创造了利益。但是基线产品如果得不到保障，那么维护的成本将会是巨大的，这是绝大多数供应商不愿意看到的，因而如果在基线版本建立的初期能够更好的理解需求，更好地在测试环节做好把关，能大大地减少维护成本，减少企业损失。

结语

软件测试，不论是在医疗软件还是在其他软件中，重点要对业务和技术有所掌握，只有两者更好的相结合，才会将测试工作做好，从而更快更好地提升医疗软件服务质量。

5.参考文献

[1] 软件测试的有效方法（原书第 2 版）/（美）佩里（Perry，W.E.）著；兰雨晴等译 --北京：机械工业出版社，2004.3 ISBN 7-111-13750-7

[2] 医院管理学.信息管理分册/李包罗等分册主编—2 版—北京：人民卫生出版社，2011.12 ISBN 978-7-117-14764-4