目 录

数据仓库特点与测试流程分析

作者：吴昊占

摘要

本文主要通过对数据仓库特点与测试流程的分析，并结合实际工作中积累的管理心得体会分享给读者朋友，希望能对数据仓库的测试有所帮助。

关键词

数据仓库　测试流程　测试管理　测试策略

正文

大数据走到今天，已经呈现出多元化竞争发展的趋势，各种数据仓库产品与技术层出不穷，在数据仓库开发技术如火如荼行进的过程中，针对数据仓库的测试技术也在逐渐起步中走向稳定成熟。

今天，结合在实际工作中的具体情况，从数据仓库的测试特点与测试流程两个角度略述己见。

测试特点

作为大数据的存储单元，数据仓库一般都处于其他外围系统下游，上游数据经过简单处理甚至不处理，直接同步到数据仓库系统中，在数据仓库系统中定义一系列组织级别的统计逻辑和报表规范，按照该统计规则将上游基础数据经过层层运算以指定的报表格式呈现在用户面前，这就是数据仓库的“职责”所在。

正是处于这样的系统角色，数据仓库自身有以下几个特点：

1.数据量大

各种外围上游系统每天上百万条数据进入数据仓库，在仓库中进行存储、转换、装载，从这个角度看，性能指标是必然要考虑的测试要点。

2.被动需求多

由于处于系统群的下游，上游数据结构或数据格式发生变化时，数据仓库必须及时调整对此次上游变更的兼容处理。例如上游系统添加枚举值，则仓库也要添加对枚举值的处理；上游系统更改表结构，仓库及时变更取数逻辑；上游系统增加新功能、新模块，仓库需要增加对新数据流的处理逻辑等

3.业务数据范围广

不同的上游系统有不同的业务场景，在不同的业务场景下所生成的业务数据需正常进入数据仓库，并纳入统计数据源中。例如一笔交易分为报价成交、应急成交、撤销、应急撤销等，在不同的状态业务流下所生成的数据均需要顺利同步到数据仓库系统中。

4. 与业务系统分离

数据仓库系统只有统计功能，它与业务系统分离，并且经过多层处理，例如数据仓库的数据，经过抽取，清洗，展现前会经过数据挖掘，数据再处理，有些字段在原始数据表中根本就没有。这样的数据准确性测试比较复杂，当然检查出数据错误，修改定位也是很不容易的

5. 权限控制严格

数据仓库所统计出的数据在很大程度上提供中高层提供决策分析依据，在整个战略高度上有重要意义，如何保护数据的安全在数据仓库的权限控制环节中显得尤为突出。

针对以上数据仓库五个特点，在实际的测试过程中如何规避测试盲点？如何更有效、更高效的执行测试活动呢？依据以往的测试经验，并不是毫无章法，现简述在工作中积累下的一些心得与广大测试同行分享下。

1. 提高对上游系统业务的熟悉度

对任何一个软件系统进行测试，必然要熟悉它的业务，但是数据仓库报表同一般的业务功能还是有较大区别的。例如对于报价单的增、删、改，通过对界面的浏览和探索性的操作，大概就可以弄明白它的业务流程和业务规则，因为这些内容比较直观易懂，但是数据流转到数据仓库系统中，就很难直观的看到我们所需要了解的内容了。简单来讲，对于报表业务的熟悉，主要是两个方面：数据项的算法和数据来源，也就是说要明白一个数据项同具体的业务有什么关系，单据的增、删、改或者状态的变化，对报表中各个数据项的计算会产生什么不同的影响。如果不知道这些，那么就无法验证报表中的数据是否准确，也无法通过报表去检查业务系统的正确与否。

2. 覆盖全部客观的查询统计方式

对于一个报表的用户而言，可能是企业的中层或高层，每个人都有自己的报表使用特点，例如在某交易系统中，可能按照优惠券代码统计查询，也可能按照优惠券类型统计查询，也可能按照优惠券优惠额度统计查询等，这些基本都是用户在实际的业务过程中的需要决定的，所以在测试用例的设计过程中切忌主观臆念，以自己使用习惯为准，而忽略了客观全面的查询统计方式。

3. 建立数据可控的数据源环境

数据是数据仓库报表的灵魂。报表的基本功能就是通过各种查询统计分析的方法，分析处理数据源为用户提供准确的统计数据，帮助用户做出决策。那么如何捍卫报表的“灵魂”呢？

首先，保证足够多的有效数据。在报表测试时，应当尽可能的覆盖到报表所

提供的各种查询统计方法，因此至少应该保证每一种查询统计方法都应该有对应的数据，得到的结果都不会是 0，否则等于没有覆盖到这个被测的查询统计算法。当然数据也不是越多越好，能保证全部覆盖，并且刚好够用就可以了，因为数据的准备和生成也是很花时间的。

其次，要保证数据的可控。数据并不是随意生成的，如果使用通过自动化工具或者通过业务测试时随意的输入的数据来进行报表测试，一般来说是不太可能的。因为如果无法控制数据来源，那么即使知道报表中每个数据项的算法，也无法最终验证报表的查询统计结果是否正确。

所以如果希望更有效、更高质量的完成报表的测试，那么就要重视并增加对于数据准备工作的关注：用于验证报表功能的数据，一定是专门为报表准备的，并且是经过精心设计，要分析影响数据项算法的各种因素，以及每个因素可能出现的不同变化，这样才有可能覆盖各种查询统计方法；同时，才能保证无论使用哪个数据项的算法进行计算，其结果都是可以预知的——因为数据来源已经被我们控制了。

4. 业务功能测试通过后数据仓库方可测试

如果业务功能本身存在缺陷，导致的数据不准，那么最后进行报表测试也就没有什么意义了。所以，应该在保证各项同报表有关的业务的功能测试通过之后，才开始考虑对报表进行测试。

5. 特别注意四舍五入对统计数据的影响

在一般的交易系统中，都会存在这种情况，无论小数点后保留几位，总是难以避免明细和汇总之间的差别。原因可能是因为规格换算不一致，例如 1/9*9≠1；或者由于利率、拆借期限等因素导致的不一致。

6. 不同业务流、不同状态流对报表数据的影响

例如在某报价单中，A 报价金额共 200 万，B 与之成交 40 万，C 与之成交 50 万，然后 B 撤销成交，C 又成交 70 万，最后被最高权限管理员应急 B 成交 30 万，撤销 C20 万，如此复杂的场景在实际的业务流程中是真实存在的，在此场景下特别注意不同业务流不同状态流对报表统计逻辑的影响。

7. 大数据量下报表的性能指标

报表测试一般都会涉及到性能测试，主要是在大数据量查询统计时的响应速度。大数据量一是说原始数据多，二是被操作、计算的数据多，三是某个数据项被是经过多次计算得出的。特别是对于一些算法比较复杂的报表，10 万条数据和 100 万条数据时的响应时间将表现出巨大的差别。

8.权限控制与访问控制测试

不同的报表是设计给不同的人看的，例如在进销存系统中出入库报表是给仓库管理人员看的，里面不会包含商品的价格，而只会包含数量；而财务报表中，只会包含采购、销售的金额，而不会包含数量，这样才能保证可以相互对照，不会出现营私舞弊的行为。那么在测试时，应当考虑报表是否泄漏了不该泄漏的信息。当然，这里对业务的熟悉程度就是更高的要求了。

测试流程

在测试的过程中如何处理需求与质量的关系？又如何确定分析测试需求编写测试用例？虽然数据仓库的测试过程在一定程度上更倾向于数据库的测试，显得既神奇而又神秘，但实际上它与其他测试项目并没有太大区别，基本的质量控制与测试流程控制在数据仓库项目中仍可以行之有效。

简单描述下其中重要的几步：

分析需求文档与设计文档

这些文档详细说明了数据的来源、如何对数据进行操作，以及存储到哪里。

通过分析需求文档与设计文档，了解系统设计的整体框架与业务流、数据流、状态流，对于创建基本的测试策略非常有用。

编写测试计划与制定测试策略

分析了各种各样的源文件后，就要开始编写测试计划、制定测试策略。从生命周期和质量的角度来看，迭代增量测试是测试数据仓库的最好办法。这个办法的主要优点可从研发早期开始检验缺陷，并使调试变得简单。此外，这个方法还有助于在开发与测试周期中建立详细的过程。

制定数据仓库测试策略的主要问题是基于分析(analysis-based)的测试方式和基于查询(analysis-based)的测试方式的选择。纯基于分析的方法是让测试分析师通过分析目标数据和相关标准计算出预期结果。基于查询的方法有相同的基本分析步骤，但更进一步，用 SQL 查询语言编写预期结果。这为将来建立回归测试过程节省了很大精力。两者各有利弊，在实际工作中权衡使用。

编写测试用例与执行用例

数据仓库的测试计划中确定了测试策略，则测试用例的编写也有了依据，常用的测试用例设计方法在数据仓库报表中均可以得到实际应用，例如等价类划分法、因果图、判定表、正交分析法、错误猜测法、业务流程法等。不同的报表模块不同的查询组合可灵活并交叉利用测试用例方法。

提交 BUG 与回归测试

寻找 BUG 是每一位测试同行的天职，提交 bug 并持续跟踪直至回归测试通过

缺陷关闭，基本流程与普通软件系统并无区别，不再赘述。

总结

整体看来，数据仓库作为一个数据容器，从测试流程和测试策略上并没有太大区别，基本的质量控制体系均可以适用，只需要在一些数据管控、权限管控、性能指标管控下多加注意即可。

ItemCF 算法及其测试方法介绍

作者：敏仪

摘要：随着大数据的广泛应用，淘宝个性化的业务场景越来越多。基于物品的协同过滤（item-based collaborative filtering）算法是推荐算法领域应用比较广泛的一种算法 。本文首先介绍了 ItemCF 算法基础概念；其次介绍了基于 Mapreduce 的算法开发过程分析；最后介绍了如何对 Mapreduce 进行单元测试和本地集成测试。

关键词：个性化推荐、ItemCF 算法、Mapreduce 单元测试

正文：

一、ItemCF 算法-基于物品的协同过滤算法介绍

基于物品的协同过滤（ItemCF）算法是目前业界应用最多的算法。在淘宝网的很多个性化推荐场景，ItemCF 算法都被广泛使用。

基于物品的协同过滤算法为用户进行个性化推荐主要分为两步：

1）计算物品之间的相似度；

2）根据物品的相似度和用户的历史行为给用户生成推荐列表；

ItemCF 算法演变经历了几个重要的阶段。通过几次算法优化，降低了热门的物品和活跃的用户对物品之间的相似度的影响。下面通过算法核心计算公式 A 到公式 C 的介绍，详细介绍 ItemCF 算法。

公式 A：$$W_{ij}=\frac{|N(i)\cap N(j)|}{|N(i)|}$$

公式 A 中，分母|N(i)|是喜欢物品 i 的用户数，分子|N(i) N(j)|是同时喜欢物品 i 和物品 j 的用户数。公式 A 的缺点：如果物品 j 很热门，很多人喜欢，那么 w_{ij} 就会很大，接近 1。因此，该公式会造成任何物品和热门物品的相似度都有很大的相似度。为了避免推荐出热门物品的问题,提出公式 B。

公式 B：$$W_{ij}=\frac{N(i)\cap N(j)}{\sqrt{|N(i)||N(j)|}}$$

公式 B 中，将热门的宝贝 j 被浏览的次数作为分母的一个因子，降低了热门宝贝 j 对 W_{ij} 的影响。

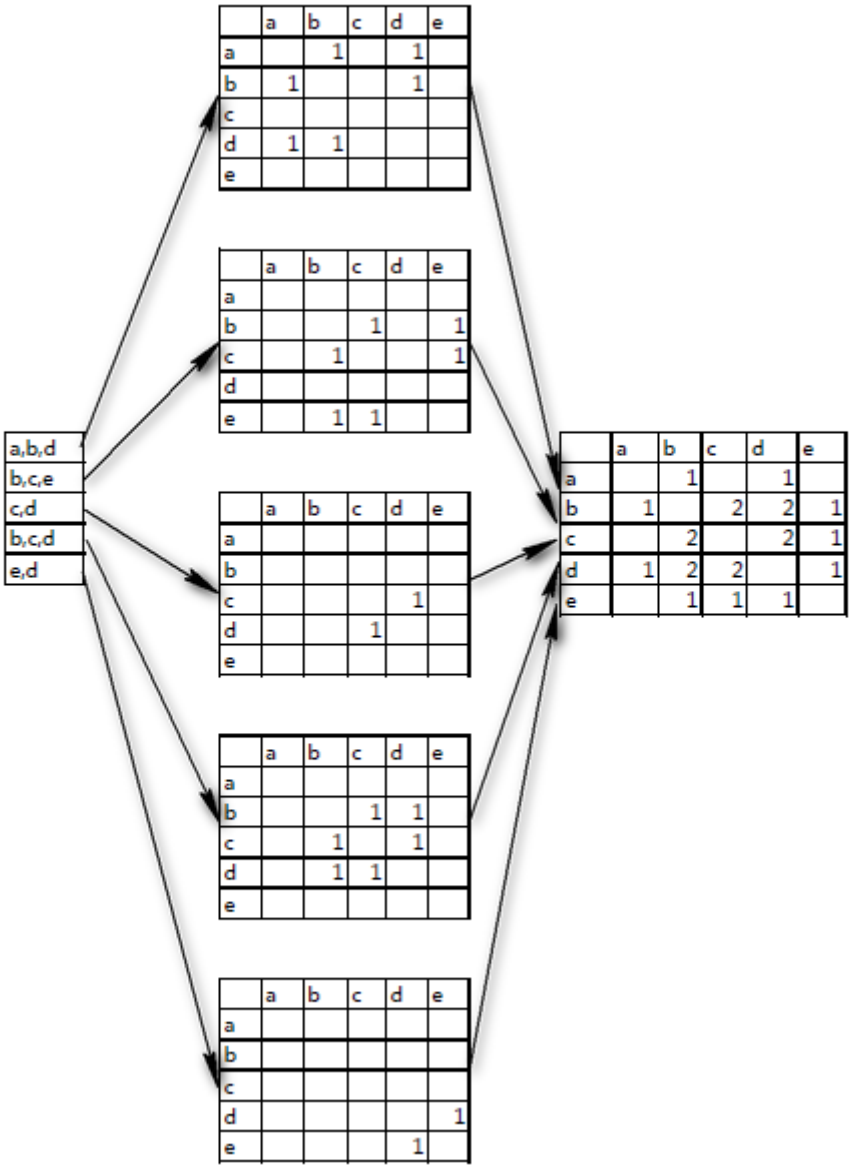


图 1:用户-物品关系表

对用户进行基于 ItemCF 的个性化推荐，需要首先建立用户-物品的关系表（如：图 1）。将用户浏览记录转化为物品共现的二维矩阵。矩阵 C[i][j]为有多少用户不仅浏览了物品 i 并且浏览了物品 j。在公式 B 的基础上，可以得到用户感兴趣的物品。

$$P_{uj}=\sum_{i\in N(u)\cap S(j,k)} W_{ij}R_{ui}$$

W_{ij} 是物品 i 和物品 j 之间的相似度，R_{ui} 是用户对物品 i 感兴趣，S（i,k）是

物品 i 最相似的 K 个物品集合。公式 B 的缺点：活跃用户对于物品之间相似度的影响。假如，一个非常活跃的用户一天看了 100 万的商品，其中包括物品 i 和物品 j，那么物品 i 和物品 j 就因为这个非常活跃的用户产生了相似度。学者认为活跃用户对物品相似度的贡献应该小于不活跃的用户，需要对活跃用户做软性的惩罚。提出了公式 C。

公式 C：$$W_{ij}=\frac{\sum_{u\in N(i)\cap N(j)}\frac{1}{\log 1+|N(u)|}}{\sqrt{|N(i)||N(j)|}}$$

N(u)是用户喜欢的物品集合。根据公式 C 计算的物品之间的相似度计算公式为用户进行的个性化推荐解决了热门物品和活跃用户的噪音问题。

二、基于 Mapreduce 算法开发过程分析

将 ItemCF 算法转化为一系列首尾相接的 Mapreduce 处理。同一个 Job 的 Mapper 的输出是同一个 Job 的 Reducer 的输入；前一个 Job 的 Reducer 的输出是后一个 Job 的 Mapper 的输入:

1）Mapper1: (k1, v1)→list(k2, v2); Reducer1: (k2, list(v2))→list(k3, v3);

2）Mapper2: (k3, v3)→list(k4, v4); Reducer2: (k4, list(v4))→list(k5, v5);

基于 Mapreduce 的 ItemCF 算法实现过程如下：

1. Mapreduce 计算过程的输入数据

假设有 5 个用户的浏览情况如下图（图 2）所示，需要对输入数据进行预处理。首先要计算每个用户浏览的宝贝个数，还需要求出每个宝贝被多少用户浏览过。（冗余字段的设计，可以针对不同的用户来源采用不同的计算公式，以便在 Map 阶段进行加权计算，本文中不作介绍）。Mapreduce 计算的数据输入结构如下：

用户	浏览宝贝记录		浏览宝贝个数	冗余字段
	宝贝ID	该宝贝被全网用户浏览次数		
1	a,1;b,3;d,4		3	2
2	b,3;c,3;e,2		3	2
3	c,3;d,4		2	2
4	b,3;c,3;d,4		3	2
5	e,2;d,4		2	2

图 2:ItemCF 算法输入数据

2. Map 阶段计算过程

在 Map 阶段完成对每一个用户完成$\frac{1}{\log 1+|N(u)|}$的计算。具体的计算逻辑如下图（图 3）所示。对于用户 1 来说，经过 Map 阶段的计算可以得到 W_{ab}=1.66097, W_{ad}=1.66097, W_{bd}=1.66097 的三对物品之间的相似度分值。

经过 Map 阶段通过 5 个用户的浏览行为，求出 11 对物品之间的相似度分数 W_{ij}。

用户	输入	MAP	输出
1	a,1;b,3;d,4	计算公式1/(log1+|N(u)|)	a 1 b 3 ,1.66097 a 1 d 4 ,1.66097 b 3 d 4 ,1.66097
2	b,3;c,3;e,2		b 3 c 3 ,1.66097 b 3 e 2 ,1.66097 c 3 e 2 ,1.66097
3	c,3;d,4		c 3 d 4 ,2.09590
4	b,3;c,3;d,4		b 3 c 3 ,1.66097 b 3 d 4 ,1.66097 c 3 d 4 ,1.66097
5	e,2;d,4		e 2 d 4 ,2.09590

图 3:Map 阶段计算过程

3. Reduce 阶段：计算物品之间的相似度

Reduce 阶段的输入为 Map 阶段的输出。在 Reduce 阶段完成$\frac{\sum_{u\in N(i)\cap N(j)}\frac{1}{\log 1+|N(u)|}}{\sqrt{|N(i)||N(j)|}}$的计算，求得最终的 W_{ij}。根据 5 个用户的浏览记录，最终可以得到 8 对 W_{ij} 物品相似度分数，如下图（图 4）所示。

输入	REDUCE	输出
a 1 b 3 ,1.66097	计算公式 $W_{ij}=\frac{\sum_{u\in N(i)\cap N(j)}\frac{1}{\log 1+\lvert N(u)\rvert}}{\sqrt{\lvert N(i)\rvert\lvert N(j)\rvert}}$	a b 0.9589614766158928
a 1 d 4 ,1.66097		a d 0.830485
b 3 d 4 ,1.66097		b d 0.9589614766158928
b 3 c 3 ,1.66097		b c 1.1073133333333334
b 3 e 2 ,1.66097		
c 3 e 2 ,1.66097		b e 0.6780881630117627
c 3 d 4 ,2.09590		
b 3 c 3 ,1.66097		c e 0.6780881630117627
b 3 d 4 ,1.66097		c d 1.2100684291945367
c 3 d 4 ,1.66097		
e 2 d 4 ,2.09590		e d 0.7410125513444424

图 4:Reduce 阶段计算过程

最后，根据用户的浏览行为和被浏览物品之间的相似度分数，为用户推荐可能的最感兴趣的物品。

三、ItemCF 算法的 Mapreduce 单元测试和本地集成测试

本节所介绍的 ItemCF 算法测试方法所用测试数据将沿用上一节的分析案例。通过单元测试和本地集成测试验证基于 Mapreduce 开发的 ItemCF 算法正确性。

针对 hadoop 的 Mapreduce 算法开发代码，通过集群运行结果来定位问题比较低效。当数据量大时，无法准确定位问题，调试代码比较耗时。因此，对 Mapreduce 的单元测试和 Mapreduce 的本地集成测试是非常有必要的。

MRUnit 是一款由 Couldera 公司开发的专门针对 Hadoop 中编写 MapReduce 单元测试的框架。可以用 MapDriver 单独测试 Map，用 ReduceDriver 单独测试 Reduce，用 MapReduceDriver 测试 MapReduce 作业。

根据 ItemCF 算法的 Mapreduce 计算过程，进行单元测试。从上一节案例的算法分析过程来看，要进行 MR 的单元测试，分别需要对 Map 阶段和 Reduce 阶段进行测试，以及进行 mapreduce 的集成测试。

ItemCF 算法的单元测试点如下（见图 5）：

测试阶段	测试点
Map 阶段	a) 输出 OutputCollector<Text, Text> output 返回值正常; b) 公式计算逻辑正确; c) 异常值、边界值处理逻辑校验;
Reduce 阶段	a) 输出 OutputCollector<Text, NullWritable> output 返回值正常; b) 公式计算逻辑正确; c) 异常校验，优先级比较低，从 map 阶段传入的空值等异常情况较少;
Mapreduce	a) 从本地读取文件进行计算逻辑校验;

图 5:MR 单元测试点

1. Map 阶段的单元测试

Map 阶段的单元测试通过三种方式进行测试代码设计，并给出脚本设计区别。测试数据输入为用户 1 的浏览行为，见图三。根据公式计算 Map 阶段的会产出三个输出结果，分别为：物品 a 和物品 b 的相似度，物品 a 和物品 d 的相似度，物品 b 和物品 d 的相似度。通过以下的单元测试验证逻辑正确性。

mapper 和 mapDriver 的定义如下（见图 6）：

```
//被测试map定义
private MapClass mapper;
private ReduceClass reducer;
private MapDriver<LongWritable, Text, Text, Text> mapDriver;
private ReduceDriver<Text, Text, Text, NullWritable> redDriver;
private MapReduceDriver<LongWritable, Text, Text, Text, Text, NullWritable> mapredDriver;

@SuppressWarnings({ "unchecked", "rawtypes" })
@Before
public void setUp() throws Exception {
    // Real Mapper and Reducer
    mapper = new ItemSim.MapClass();
    reducer = new ItemSim.ReduceClass();
    // Drivers for mapper and reducer
    mapDriver = new MapDriver(mapper);
    redDriver = new ReduceDriver(reducer);
    mapredDriver = new MapReduceDriver();
    mapredDriver.setMapper(mapper);
    mapredDriver.setReducer(reducer);
}
```

图 6:mapper 和 mapDriver 的定义

单元测试用例描述如下（见图 7、8）：

```
/**
 * MAPPER测试：测试正常的MAP计算逻辑（方法一）
 * 111表示被浏览的物品a;222表示被浏览的物品b;444表示被浏览的物品d;
 * \t为分隔符
 */
@Test
public void test_01_Normallogic_testMapper() {
    // MAP阶段测试数据输入，输入用户1的浏览行为，该用户浏览了3个宝贝，每个宝贝被多少用户浏览
    mapDriver.setInput(new LongWritable(10), new Text(
            "1\t111,1;222,3;444,4\t3\t2"));
    //输入期望值物品a和物品b的相似度得分为1.66097
    mapDriver.addOutput(new Text("111\u00011\u0001222\u00013"),
            new Text("1.66097"));
    mapDriver.addOutput(new Text("111\u00011\u0001444\u00014"),
            new Text("1.66097"));
    mapDriver.addOutput(new Text("222\u00013\u0001444\u00014"),
            new Text("1.66097"));
    mapDriver.runTest();
}
```

图 7:map 单元测试方法一

```
/**
 * MAPPER测试：测试正常的MAP计算逻辑（方法二）
 *
 * @throws IOException
 */

@Test
public void test_02_Normallogic_testMapper() throws IOException {
    //调用run方法获取输出结果，再跟期待结果相比较
    List<Pair<Text, Text>> out = mapDriver.withInput(
            new LongWritable(59848),
            new Text("1\t111,1;222,3;444,4\t3\t2")).run();
    List<Pair<Text, Text>> expected = new ArrayList<Pair<Text, Text>>();
    expected.add(new Pair(new Text("111\u00011\u0001222\u00013"),
            new Text("1.6609")));
    expected.add(new Pair(new Text("111\u00011\u0001444\u00014"),
            new Text("1.66097")));
    expected.add(new Pair(new Text("222\u00013\u0001444\u00014"),
            new Text("1.66097")));
    //当预期结果和实际结果不符合时的异常信息，断言结果更加明确
    assertEquals(expected, out);
```

图 8:map 单元测试方法二

以上两种测试方法，主要区别在于断言的方式，方法二更加清晰。

```
/**
 * MAPPER测试：测试正常的MAP计算逻辑（方法三）
 *
 * @throws IOException
 */
@SuppressWarnings("unchecked")
@Test
public void test_03_Normallogic_testMapper() throws IOException {
    Text value = new Text("1\t111,1;222,3\t2\t2");
    OutputCollector<Text, Text> output = mock(OutputCollector.class);
    mapper.map(null, value, output, null);
    verify(output).collect(new Text("111\u00011\u0001222\u00013"),new Text("2.09590"));

}
```

图 9:map 单元测试方法三

方法三（见图 9）采用了 mockito 框架的 mock()方法对 OutputCollector.class 进行 mock。采用 mockito 的 verify()方法验证，output.collect 被调用的参数是否正确。

2. Reduce 阶段的单元测试

Reduce 阶段的单元测试，测试数据输入是 Map 阶段的结果输出。通过上一节的算法过程分析可以看出，对于物品 b 和物品 d 在 map 阶段分别有两次输出（见图 3）。

```
/**
 * reduce测试：测试正常的reducer计算逻辑
 *
 * @throws IOException
 */
@Test
public void test_01_Normallogic_testReduce() throws IOException {
    Text key = new Text("222\u00013\u0001444\u00014");
    Iterator<Text> values = Arrays.asList(new Text("1.66097"),
            new Text("1.66097")).iterator();
    OutputCollector<Text, NullWritable> output = mock(OutputCollector.class);
    reducer.reduce(key, values, output, null);
    verify(output).collect(
            new Text("222\u0001444\u00010.9589614766158928"),
            NullWritable.get());
}
```

图 10:reduce 单元测试

通过 reduce 阶段的单元测试，验证公式计算逻辑的正确性（见图 10）。

3. Mapreduce 集成测试

通过 Map 阶段的单元测试和 Reduce 阶段的单元测试可以比较方便的排查问题和对代码进行调试。但是，单元测试无法将第二部分的 ItemCF 算法过程分析中的 5 个用户浏览行为作为测试数据统一输入。需要对 mapreduce 进行集成测试。可以通过 MapReduceDriver，将 Map 和 Reduce 连贯起来进行集成测试。

```
@Test
public void test_02_Normallogic_testMapReduce() throws IOException {
    List<Pair<Text, NullWritable>> out = mapredDriver.withInput(new LongWritable(1), new Text(
            "1\t111,1;222,3;444,4\t3\t2")).withInput(new LongWritable(1), new Text(
            "2\t222,3;333,3;555,2\t3\t2")).withInput(new LongWritable(1), new Text(
            "3\t333,3;444,4\t2\t2")).withInput(new LongWritable(1), new Text(
            "4\t222,3;333,3;444,4\t3\t2")).withInput(new LongWritable(1), new Text(
            "5\t555,2;444,4\t2\t2")).run();
    List<Pair<Text, NullWritable>> expected = new ArrayList<Pair<Text, NullWritable>>();
    expected.add(new Pair(new Text("111\u0001222\u00010.9589614766158928"),
            NullWritable.get()));
    expected.add(new Pair(new Text("111\u0001444\u00010.830485"),
            NullWritable.get()));
    expected.add(new Pair(new Text("222\u0001333\u00011.1073133333333334"),
            NullWritable.get()));
    expected.add(new Pair(new Text("222\u0001444\u00010.9589614766158928"),
            NullWritable.get()));
    expected.add(new Pair(new Text("222\u0001555\u00010.6780881630117627"),
            NullWritable.get()));
    expected.add(new Pair(new Text("333\u0001444\u00011.0845178396565607"),
            NullWritable.get()));
    expected.add(new Pair(new Text("333\u0001555\u00010.6780881630117627"),
            NullWritable.get()));
    expected.add(new Pair(new Text("444\u0001555\u00010.7410160868783483"),
            NullWritable.get()));
    assertEquals(expected, out);
}
```

图 11:Mapreduce 集成测试

通过上图可以看出（见图 11），mapreduce 集成测试的测试输入数据是通过 MapReduceDriver 传入的，当测试数据输入量较大时，上述方法无法满足期望。需要通过本地文件读取的方式传入测试数据。

例如，本地文件 a.in 为 5 个用户的浏览信息（见图 12），分别记录了 5 个用户的浏览宝贝和浏览次数，字段设计和图 3 保持一致。

```
1   111,1;222,3;444,4   3   2
2   222,3;333,3;555,2   3   2
3   333,3;444,4 2   2
4   222,3;333,3;444,4   3   2
5   555,2;444,4 2   2
```

图 12:本次测试数据

通过引入 package org.apache.hadoop.mrunit.enhance;读取本地文件，并对 mapreduce 进行集成测试（见图 13）。通过本地文件的读取，不仅可以测试数据量较大的场景，而且方便测试数据管理，提高 Mapreduce 集成测试效率。

```
@RunWith(MRUnitJunit4TestClassRunner.class)
public class ItemSimforMr_Test {

    MapDriver<LongWritable, Text, Text, Text> mapDriver;

    ReduceDriver<Text, Text, Text, NullWritable> reduceDriver;
    @MapInputSet
    @MapReduce(mapper = ItemSim.MapClass.class, reducer = ItemSim.ReduceClass.class)
    MapReduceDriver<LongWritable, Text, Text, Text, Text, NullWritable> mapredDriver;

    @Test
    @MapInputSet("a.in")
    public void test_01_Normallogic_testMapReduce() throws IOException {
        mapredDriver.withOutput(new Text("111\u0001222\u00010.9589614766158928"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("111\u0001444\u00010.830485"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("222\u0001333\u00011.1073133333333334"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("222\u0001444\u00010.9589614766158928"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("222\u0001555\u00010.6780881630117627"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("333\u0001444\u00011.0845178396565607"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("333\u0001555\u00010.6780881630117627"),
                NullWritable.get());
        mapredDriver.withOutput(new Text("444\u0001555\u00010.7410160868783483"),
                NullWritable.get());
        mapredDriver.runTest();
    }
}
```

图 13:本地文件读取的 Mapreduce 集成测试

总结

本文介绍了一种推荐算法领域较为常见的基于物品的协同过滤算法，及基于 Mapreduce 的算法分析过程，并通过 Map 阶段的单元测试、Reduce 阶段的单元测试对该算法进行正确性验证。最后，通过本地读文件的方式进行 Mapreduce 的集成测试，可以高效的验证算法的正确性。

【特刊】自动化测试框架设计实践

—实现基于 ruby 及 watir-webdriver 的自动化测试框架

作者：郝强

摘要

本文讲述了如何运用 ruby 和 watir 来实现一个可配置化的关键字驱动的 web 自动化测试框架，文中既讲述了框架实现思想又给出了具体实现，是目前为数不多讲述自动化测试框架实现的文章。

关键词

自动化测试框架，关键字驱动

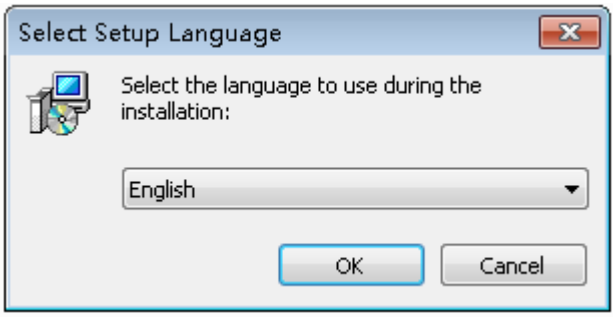
正文：

第一章、 框架开发环境安装

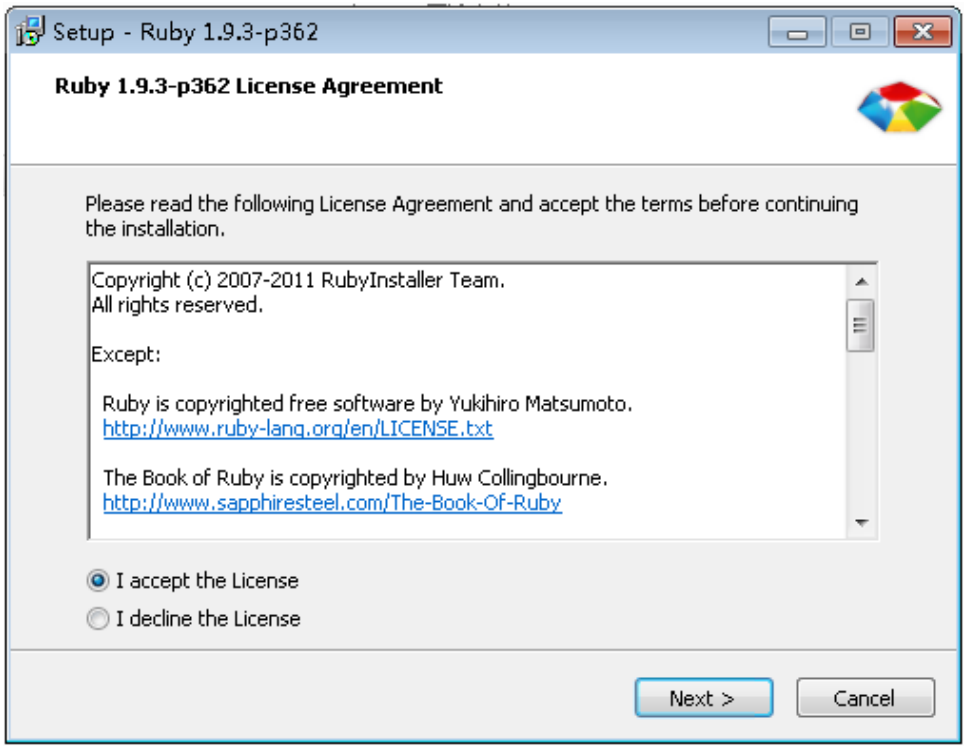
1.1 安装 Ruby 1.9.3

Ruby 1.9 的 Windows 版本安装文件位于其官方网站：http://rubyinstaller.org/downloads/，各位可以直接选择对应版本进行安装，以下是本人将相关软件下载到本地的文件目录截图。

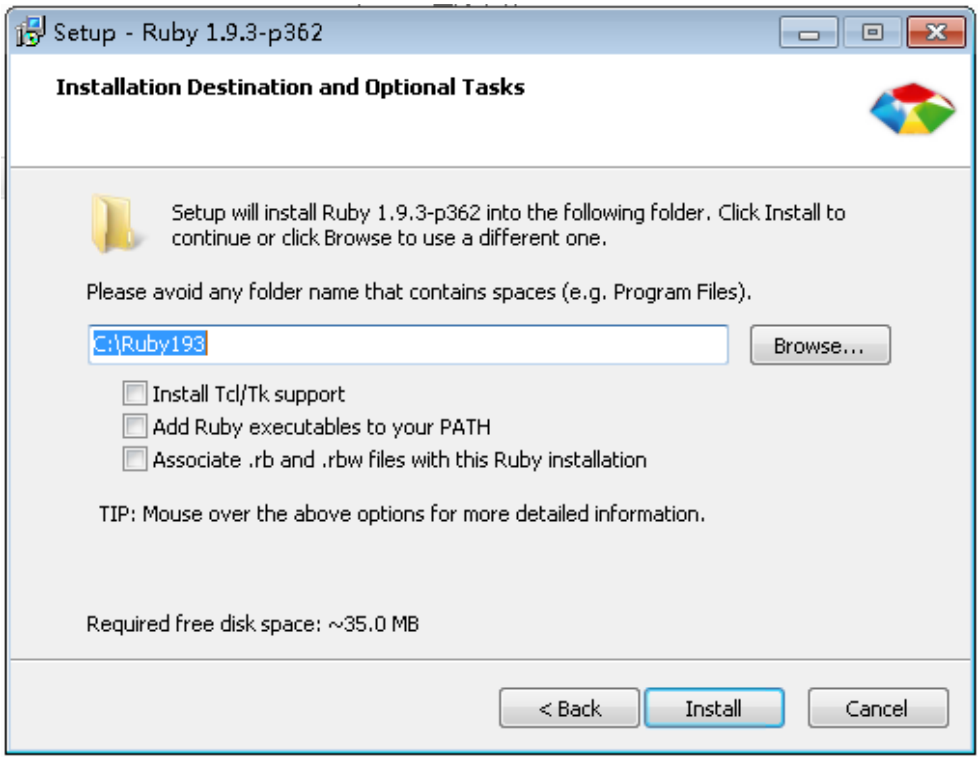
双击 rubyinstaller-1.9.3-p362.exe，弹出选择安装语言界面，如下图所示：



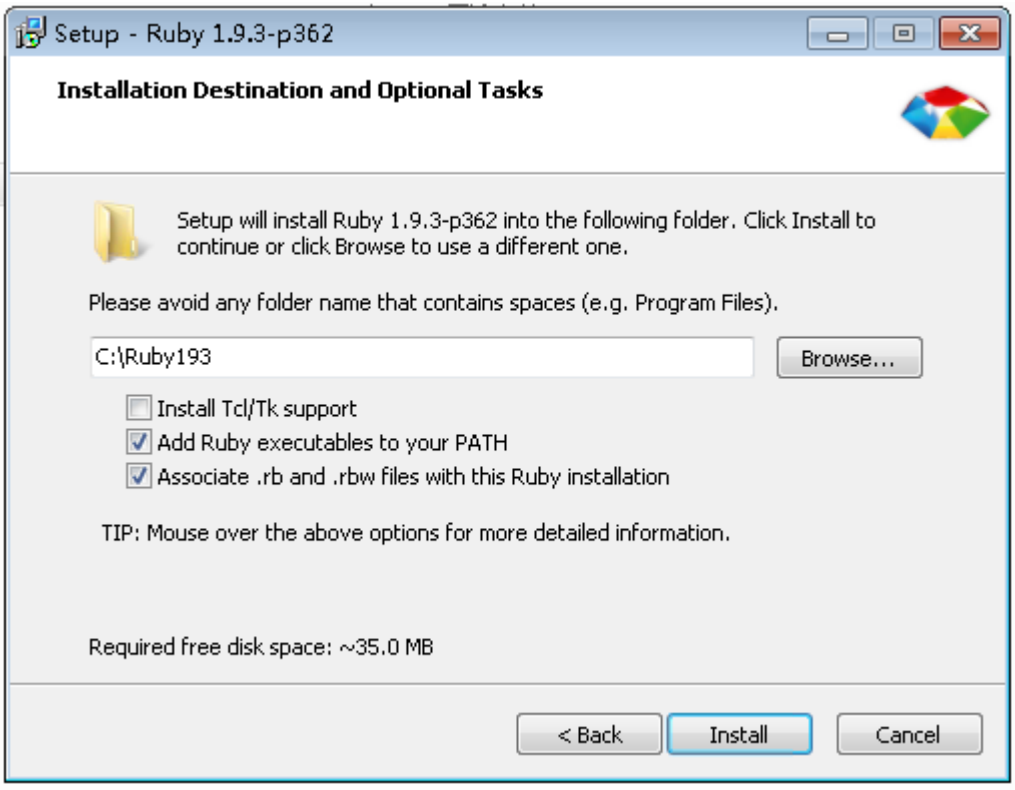
选择安装语言为英语(English)后，点击 OK 按钮，系统提示许可协议，参见下图：



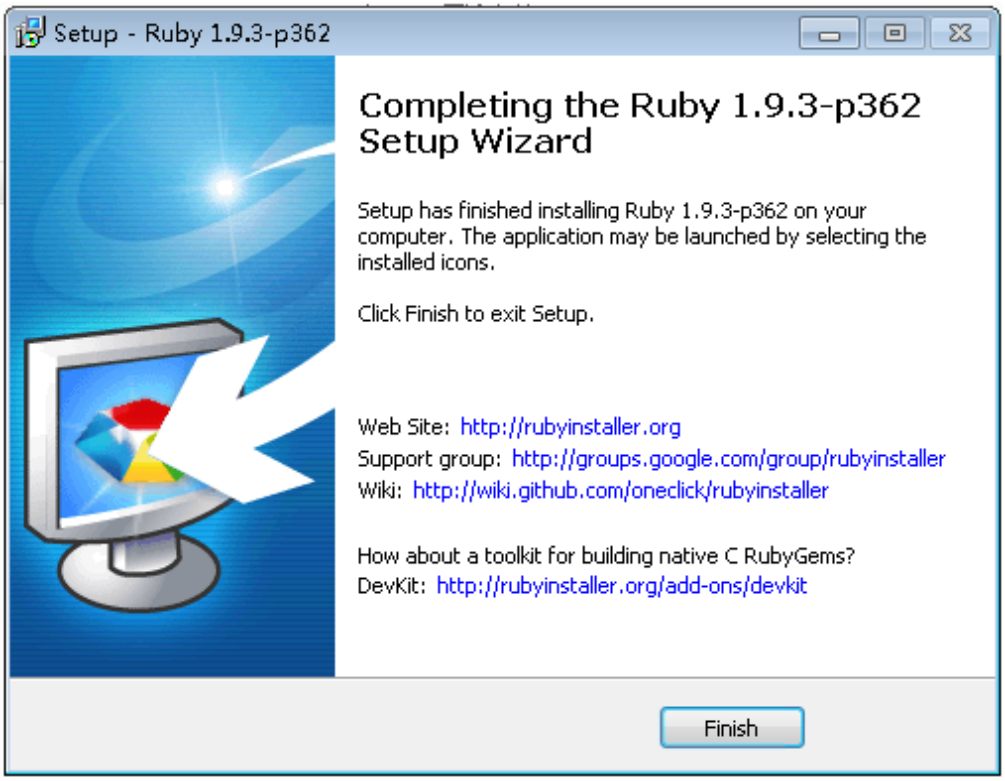
选择接受协议并点击 Next 按钮。



如上图，将安装目录设置为 c:\Ruby193 目录，并点击 Install 按钮，

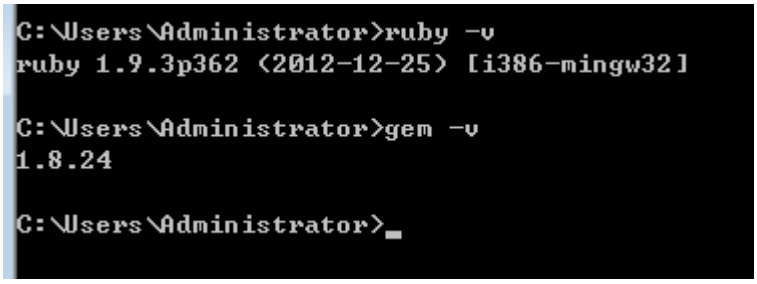


选中 Add ruby executables to your PATH 及 Associate.rb and .rbw files with this Ruby installation 并点击 Install 按钮，系统将进行安装操作，完成后如下图所示:



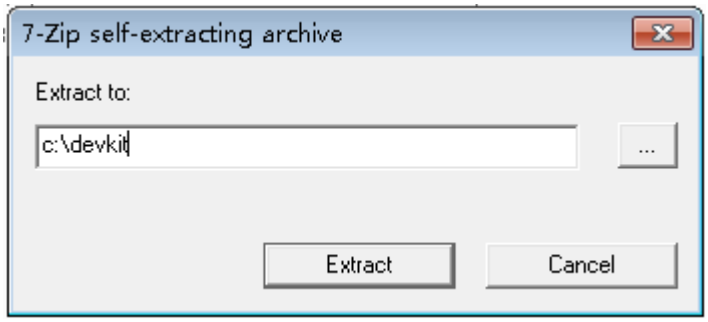
点击 Finish 完成安装 Ruby 1.9.3 操作。

安装完成后，我们测试一下安装是否正确。我们打开一个 cmd 命令窗口，输入 ruby –v 测试一下 ruby 的版本号；再输入 gem -v 测试一下 rubygems 版本号，参见下图：



1.2 安装 DevKit

DevKit 也是位于 http://rubyinstaller.org/downloads/下载页中，ruby 1.9.3 版本需要选择 DevKit-tdm-32-4.5.2 版本来安装，请下载后运行该程序。



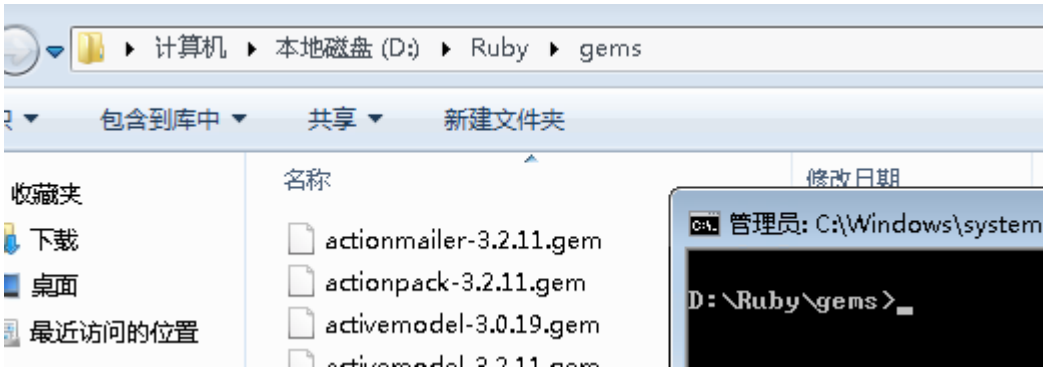
自解压程序要求我们指定一个安装，通常我们设置为 c:\devkit，之后我们要在命令窗口中运行命令安装 DevKit.

安装只需要运行两条命令即可：

首行需要进入到 c:\devkit 目录下，执行的命令是 cd c:\devkit，运行 ruby dk.rb init 进行初始化操作，之后运行 ruby dk.rb install 完成安装，具体操作过程参照下图所示：

1.3 安装 Watir

Ruby 的包通过 gem 来管理，官方网站是 http://rubygems.org/ ，大家可以在上边下载所需要的包，我们主要是要安装 waitr 及 watir-webdriver 包。本人习惯将所用到的包先下载到本地进行安装。



Ruby 用 gem 安装包非常简单，通过 gem install 命令可以直接安装。

Gem install watir --no-ri --no-rdoc

Gem install watir-webdriver --no-ri --no-rdoc

以上两条命令可以完成 watir 及 watir-webdriver 的安装操作

```
D:\Ruby\gems>gem install watir --no-ri --no-rdoc
Fetching: ffi-1.3.1-x86-mingw32.gem (100%)
Fetching: childprocess-0.3.7.gem (100%)
Fetching: websocket-1.0.7.gem (100%)
Fetching: selenium-webdriver-2.29.0.gem (100%)
Successfully installed multi_json-1.5.0
Successfully installed rubyzip-0.9.9
Successfully installed ffi-1.3.1-x86-mingw32
Successfully installed childprocess-0.3.7
Successfully installed websocket-1.0.7
Successfully installed selenium-webdriver-2.29.0
Successfully installed watir-webdriver-0.6.2
7 gems installed

D:\Ruby\gems>gem install watir-classic-3.4.0.gem --no-ri --no-rdoc
Fetching: win32-api-1.4.8-x86-mingw32.gem (100%)
Fetching: nokogiri-1.5.6-x86-mingw32.gem (100%)
Fetching: hoe-3.5.0.gem (100%)
Successfully installed win32-process-0.7.1
Successfully installed win32-api-1.4.8-x86-mingw32
Successfully installed windows-api-0.4.2
Successfully installed windows-pr-1.2.2
Successfully installed nokogiri-1.5.6-x86-mingw32
Successfully installed rautomation-0.7.3
Successfully installed xml-simple-1.1.2
Successfully installed hoe-3.5.0
Successfully installed s4t-utils-1.0.4
Successfully installed builder-3.1.4
Successfully installed user-choices-1.1.6.1
Successfully installed subexec-0.0.4
Successfully installed mini_magick-3.2.1
Successfully installed win32screenshot-1.0.7
Successfully installed watir-classic-3.4.0
15 gems installed
```

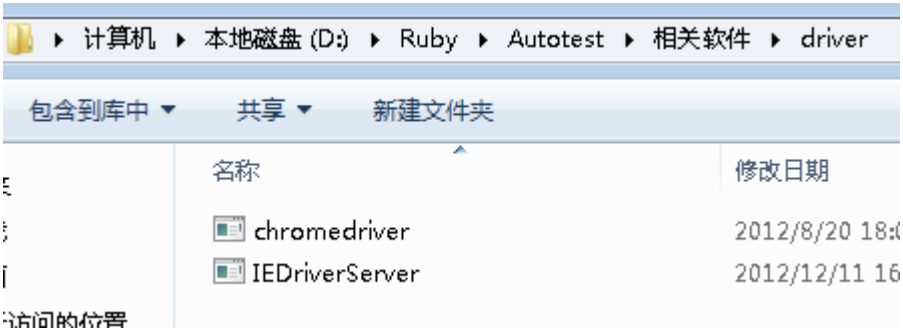
1.4 安装 IE 及 chrome 驱动

IEDriver 驱动地址：http://code.google.com/p/selenium/downloads/list

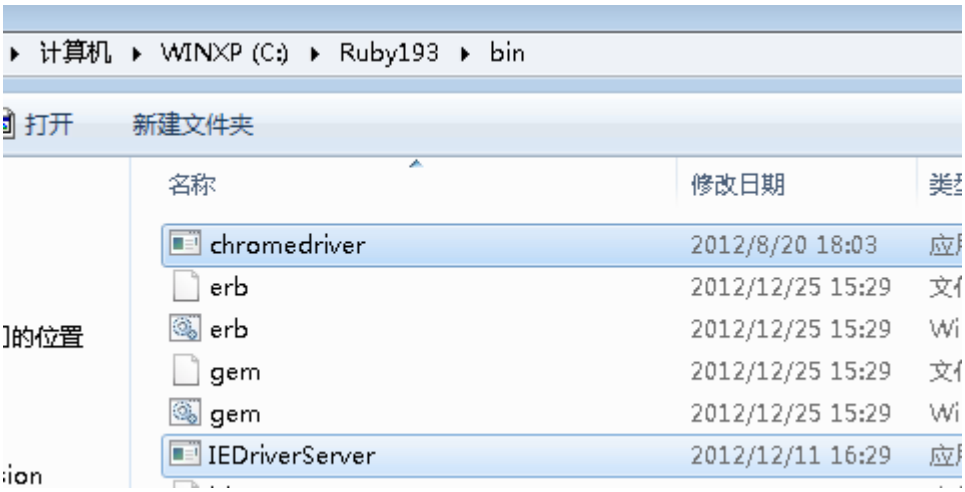
ChromeDriver 驱动地址：

http://code.google.com/p/chromedriver/downloads/list

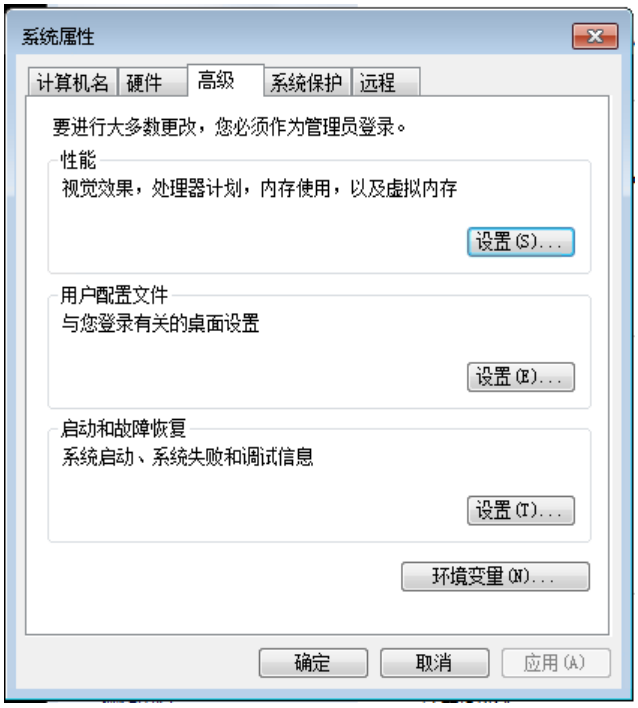
请先将两个 driver 文件下载到本地。



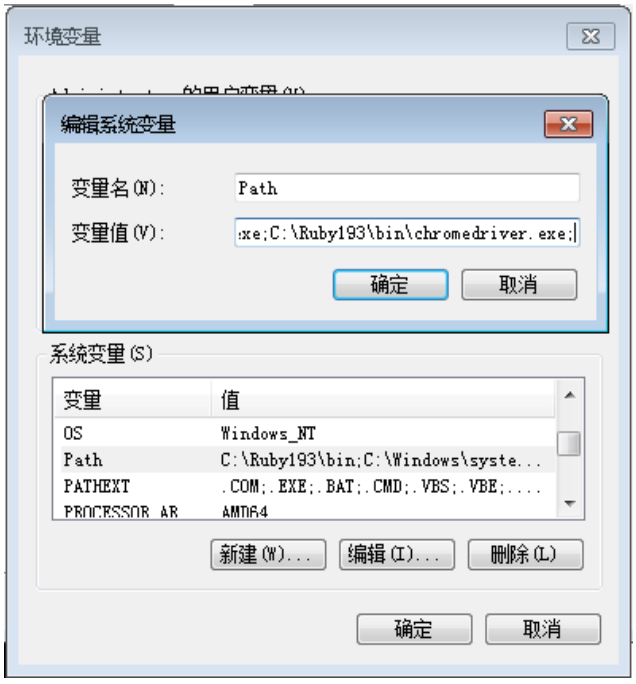
将 IEDriverServer.exe 及 Chromedriver.exe 复制到 ruby 的 bin 目录下，即 c:\Ruby193\bin 下



之后我们需要将两个 driver 文件加到系统的 path 路径中，在 windows 的系统属性里选择环境变量选项。



在 path 中添加执行文件路径。



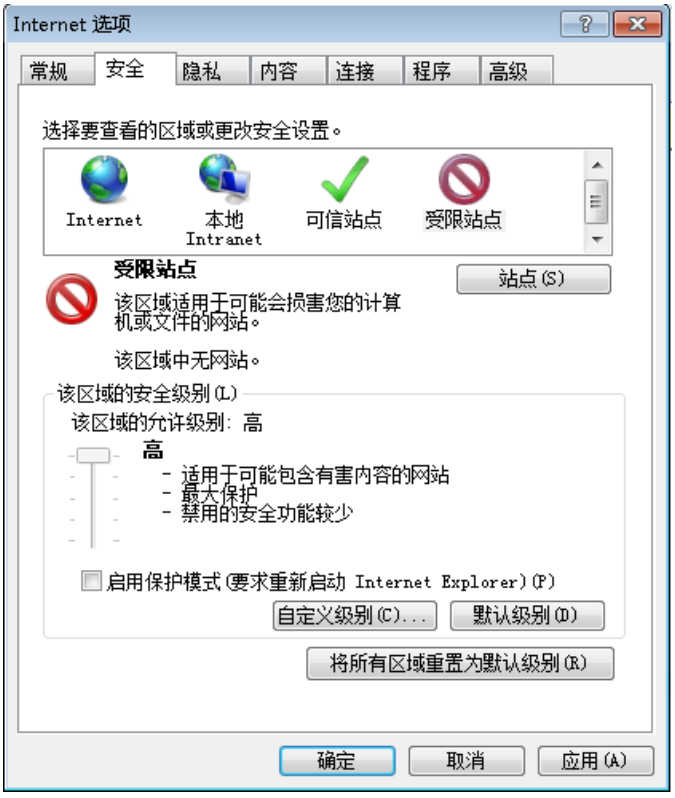
点击“确定”后环境变量添加完成。

之后我们需要在 IE 的选项里全部取消选中保护模式或全部选中保护模式以保证 driver 生效。这里以全部取消选中保护模式为例。

如上图所示，在 Internet 上取消选择“启用保护模式”。

在本地 Intranet 上也取消选择“启用保护模式”。

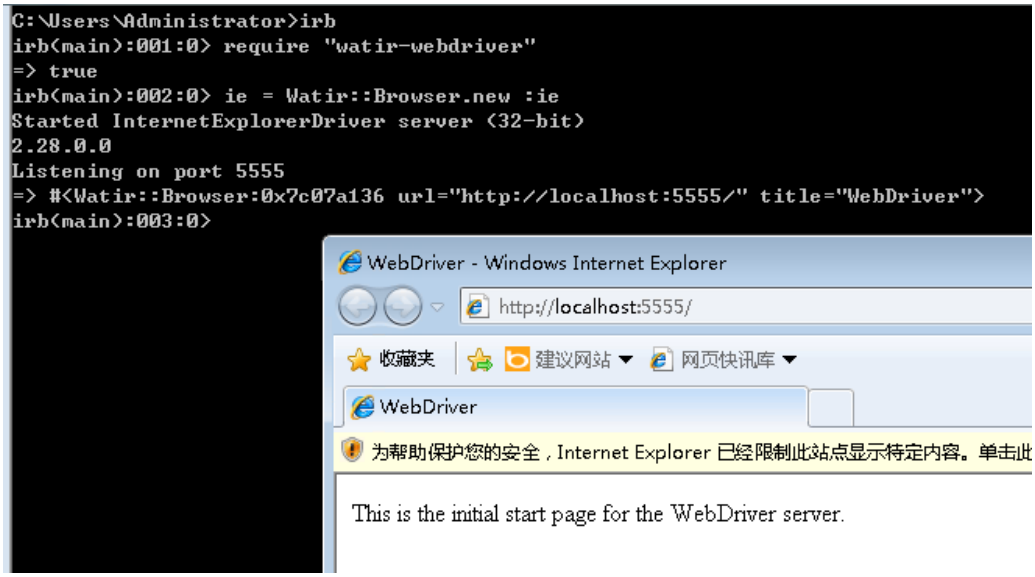
在可信站点上也取消选择“启用保护模式”。



在受限站点上也取消选择“启用保护模式”。下面我们来测试一下 IEdriver 是否生效。

我们在命令行运行 irb 命令。输入 require “watir-webdriver”, 引入 watir-webdriver 包。

再输入 ie = Watir::Browser.new :ie 来启动一个 IE 实例。



如上图所示，如果你看到一个 IE 窗口被打开，就表示配置正常了。

至此，ruby、Watir 以及 IE 和 Chrome 的基本配置工作完成。

第二章 全面了解 Waitr

Waitr 是 Web Application Testing in Ruby 的缩写。在 Watir 官方网站有一句广告语，是“Automated testing that doesn’t hurt”，翻译过来就是自动化测试不再受伤。

2.1 Watir 是什么？

Watir 是用于自动化 WEB 浏览器的一组开源（BSD）的 Ruby 代码库，发音等同于英语中的 Water。Watir 目前可以驱动微软的 IE，Mozilla 的 Firefox，苹果的 Safari，Google 的 Chrome 以及 Opera，它目前支持主流的操作系统，包括微软 Windows，苹果的 Mac OS X 以及 Linux。

2.2 Watir 能做什么？

简单来讲，Watir 几乎可以做任何手工与浏览器打交道的动作。例如：打开一个页面，点击一个链接或按钮，在输入框中输入内容等等。除此外，它也能够

检查在浏览器中打开的页面，例如：列出当前页的所有链接，复选框是否选中，单选钮是否显示等等。了解到这些，我相信，聪明的读者已经知道我们可以利用 Watir 做一些软件测试需要做的事情了。

2.3 Watir 不能做什么？

非常重要的一点是，虽然 Watir 能够驱动浏览器，但它不能够控制浏览器插件，像 Flash，Java applet 以及微软的 Silverlight。

2.4 Watir 对 HTML 元素的支持

不同的 Watir Gem 包对 HTML 元素的支持也不尽相同。Watir-webdriver 的 Gem 包支持所有的 HTML 元素。Watir Gem 包支持大多数公共的 HTML 元素，但它比较容易扩展成为支持更多的元素。SafariWatir 对于大多数 HTML 元素都不支持，它只支持最常用的那些。

网站 http://wiki.openqa.org/display/WTR/HTML+Elements+Supported+by+Watir 列出了 Watir-Webdriver 对 HTML 元素支持的表格。

Watir method	HTML tag	:action	:after?	:alt	:class	:css	:for	:href	:html	:id	:index	:method	:name	:src	:text	:title	:url	:value	:xpath	Multiple Attributes
area	<area>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
button	<button> <input type="button"> <input type="image"> <input type="reset"> <input type="submit">	✗	?	✓	✓		✗	✗	?	✓	✓	✗	✓	✓ he input type is image	✓	✗		✓	✓	✓
cell	<td>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
checkbox	<input type="checkbox">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
dd	<dd>																			
div		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
dl	<dl>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
dt	<dt>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
em		?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
file_field	<input type="file">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✗	✓		✓	✓	✓
form	<form>	✓	?	?	✓		✗	✗	?	✓	✓	✓	✓	✗	✗	✗		✗	✓	✓
frame	<frame> <iframe>	✗	?	?	✗		✗	✗	?	✓	✓	✗	✓	✓	✓	✗		✗	✗	✓
h1 h2 h3 h4 h5 h6	<h1> <h2> <h3> <h4> <h5> <h6>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
hidden	<input type="hidden">	✗	?	?	✓		✗	✗	?	✓	✓	✓	✓	✗	✓	✗		✓	✓	✓
image	<img>	✗	?	✓	✓		✓	✗	?	✓	✓	✗	✓	✓	✓	✗		✓	✓	✓
label	<label>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
li		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
link	<a>	✗	✓	?	✓		✗	✓	✓	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
map	<map>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
p	<p>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
pre	<pre>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
radio	<input type="radio">	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
row	<tr>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
select_list	<select>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
span		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
strong		?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
table	<table>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
tbody	<tbody>	?	?	?	?		?	?	?	?	?	?	?	?	?	?		?	?	?
text_field	<input type="password"> <input type="text"> <textarea>	✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✓	✓	✓
ul		✗	?	?	✓		✗	✗	?	✓	✓	✗	✓	✗	✓	✗		✗	✓	✓
New Browser Windows	n/a	✗	?	?	✗		✗	✓	?	✗	✗	✗	✗	✗	✗	✓		✗	✗	?

2.5 Watir-WebDriver 介绍

Watir-WebDriver 目前支持四种浏览器：分别是 IE，Chrome，FireFox，Safari，同时它也对移动设备的测试提供了支持。

由于 Watir-WebDriver 对所有 Html 元素支持，并且可以同时支持四个浏览器，所以理所当然当你要实现一个测试框架时，Watir-WebDriver 就成为了首选。

参见如下代码可以启动 Chrome 浏览器:

```
require "watir-webdriver"
b = Watir::Browser.new :chrome
```

如果要启动其它浏览器，参见如下代码：

```
ie = Watir::Browser.new :ie
ff = Watir::Browser.new :firefox
sf = Watir::Browser.new :safari
```

2.6 Watir-WebDriver 元素识别

所有的 html 元素都被 watir-webdriver 支持，这里我们准备一些例子供大家参考：

2.6.1 Text Fields 输入框

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
t = b.text_field :id => 'kw'
p t.exists?
t.set '自动化测试'
p t.value
```

2.6.2 Buttons 按钮

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
t = b.text_field :id => 'kw'
t.set '自动化测试'
btn = b.button :value, '百度一下'
p btn.exists?
```

2.6.3 Links 超链接

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
l = b.link :text => '百科'
p l.exists?
l.click
```

2.6.4 Divs & Spans

```
require 'watir-webdriver'
b = Watir::Browser.start 'www.baidu.com'
d = b.div :id => 'content'
p d.exists?
p d.text
s = b.span :class => ' s_btn_wr'
p s.exists?
p s.text
```

2.6.5 处理等待

当你访问动态 web 界面时等待通常是不确定的，特别是拥有大量 AJAX 异步请求的站点。通过情况下大家会用 sleep 进行延时等待，但在不同的网络环境 sleep 的时间是不确定的。

Explicit waits 显式等待

Watir::Wait.until { ... }：等待直到块即{}内的内容为真

object.when_present.set：对象出现再做动作

object.wait_until_present：一直等待到对象出现

object.wait_while_present：一直等待到对象消失

Implicit waits 隐式等待

隐式等待用于设置 Watir-WebDriver 最大的超时时间，以秒为单位。

```
require 'watir-webdriver'
b = Watir::Browser.new
b.driver.manage.timeouts.implicit_wait = 3  #3 秒
```

2.7 Watir-WebDriver 高级交互技术

Watir-WebDriver 高级交互技术主要用于处理浏览器认证，证书，下载，代理，弹出窗口，javascript 对话框等，这里会分别讲解相关用法。

Basic Browser Authentication

浏览器的 http basic authenticaio 认证方式，watir 处理起来是非常容易，只需要在 url 中将相关参数传送过去即可以完成认证。

```
require 'watir-webdriver'
b = Watir::Browser.start 'http://admin:password@TestSolution.cn'
```

处理浏览器下载

自动处理下载并保存到某个目录 。

Firefox

```
download_directory = "#{Dir.pwd}/downloads"
download_directory.gsub!("/", "\\") if Selenium::WebDriver::Platform.windows?
profile = Selenium::WebDriver::Firefox::Profile.new
profile['browser.download.folderList'] = 2 # custom location
profile['browser.download.dir'] = download_directory
profile['browser.helperApps.neverAsk.saveToDisk'] = "text/csv,application/pdf"
b = Watir::Browser.new :firefox, :profile => profile
```

Chrome

```
download_directory = "#{Dir.pwd}/downloads"
download_directory.gsub!("/", "\\") if    Selenium::WebDriver::Platform.windows?

profile = Selenium::WebDriver::Chrome::Profile.new
profile['download.prompt_for_download'] = false
profile['download.default_directory'] = download_directory
b = Watir::Browser.new :chrome, :profile => profile
```

处理 JAVAScript 对话框

JavaScript Alerts

检查 Alert 是事被显示

browser.alert.exists?

获得 Alert 的文件内容

browser.alert.text

关闭 alert

browser.alert.ok

browser.alert.close

JavaScript Confirms

点击确定

browser.alert.ok

点击取消

```
browser.alert.close
JavaScript Prompt
# 输入文本
browser.alert.set "Prompt answer"
# 点击确定
browser.alert.ok
# 点击取消
browser.alert.close
```

后续内容请访问 51Testing 官网，《自动化测试框架设计实践》特刊>>

文章详细目录：

测试工程执行时间优化

作者：秦广

sell-test 是 sell 应用的 webx 层接口测试工程，有 1600 多个 case，执行时间约为 40min，最近对这个工程作了一些运行时间调优，成果显著，现在执行时间约 17min。

优化主要方法就是提取数据，分析数据，改进耗时部分、减少重复工作。

1．分析 maven 输出的统计信息

```
>Running
com.taobao.qatest.editItem.csellerEditItem.checkSpuEditItem.CCheckSpuInfoItemTest
>Tests run: 1, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 1.384 sec <<< FAILURE!
>Running com.taobao.qatest.editItem.csellerEditItem.checkSpu.CheckSpuTest
>Tests run: 19, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 17.951 sec
>Running
com.taobao.qatest.editItem.csellerEditItem.checkSkuSort.CheckSkuPropertySortTest
>Tests run: 4, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 3.305 sec <<< FAILURE!
```

maven 用来执行测试的 surfire 插件会输出如下的基本统计信息，

信息非常有限，只能提取到每个类执行的时间，类里面的@Test 执行个数。

从以上信息里，我们可以分析出每个类执行的时间，每个类里面 case 的平均时间，绝大部分的 case 执行时间都在 0-3s，有一个类：com.taobao.qatest.editItem.csellerEditItem.checkChangeItemProperty.CSellerChangeItemPropertyTest 的执行时间特别长，总时间 340s，平均每个 case 执行时间 85s。

CSellerChangeItemPropertyTest 类 5 个 case 耗时接近 7 分钟，重点解决。接下来 review 这个类，没有发现问题，只好在每个@Test 方法开始和最后都加上时间统计，输出@Test 方法本身的执行时间。执行过后发现@Test 耗时约 200ms，本身并不慢。

> （ps：这里需要补充一下每个 case 执行的过程。

>先是 ITest 的各种注解(可以查看 AbstractITestWebBaseCase 类上的注解)对应的 Listener 执行，然后 Junit 的非静态注解(@Before, @Test, @After)执行，

>总体顺序是： @ITestxx --> @ITestxx --> @ITestxx --> @Before --> @Test -- @After

>@ITestxx 各注解完成自己的职责， @Before 完成一些初始化工作，@Test 是测试用例本身，@After 完成数据清理工作

>假设一个类是这么写的：

> @ITestxx

> @ITestxx2

> @ITestxx3

> public class TitleTest() {

> @Before public void before_TitleTest() {}

> @After public void after_TitleTest() {}

> @Test public void test() {}

> }

>执行顺序将是 ITestxxListener#xx(), ITestxx2Listener#xx(), ITestxx3Listener#xx(), TitleTest#before_TitleTest(),TitleTest#test(),TitleTest#after_TitleTest()

>后文中，将称所有@ITestxx 注解的执行时间称为**ITest 时间**，@Before-->@Test-->@After 的时间称为**Junit 时间**

>）

再次 review @After 方法的数据清理代码，依然没发现问题。（其实这里应该能发现，这个类测试的都是宝贝上 sku 数量 600 个的极端场景，在@After 清理数据的时候，删除宝贝和 sku 数据，sku 数量比较大应该能看出来，不过没想到 sku 删除的方法招执行的这么慢）

2. btrace 提取耗时用例的执行情况

静态代码 review 没有结果，就只能提取运行时数据了，写了一个简单的 btrace 脚本用来提取每个方法（com.taobao.*包）的耗时，单独运行：CSellerChangeItemPropertyTest 这个类并提取数据。

取耗时 top 前 10w，发现 SkuDB#queryCountBySkuId()的次数非常多（9.7w），每次约 20ms。

>（ps：

>SkuDB 是 auction-db.jar 提供的工具类，auction-db 为了屏蔽商品数据库分

库分表的细节，封装了基本的删、改、查操作，提供透明的数据访问方法，可以"简单"理解成 mini TDDL。）

* 分析： SkuDB#deleteSkuByItemId()方法实现的不好，我们都是逻辑删除（status = -1），当初写的时候为了省事，查询出所有 sku 再遍历调用 update（因为先实现 query, update 再实现 delete，图省事少写几行代码@_@），而 update 为了通用性会先查一把数据是否存在，也就是调用一次 queryCountBySkuId()。由于这个设计的存在，所有的操作（删、改、查）最终都会调用一次 queryCountBySkuId()，造成了 query 方法调用次数膨胀。

执行时间 20ms 也超出预期，review queryCountBySkuId()方法，发现拼出的 sql 里没有带上 item_id，没用到索引。

* 解决方案：调整 deleteSkuByItemId()方法的实现，直接拼 sql 更新 status 字段，缩短调用链；

修改 sku、auction_image 这 2 张大表的所有 sql，利用索引提升性能；

* 收益： CSellerChangeItemPropertyTest 执行时间从 340s 降到 10s，提升约 6min；

3．增加日志打点，提取更多数据

从现有日志数据中发现的问题已经解决，添加更丰富的日志，输出@Before-->@Test-->@After 的时间，@Test 的时间，由于我一直怀疑数据清理耗时太多，顺带也打出了数据清理的时间，见下

```
>Running com.taobao.qatest.editItem.csellerEditItem.checkPriceLimit.CSellerCheckItemPriceTest
>time:420|dataCleanTime:65|count:44|now:16:40:32
>time:557|dataCleanTime:84|count:45|now:16:40:32
>Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 1.335 sec
>Running com.taobao.qatest.editItem.csellerEditItem.checkPriceLimit.CheckPriceForWujiajuTest
>time:286|dataCleanTime:57|count:46|now:16:40:33
>time:270|dataCleanTime:41|count:47|now:16:40:33
>time:266|dataCleanTime:43|count:48|now:16:40:34
>Tests run: 7, Failures: 0, Errors: 0, Skipped: 4, Time elapsed: 1.472 sec
>Running com.taobao.qatest.editItem.csellerEditItem.checkPriceLimit.CheckPriceLimitTest
>time:451|dataCleanTime:42|count:49|now:16:40:34
>Tests run: 3, Failures: 0, Errors: 0, Skipped: 2, Time elapsed: 0.567 sec
```

maven surfire 输出了每个类执行的总时间，日志输出了 Junit 所有注解的执行时间，因此可以计算出@ITestxx 各种注解的执行时间，分析一次 30min 的日志（实际 test 时间为 27min），Junit 时间为 10min，ITest 时间为 17min。

数据清理时间总共才 1min（含在 Junit 时间内），日志提取的数据告诉我，数据清理速度很快，不需要再继续优化了。

>（ps：这里的运行时间指的是一次构建所用的全部时间，从调度开始到运行日志回写结束，调度开始、启动脚本、更新代码、回写日志时间约 1min，由于 sell-test 测试的是 web 工程，还有一个打包过程，约 2min，因此有约 3min 的时间差）

4．btrace 提取全局执行情况

日志数据表明数据清理不需要优化，itest 时间最耗时，考虑优化这方面。

review itest 各个注解对应 Listener 的实现，发现每个 case 执行的时候都有很多重复，有优化空间。

使用 btrace 提取全局数据，输出 com.taobao.*包下所有执行时间超过 20ms 的方法，依然取 top 10w 分析，收获很丰富。

* @ItestDataSet 注解对应的实现类 ITestDataSetListener# beforeTestMethod() 调用链中需要获取数据表的元数据，

com.taobao.itest.dbunit.operation.AbstractOperation#getOperationMetaData 这个方法执行非常耗时，20ms -- 700ms 不等，数量也很可观（后期统计总执行次数约 1.4w，总共 182 个 excel，执行次数略有点多@_@）；

>（ps：@ITestDataSet 的作用是数据准备，它会把注解中标识的 excel 中的数据根据配置写入数据库，确保在 case 执行中数据与预期一致）

* @ITestWebContext 注解对应的实现类 ITestWebContextListener#prepareTestInstance() 会处理 web 工程的总控文件 web.xml，去掉其中关于安全的 filter 节点，每个 case 都会执行一次，而事实上一个 web 工程只有一个 web.xml，处理一次即可；该方法耗时约 400-600ms；

>（ps：@ITestWebContext 的作用是 web 测试工程入口，它指明被测试工程的 web.xml 的位置，根据 web.xml 启动 web 容器）

*@ITestClassLoader 注解对应的实现类：ITestClassLoaderListener#prepareTestInstance() 会重新 build classpath，每个 case 都会执行一次，其实针对一个@ITestClassLoader 注解执行一次就行了；

>（ps：@ITestClassLoader 的作用是运行时修改 classpath，用来解决 jar 包易冲突的问题，把开发工程和 WEB-INF/lib 目录下的 jar 包放到最前面，尽力确保被测工程的真实性，同时也可以排除部分确认有冲突的包）

5. 改进 itest 框架减少重复次数

针对上面的 3 个注解，每个都添加缓存或是判重，减少重复次数。

* ITestDataSetListener 这里获取元数据的地方，作一个缓存，最直观的缓存方案是 "连接 url + username + tablename"作为 key 存储，考虑到淘宝网特有的分库分表模式，还可以进一步优化，例如 icdb 的数据库连接：jdbc:mysql://10.232.31.67:3306/icdb0 ，16 个库只有最后面的数字序号在变化，表名 auction_auctions_0000 ~ auction_auctions_1024，也只有最后面的数字序号在变化，使用"连接 url 去掉最后数字+username+tablename 去掉最后数字"作为 key 缓存；

后期提取到的数据，去重以后只用到了 17 张表，而总调用次数是 1.4w，优化效果很显著；

* ITestWebContextListener 这里其实有一个启动 spring 容器的操作，耗时约 90s，作了判重，处理 web.xml 的地方，以文件绝对路径为 key，也添加一个判重处理，从 1600 次（case 数量）优化到 1 次；

* ITestClassLoaderListener 这里重复 build classpath，判重的 key 不太好找，最好是能用这个注解所在类的类名，但是由于取注解的过程封装了几层，不太好

拿，于是用这个注解最核心的排除包的字面值作为 key 判重，从 1600 次（case 数量）优化到 1 次。

例如：@ITestClassLoader(testSpecExcludedLibNamas = { "dbunit-2.1.jar", "log4j-1.2.14.jar", "log4j-1.2.13.jar", }, loadQuietly = true) 这个注解的 key 就是 "[dbunit-2.1.jar, log4j-1.2.14.jar, log4j-1.2.13.jar]"；

* 完成以上改造后，执行时间缩短到约 17min

6. 未来可改进的空间

* 分析一次 17min 的日志（test 时间 14min），junit 时间 10min（其中数据清理 1min），itest 时间 4min

junit 时间 10min，还有部分 case 运行时间超过 3s，可以考虑优化，有一定成本

@ITest 时间 4min，还有部分可缓存的 Listener，但是这部分 Listener 的重复代码本身执行时间不长，在数据中不够显眼，成本不高，价值也不大

数据清理，目前看起来不必要，但是这部分受环境影响较大，依赖的基础应用慢的情况下会放大影响，可优化

kelude 调用本身占去 2-3min 的时间，也不可能优化了

以上所有的优化，可能最多还能降 2-3min，总 14min 是极限

在 case 数量不变的情况下，很难再压缩执行时间，不过我们也可以考虑给用例分优先级，在需要的时候，只执行 p1、p2 的用例，这样也可以缩短运行时间。

ps：还可以考虑多线程、分布式调度，不过 sell 业务复杂，这 2 种模式都会引入系统的复杂性，需要酌情考虑。

Android Crash 问题定位与实践

作者：李红玲 王晓芹

摘要: 本文介绍了如何在 Android 手机发生 Crash 时进行 Log 分析的方法，它可以帮助测试人员快速定位 Android 手机 Crash 发生的原因，同时给研发人员提供有效修改 Bug 的 Log 信息。

关键词：Android；Crash；Log；Bug

引言

用自动化测试工具对 Android 手机进行压力测试和稳定性测试，往往通过模拟实际使用场景中所发生的业务量来测试手机性能是否满足要求，测试过程中系统状态信息被实时记录到 Log 文件中，通过对 Log 信息的分析，可以检查到错误发生的根源和运行痕迹，因此，有效利用 Log 信息并对其进行分析与实时的监控管理，对于分析 Android 手机发生 Crash 的原因具有极为重要的作用。

自动化测试工具在运行过程中会产生多种类型的 Log 信息，而这些 Log 信息又往往分布在不同的目录中，在超大规模的 Log 数据上做分析不是一件容易的事情，需要测试人员分析大量的脚本文件和代码去寻找自己所关心的答案。基于以上问题，本文介绍了如何进行 Log 分析的方法，有了这些方法会让 Crash 的问题定位变得简单，并让复杂的分析变得可行。

1. Android Log 文件类型

由于 Android 上的应用程序千差万别，出现的问题也不尽相同。不过 Bug 类型还是有规律可循的，可以根据生成的 Log 文件找到相应的错误，通常错误信息里记录了错误的大致位置，据此可以捕获到问题的关键信息。

Log 文件记录着每次操作的信息，在出现问题后可以借助 log 信息分析以达到解决问题的目的，Log 文件类型主要分为以下几种：

(1) Logcat: Main 缓存日志，通过运行 logcat 命令，可以获得系统中使用的标记和优先级的列表，也可以加上过滤器进行表达式限制，只输出测试人员及研发人员感兴趣的标记-优先级组合。

Logcat-Kernel: Kernel(Linux)信息，包括多个进程并发信息，进程所使用的内存情况，进程访问磁盘的请求等信息。

Logcat-Radio: Radio And Telephoney 信息，面向手机基本信息提供的 api，包括网络类型、连接状态、电话功能、电话号码字符串处理的实用程序等。

Logcat-Event: 系统级别的 Events，比如：垃圾回收，Activity 管理状态，系统的 Watchdogs 和其他底层 Activity；

(2) Bugreport: Java 应用程序 Crash 时会产生一个 Bugreport 文件，该文件主要包括三个方面的内容：

Dumpstate：内存信息，Cpu 信息，Procrank 信息，系统日志，Vm Trace 信息等。

Build.Prop：当前版本、当前命令、显示系统 Build 的一些属性等；

Dumpsys：Dump Of Service Meminfo(显示某个进程更详细的内存消耗情况以及 Native And Java (Dalvik)堆栈的统计数) ；

(3) Crashdump: 每次 Crash 都会产生一个 Crashdump 文件，文件包括主日志，Java 堆栈信息，本地调用堆栈，虚拟机/进程堆，Log 缓存，内存信息，进程列表，Modem 信息，Adb Log 等信息；

(4) Bratlog: 测试用例及详细信息；

(5) Logalong: 事件，如手机通讯功能信息等；

(6) Pullfs: Traces(Java 堆栈信息)；

(7) Procrank: Uss(Unique Set Size) 值，进程独自占用的物理内存。

Android Crash 类型

程序员开发 android 应用程序时追逐的最终目标是要尽量避免程序 Crash 的发生。但是现实的情况是，在进行轮番测试和验证的时候，Android 应用程序几乎不可能完全杜绝 Crash 的发生。Crash 的类型主要有以下几种：

(1) Java Application Crash：不完善的 Java 应用程序很容易导致 Crash 的发生如：某个应用程序由于源代码不完善，会做一些它本不应该做的操作而导致 Crash 的发生。 另外，Java 虚拟机自身的 Bug、系统的库文件、Api、第三方的库文件、系统资源的短缺都有可能造成 Java Crash 的发生。

(2) System Crash：当程序试图访问不被允许访问的内存区域、指针越界、错误的访问类型、访问不存在的内存、访问不属于进程地址空间的内存、栈溢出、函数非法跳转、非法的系统调用、数据中断等操作容易引起 System Crash。

(3) Modem Crash：当手机出现无信号、死机、网络中断等现象，通常会出现 Modem Crash。

(4) Kernel Crash：由于手机应用软件出现错误而导致系统崩溃的时候，会提示 Kernel Crash 的信息，并且错误发生时 kernel 的存储图像会保存起来，当系统重新启动后，会恢复 kernel 的存储图像，然后根据现象判断是哪类错误发生。

(5) Watch Dog Crash：当手机发生 Crash 时系统停止运行，说明预先设定的 Watch Dog 发生了 Crash。

Android Crash 问题定位方法

3.1 Java Application Crash

3.1.1 Java Anr Crash

手机应用程序在预先设定的时间内不响应或响应不够灵敏，系统会向用户显示一个对话框，这个对话框被称作 anr(Application No Response) Crash。这一般都是由于应用程序错误导致的，测试人员可以在 Logcat 文件中定位是哪个进程导致 anr 的发生，为什么会发生 anr，发生 anr 之前的历史日志，事件响应间隔中 Cpu 的使用状态等。

当出现 Anr Crash 后，测试人员可以在 Log 文件里查找关键字：Anr，定位到的相应的关键信息，然后根据定位的具体信息再进行分析。下述示例说明了造成了 anr Crash 的三种情况：

⑴ **Keydispatchingtimedout:**测试人员在 Log 文件里查找关键字：Anr，定位到的关键信息如下所示：

```
11-02 13:00:45.878  1200  1211 I Activitymanager: Anr In Process: Com.*******.Camera (Com. *******.Camera/Com. *******.Camera.Video.Videocamera)
11-02 13:00:45.878  1200  1211 I Activitymanager: Annotation: Keydispatchingtimedout
11-02 13:00:45.878  1200  1211 I Activitymanager: Cpu Usage:
11-02 13:00:45.878  1200  1211 I Activitymanager: Load: 11.09 / 11.24 / 10.73
11-02 13:00:45.878  1200  1211 I Activitymanager: Cpu Usage From 7739ms To 2694ms Ago:
11-02 13:00:45.878  1200  1211 I Activitymanager:  System_Server: 17% = 9% User + 7% Kernel / Faults: 149 Minor
11-02 13:00:45.878  1200  1211 I Activitymanager:  Adbd: 2% = 0% User + 2% Kernel
11-02 13:00:45.878  1200  1211 I Activitymanager:  Mediaserver: 1% = 0% User + 0% Kernel
11-02 13:00:45.878  1200  1211 I Activitymanager:  Logcat: 0% = 0% User + 0% Kernel
11-02 13:00:45.878  1200  1211 I Activitymanager:  M.Android.Phone: 0% = 0% User + 0% Kernel / Faults: 15 Minor
11-02 13:00:45.878  1200  1211 I Activitymanager: Total: 22% = 12% User + 10% Kernel
  ……
11-02 13:00:46.128  1832  1832 D Jcrashcatcher: Create Bugreport Instance(/Sdcard/Bugreport/Bugreport-004402140726351-20111102-130045)
User: The System Spent In User Mode, User Mode In Low Priority
Kernel:The System Spent In System Mode.
Iowait:The System Spent For I/O Wait
,……..
```

以上代码中 com.*******.Camera 应用程序包下的 com.*******.Camera.Video.Videocamera 出现了 keydispatchingtimedout Anr，表示输入事件(比如：按手机键盘，触摸屏幕等操作)在预设时间之内（通常是 10 秒）无响应，当这种情况发生时，在目录/Sdcard/Bugreport/下会生成一个以当前日期命名的文本文件，例如文件名：Bugreport-004402140726351-20111102-130045，此文本文件里记录了发生 Crash 时的多个 I/O 操作,但具体是哪个主线程引起的 Crash，还需要我们做进一步的分析。通过在生成的 Bugreport 文件中搜索关键字：Dalvik Threads，可以定位到本应用程序的虚拟机信息。

```
----- Pid 1368 At 2011-11-16 21:39:00
Cmd Line: Com.*******.Camera
Dalvik Threads:
"Main" Prio=5 Tid=3 Vmwait
| Group="Main" Scount=1 Dscount=0 S=N Obj=0x40026240 Self=0xbda8
| Systid=1368 Nice=0 Sched=0/0 Cgrp=Unknown Handle=-1344001376
  At Dalvik.System.Vmruntime.Trackexternalallocation(Native Method)
At Android.Graphics.Bitmap.Nativecreate(Native Method)
At Com.Android.Camera. Video.Videocamera (Bitmap.Java:468)
At Android.View.View.Builddrawingcache(View.Java:6324)
At Android.View.View.Getdrawingcache(View.Java:6178)
At Android.View.Viewgroup.Drawchild(Viewgroup.Java:1541)
At Android.View.Viewgroup.Dispatchdraw(Viewgroup.Java:1343)
At Android.Widget.Abslistview.Dispatchdraw(Abslistview.Java:1474)
At Android.Widget.Listview.Dispatchdraw(Listview.Java:2978)
  At Android.View.View.Draw(View.Java:6733)
  At Android.Widget.Abslistview.Draw(Abslistview.Java:2327)
  At Android.View.Viewgroup.Drawchild(Viewgroup.Java:1616)
  At Android.View.Viewgroup.Dispatchdraw(Viewgroup.Java:1343)
  At Android.View.Viewgroup.Drawchild(Viewgroup.Java:1614)
  At Android.View.Viewgroup.Dispatchdraw(Viewgroup.Java:1343)
  At Android.View.View.Draw(View.Java:6630)
```

从上面的代码中可以看到关键问题出现在 java 代码的 468 行：At Com.Android.Camera. Video.Videocamera (Bitmap.Java:468)

至此，测试人员和研发人员就可以在 eclipse 开发环境中相应的 java 代码块中去定位问题类型，然后去进一步寻找解决问题的具体方法。

⑵ 在 Log 文件里查找关键字：Anr，定位到的关键信息如下所示：

```
[ 11-26 23:01:57.169    8353:0x20ac W/Activitymanager ]
Timeout Of Broadcast Broadcastrecord{45ec73c8 Android.Intent.Action.Media_Mounted} -
Receiver=Android.Os.Binderproxy@45f76d38
[ 11-26 23:01:57.169    8353:0x20ac W/Activitymanager ]
Receiver During Timeout:
Resolveinfo{45e24348 Com.Android.Providers.Media.Mediascannerreceiver P=0 O=0
M=0x208000}
[ 11-26 23:01:57.231    8353:0x20ac I/Activitymanager ]
Anr In Process: Android.Process.Media
Annotation: Broadcast Of Intent { Act=Android.Intent.Action.Media_Mounted
Dat=File:///Sdcard Cmp=Com.Android.Providers.Media/.Mediascannerreceiver (Has Extras) }
Cpu Usage:
Load: 10.11 / 8.02 / 6.84
……
```

上述代码说明 com.Android.Providers.Media 应用程序包下的 mediascannerreceiver 发生了 broadcast Timeout Anr，广播事件在 20 秒内未完成执行。当这种情况发生时，在目录/Sdcard/Bugreport/下会生成一个以当前日期命名的文本文件，测试人员可用类似于 keydispatchingtimedout Anr 的思路去找到问题的所在，也可以在目录/Data/Anr/Traces.txt 下查找 Crash Log 的关键信息，最终定位到 Java 代码行中。

⑶在 log 文件里查找关键字 Anr，定位到的关键信息类似如下代码：

```
10-01    05:01:42.938: Warn/Activitymanager(1149): Timeout Executing Service:
Servicerecord{45c06118com.*******.Android.Debug.Jcrasher/.Jcrasherservice}
10-01    05:01:42.948: Info/Activitymanager(1149): Anr In Process:
Com.*******.Android.Debug.Jcrasher
10-01    05:01:42.948: Info/Activitymanager(1149): Annotation: Executing Service
Componentinfo{Com.*******.Android.Debug.Jcrasher/Com.*******.Android.Debug.Jcrashe
r.Jcrasherservice}
……
```

以上代码说明 com.*******.Android.Debug.Jcrasher 应用程序包下的 Jcrasherservice 发生了 Service Timeout Anr，程序对一个服务进程未完全响应。可以在/Sdcard/Bugreport/下生成的文本文件或者在/Data/Anr/Traces.Txt 文件中找到

问题所在。

3.1.2 Java Uncaught Exception

还有其他一些 Java Crash 由 Java Uncaught 异常引起。对于这些异常的产生原因，又有多种情况，以下列出常见的四种：

（1）在 Bugreport 文件里查找关键字：Uncaught Exception，定位到的关键信息类似如下代码：

```
01-01 00:19:45.764: Error/Androidruntime(2251): Uncaught Handler: Thread Main Exiting Due To Uncaught Exception
00:19:45.764:Error/Androidruntime(2251):Java.Lang.Nullpointerexception
01-01 00:19:45.764: Error/Androidruntime(2251): At Com.*******.Android.Debug.Jcrasher.Jcrasheractivity.Causeexception(Jcrasheractivity.Java:86
……
```

以上内容说明问题是由 java.Lang.Nullpointerexception 引起的主线程异常退出所造成。一般情况下，Uncaught 异常发生在 androidruntime，还有很多原因使得系统杀死应用程序进程，Uncaught 异常时 sig 数值一般是 3，象以下代码中显示 Sig 数值为 9，可以断定这不是一个 uncaught Exception Crash。

```
At Android.Os.Binderproxy.Senddeathnotice(Binder.Java:353)
At Dalvik.System.Nativestart.Run(Native Method)
[ 12-08 04:05:35.451  1189:0x4a9 I/Process  ]
Sending Signal. Pid: 30392 Sig: 9
```

（2）在 Bugreport 文件里查找关键字：Uncaught Exception，定位到的关键信息类似如下代码：

```
11-07 17:19:05.733 18621 18621 W Dalvikvm: Threadid=3: Thread Exiting With Uncaught Exception (Group=0x4001e160)
11-07 17:19:05.733 18621 18621 E Androidruntime: Uncaught Handler: Thread Main Exiting Due To Uncaught Exception
11-07 17:19:05.743 18621 18621 E Androidruntime: Java.Lang.Illegalstateexception: The Content Of The Adapter Has Changed But Listview Did Not Receive A Notification. Make Sure The Content Of Your Adapter Is Not Modified From A Background Thread, But Only From The Ui Thread. [In Listview(16908298, Class Android.Widget.Listview) With Adapter(Class Android.Widget.Headerviewlistadapter)]
……
```

以上代码说明 crash 是由 java.Lang.Illegalstateexception 引起。根据相应的应用程序包可以准确定位到 java 的代码行中。

（3）Java Heap Leak 内存泄漏。如果应用程序内存超过了所允许的限度（24m）而导致了泄漏，在 Log 文件里会出现如下代码：

```
890 22613 22613 W Dalvikvm: Threadid=3: Thread Exiting With Uncaught Exception (Group=0x40023160)
11-24 20:40:10.890 22613 22613 E Androidruntime: Uncaught Handler: Thread Main Exiting Due To Uncaught Exception
11-24 20:40:10.900 22613 22613 E Androidruntime: Java.Lang.Outofmemoryerror: Bitmap Size Exceeds Vm Budget
……
```

（4）native Heap Leak 内存泄漏。在应用程序占用的内存超过分配的限度（Gref）(2000). Native Crash 时，异常会发生，这时会在 log 文件里出现如下类似代码：

```
01-01 00:05:55.414 D/Dalvikvm( 1330): Gref Has Increased To 2001
01-01 00:05:55.414 W/Dalvikvm( 1330): Last 10 Entries In Jni Global Reference Table:
00:05:55.414 W/Dalvikvm( 1330):   1991: 0x457b2510
Cls=Landroid/Database/Cursortobulkcursoradaptor; (44 Bytes)
……
```

需要说明的是 Procrank 文件可以捕获当前系统中各进程的内存使用情况，其中的 uss 值非常重要，如果 uss 值总是上升而没有下降，说明容易发生内存泄露，如果 uss 有时上升然后会下降到一个正常值，说明将不会有内存泄露发生如图 1 所示：

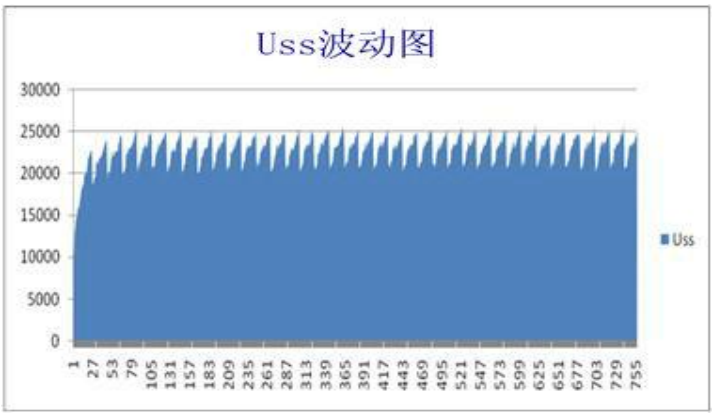


图 1　Uss 波动图

3.2　System Crash

3.2.1 Sigsegv

Sigsegv: Segmentation Fault 意味着进程执行了一个无效的内存引用，没有物理内存对应该地址，程序可能抛出 sigsegv 异常，会在/Data/Tombstone/下找到这些信息。

⑴ 搜索关键字 Sigsegv，如果定位到类似下面代码，说明发生了 data Abort 异常，若发生这类问题，一般都是深层的 so 在调用时由于深层代码的 nullpointerreference 所导致。

```
01-01 00:41:52.751: Info/Debug(1061): Build Fingerprint:
'Semc/X10i_1232-9897/X10i/Es209ra:2.1-Update1/2.0.A.0.468/1afa:Eng/Release-Keys'
01-01 00:41:52.751: Info/Debug(1061): Pid: 1896, Tid: 1896 >>>
Com.********.Android.Debug.Jcrasher <<<
01-01 00:41:52.751: Info/Debug(1061): Signal 11 (Sigsegv), Fault Addr Deadcafe
00:41:52.751: Info/Debug(1061): R0 Deadcafa R1 00000001 R2 Afe3db7c R3
00000000
01-01 00:41:52.761: Info/Debug(1061): #00 Pc 0000ae74 /System/Lib/Libc.So
01-01 00:41:52.761: Info/Debug(1061): #01 Pc 000003ba
/Data/Data/Com.********.Android.Debug.Jcrasher/Lib/Libjnicrasher.So
……
```

⑵ 搜索关键字 Sigsegv，如果定位于类似以下代码，说明地址 0xdeadd00d 发生了 sigsegv 异常，这个地址发生了 dvmabort。

01-01 00:35:47.551: Info/Debug(1061): Pid: 1296, Tid: 1305 >>> Android.Process.Acore <<<

00:35:47.551: Info/Debug(1061): Signal 11 (Sigsegv), Fault Addr Deadd00d

……

3.2.2 Sigill

Sigill: 非法指令异常是 cpu 处理非法指令之后所发生的一个异常，程序收到了 sigill 信号，一般就报告 illegalinstruction 错误信息。下面的代码说明地址

```
01-01 00:44:41.246: Info/Debug(1061): Pid: 1914, Tid: 1914 >>> Com.*******.Android.Debug.Jcrasher <<<
01-01 00:44:41.246: Info/Debug(1061): Signal 4 (Sigill), Fault Addr 810003c4
00:44:41.246: Info/Debug(1061): R0 0000ab50 R1 45689080 R2 810003a8 R3 810010b0
……
810003c4 发现了非法指令。
```

3.3 Modem Crash

顾名思义，Modem Crash 往往是由调制解调器所引起。调制解调器可以提供快速可靠的连接，使得手机能够连接到互联网，但是有些操作可能导致 Modem Crash 的发生，如：软件版本与硬件型号不兼容。当发生 Modem Crash 时，可以在生成的 kdumppdf 文件里定位到关键信息。如以下代码说明是调制解调器发生了 crash。

07-12 17:15:01.321 <3>[110.120141] Arm9 Has Crashed

07-12 17:15:01.322 <6>[110.120283] Smem: Diag ’Mod Gsdi:Pfault 00000 ’

07-12 17:15:01.322 <3>[110.121191] Smem: Crash Log

7:15:01.434 <0>[110.232524] Kernel Panic - Not Syncing: Modem Has Crashed...

3.4 Kernel Crash

一旦手机发生了 kernel Crash，相关的日志信息很少，而且又很难重现，因此遇到这类问题比较难解决，比较简单的方法是：如果手机彻底被锁定，不能再使用；数字键(Num Lock)、大写锁定键(Caps Lock)、滚动锁定键(Scroll Lock)不

停闪烁；手机蓝屏并且在手机屏幕上看到内核 Cp Assert 出来的信息，如出现以上任何一种现象则可以断定发生了 Kernel Crash。也可以在 Crash Dump 文件里搜索关键字：Kernel Panic，然后检查 Call Stack 信息，将会看到 Backtrace 信息。在手机/Proc/Kpanic 目录或者 /Data/Cp_Panic.Bin 中可以找到 panic Logs（），通过 adb 工具可以导出所需要的 log 信息，需要时可以将该信息提交给研发人员。

3.5 Watch Dog Crash

如测试人员在 crash Dump 文件和 event Log 文件里的最后几行代码中看到 2802 标识，意味着发生了 Watch Dog Crash。

```
12-15 16:36:14.076 I/Process ( 1150): Sending Signal. Pid: 1150 Sig: 3
12-15 16:36:14.076 I/Dalvikvm( 1150): Threadid=7: Reacting To Signal 3
12-15 16:36:14.058 I/[2802] ( 1150)  (Or Watchdog)
Com.Android.Server.Am.Activitymanagerservice  (Or Null)
```

4 结论

Android 程序测试过程中会遇到很多 Crash 的问题，一般出现的问题都可以通过本文所述的这种方法查看代码得到解决。该方法抽取出易于理解，精度高的代码分析规则集，在理论上是合理的。该方法已经通过实际案例验证，可以用于解决一些比较深层的定位问题，具有一定的理论和现实意义。

参考文献

[1] 韩超. 《Android 系统级深入开发--移植与调试》

[2]android 输出 log 相关应用技巧剖析 http://Developer.51cto.Com/Art/201001/180265.Htm.

[3] Android 中级教程之----Log 图文详解 Http://Weizhulin.Blog.51cto.Com/1556324/311692

[4]Android Log 详解

Http://Hi.Baidu.Com/Lesley_Gyn/Blog/Item/C7db803c5a0c20fda2cc2bc5.Html

测试复杂系统

译者：于芳

好的设计习惯可以拯救你的系统和头脑清醒

每个人都知道单元测试很重要，应该努力去测试所写的每一块代码。提倡以测试为驱动的开发者们甚至认为你应当先写下测试用例，然后用他们作为一个设计工具。不幸的是，对大部分团队和系统来说，真实世界粗鲁的打断和自动化测试覆盖面远远不够全面（包括单元测试，集成测试和完整的系统测试）。造成这种明显忽略良好习惯的原因多种多样。对测试爱好者来说令人震惊而且痛苦的真相是成功的复杂系统可以不需要充足的自动化测试就可以开发。成本可能更高，需要更多手工测试。但是，它是可能的。甚至那些大多数以测试为驱动的开发团队也会同意她们的代码是经过测试的(除了那些可能在一些生命攸关和任务攸关的行业)。在这篇文章中，我会一一详述并且展示怎样给传统意义上比较难测的代码块写测试用例。 就像你将会看到的，测试复杂系统与系统架构和你代码的设计有很大关系。这对软件系统的其它方面也是适用的，比如系统的可扩展性，性能和安全性。你不能在一堆恐怖的乱七八糟的代码上将他们吞咽。你必须从开端就设计它的可测试性或者重构的方法。好消息是好的设计（模块化和松散的连接起来的元素有良好设计性能和模块间的接口）可以形成一个系统，让它具有更好的可测试性，可扩展性，预先格式化和安全性。

基本知识

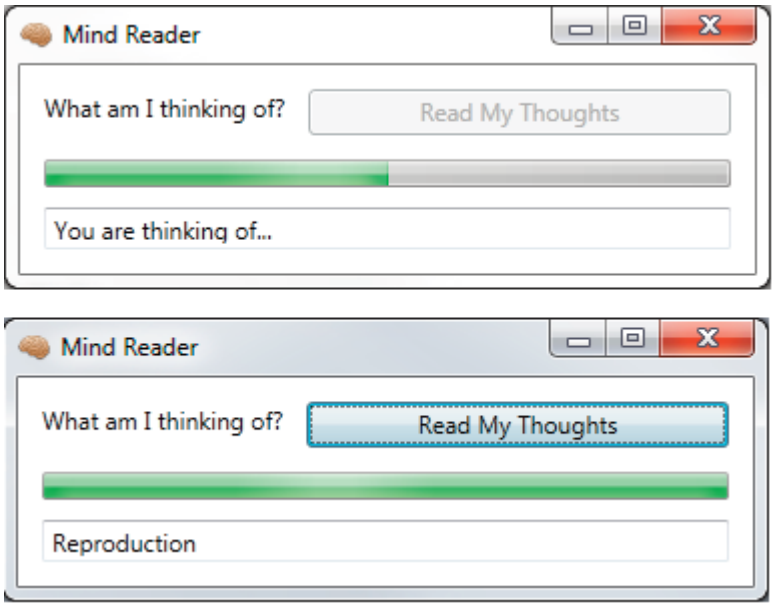
开发复杂的软件常常需要融合多个团队—可能不是在同一个地点或者不在同一个时区。你必须有一个很扎实的开发流程，它可以融合源头控制，一个自动化创建/配置的系统和自动化测试。这样一个环境的一个主要担心是怎样避免打破创建，因为有一个开发人员引起的破坏性的创建可以阻止到每个其它的开发者。一个破坏性的开发包括代码没有像预测中的那样表现。

比如，如果我引进一个缺陷到数据访问层，试图访问数据的每个代码块都会失败。系统越复杂，就越难修改这种特殊变化带来的影响（而这个可以通过好的设计来最小化）。自动化测试在这里就变得很关键了。如果所有的测试在你做出改变后通过了，你就能很确信说你没有破坏任何其它人的代码（假设是合理的测试覆盖范围）。

思维读取器

为说明目的，我用一个 C#程序，叫做‘Mind-Reader’， 来读取你大脑在想什么，告诉你你正在想什么。图 1 表示在行动中它是什么样的。它有个单独的按

钮来初始化思维读取过程，一个报告进程的进程条，和一个输出块来告诉你你正在想什么。在任何时候只能有一个思维读取进行，所以当思维读取在进行的时候这个按钮是被灰色掉的。这是个很简单的程序，但是它会让我们探索复杂的测试技术。



第三方代码

在一个复杂系统里，你的团队可能只要写一部分代码。你可能使用开源的库，从外部服务提供商获得的经过许可的代码，（更多时候是）从其它团队拿过来的代码。从经验之谈上来说，你不应当给不是自己写的代码创建测试用例。你通常通过一个 API 来使用第三方代码。在某些情况下，你直接就在代码中使用 API。当在这种模式下操作时，第三方就等同于将库内建于你的编程语言或者操作系统中。然而，原始的 API 访问常常并不是最好的办法。API 可能是很复杂的，它可能在各个版本间不稳定，也可能太地基，可能是一个 C 语言写成的 API 而你的系统是在 C++环境中运行的。在这所有的情形下，团队人员可以选择去写一个包裹层在第三方代码上，这样可以暴露需要的功能，然后使用包裹层。这些包裹层对测试来说会很方便，因为（如果设计合理）他们很容易模仿。

另外一个唯一需要担心的关于第三方代码的是如果你模拟它，你需要真正的理解它的行为—尤其是对那些错误的情形要真正理解。如果一个 API 方法返回一个错误状态或者抛出一个异常，而你的模仿程序没有模拟它，你就不能针对这个 API 很好地测试你的代码。另外一个第三方代码的领域涉及到框架和插件容器。在这些情形下，你应当开发你的代码成尽可能不是主人/容器知论，并且分

别测试它。Mind-Reader 用户界面在程序开始执行的时候使用 WPF 作为第三方代码和框架。WPF 通常通过事件处理器与你的代码连接在一起，这个事件处理器可能会被认为是松散连接的，但是它代表你必须处理 WPF 的签名和数据类型（所以当点击按钮的时候，你的事件处理器机会得到一个 Systems.Windows.RouteEventArgs 对象它需要处理的）。IMainWindow 界面将 Mind-Reader 程序在主要窗口上作的操作抽象化为：

```
public interface IMainWindow
{
void EnableGoButton(bool enable);
void SetThoughtText(string thought);
void UpdteProgressBar(int percent);
}
```

IMainWindowEvents 界面抽象了主要窗口产生的和程序想要响应的所有相关事件：

```
{
void OnGoButtonClick();
void OnKey(string key);
void OnClose();
}
```

注意这些界面是完全 WPF 不知的。你可以在任何时候切换 UI 技术或者为测试目的来模仿，稍后你就会看到。MainWindowWrapper 类（看清单 1）运行了 IMainWindow 界面，将调用转发给实际的 MainWindowWPF 类。它也聆听特定的 WPF 事件（通过事件处理器），并将他们转发给他的 IMainWindowEvents 接收端。

Listing One

```
using System;
using System.Windows.Controls;
namespace MindReader
{
class MainWindowWrapper : IMainWindow
{
MainWindow _window;
IMainWindowEvents _sink;
Button _goButton;
public MainWindowWrapper(MainWindow window)
{
_window = window;
_goButton = (Button)_window.FindName("goButton");
// Hook up event handlers
_goButton.Click += new System.Windows.
RoutedEventHandler(_goButton_Click);

_window.KeyUp += new System.Windows.Input.
KeyEventHandler(_window_KeyUp);
_window.Closed +=
new EventHandler(_window_Closed);
}
public void AttachSink(IMainWindowEvents sink)
{
_sink = sink;
}
// MainWindow events
void _window_Closed(object sender, EventArgs e)
{
_sink.OnClose();
}
void _window_KeyUp(object sender, System.
Windows.Input.KeyEventArgs e)
{
_sink.OnKey(e.Key.ToString());
}
```

```
void _goButton_Click(object sender,
System.Windows.RoutedEventArgs e)
{
_sink.OnGoButtonClick();
}
// IMainWindow
public void EnableGoButton(bool enable)
{
_window.Dispatcher.Invoke((Action)(() =>
{
_goButton.IsEnabled = enable;
}));
}
public void SetThoughtText(string thought)
{
_window.Dispatcher.Invoke((Action)(() =>
{
_window.thoughtBox.Text = thought;
}));
}
public void UpdteProgressBar(int percent)
{
_window.Dispatcher.Invoke((Action)(() =>
{
var min = _window.progressBar.Minimum;
var max = _window.progressBar.Maximum;
var range = max - min;
_window.progressBar.Value = min +
(percent / 100.0) * range;
}));
}
}
}
```

测试跨平台系统

这个部分讨论在不同操作系统上有相同行为的系统。我写过一些跨平台的代码，大多是用 C++ 写的。很经常，脏乎乎的跨平台问题证实是一个创建问题（错误的标志，错误的依赖性，错误的版本等等）。做跨平台开发的关键是将平台依赖的代码尽可能最少化。这大部分可以通过使用跨平台库和避免直接系统调用来实现。对可能会出异常的用户界面（这个我接下来会讨论），用一个单独的代码基地去开发一个整个的跨平台系统是可能的。

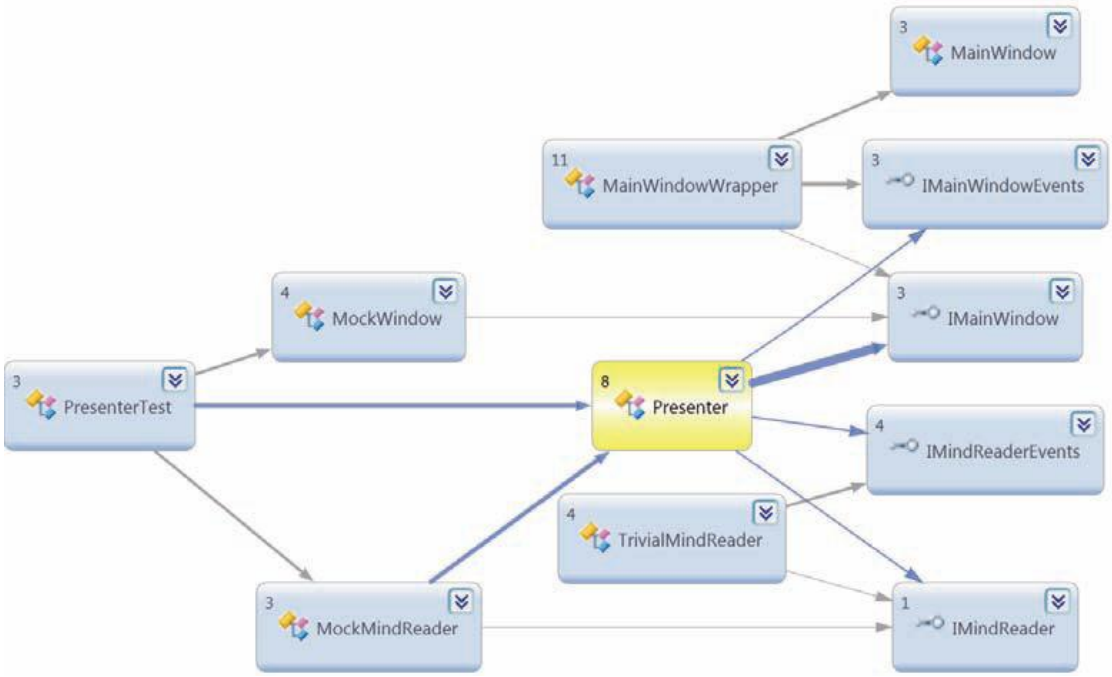
这使得测试变得简单多了。开发者可以在他们的开发系统上跑测试用例直到他们接触到一些平台特定的代码，他们可以很确定说他们的改动在其他系统上行得通。你创建的系统应当考虑到在所有平台上的测试。这个测试取决于你的开发过程。有些团队对每一个变化都会在每个平台上跑测试用例。有些团队有底层测试政策，这里更有扩展性和穷尽的测试用例会周期性地跑起来（比如说，晚上或者在周末时跑）。有些系统必须用用不同的性能来达到各个平台的目标。它可能是多种硬件设备，不同的浏览器，同一操作系统或者浏览器的不同版本，等等。这种情况通常会导致目标平台的组合式爆炸。你需要首先以好的设计来发布。通常的一个方法是创建灵活的系统，这个系统的性能能够被动态发现或者配置并且能作为插件整合进主系统。从测试角度来看，这种方法也可能会减少测试的负担。如果你有五个可选的性能，而你的目标平台可以提供这些性能中的任何子集，那么你有 32 个目标要测。但是，如果你可以单独测试每个性能，那么你只需要测试五个。这是和重要的分别。

另外一种方法可以减少负担的是模仿丢失的性能。举个例子，在 3D 图片世界里，软件供应是很常见的当硬件加速不再可能。从测试的角度来看，这是个双刃剑。代码本身可能变得更流水线化，仅仅想一个性能可以做到，不用关心它是否能被模仿。但是，你会需要在实际和模拟的两种模式下测试系统的行为。这对如果你是开发模拟器的情况来说尤其正确。

测试用户界面

啊，测试用户界面.....自动化测试的祸根。任何一个用户界面有几个方面需要去测试。在大多情况下，你会使用某些库，它们知道怎样呈现一些小组件如按钮，文本框，屏幕上的图片并且允许你将事件处理器与像按钮点击或者选项变化的事件连接起来。如果你开发自己定制的 UI，事情就会完全不同，完全跳出本篇文章的范畴。尝试自动化它来直接测试一个 UI 是一个噩梦。你必须处理消息群，事件队列，定时器，输入设备延迟和考虑屏幕分辨率和用户反应时间。这可以做到，但是这很闷并且脆弱。更好的方法会将显示和低级别的时间处理从 UI

状态的管理中隔离开来。这种方法使用著名的 MVC 模式的演化--MVP 模型观看呈现者。这些模型是业务对象，他们对 UI 完全没有概念。这个检查是在 UI 组件附近的一层很薄的层（和合成物，例如窗体，有很多组件的对话框）。呈献者是那个聪明的小伙。它与模型/ 业务对象对话并且负责更新合适的视图，管理 UI 状态。他让你从测试角度学到了什么？一个完全自由的测试你的业务对象和用户界面逻辑和状态的。你依赖于你的组件库来做正确的事情然后手工验证视图设计。



多种多样的类接口以及他们的关心在图 2 中呈现出来。在思维读取程序中，业务或者模型是通过 TrivialMindReader 类运行的，这暴露了 IMindReader 接口：

```
public interface IMindReader
{
void Read();
它也会通过即将离开的 IMind-ReaderEvents 接口产生思维读取事件：
public interface IMindReaderEvents
{
void OnProgress(int percent);
void OnReadComplete(string thought);
}
}
```

这两个接口将业务从程序的其他部分隔离开来并且允许以测试的目的来模拟业务，你接下来就会看到。用户接口状态和逻辑是通过 Presenter 类来管理的。这个 Presenter 类接收它的构建器在 IMainWindo 和 IMainReader 引用，而它运行 IMainWindwoEvents 和 IMainReaderEvents 接口。

```
Listing Two
namespace MindReader
{
/// <summary>
/// In charge of MainWindow
/// </summary>
public class Presenter :
IMainWindowEvents,
IMindReaderEvents
{
IMainWindow _window;
IMindReader _reader;
public Presenter(IMainWindow window, IMindReader reader)
{
_window = window;
_reader = reader;
}
public void OnGoButtonClick()
{
_window.EnableGoButton(false);
_window.SetThoughtText("You are thinking of...");
_reader.Read();
}
```

```
public void OnKey(string key)
{
}
public void OnClose()
{
}
public void OnProgress(int percent)
{
_window.UpdteProgressBar(percent);
}
public void OnReadComplete(string thought)
{
OnProgress(100);
_window.SetThoughtText(thought);
_window.EnableGoButton(true);
}
}
}
```

Presenter 负责下面的 UI 状态:

当 go 按钮按下，将会启动一个思维读取（读心）会话，重置进度条到 0，重置文本框为基本消息：“你在想/思考……”并且让按钮灰掉。

当进度报告通过 OnProgress（）到达时，应当更新进度条。

当 OnReadComplete（）方法被调用时，应当显示新的想法，更新进度到 100%并且让 go 按钮恢复功能。

PresenterTest 通过下面几行代码中的两个模拟对象验证了这整个序列：

```
[TestMethod]
public void Test()
{
_presenter.OnGoButtonClick();
// The presenter should first disable the button and in
// the end re-enable it.
CollectionAssert.AreEqual(new List<bool>()
{ false, true }, _mockWindow.EnableGoButtonCalls);
// The presenter should set the thought twice, first to
// the generic message and then to the test thought
CollectionAssert.AreEqual(new List<string>() {
"You are thinking of...", "whatever" },
_mockWindow.SetThoughtCalls);
// The presenter should update the progress bar 5 times
// (initially to 0, finally to 100)
CollectionAssert.AreEqual(new List<int>() {
0, 25, 50, 75, 100 }, _mockWindow.UpdateProgressCalls);
}
```

当我跑这个测试时，我发现了一个缺陷--忘记在 Presenter 上重置进度条到 0。这就是那种可能从系统崩溃漏掉而只能在生产环节发现的缺陷类型。

这里是将真实的模仿窗口的 Presenter 与模拟思维读取器连接起来的测试启动：

```
[TestInitialize]
public void Setup()
{
_mockWindow = new MockWindow();
_mockMindReader = new MockMindReader(new int[] {
25, 50, 75 }, "whatever");
_presenter = new Presenter(_mockWindow,
_mockMindReader);
_mockMindReader.AttachPresenter(_presenter);
}
```

MockWindow（见列表三）运行 IMainWindow 界面，但是它做的所有动作都是记录对它的方法的调用。

```
Listing Three
using MindReader;
using System.Collections.Generic;
namespace MindReaderTest
{
public class MockWindow : IMainWindow
{
public List<bool> EnableGoButtonCalls = new List<bool>();
public List<string> SetThoughtCalls = new List<string>();
public List<int> UpdateProgressCalls = new List<int>();
public void EnableGoButton(bool enable)
{
EnableGoButtonCalls.Add(enable);
}
public void SetThoughtText(string thought)
{
SetThoughtCalls.Add(thought);
}
public void UpdteProgressBar(int percent)
{
UpdateProgressCalls.Add(percent);
}
}
}
```

MockMindReader（见列表 4）运行 IMain-Reader 接口。

```
Listing Four
using System;
using MindReader;
using System.Collections.Generic;
namespace MindReaderTest
{
public class MockMindReader : IMindReader
{
int[] _progressReports;
string _thought;
Presenter _presenter;
public MockMindReader(int[] progressReports,
string thought)
{
_progressReports = progressReports;
_thought = thought;
}
public void AttachPresenter(Presenter presenter)
{
_presenter = presenter;
}
public void Read()
{
foreach (int progress in _progressReports)
{
_presenter.OnProgress(progress);
}
_presenter.OnReadComplete(_thought);
}
}
}
```

MockMindReader 也提供了一个预录的思维读取会话，通过接收他构造器一个进度报告的清单和硬编码想法来实现：

```
public MockMindReader(int[] progressReports, string thought)
{
_progressReports = progressReports;
_thought = thought;
}
```

稍后，这个测试将 Presenter 作为附件添加为一个接收端。当 Presenter 调用 Read（）方法时（作为对模拟按钮点击的回应），这个模拟思维读取器将他的预录进度报告通过 On-Progress（）发送出去最后调用 nReadComplete()方法。Presenter 调用模拟主窗口的合适的方法，它记录了所有的东西。最后，这个测试证实正确的方法是通过检查模拟窗体的记录调用被调用。

稍后，这个测试将 Presenter 作为附件添加为一个接收端。当 Presenter 调用 Read（）方法时（作为对模拟按钮点击的回应），这个模拟思维读取器将他的预录进度报告通过 On-Progress（）发送出去最后调用 nReadComplete()方法。Presenter 调用模拟主窗口的合适的方法，它记录了所有的东西。最后，这个测试证实正确的方法是通过检查模拟窗体的记录调用被调用。

这整个启动的操作只需要花费很少的时间来写。它允许完全测试 UI 管理状态，而不需要处理实际的难测的 UI，而且当实际的 UI 变化的时候可以很容易进行扩展或者修改 。 我使用的可以使得测试能够超越.NET, WPF 和其用户界面的步骤是：

1) 通过接口（IMainWindow, IThought - Reader）访问所有的依赖性，用简单的模拟对象来模仿这些依赖关系，模拟的对象可以捕捉你想要测试的状态和流程

2) 通过测试场景来运行对象/系统

3) 验证对象/系统如期待中运行

测试联网代码

网络连接变得很盛行。今天，人们经常调用网络服务，他们常常习惯于添加一个库到应用程序中或者上帝禁止的话，自己写一些代码。设计一个安全，健壮和互相连接的系统比一个独立的系统复杂得多。原因是你必须处理更多的失败模式和很多杂物，它们与你的代码功能如安全性等等并不相关。远程系统可能在任何时候会消失，变慢，或者变化，包括在发送数据的中途或者就在一次复杂的握手国政之间。相反的情况也是这样，如果你的系统是远程服务的话。你怎么支持老客户？你怎么确认系统规模以及它是高度可实现的？你怎么确保攻击者不会

窃取你的数据或者将你的系统搞垮掉？这些是重要的考虑需要很多工程的工作。为了确保你正确理解了，你必须能够模仿和测试所有这些场景。

首要法则是将你的系统代码从低级别的网络连接隔离开来。这与适用于用户界面的 MVP 模式很相似。你想要你的域对象在一个对网络不可知的环境中运行。但是，在设计接口的时候必须仔细点，因为当你必须在电线上发送大量数据时非正式的或者阻塞的接口不会切断电线的。一般来说，对高性能的联网，你会想要异步的接口。我会稍后多说一些有关异步代码的事。为了说明这个过程，我创建了一个小的思维读取/读心 TCP 服务器。这个协议很简单。服务器等待一个客户发送命令："远程读取想法。"服务器以"进度："（比如说，"进度：25"）的格式并发的客户命令。我还创建了一个 Client 类，它暴露了 IMindReader 接口将 IMindReaderEvents 发送到接收端（看列表 5）

```
var s = client.GetStream();
// Send read command to server
var bytes = ASCIIEncoding.ASCII.GetBytes
("Read Thought Remotely\r\n");
s.Write(bytes, 0, bytes.Length);
// Read progress reports and finally the thought
var reader = new StreamReader(s);
while (true)
{
var line = reader.ReadLine();
if (line.StartsWith("progress:"))
{
var progress = int.Parse(line.
Split(':')[1]);
_sink.OnProgress(progress);
}
if (line.StartsWith("thought:"))
{
var thought = line.Split(':')[1];
_sink.OnReadComplete(thought);
}
}
}).Start();
}
}
}
```

这个 Client 类是放过来代替本地思维读取器的，而且需要一单行工厂代码的变动来连接所有的组件：

```
var wrapper = new MainWindowWrapper(mainWindow);
//var mindReader = new TrivialMindReader();
var mindReader = new cli.Client("localhost", 54321);
var presenter = new Presenter(wrapper, mindReader);
wrapper.AttachSink(presenter);
mindReader.AttachSink(presenter);
mainWindow.Show();
```

IMindReader 和 IMindReaderEvents 已开始是为异步交互设计的，所以这是个很简单的转换。Client 单独跑一个线程，用阻塞的 API 给服务器交互。在这个例子中，我创建了两个简单的测试用例来验证整个会话的正确性。我有一个简单的 Sink（接收端）类运行 IMindReaderEvents 方法，并且收集其调用（与 PresenterTest 里的 MockWindow 类似）。

```
class Sink : IMindReaderEvents
{
public List<int> OnProgressCalls = new List<int>();
public List<string> OnReadCompleteCalls =
new List<string>();
public void OnProgress(int percent)
{
OnProgressCalls.Add(percent);
}
public void OnReadComplete(string thought)
{
OnReadCompleteCalls.Add(thought);
}
}
```

我将 Sink（接收端）对象传递给 Client（客户），调用它的 Read()方法，开始一个对服务器端的思维读取会话：

```
[TestMethod]
public void SingleClient_Test()
{
var cli = new Client(_hostname, _port);
cli.AttachSink(_sink);
cli.Read();
while (_sink.OnReadCompleteCalls.Count == 0)
{
Thread.Sleep(100);
}
Assert.AreEqual(3, _sink.OnProgressCalls.Count);
Assert.AreEqual("I think therefore I am", _sink.
OnReadCompleteCalls[0]);
}
```

我将 Sink（接收端）对象传递给 Client（客户），调用它的 Read()方法，开始一个对服务器端的思维读取会话：

```
[TestMethod]
public void SingleClient_Test()
{
var cli = new Client(_hostname, _port);
cli.AttachSink(_sink);
cli.Read();
while (_sink.OnReadCompleteCalls.Count == 0)
{
Thread.Sleep(100);
}
Assert.AreEqual(3, _sink.OnProgressCalls.Count);
Assert.AreEqual("I think therefore I am", _sink.
OnReadCompleteCalls[0]);
}
```

模拟服务器和客户

在联网代码中，谈到客户端和服务器端是很常见的。有很多拓扑结构和协议，但是总体来说，客户是配置过的或者直到怎么寻找他们的服务器并且初始化通信。在理想的测试场景中，你通过接口将代码完全隔离开。如果你的代码是一台服务器，你创建一个模拟客户然后在测试中，模拟客户会开始与你的服务器讲话。如果你的代码是一个客户，创建一个模拟服务器然后在测试中，你的代码会与模拟的服务器讲话。有时候，同一段代码既是某些客户的服务器也是其他服务器 / 服务的客户。在这种情况下，你要提供很多模拟，单独测试服务器角色和客户角色。

对一台模拟服务器或者客户测试有意思的事情是你拥有完全的控制并且能够容易地模拟任何失败，例如一台到达不了的服务器，任意点的突然断连，丢失的响应，损坏的响应和缓慢的响应。你甚至可以插入断点逐行检查代码和模拟中的每一个交互。当对实际的服务器 / 客户测试的时候，常常很难创建很多错误条件和行为。模拟服务器对模拟簇也很有用，在簇里能够在测试与测试之间或者在一个测试过程中间动态重新配置模拟簇。

MindReader 客户和服务器精确地重复相同的概念。不仅要直接与.NET TcpListener 和 TcpClient 工作，你还要创建 tcpListener, ITcpListenerEvents 和 ITcpClient 接口（如果你想要一个不阻塞的客户，那么添加 ITcpClientEvents）。然后添加 TcpListenerWrapper 和 TcpClientWrapper 将所有的东西连接起来，将它传递到 Mind-Reader 客户端和服务器。从这里，你可以模拟任何 TCP 联网的场景。

本地测试服务器

有时候忠实地模拟一个服务器或者客户是很复杂的。在这些案例中，接下来最好的东西就是本地测试服务器（或者客户）。你不能看到所有的交互（通常只能通过日志文件才能看到），但是至少该连接是大大简化的。你可以在自己的机器上不影响任何其他人启动或者结束测试服务器。你的代码页和实际的服务器交互，这给你带来自信，它在自然环境下也能正常运行。在这种情况下调试程序的经验通常不那么重要。

有时候忠实地模拟一个服务器或者客户是很复杂的。在这些案例中，接下来最好的东西就是本地测试服务器（或者客户）。你不能看到所有的交互（通常只能通过日志文件才能看到），但是至少该连接是大大简化的。你可以在自己的机器上不影响任何其他人启动或者结束测试服务器。你的代码页和实际的服务器交互，这给你带来自信，它在自然环境下也能正常运行。在这种情况下调试程序的

经验通常不那么重要。

专用的测试服务器和环境

本地服务器可能很难配置（举个例子，当服务器依赖很多需要被模拟或者配置的其他的服务时）。经常，你需要测试代码在许多服务器和/或客户的性能。这就是专门的测试服务器或者完全成熟的测试环境需要应用到自然中来的地方。测试环境尝试将生产环境减少到一定的程度。有两种途径可以在测试环境下工作：

本地运行代码并连接到测试环境中。这个选项通常适用于测试客户代码。

部署代码到测试环境中。通过暴露的 API（REST，SOAP，普通的 HTTP，定制的 TCP/UDP）来测试。这个选项通常适用于测试服务器代码。

测试环境对集成测试很有帮助。需要关注的主要事情是测试环境并没有偏离生产环境：你必须有一个替代的流程（理想的是自动化实现）。另外一个担心是不同组件和处于测试下的资源的版本控制。测试环境是个稀有资源，为很多开发者共享的。如果多个开发者在同一时间部署他们最新的代码，他们可能很容易就踩到彼此的代码从而破坏这个为每个人搭建的测试环境。

延迟，吞吐量和压力测试

在处理分布式系统时，两个至关重要的度量是吞吐量和延迟。吞吐量定义了多少数据可以在电线上传输，而延迟定义了多快可以响应一个请求。由于分布式系统的性质，这些属性依赖于很多因素，比如硬件的规格有不同的节点，节点间可用的带宽，在用的协议以及你系统上的负载。在延迟和吞吐量之间也常常会想户影响。总体上来说，当系统上的负载增加时延迟会开始上升，而吞吐量则会下降。这很困难，如果是不可能的，来预测所一个处在沉重负载下的分布式系统的行为因为它是非线性的。各种各样的阈值和交互在不同的负载下会……知道你的系统能够处理多大的负载是确保服务质量和规模至关重要的。

获得这方面知识的方法是测试系统在各种模仿的负载下，观察它的行为。获得这方面知识的方法是测试系统在各种模仿的负载下，观察它的行为。有时候，你将系统不同部分的组件隔离开，然后单独做压力策划四。有很多做压力测试的工具可用，但是对复杂的系统来说，你会想要开发自己的压力工具，这样可以模拟现实真正的负载和交互模式。

有限的生产测试和逐渐的部署

压力/负载测试很重要，但是它还不足够。另外一个测试代码变化的好技术是部署新版本的代码到与现有系统并行的少数几个服务器，然后观察新服务器的行为。如果有任何事情出错了恶，你切换回现有版本分析问题出在哪。如果一切都运行正常，你可以传递更多的服务器到新系统中。这种逐步部署的方式是一个

很有效的技巧，但是却是有着多个步骤的复杂过程，尤其是如果代码变化涉及到数据库制式，协议或者文件格式的话。

测试连续存储

连续存储是另外一个总是不易测试的区域。更相似的是，你不能在实际的生产存储上运行测试（尤其不会测试那些修改数据的情况）。这意味着你必须针对测试数据测试，这有着它自己的挑战，因为涉及到与生产代码同步。 这里有两个级别要处理：

(1) 数据库制式或者用作连续的文件格式

(2) 实际的内容

当你保留连续测试数据时，每个级别或者两个可能不能同步。你必须很勤奋地跑在弧线的前面。攻击的第一行还像平常一样---抽取数据访问，然后在模拟数据资源上测试你的代码。这会 让你不受实际的访问实际数据商店的情况下很快地运行很多测试。在某些情况下，你可能需要测试更复杂的数据访问场景涉及到在不同级别的缓存问题。关键是要能够控制你想要测试的数据访问过程的每个方面。直到你通过接口和使用依赖介入与依赖性交互，你就可以开始进行操作了。

测试异步代码

异步代码，经常与联网或者 UI 相关，可能是最难测的代码。当你开始一段异步操作时，控制流立即就返回到调用的代码，但是操作还没有完成。在你的测试代码中，你需要等待操作完成，然后像平时一样继续，检查结果，状态和副作用。异步的 API 可以划分为三大块：

1) 调回为基础的 API

2) 以未来为基础的 API

3) 以队列为基础的 API

有调回为基础的 API， 你提供了一个调回功能给异步操作；当它完成的时候，你的调用函数就被调用了。以未来为基础的 API 返回你可以定期查询的对象来检查那个操作是否完成了。以队列为基础的 API 存储你可以跟踪的响应在一个队列里。

在你的测试中，你想要边忙边测试异步操作来完成或者是在调用函数里设置一个标志或者是直接检查未来对象或者对 ie 然后像平时一样继续测试。

事件接口我最喜欢的一个以调回为基础的 API 格式。 在我的网络连接测试中，我用下面的代码来等待思维读取会话来完成：

```
while (_sink.OnReadCompleteCalls.Count == 0)
{
```

Thread.Sleep(100);

}

这不是理想的。如果代码失败了， OnReadComplete（）就永远不会被调用，测试就会悬吊。一个很常见和简单的方法就是只让他睡去：

// Sleep for 5 seconds to give the operation enough time to

// complete

Thread.Sleep(5000);

这很糟糕，也是。如果操作在一个瞬间的时间内完成，你仍然需要等待 5 秒钟；而如果你运行很多测试， 它可以显著地减慢你的测试运行。我最近测试了很多代码，有些我不得不杀掉，启动然后远程配置 RabbitMQ 的簇在大约 50 个测试用例里完成。如果我用硬编码的睡眠来等待每个操作完成，我就不会坐在这里写这篇文章里。最好的方法是做些定期的检查，在有超时的情况下进行。假的代码看起来像下面这样的某些东西：

```
timeout = Now + 5 seconds
while (!ok and Now() < timeout)
{
ok = is operation complete?
sleep 100 milliseconds;
}
if (ok)
{
// good
}
else
{
// oh-oh. operation times out
}
```

这是笨重的，每次你想在等待一个操作完成的时候写成的代码。在将来一片文章里，我会说明一个简洁的类，它将代码压缩成下面的一小段：

var ok = Wait.For(3000, () =>

{ _sink.OnReadCompleteCalls.Count > 0 });

这个声明会等待 3 秒钟，定期检查表达式，会立即跳出来如果值为真的话。如果永远都不为真，在 3 秒钟过后，它就会退出，OK 就不会成真。

“多种语言创建的系统的问题是错误汇报常常没有流水线化。”

有些异步代码依靠计时器和超时。你应当确保这些超时是可以配置的，这样你的测试不必要等待 10 分钟才发现你的代码在做什么而这时有些操作已经超时了。

测试多线程或者多进程代码

有着低级别锁的多线程代码和修改共享资源的多线程对测试来说就是一个噩梦。这个应当让你犹豫不决。在每个情况下，我都遇到过多线程代码太复杂以至于不能测的情况，代码本身就是太复杂了。有些肮脏的缺陷与细密纹理的锁（不用提在不需要锁的算术英雄尝试了）。考虑像消息传递和共享为零设计的现代的方法。如果你必须管理低级别的并发，尝试最小化表面区域---代码评审加分析，然后用各种测试方法炮轰代码。我不推荐尝试模拟线程。你永远都不能去模拟实际的线程工作的方式那样会毁掉你充满缺陷的代码，也只会给你一个虚假的安全感。

测试多进程代码提出了他自己的挑战。这跟多线程化的代码类似，他们里的某些操作在你的测试控制流范围之外发生。不同点在于在其他进程中检查对象的状态更难。有很多进程间通信机制，你应当察看一下他们来查看进程的状态。你也可以模拟进程（就像你在测试网络连接代码时模拟服务器时一样）。如果你想等在另外一个进程的某个操作完成，可以使用一个文件作为锁（每个进程尝试拥有一个锁文件的唯一的访问权）。

测试用多种编程语言创建的系统

很多复杂的系统有的组件是用多种不同编程语言写的。因为组件间的交互是通过相同的网络协议完成的，这种测试不是很有意思。

然而，有时候多种编程语言通过语言邦定以更直接的方式交互。举个例子来说，很多动态输入的语言（脚本语言）像 Python 和 Ruby 提供了一个 C 扩展的 API，他允许与任何 C 库集成。另外一个方向也很常见，可以把一个脚本语言嵌入到 C/C++程序中然后允许用户来编写各种项目或者用动态语言写插件。比如，Lua 通常嵌入在游戏引擎中。Javascript，通过 Google Chrome 的 V8， 也可以和 C 一起操作。Java 虚拟机当遇到多种语言时就是一个力量房，而在最近几年，瞄准用很多语言如 Scala， Clojure， Jython 和 JRuby 的 JVM 已经变成时髦的事情。Java 本身总是有 Java 本地的接口（JNI）与 C 交互。但是跨语言开发的海报孩子绝对是.NET 框架，这个框架是已开始设计就作为多语言框架的。VS 给多

语言开发提供的支持是很巨大的， 甚至可以允许你做调试，跨进不同的语言。

多种语言系统的问题是错误汇报常常没有流水线化，跨语言界限的失败可以伪装然后悄悄地失败（或者程序只可以退出）。为测试这样的系统，你需要采用一种系统的途径来收集语言界限两方面的信息。设计语言邦定是一门妖术。配制整齐数据类型常常是主要起因。如果你曾经尝试吞咽一个很酷的 C++类到一打 STL 类型和定制模板，你永远不会忘记它。Swig 几乎可以产生绑定到任何语言的 C/C++绑定而且很强大。但是就像那句话所说的：“力量越大，困惑越多”。如果你用 Swig/C++，我强烈推荐你将绑定本身视为仅仅是一层薄薄的外表并且避免任何延伸。

测试多种语言有核心 C/C++引擎的系统和邦定层是最好做的层。总体来说，本土的代码应当对系统可以从其他语言访问的事实丝毫无所谓。

你应当能够单元测试 C/C++写的核心引擎。如果你有一层脚本代码（比如，Python）通过绑定来运用本土代码，考虑模拟本土代码。这对通过 Monkey 补丁的动态语言来说通常都是很简单的。

有时，系统的多种语言性质正好与你的优势互补。写测试用例和运行测试用例在 Python 中比在 C++中容易多了，你可以发现很多好的 Python 单元测试框架。你不必编译并连接你的测试，你甚至可以在 REPL 中探寻你的对象。

测试失败代码路径和错误处理

复杂系统的巨大挑战之一是测试失败和错误处理。 主要困难是对每个成功的交互，通常都有一个单独的路径：一个输入事件到达（网络包，按钮点击，另外一个进程的消息，计时器过期），它被处理，然后系统返回一个结果并/或者记录这个结果。每个人都知道怎样测试这种令人高兴的路径。但是每一个在路径上的步骤都可能失败。对每次失败，会有一个不同的回应和可能的撤退。突然，原本干净的交互进入一个难题而你需要去测每个头。下面是一些常见的失败你可能从来都不会综合区测试的：

- 输入文件或目录不存在/有错误接入信息/包含损坏的数据
- 任意操作上产生的内存不足
- 硬盘空间不足
- 网络连接突然断开
- 调用的外部组件悬挂
- 首个名字字符串是 30MB（有人在替你做缓存溢出测试！）

如果你忽略测试你的代码在这些情况下的表现，那么你不会知道系统在最关键时候会怎样表现。比如，你的代码可能需要警示操作团队当某些服务器的响应

时间超过 5 秒钟。现在，考虑一下在这片代码中有个缺陷的后果。。。

好的，你知道我在讲什么。所有的东西都可能失败，错误处理代码本身应当被测试。但是怎样描述看起来是失败代码路径的组合爆炸？它都是以合理的设计开始的。同样老的模块化，松散组合高度连接的组件原则会给你带来好的服务。识别每个组件的风险，设计一个错误处理政策。举个例子，对敏感数据，使用事务来确保你不会结束与不连续的状态。

对必须经常要运行的组件，在冗余和可切换性能中开发。对大型的暴露 API 的子系统，确保控制和保护表面区域这样没有崩溃或者意料之外的异常情况发生在用户身上（抛掉异常作为设计的一部分是可以的）。

现在你的系统是一个定义良好的交互组成的一系列组件，你可以系统地测试如果任何组件在那些定义良好的方式中失败的话系统怎样表现。听起来有很多东西，而事实上的确是。唯一的安慰是这种方法推出了真正出色的 API：很小，很少的交互，尽可能少的副作用，以及很少外部暴露的失败。

让我们检查一下怎样在 MindReader 例子中测试错误。我的目标是测试 Presenter 怎样处理异常情况。用模拟的 mind reader 这很容易做到。测试传递空的参考值给进度报告和想法。这会引起模拟的 mind reader 抛出空值参考异常在 Read（）方法中：

```
[TestMethod]
public void MindReader_CrashTest()
{
var mockMindReader = new MockMindReader(null, null);
var presenter = new Presenter(_mockWindow,
mockMindReader);
mockMindReader.AttachPresenter(presenter);
presenter.OnGoButtonClick();
}
```

问题是，“Presenter 怎样处理 MindReader 异常？”，答案是它没有处理。它会崩溃并且燃烧。测试暴露了 Presenter 的脆弱。现在你可以提出一个政策（捕捉异常，展示一条消息给用户，悄悄地重试几次，或者记录下异常然后退出）。如果有某个错误处理机制适当地出现，测试可以验证它的确被合理的激起了。

在策略层面上，我推荐使用异常并且让他们传送到一个点上，在这个点上他们可以被合理的处理。确保系统的状态在遇到错误的时候保持原封不动。使用设

计模式（例如 RAII）并且随后清理掉。如果你用像 C 或者 Go 这样的语言，这种语言没有异常，你只要额外勤奋一点就好了。

结论

今天的软件正在变得越来越复杂，却很少异常，那是它没有被真正地测试过。即使软件开发过程强调测试，通常只是有有限的单元测试。由于深入测试复杂系统的极大挑战，这常常是一个精细的关于成--回报的决定。权衡是市场时间和质量。在这篇文章中，我展示了怎样通过依靠试过为真的设计原则来深入测试复杂系统最具有挑战性的方面。

基于任务驱动的第三方测试服务平台设计与实现

作者：陈磊 简炜 周波 翟艳芬 相春雷

摘要：随着软件规模的不断扩大，软件质量问题越来越突出。软件测试在软件质量保证过程中的地位也越来越重要，这种情况带动了第三方测试服务的兴起。由于第三方测试机构涉及的测试对象越来越广泛，构建了庞大的软件测试工具集和资源库，因此如何对这些工具和资源进行统一管理，并提供从测试设计到测试结果分析的整体测试服务平台是一个有待解决的问题。针对上述问题，本文设计并实现了一个基于任务驱动的第三方测试服务平台，该平台集成了所有软硬件资源的统一配置管理、平台内物理设备的全程监控、测试流程的约束以及测试过程数据的收集和测试数据的查询等功能。

关键字：任务驱动；第三方测试服务；软件测试；测试管理

1. 引言

从软件质量保证的方面来说，软件测试是软件质量保证的一个重要环节，通过软件测试来验证软件是否满足测试需求，验证产品是否满足内部质量和外部质量[1]。现在有些软件开发商已将研发力量的 40%以上投入到软件测试之中[2]。

在测试过程中，测试执行人员和测试系统的交互以及对测试工具配置、使用和切换的过程繁琐使得测试执行效率不高。[3]在较大投入的前提下，如何通过统一的测试管理方法和测试自动化方法约束测试工程师行为，最大化复用测试用例，规范测试接口等，在大型软件系统测试过程中显得尤为重要。针对上述问题，依据软件测试流程和测试项目实施方法，介绍了笔者参与设计并实现的基于任务驱动的第三方测试服务平台，该平台基于 B/S 架构设计，采用 ASP.NET 和 Microsoft SQL Server 2005 进行开发，同时兼顾了效率、可扩展性的等相关特性，保证了测试平台的质量。

2. 关键技术简介

2.1 第三方测试服务简介

第三方测试是具有独立的、测试检验能力的机构，由具有专业的测试人员应用专业的测试工具、方法对软件的质量进行全面审核的过程。第三方测试有别于软件厂商自己进行的测试，其目的是为了保证测试工作的客观公正性。[4]第三方测试工程主要包括需求分析审查、设计审查、代码审查、单元测试、功能测试、性能测试、可恢复性测试、资源消耗测试、并发测试、健壮性测试、安全测试、安装配置测试、可移植性测试、文档测试以及最终的验收测试等。[5]

2.2 任务驱动简介

所谓任务，即为完成某个测试流程所执行的过程。一个测试项目可以分成多个任务，每个任务是一个逻辑执行单元，不可再分。所谓任务驱动，就是指整个测试项目是通过执行任务来达成的。在完成所有任务测试后，即完成了测试项目。任务是任务驱动的基本元素。是测试中的一个因子，每个因子都正常运行，整个测试工作才能正常运行。[6]

3. 基于任务驱动的第三方测试服务平台的设计和实现

3.1 基于任务驱动的第三方测试服务平台工作流分析

基于任务驱动的第三方测试服务平台通过测试任务约束各种软硬件资源，通过测试服务平台中的测试管理模块进行测试需求、测试用例的设计。在测试需求设计过程中，将测试项目划分成多个测试任务，然后针对每个测试任务中的测试流程设计测试用例。测试管理人员通过将测试任务分给不同的测试组完成测试任务的分配。测试组组长依据测试任务通过软硬件资源池的软硬件设备搭建测试环境，再通过平台启动对应的自动化测试工具进行测试，同时可通过平台的监控接口对使用的测试设备和字云进行监控，将测试过程数据保存到测试管理模块中完成对测试的全周期管理。

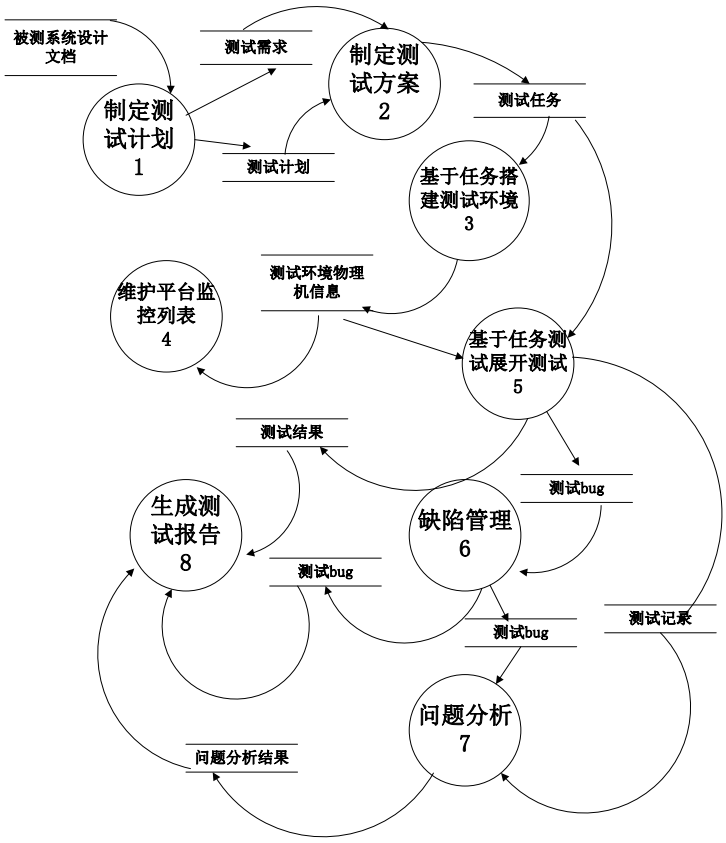


图 1 基于任务驱动的第三方测试服务平台工作流图

3.2 基于任务驱动的第三方测试服务平台工作的设计

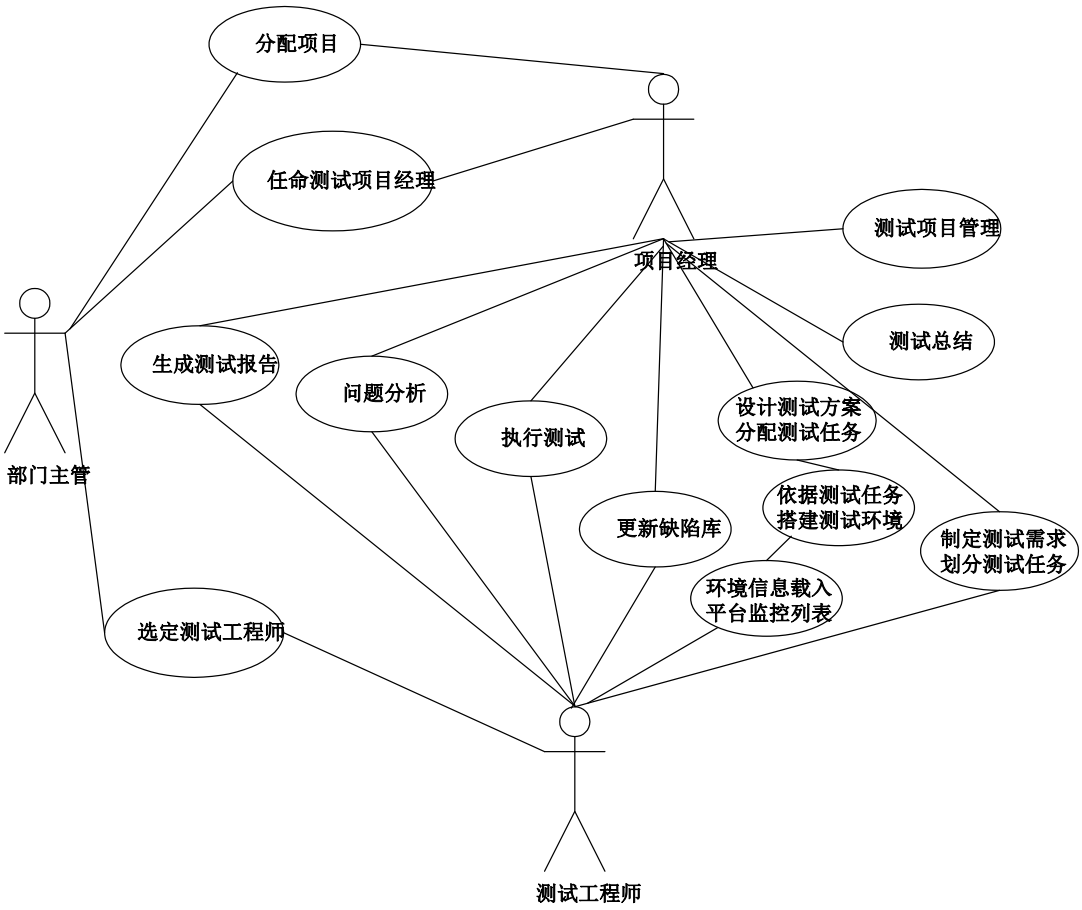


图 2 基于任务驱动的第三方测试服务平台总体用例图

其中，部门主管主要是通过测试服务平台对测试团队管理，测试项目的分发工作进行统筹规划。项目经理主要负责整体项目的管理，其中包含了测试需求和测试任务的划分、测试方案的撰写、测试执行过程的管理等，属于监管控制角色。测试工程师主要是按照测试任务执行测试、搭建测试环境、记录测试过程和结果数据等，属于测试执行角色。

基于任务驱动的第三方测试服务平台共有四个模块组成，其中包含了平台监控、测试环境部署区、测试区和资源库。

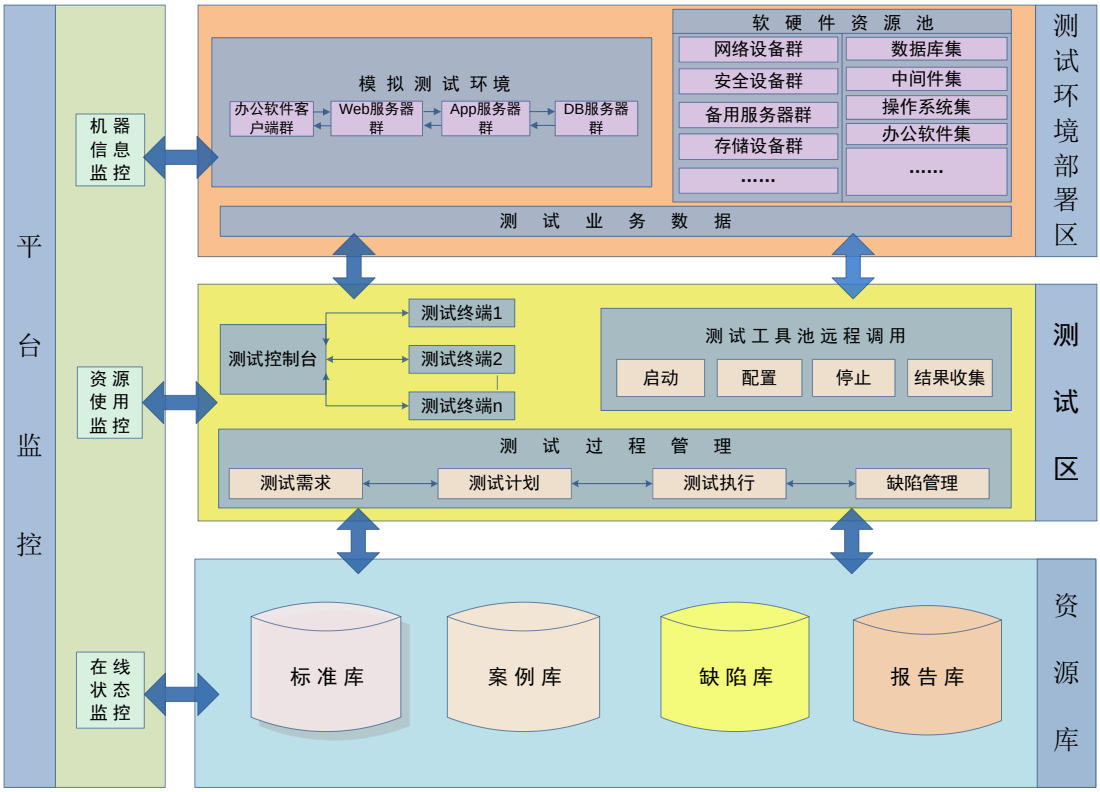


图 3 基于任务驱动的第三方测试服务平台系统结构图

3.2.1 平台监控

平台监控包含了对平台内部所有的物理机的可用状态的管理、系统资源的监控和机器信息的监控三个功能实现。平台监控主要是为了能够对平台内部的物理机的使用情况、监控程度以及软硬件配置信息进行管理，方便测试过程中对平台内部所有和测试资源的调度情况进行分析。

在技术方面通过数据接口层的公共服务接口获取资源数据，通过对原始数据进行采集、分析和处理，形成半结构化的操作系统数据资源池，以 Web Service 方式提供对外的公共服务接口，使用富客户端方式进行数据的图表展现。

3.2.2 测试环境部署区

测试环境部署区主要包含软硬件资源池和模拟测试环境区两部分组成，其中软硬件资源池包含了测试评估过程中所需的所有的网络设备、安全设备、服务器、存储等硬件环境以及数据库、中间件、操作系统、办公软件、测试工具等相关软件。模拟测试环境是依据通用基础软件测试评估的软硬件环境需求，通过调用软硬件资源池中的设备来构建的

在逻辑上测试环境部署区是以软硬件资源池为基础，以分享硬件、独占软件的设计理念为指导，采取松耦合、低内聚的设计模式，为通用基础软件测试提供

高效的、敏捷的、可靠的测试软硬件保障。

3.2.3 测试区

测试区是测试执行过程中的测试交互的主模块，是监控内部测试环境，控制测试工具进行测试实施的区域。测试区包含了测试控制台，可以通过不同的测试终端访问测试平台，通过测试业务数据的交互完成测试。在测试区中，测试工程师调用平台提供的测试工具，对工具进行启动、配置、结果收集和停止的操作。同时通过测试过程的管理，约束了测试需求、测试计划、测试执行和缺陷管理测试行为，保证了基础软件测试评估的规范性、客观性、可溯源性等。

3.2.4 资源库

资源库是基础软件测试平台的重要组成部分，主要包含标准库、案例库、缺陷库和报告库四个部分。其中标准库主要包含了通用基础软件测试标准和其他相关的国家标准、行业标准等。标准库是基础软件测试平台提供测试服务解决方案，开展测试服务，的根本依据；案例库包含了对通用基础软件产品开展测试工作所需的测试用例，分为功能测试案例库、兼容性测试案例库、安全性测试案例库、可靠性测试案例库、性能测试案例库、可扩展性测试案例库、易用性测试案例库、综合应用测试案例库等；缺陷库展现的是对软件测试过程发现的缺陷的整理、汇总和统计结果等相关信息；报告库主要完成测试项目后形成的成果报告的在线展示功能。

3.3 基于任务驱动的第三方测试服务平台工作流实现

第三方测试服务的测试过程中，用户、开发方和测试方形成了相互制约的关系。但三方的目标都是希望被测系统符合需求、能够稳定运行。本平台是通过不同的测试组完成不同的测试任务，通过测试组完成测试项目的全部任务来完成测试。

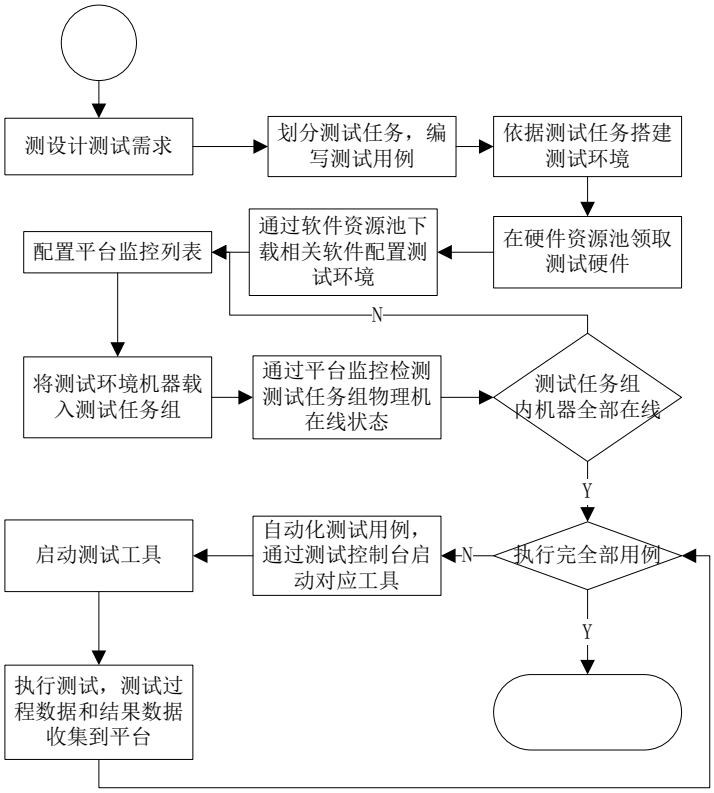


图 4 基于任务驱动的第三方测试服务平台执行流程

测试服务平台系统截图如下所示。

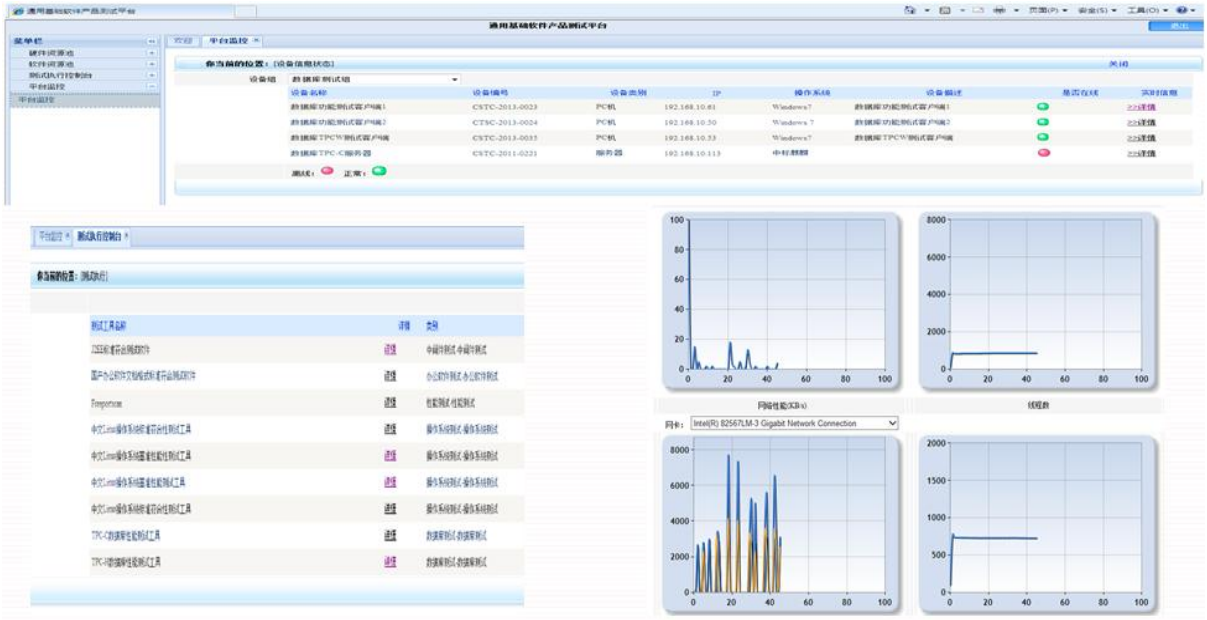


图 5 测试服务平台系统截图

4. 结论

基于任务驱动的第三方测试服务平台依据了第三方测试服务流程设计，采用了 B/S 架构避免了平台应用的系统相关性。通过本测试服务平台可以约束测试实施人员测试流程、保证测试客观公正、复用测试用例，目前该平台已经在测试项目中得到了应用，系统同时支持 100 人在线，50 人并发，该系统已经满足了第三方测试服务的要求，但是在测试工具的结果自动收集等方面，还需要进一步改进。

参考文献:

[1]吴慧韫，李卓群.基于 H 模型的软件测试管理应用模型研究[J].计算机工程与设计.2006,27(11):1993-1995.

[2]单锦辉.姜瑛.孙萍.软件测试研究进展[J].北京大学学报自然科学版.第 41 卷,第 1 期，2005：134-145.

[3]周飞宇.自动化测试平台的设计与实现[D].北京交通大学.2009.06：1-2.

[4]于秀山.第三方测试-一种提高软件质量的有效方法[M].质量与可靠性.2000(2):25-26.

[5]杨松,苏佳,陈磊. 基于第三方测试服务流程的测试管理系统的设计与实现[J].计算机光盘软件与技术.2011(1)：161-161

[6]熊登峰.基于任务驱动的 ContentEngine 自动化测试的设计与实现[D].华中科技大学.2010.11：11-12

我的 LoadRunner 使用经历

作者：冀方

有这样一件事情一直记忆犹新，刚入行时，通过 LoadRunner 实现了性能测试，过程之艰辛非三言两语能概述。

那时，公司急于推销一款产品，网页版的即时通讯系统，类似于现在的 WebQQ。公司 Boss 侃侃而谈，把产品的优点如数家珍般的对客户描绘了一遍。可是，在当客户询问 Boss，你们产品的性能如何时，我们心里顿时咯噔一下，心想完了，Boss 也面露难色，杯具了。由于当时仅对产品的功能和兼容性做了全面的测试，性能测试压根就没做过。面对精明的客户，公司也拿不出数据说服他们。最后，这单生意不出意外的泡汤了。领导当时触动很大，把测试召集起来，表情凝重，语重心长地说，生意谈不成，没关系，我们还有下次机会，关键是拿不出客户想要的东西。先前测试主要测功能，性能测试是我们的弱项，说明我们还有提高的空间，如果能把性能测试做好，一方面自己的能力也提高了，另一方面公司也能给客户提供有力的数据，说服他们买我们的产品。大家下去后好好往这方面努力吧。

带着 Boss 殷切的希望，测试负责人安排了我开展针对公司产品性能测试的探索和研究。有幸搞性能测试，可能领导也是觉得我平时工作比较踏实，比较勤奋，内心自然充满了无限感激。可是，性能测试是神马玩意，用什么测，怎么测，测到什么程度，还真是一头雾水。那就用最土最笨的方法，上网搜，泡论坛。

于是，我就成了 51testing 的常客，零零散散的收集了一些性能测试常用的测试工具和测试资料。后来 MSN 群里的同行告诉我，《LoadRunner 性能测试实战》这本书不错，适合入门看。我马上跑去西单图书大厦买了一本，翻了翻感觉还不错，后来证实这本书还真是帮了大忙。现在还能记得书的主要作者是陈绍英，陈老师在书里提供了自己的 MSN 群号，有问题随时可与他联系。幸福啊，没想到还有这么贴心的售后服务,能与作者近距离的探讨，当时心里真是乐开了花。感觉上也不是一个人在战斗，公司的性能测试真是有希望做好了。回想起来，陈老师算是我性能测试的启蒙老师了，真心感谢。

性能测试前的准备工作，前后花了将近一个月的时间，包括了解系统的架构，学习性能测试的理论知识，以及 LoadRunner 工具的使用等方面的内容。这一月，周例会是最难熬的，领导总是很关心的问，怎么样了，有进展吗，而我好像没什么可说的，也没什么实质性的进步，压力山大啊。这之后，有了一定积累后，就想把 LoadRunner 用在实际的项目中。

真是说起来容易做起来难。录制了一个模拟用户登入登出的脚本。经过修改后，脚本可以回放，但是数据库的表中却查不到登出的用户。奇怪啊，明明回放成功了？！后来和 MSN 群友一起分析，我们一致认为是表面上成功了，实际上账号并没有成功登入系统。这个结论也得到了研发同事的认可。可是为什么会出这样的问题呢？经过查找，发现问题似乎出在了协议选择上。录制的时候只选用了 Web(HTTP/HTML)协议，而实际上系统的实现不仅用到了 Web 协议，也用到了 Windows Sockets 协议。心中窃喜，以为找到了问题的答案，迫不及待的录制了多协议脚本，结果回放依然失败。真是希望越大，失望越大。怎么办？这个问题郁闷了好几天。

周例会的时候，做为重点问题，研发和测试都在讨论。不知谁冒出了一句，会不会是因为密码做了 SHA1 加密后导致的问题？对啊，我怎么没想到啊。于是和开发一起分析了脚本，发现脚本回放时，提交了上一次经过加密后的密文，而此时服务器端已经生成了新的密文，两端密文不一致，使得验证失败，自然登入失败。此时，才真正有种豁然开朗的感觉，同时也说明理解系统的工作方式对性能测试而言是多么重要。找到了症结后，解决方法就简单了，做一个加密算法的 Dll 文件，并在脚本中调用。

方法虽然简单，但操作起来还是出问题了。脚本加载了 Dll 文件后，然后通过加密接口对密码和随机数序列做加密运算，发现每次的结果依然不变。头大了，怎么可能呢，当时甚至怀疑起自己智商有问题。最后，多亏了群友的鼓励，是不是哪儿用错了？

我反复的查看脚本的每一句，这才发现每次加密用到的随机数居然是一样的。问题就来了，随机数是服务器返回的，不一样才对啊！后来把书里的相关的内容又仔仔细细的看了一遍，终于找到了问题的解决方法，为了保证脚本回放时能够动态的获取到这个随机数，需要做“关联”操作。果然，做了“关联”以后，返回值正确了，感觉又前进了一步。

随后，怀着忐忑不安的心情，参数化脚本，编译脚本。最后，还是出现了我不愿意看到的结果，报了一堆错。用现在的话说真是泪流满面。可是究竟错在哪儿呢？反复看书后，发现 LoadRunner 参数化，是按照它内置的机制执行的，符合这个机制，编译就能通过。于是，按照书中给出的方法，找到了可以更改这个机制的地方，修改完了后，再次编译，这次 OK 了。运行脚本后，在数据库表里也查到了用户的登入登出记录。周例会的时候免不了也被经理表扬了一番，记得他当时夸张地说，这个结果是真的，不是假的。估计也是因为高兴。心中窃喜，不算白忙活。

终于成功了，艰难的过了第一关，压力也少了很多。可是，兴奋劲还没持续多久，问题又来了。

在对系统做压力测试的时候，LoadRunner 提示没有足够的虚拟用户分配给这个新的参数？这又是神马问题啊？百思不得其解。只能重新录制，修改脚本。像平时电脑崩溃了以后，重启电脑一样。这次居然误打误撞地解决了问题。原来，LoadRunner 中在对用户名和密码或其他数据参数化了以后，不能删除参数，重新参数化，否则就会出现问题。

以后的测试过程就顺利了。先设计测试场景，然后录制脚本，修改脚本，运行脚本，生产结果报告，分析结果数据。再后来，Boss 与客户谈的时候，就底气很足地说，我们用 LoadRunner 对系统做性能测试，并骄傲地展示 LoadRunner 自动生成的测试报告。虽然有些图表还不知道什么意思，但在当时看来，公司在性能自动化的测试上边确实进步了不少。而这就是两个多月辛苦付出的回报。

上面经历说明，即使在性能测试上是只小菜鸟，也不要灰心丧气，轻言放弃，只要专心寻找问题的答案，也一样可以做好。

我的海外测试经历之测试管理

作者：冀方

来 ERP2 项目组之前，我便在智能交通领域这片广阔的海洋中，苦苦挣扎了两年时间。期间，虽然呛了不少水，有过教训，但同时也积累了经验，学了不少东西。或许是曾经有过这样的项目经历，刚进项目组，便有幸成为负责稽查设备测试的小组长。初期的时候也就一光杆司令，只能被管理。后来，我们小组成员增加到三人，虽然人数不多，终究手底下也有兵了。那么测试管理又如何在测试团队中实施的呢？

首先是管理人。如何管呢？这个我要说只能是靠制度了，经过多年的沉淀，公司有一套相对完善的开发测试控制流程。

具体点说，团队成员每天必须写日报。这种方式的缺点是，时间长了真的很烦，有种被监视的感觉。可是又没办法，谁让这是重点项目呢。优点是，由于每天都要汇报工作，所以几乎没有人偷懒。

除了日报，还得写周报。我做为测试的小组长，除了周一分配工作任务外，周末还要汇总团队成员的周总结，总结测试小组上周的工作进展和遗留问题，然后汇报给测试经理。

当然，周例会也是不可少的。团队的每个成员都要做汇报，重点总结上周的工作成果。对于遗留问题，大家有针对性地一起讨论，定一个解决方案。这种方式能起到好的督促作用，而且对管理的效果也起到了直接检验的作用。

随着项目的开展，测试人员会完成诸如测试方案、测试用例此类的文档。是不是文档完成后，就可以立马实施了呢，至少在 ERP2 项目组不是，实施的前提是开评审会议。缺点是，耗费时间长，有时因为文档中错误多，返工的次数也可能会比较多，虽然很烦，但是也要耐着性子改；优点是，便于后期测试有所侧重，便于测试实施。记得有次评审我写的稽查设备的测试方案，由于事先仅依照旧的设计文档编写了系统框架图，忽略了实际设计时对系统做的优化调整，结果会上挨了领导的批评，说我细节关注太多，忽略了应该关注的东西。后来我总结这次教训，方案重写是因为没有把握住系统的整体。如果再上升一点高度，那就是不管是做为一个测试人员还是一个管理者都要眼观全局，着眼于整体。这也是我通过评审会议学到的管理学。

每个人都是一座宝矿，这话一点不假。尤其是对测试团队来说，成员来自不同的公司或部门，有不同领域的从业经验，每个人都有独到的地方，有自己的优点。为此，团队建立了内部培训机制，以便于团队成员之间优势互补。这种管理

方式也算是对公司现有制度的一个补充。这种方式的最大优点是轻松自由，毕竟隔行如隔山嘛，即使有些术语听不懂也没关系，成员之间经过交流探讨，加深了了解，增进了友谊。

另外一个补充，是后备领导培养机制。这个项目，由于工作关系，测试经理需要经常出差去国外，讨论关于需求和测试方法的问题。出差期间，测试团队怎么能群龙无首呢，这就需要一个临时领导，管理整个测试团队。虽然我没有被选中，有些沮丧，但同时也发现了自己能力的不足，那就向别人学习吧。后来证明这种机制是明智的，使得整个团队能够有序运转，团队成员的工作也没因为测试经理不在而受到影响，实现了无缝管理。

其次是管理时间。这个项目从一启动，便注定要和时间赛跑，而完成这个项目也注定是一场艰苦的战役。个人感觉时间总不够用，领导也一再强调，面对强大的对手，所有人都不能偷懒，只有比对手更勤奋才能有胜出的希望。为此，项目组每天晚上都会额外加班两小时。而且为了赶进度，周六也要加班。这当然不能算是一种好的时间管理的方法，但是，在世界舞台上与国外强手对决，如果非要期待一种好的结果，中国人唯有凭借一股拼劲。

加班，是每个做项目的人都会经历的过程。除此之外，提高单位时间内的工作效率也是测试团队成员追求的目标。之前提到的临时领导，也是因为他能合理安排时间，工作效率高，而深得测试经理的器重。有次还在会上表扬他，今日事，今日毕，是我们学习的好榜样。在他的影响和带动下，大家也都尽量在下班之前结束一天的工作。由此看来，树立典型是管理好时间一个行之有效的方法。

还有一种方法也是实践中总结出来的，不做重复劳动。可真是说起来容易，做起来难。

内部测试开始后，团队成员用邮件的方式整理 BUG，并汇报给开发人员。后来公司 QA 要求测试团队使用 CQ，团队成员可能也是考虑到邮件的方便快捷，仅把之前的 BUG 都提到 CQ 上去，但同时也保留了使用邮件汇报 BUG 的方式。这种不规范也不专业的 BUG 提交方式，立刻引来了项目副总监的注意。他转发了 QA 的邮件，要求使用 CQ 的方式提交 BUG 即可，没必要使用邮件方式汇报，这样也就避免了重复劳动。我们这才完全放弃了使用邮件汇报 BUG。这件事印象深刻，幸亏问题纠正的早，如果按照之前的工作方式，我们又需要花不必要的时间。

最后是管理资源。这个相对容易些，测试过程中编写的脚本，以及产生的文档，比如说测试计划，测试方案，测试用例，评审记录等文档用 CC 管理，而发现的 BUG 则用 CQ 管理追踪。至于车辆，工具箱，测试仪器，待测设备之类的

资源则分配到个人，由专人负责。这样，物品的出借归还都能追踪到个人，避免了遗失。

本文结合实际项目从人、时间和资源的角度浅谈了实际工作中的测试管理，希望能藉此抛砖引玉，与大家探讨交流测试中的管理学。

作者简介：冀方，有 7 年测试经验，从事智能交通领域设备的硬件测试，嵌入式软件测试，担任过测试主管，负责测试团队管理和测试执行。

深入解读软件性能测试中的基本术语

作者：茹炳晟

摘　要：

本文对软件性能测试中最常使用并且大家耳熟能详的的几个基本术语从不同的维度与视角进行了深入解读，力求使读者能够对性能测试的这些基本术语有一个深入的理解和深层次的思考。

关键字：

软件性能衡量、并发用户数、响应时间、吞吐率

正文：

谈及软件的性能，最普遍的理解就是对软件处理的及时性要求，对于不同类型的系统,性能的关注点各不相同。对于典型的 WEB 交互式应用，一般以终端用户感受到的端到端的响应时间来描述系统的性能，而对于非交互式应用，比如典型的电信和银行后台处理系统，响应时间更多关注的可能是软件对系统事件（Event）处理并产生响应时间以及单位时间的事件吞吐量。但是，最终对于系统性能指标的描述大多还是采用响应时间、吞吐量、并发用户数和可扩展性等，这些都是性能测试用最常使用的基本术语。本文拟以这些基本术语为切入点，深入讨论这些指标所代表的本质含义，并从不同的维度和视角对其进行解读。首先从软件性能的不同视角来展开讨论。

衡量软件性能的不同视角

不同的对象群体对软件性能的关注点和要求不是完全相同的，有时甚至是对立的。以下首先从 4 个不同的视角来探讨软件的性能：终端用户、系统管理员、软件设计开发人员和性能测试人员。

从软件系统使用者的维度，也就是终端用户来讲，性能体现为用户在进行业务操作时的主观响应时间，具体来讲也就是用户在界面上完成一个操作开始到系统把本次操作的结果以用户能察觉的方式展现出来的全部时间，这个时间包含了系统响应时间和前端展现时间，这也是终端用户对系统性能的最直观印象。其中系统响应时间由系统响应和网络传输等部分构成，通常反映的是系统的能力，而前端时间则取决于用户端的处理能力。总之，最终用户希望整个时间越短越好；

从软件系统运维的角度，也就是系统管理员来讲，所关注的软件性能不仅仅是单个用户的响应时间了，更多的是关注大量用户并发访问时的负载以及可能的更大负载情况下的系统健康状态、并发处理能力、当前部署的系统容量、可能的系统瓶颈、系统配置层面的调优、数据库的调优，长时间运行稳定性和可扩展性。虽然系统运维方在大多数时候是和终端用户站在一条战线上的，希望提供尽可能快速的系统响应速度，但是某些时候确实对立的，最常见的就是系统运维方必须在最大并发用户数和系统响应时间之间做出必要的权衡。比如当提供 10K 并发访问的时候用户登陆时间是 5 秒，而当提供 20K 并发访问的时候用户登陆时间是 10 秒或更长。目前有些系统在处理极大并发用户的时候采用了排队机制以尽可能提高系统容量而增大用户感受到的响应时间。典型的例子就是火车票购票网站；

从软件系统开发的角度，也就是对软件设计和开发人员来讲，他们更关注性能相关的设计和实现的细节。这个关注面几乎涵盖软件设计和开发的全过程，主要会涉及以下多个方面：

- 核心算法的设计与实现是否高效
- 必要的时候设计上是否采用 buffer 机制以提高性能，降低 I/O
- 代码实现是否遵守开发语言的性能最佳实践
- 关键代码是否在白盒级别进行性能测试
- 必要的时候是否采用数据压缩传输
- 是否考虑前端性能的优化
- 是否存在潜在的内存泄露
- 是否存在并发环境下的线程安全问题
- 是否存在不合理的线程同步方式
- 是否存在不合理的资源竞争
- 数据库表设计是否高效
- 是否引入必要的索引
- SQL 语句的执行计划是否合理

- SQL 语句除了功能是否考虑性能要求
- 是否提供系统容量和性能的可扩展性
- 是否为性能 Profiler 提供必要的接口支持

注意，软件开发人员一般不会涉及系统部署级别的性能考量，比如软件目标操作系统的调优，应用服务器的调优、数据库的调优，网络环境的调优等。这部分工作目前一般是在系统性能测试阶段或者系统容量规划阶段，由性能测试人员和系统架构师和 DBA 共同协作完成。

最后就是软件性能测试的角度，作为性能测试与调优工程师，需要在系统架构师，DBA 和开发人员的共同协助下，全面关注以上提到的各个方面，既要能够准确把握软件的性能需求，又要能够准确分析定位引起“不好”性能表现的制约因素和根源，并提出相应的解决方案。

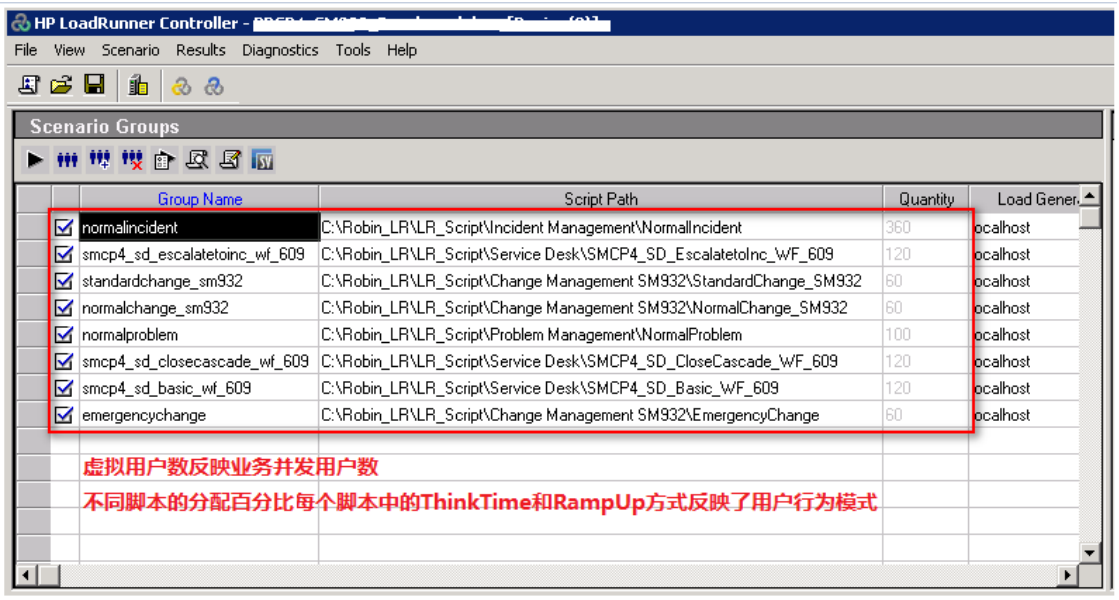
衡量并发用户数的不同视角

并发用户数是性能需求与测试中提到最多的也是最重要的指标之一，但是这个术语本身其实包含了两层不同的含义，一个是业务层面的，一个是服务器层面的。通常，我们所说的并发用户数多数指的是业务层面的含义，那么两者的根本区别在哪里，又有那些内在联系？为了更好的说明这个问题，先看一个实际的例子：

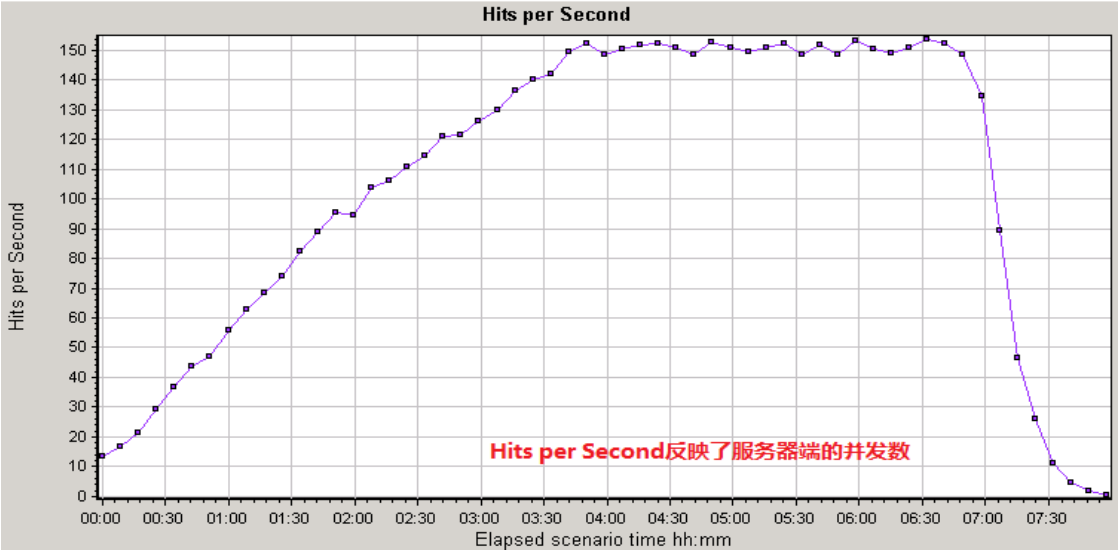
一个已经投入运行的 ERP 系统，该系统所应用企业共有 5000 名员工并都拥有账号，也就是说这个系统一共拥有 5000 个潜在用户。同时根据系统日志分析得知，该系统最大在线用户人数是 2500 人，那么根据通常的并发用户数的定义，这 2500 就是这个系统的最大并发用户数。显然，这 2500 个用户仅仅表明在最高峰的时候有 2500 个用户登陆了系统，但这个不能实际的反映此时此刻服务器所承受的压力，服务器所承受的压力取决于登陆的用户的行为。假设在某一时间点上，在这 2500 个“同时使用系统”的用户中，30%用户处于页面浏览状态（对服务器没有发起请求），20%用户在填写订单过程中，5%用户在递交订单，15%用户在做订单查询，而另外的 30%登陆了系统但是什么操作也没有做，那么在这个时间点上，这 2500 个“并发用户”中真正对服务器产生压力的只有 500 个用户（（5%+15%）*2500=500）。在这个例子中，我们把 5000 称为“潜在最大系统用户数”，2500 称为“最大业务并发用户数”，而 500 称为那个时间点上的“服务器并发请求数”。而实际上在那个特定的时间点上服务器端收到的并发请求数是有延迟的，这个我们在这里就不继续细究了。

从这个例子就可以看出，在系统运行的每一个时间点上，都有一个“同时向服务器发送请求的用户数”，这个就是服务器层面的并发用户数的含义，服务器

层面的并发用户数同时取决于业务并发用户数和用户行为模式，而且用户行为模式占了很大的比重。而分析得到准确的用户行为模式往往是性能测试中除了性能需求获取外最困难也是最重要的工作。对于已经上线的系统，常用的做法是采用系统日志分析法得到用户行为统计和峰值并发量等重要信息。对于一个未上线的全新系统，通常只能参考行业里面类似系统的统计信息来建模。



这里为了便于读者更好的理解，我们可以把这些概念对应到 HP LoadRunner 的概念上去。首先，就性能场景设计而言，LR 不会去模拟“潜在系统用户数（5000）”。其次，最大业务并发用户数（2500）就是 LR 中场景设计中的虚拟用户 VUser 的数量，并且用户行为模式通过场景中各个业务脚本的百分比以及脚本中各个操作步骤之间的思考时间 ThinkTime 实现，而 LR 中实时的 Hits Per Second 则反映的是服务器层面的并发数。当然为了比较精确的控制服务器层面的并发数，我们通常还会采取没有 ThinkTime 的脚本场景或者是在脚本中别加入必要的集合点 Rendezvous 以控制服务器端的并发数。（注意这里的 Hits Per Second 指的是每秒服务器端收到的 HTTP 的请求数量，而不是指用户在 WEB 页面上的实际点击事件的次数，因为一次页面点击操作通常会产生不止一次的请求。）



衡量响应时间的不同视角

通俗来讲响应时间反映了完成某个操作所需要的时间，其标准定义是“应用系统从请求发出开始到客户端接收到最后一个字节数据所消耗的时间”，是作为用户视角软件性能的主要体现。响应时间分为“前端时间”和“系统响应时间”两个部分。其中“前端时间”又称“呈现时间”取决于数据在被客户端收到响应数据后渲染页面所消耗的时间，而“系统响应时间”又可以进一步划分成 Web 服务器时间、应用服务器时间、数据库时间以及各服务器间通讯的网络时间。

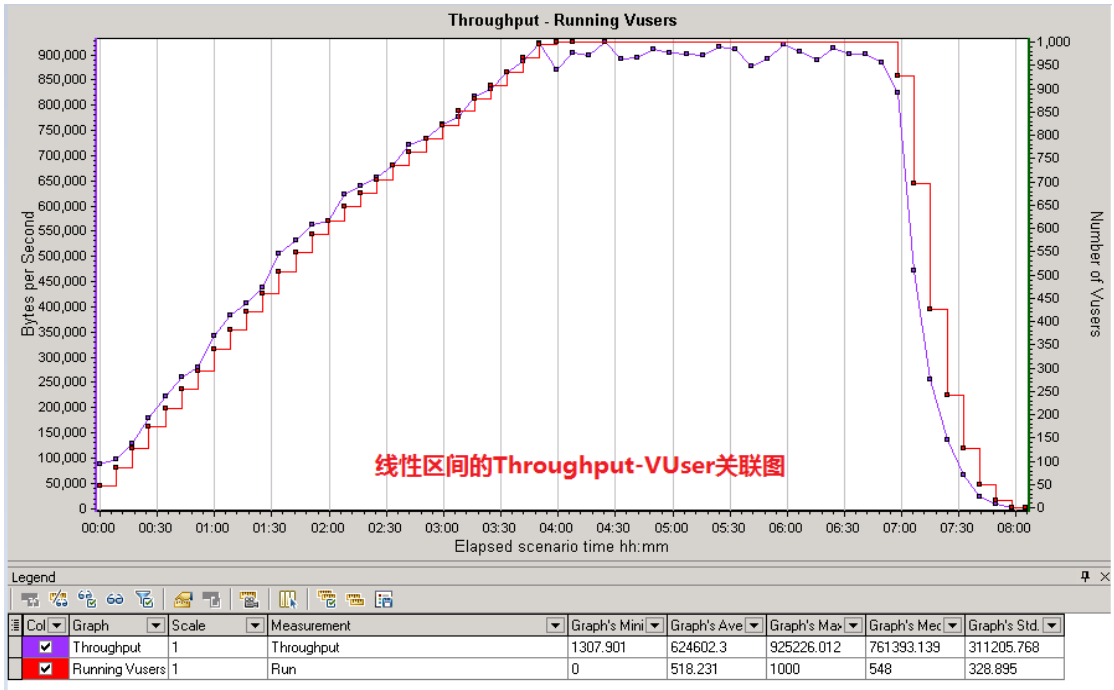
除非是针对前端的性能测试，软件的性能测试一般不关注“前端时间”，因为前端时间很大程度上取决于客户端本身的处理能力，同时可以使用一些编程技巧在数据尚未完全接收完成时进行呈现来减少用户实际感受到的主观响应时间。鉴于此，个人认为响应时间的标准定义视乎不尽合理，尤其对于“接收到最后一个字节”，因为在普遍采用“提前渲染”的技术的今天，实际用户感受到的的响应时间通常要小于标准定义的响应时间。用一个实际例子来讲可能更有说服力，一个网页的加载，如果过了 10 秒都白屏，用户一定感觉很慢，性能无法接受，但是大家肯定上过新浪网吧，当加载新浪首页的时候，大家应该都没有觉得非常慢吧，但实际情况是 sina 首页的加载时间要远大于 10 秒，只是由于 sina 采用了数据尚未完全接收完成时进行呈现的技术使用户主观感受到的时间大大缩短了。所以深究起来，响应时间应该包含两层含义，技术层面的标准定义和用户主观感受的时间。至于在性能测试过程中，我们会使用哪个层面的含义将取决于性能测试的类型。显然，对于软件服务器端的性能测试肯定会采用标准定义，而对于目前

越来越普及的前端性能评估，则会采用用户主观感受到的时间，当然我们在前端性能测试中，会利用一些事件的触发（比如 DOM-Load、Page-load 等）来客观地衡量“主观的前端性能”。

衡量吞吐量的不同视角

吞吐量是最能直接体现软件系统负载承受能力的指标。所有对吞吐量的讨论都必须以单位时间为基本前提，其实个人认为 Throughput 翻译成吞吐率更为贴切，因为我们可以这样理解：吞吐率=吞吐量/单位时间。但是通常情况下，我们不会刻意去区分吞吐量和吞吐率，统称为吞吐量。

对于性能测试而言，通常用“Requests/Second”、“Pages/Second”、“Bytes/Second”来衡量吞吐量，当然从业务的角度，吞吐量可以用单位时间的业务处理数量或者单位时间的用户人数来衡量。以不同的方式表达的吞吐量可以说明不同层次的问题。例如，“Bytes/Second”和“Pages/Second”主要受网络基础设置、服务器架构、应用服务器制约；“Requests/Second”表示的吞吐量主要



受应用服务器和应用本身实现的制约。

吞吐量和并发用户数之间存在紧密的联系。首先在没有到达性能拐点即没有出现性能瓶颈的时候，吞吐量往往和并发用户的数量成正比。这个很好理解：比如在一个测试场景中只有 10 个并发用户，每个用户每隔 1 秒发出一个 Request，此时服务器能够每秒处理完成 10 个 Request，那么服务器端吞吐量为(10*1)/1=10 Requests/Seoncd，接下来进一步假定现在用户数增加到 100 个并且没有出现性能

瓶颈，还是每个用户每隔 1 秒发出一个 Request，此时服务器依然能够每秒处理完成 100 个 Request，那么此时的吞吐量为(100*1)/1=100 Requests/Seoncd，继续当用户增加到 200 的时候，用户还是以 1 秒的间隔发送请求，但是服务器由于某些原因（过高的 CPU 占用、内存不足、I/O 过度频繁、资源死锁或等待等等）出现了性能瓶颈，此时服务器端每秒只能处理 150 个 Request，响应的响应时间也随之增大，此时的吞吐量显然不再是(200*1)/1=200 Request/Seoncd，也就是说一旦进入系统瓶颈区间后并发用户数不再和吞吐量成正比关系了，在极端情况下甚至会呈现反比关系，即用户数越多，系统响应时间变得非常长，吞吐量急剧降低，这是我们在做系统容量规划时必须避免的。通常在系统容量规划测试中，常采用“Throughput-VUser”关联分析法。

其次，虽说吞吐量可以被看做服务器承受负载的体现，但是在不同并发用户数的场景下，即使系统具有相接近的吞吐量压力，但是得到的系统性能瓶颈结果却是不相同。比如某个测试场景中采用 100 个并发用户，每个用户每隔 1 秒发出一个 Request，另外一个测试场景采用 1000 个并发用户，每个用户每隔 10 秒发出一个 Request。显然这两个场景具有相同的吞吐量，都是 100 Requests/second，但是两种场景下得到的系统性能拐点肯定是不同的。