

# 测试人

Testers



本期导读：

- 从入门到精通
- 自动化测试
- 探索式测试
- 测试的万花筒



# Contents

## 卷首语

## 从入门到精通

软件测试入门 .....	<b>4</b>
究竟什么是软件测试 .....	<b>10</b>
测试有效性评估的优化 .....	<b>13</b>
测试管理艺术的未来 .....	<b>16</b>

## 自动化测试

千里之行，始于足下——自动化测试 ROI 实践 .....	<b>19</b>
无线页面转换系统抓取系统性能优化测试回顾 .....	<b>22</b>

## 探索式测试

2011 回顾之探索式测试实践 .....	<b>26</b>
思维成就测试--探索式测试实践篇 .....	<b>31</b>
测试的心理定势 .....	<b>35</b>

## 测试万花筒

从医学检测来看软件测试 .....	<b>38</b>
代码检视三步走 .....	<b>42</b>
接口验证模式 .....	<b>51</b>

## 卷首语

在俱乐部成立半年后，我们迎来电子杂志《测试人》的问世，可喜可贺！

当初建立软件测试俱乐部就是为了给测试人员提供一个交流的平台，此后搞了几次软件测试技术沙龙，但由于大家工作忙，加上物理空间的限制，参加沙龙的人数还是有限的。于是，我们想还能软件测试界做一点什么事呢？或者说，如何更好地发挥俱乐部的作用呢？于是，2011年12月11日上午，俱乐部的主要成员聚集在一起，商量做点什么事。那我们首先就会问，软件测试业界目前存在哪些问题？有哪些需求？分析问题，找出需求，然后，我们就知道做什么事情，或者采用什么方法可以解决软件测试业界中存在的某突出的问题，至少满足大多数测试人员的需求。基于需求，我们的工作才有价值。

为此，我们就开始认真分析测试人员的需求，在头脑风暴之后，将大家想出来的各种需求列了出来，即归为下面8个方面：

- 1) 软件测试人员的职业规划
- 2) 软件测试工作入门指导
- 3) 在组织层面上实施敏捷方法所带来的困惑
- 4) 软件测试工程师的积极性
- 5) 如何将测试理论与实践相结合
- 6) 测试过程的改进
- 7) 测试工具与测试自动化的实施或落地
- 8) 技术交流的平台或环境

再后来，我借助新浪微博的投票功能，做了一次调查（<http://vote.weibo.com/vid=1243341>），投票结果显示和我们当初讨论的结论也基本一致。“测试人员的职业发展比较困惑”排在第一，而“如何将测试理论和实践结合起来”、“自动化测试的 ROI 问题”等也是测试人员困惑的地方。

有了这些问题，就可以进一步展开讨论我们能做的事情，例如可以写书、建网站、举办技术沙龙、举办测试技术大会、开展测试培训、办电子杂志以及可以测试咨询。将需求和可做的事情结合起来，就可以构成一个矩阵，从中比较容易确定做哪件事是收益比较大的或对测试人员帮助最大的，又相对比较容易实施的。最后，大家认为办一份电子杂志是一个不错的选择。

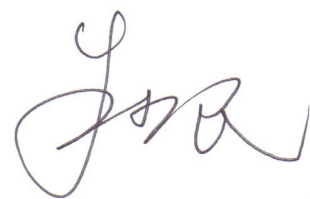
办杂志，首先得取一个好名字，再次发挥微博的作用，发起了另一个投票：<http://vote.weibo.com/vid=1274307>，最终确定杂志“测试人”。至此，事情并没有结束，而仅仅才开始，然后征集稿件、评审、再评审、排版，大家忙到不亦乐乎。

首期《测试人》终于和大家见面了，它还是襁褓中婴儿，需要大家的呵护和支持。希望它一天天长大，逐渐得到大家的喜欢，也为国内软件测试业的发展尽一份绵薄之力。

不耽误大家的时间了，你们可以翻到自己喜欢的文章，开始快乐的阅读历程吧！



软件测试俱乐部轮值主席





# 软件测试入门

**作者：郑银华** 毕业于成都电子科技大学。从事软件开发、测试方面的工作 10 多年，目前在上海科泰世纪科技有限公司担任软件测试主管。

Email: [zzyyxxhh@sina.com](mailto:zzyyxxhh@sina.com)

## 编者之话

亲爱的朋友，本文主要是针对打算进入软件测试领域的朋友们编写的。在我的面试经历中，我经常会遇到很多刚走出大学校门的朋友。他们都非常优秀，但是由于各种原因，例如因经济原因无法参与专业培训、短期无法找不到工作而准备放弃在大城市奋斗等，在职业选择方面非常迷茫。我非常希望能将自己的经验跟大家分享，如能对年青朋友们有一些帮助，我就非常高兴。感谢你们阅读本文，谢谢！

## 职业发展

“你为什么选择软件测试？”面试时很多人回答：“因为软件测试简单”。这样的回答其实很糟糕。如果你是真心喜欢、热爱这个行业，再加上你的认真、踏实、负责、以及良好的团队合作等，恭喜你，不管你的计算机基础如何，你都能在软件测试行业有很好的发展前途！

## 入门

当你决定进入软件测试行业，若你对软件测试还不太了解，我建议你去书店选择软件测试相关书籍学习几天（特别提醒：对于重点知识、疑问等需做好笔记，并及时查阅资料将疑问解决掉）。经过这一阶段的学习，你可以知道哪些书写得比较好，可以从中买下 1-2 本书带回家仔细研究！

## 面试

恭喜你获得面试机会。这时候，你应该真诚、勇敢地参加面试。面试的时候，眼光请一定要正视考官，把你自信、优秀的一面充分展现出来。

## 入职

恭喜你进入软件测试行业。通常，通过一个月左右的时间熟悉、学习业务知识，如果你能顺利地把测试理论知识很好地应用于实际工作中，并按时完成上级安排的测试任务，到第二、三个月时你就基本具备独立执行测试任务的能力了。我相信，你一定能顺利转正。

## 测试用例设计

理论与实践相结合是非常重要的。不知道其他公司对测试用例设计如何看待，而我始终是特别重视的。



对于踏入这个行业的新人，我通常会花一周左右的时间对他们进行测试用例设计方面的培训，重点指导新人们如何将理论用于实践。

## 测试用例模板

我相信每家公司都有自己的测试用例设计模板。我采用的测试用例设计模板主要包含：

- **最小功能测试集**: 用于简单、快速地验证系统是否满足基本的功能需求（最小功能集最好能够做到全部自动化）；
- **复杂功能测试集**: 用于进一步验证系统能否在复杂、或不常见的合法输入和操作下正常运行；
- **健壮性测试集**: 用于测试系统能否在各种异常输入、异常操作或者异常环境下正常响应，以及检测在出错之后系统能否正常运行，是否造成数据丢失、是否毁坏其它相关的软件和硬件等；
- **UI 测试集**: 编写跟 UI 设计相关的测试集。

模块(层次)划分				优先级	测试简述	前置条件	运行过程	预期结果	测试结果
test	test1	test1-1	test1-1-1	P0	TEST1	ww1	1. TEST1	1X	
			test1-2	P1	TEST2	ww2	1. TEST2	2X	
		test1-2	test1-2-1	P2	TEST3	ww3	1. TEST3	2X	
			test1-2-2	P3	TEST4	ww4	1. TEST4	4X	
			test1-2-3	P2	TEST5	ww5	1. TEST5	5X	
	test2	test2-1		P0	TEST6	ww6	1. TEST6	6X	
		test2-2	test2-2-1	P1	TEST7	ww7	1. TEST7	7X	
			test2-2-1	P0	TEST8	ww8	1. TEST8	8X	
				P1	TEST9	ww9	1. TEST9	9X	

说明：

最小测试集、复杂测试集、以及健壮性测试集都是根据需求、使用测试用例设计方法编写的。UI 是根据产品 UI 设计文档编写的。

在编写测试用例的时候，需要思考以下几个问题：

- **为什么功能性测试用例必须覆盖全部需求？**  
这问题不回答了，大家一定理解。
- **哪种测试用例便于他人审核是否有效？哪种测试用例便于增加、删除、修改？**  
具有树型结构、清晰层次关系的测试用例。审核人员一般会先审核树枝是否全面覆盖需求、是否有冗余，然后再审核树叶是否全面、是否有冗余。如果具有这样的层次关系，用户也能很好地维护测试用例。
- **哪种测试用例便于多项目共用？为什么要将功能与 UI 测试测试集分开？**

在测试用例设计中，将功能与 UI 测试用例分开，这样对于功能相同的需求，功能性测试用例就可以在多个项目中通用。为了功能性测试用例能够在多项目中通用，功能性测试用例需要使用通用词语描述。UI 用例应该只描述各产品 UI 的一些约束部分，参考后面电话模块测试用：当电话拨号盘没输入号码，键盘“灰显”等，这约束跟具体项目有关，属于 UI 用例。





## 需求模块划分

在设计测试用例前，充分理解需求是非常必要的。在此基础之上再对需求进行模块划分，形成一棵需求树（说明：划分模块的时候，需求可以重复。但重复不宜太多，否则需要思考划分的模块是否合理？）。

电话模块需求树例子：

大模块	子模块	功能	需求编号	是否需要完成
呼叫相关	编辑电话号码	设置 IP 号码	Phone_Req_ID10	Yes
		国际冠码 +	Phone_Req_ID07	Yes
		暂停 p	Phone_Req_ID08	Yes
		编辑	Phone_Req_ID12	Yes
		紧急呼叫	Phone_Req_ID36	Yes
		保存输入号码到地址簿	Phone_Req_ID37	Yes
		快速拨号		
	新呼叫	从地址簿拨号	Phone_Req_ID5	Yes
		从最近通话拨号	Phone_Req_ID6	Yes
		设置 IP 拨号	Phone_Req_ID10	Yes
		通话中执行其他应用	Phone_Req_ID17	Yes
		自动重拨开关	Phone_Req_ID19	Yes
		呼叫等待	Phone_Req_ID25	Yes
		电话会议	Phone_Req_ID28	Yes
		.....		
	新呼叫号码识别相关	拨打电话图片、号码识别	Phone_Req_ID1	Yes
通话中	通话中	通话中提示音	Phone_Req_ID16	Yes
		免提通话	Phone_Req_ID23	Yes
		通话保持控制	Phone_Req_ID24	Yes
		呼叫等待	Phone_Req_ID25	Yes
		选择接通或挂断等待的呼叫	Phone_Req_ID26	Yes
		多方通话	Phone_Req_ID27	Yes
		.....		

（未完，见下页续表）

(续表)

大模块	子模块	功能	需求编号	是否需要完成
来电相关	来电相关	黑名单	Phone_Req_ID13	Yes
		白名单	Phone_Req_ID14	Yes
		通话保持控制	Phone_Req_ID24	Yes
		.....		
	来电号码识别相关	来电图片号码识别	Phone_Req_ID03	Yes
		来电铃声识别	Phone_Req_ID04	Yes
		来电地区显示	Phone_Req_ID43	Yes
	来电提示相关	拒听	Phone_Req_ID20	Yes
		接听	Phone_Req_ID21	Yes
		.....		
通话结束	通话结束	通话报告	Phone_Req_ID15	Yes

## 根据需求编写测试用例

基于需求的模块划分结果，结合边界值、等价类等测试用例设计方法，根据测试用例设计模板，编写功能性测试用例，即编写基本功能、复杂功能、健壮性测试用例。

**注意事项：**

1. 理解测试用例设计方法特别重要。常用的测试用例设计方法有等价类、边界值等，建议大家能深入理解，针对不同类型的需求就可以选择一种或多种适宜的测试用例设计方法编写测试用例。
2. 建议每个测试目的下的测试用例不超过 10 条。如超过 10 条，需要再提出一层。这样做的目的，是便于自己与他人审核，因为单个目的下的测试用例如果太多，容易导致审核人员的思路混乱，从而很难对测试用例提出有效的改进意见！







## 完成测试任务

在大多测试职位中，通常都会提交 Bug。Bug 质量和描述都很重要，建议：

- Bug 简述应一目了然，不能含糊，长度不得超过 30 个字。
- 提交的 Bug 应用客观的书面语，避免使用口语。
- 测试中一旦发现 BUG，需要及时提交 BUG。
- 在提交 Bug 前，应查询库里已有的 Bug，防止同样的 Bug 重复提交。
- 优先级别、严重性级别、重复性定义尽可能准确。
- Bug 描述中千万不要有错别字，在细节处也都要随时体现质量人员的素质。

总之，按时、高质量地完成安排的任务，这是最重要的。在这一过程中，如遇上问题，需要及时想办法解决；若自己无法独立解决，应及时请教别人。总之，一定要准时、高质量完成任务，如无法完成，一定要提前报告给你的上级。

## 你做到了吗？

进入公司后，能够快速熟悉公司文化、开发及项目流程，并融入其中。转正前能够达到：

- 熟悉基本测试理论，熟悉业务标准，能很好地运用测试理论知识，独立编写测试用例设计。
- 熟练运用必要的测试工具。
- 独立、按时完成测试任务。



## 总结

亲爱的朋友，不知道这些内容对你是否有帮助？我只想告诉你们，不管遇上何种困难，只要有信心，努力后一定可以解决的。我的一位老师曾经在我困难的时候说，可能这个世界从来都不是公平的，有的人生下来就拥有很多，而有的人注定要非常努力后才能获取那么一点点，但是永远别失去信心，相信自己努力后，明天一定比今天好！感谢曾经给我指导、帮助的朋友！

## 究竟什么是软件测试?

**作者:**朱少民 CSTQB 专家工作组成员; 同济大学软件学院教授, Certified ScrumMaster、中国科技大学软件学院教指委委员。从事软件开发、测试、QA 和过程改进等工作近二十年, 在软件工程领域有很高的造诣, 在软件测试流程改进、自动化方法和测试管理等方面进行了大量探索和实践, 提倡“全过程软件测试”和“缺陷预防”等先进的软件工程思想, 先后出版专著《全程软件测试》、《软件测试》、《软件工程导论》和主编《软件测试方法和技术》、《软件项目管理》、《软件质量保证和管理》、《软件过程管理》等多部高等学校精品教材。微博: <http://weibo.com/kerryzhu>



“什么是软件测试”是一个最基本的问题, 但凡从事软件测试的人都能回答这个问题, 而且多数测试人员的答案不外乎是下面两种叙述的一种:

软件测试是为了发现程序或软件系统中问题而进行的一系列活动。

软件测试是为了验证程序或软件系统是否符合设计要求、是否符合满足客户的需求。

这并没有错, 也就是软件测试认识的正向思维和逆向思维的不同观点的表现:

证明软件是“不工作的”, 以反向思维方式, 不断思考开发人员理解的误区、不良的习惯、程序代码的边界、无效数据的输入以及系统的弱点, 试图找出各种各样的问题。毕竟开发人员力求构造软件, 所以开发人员的思维方式以正向为主, 测试人员的工作可以看做开发工作的一个补充, 逆向思维更为重要, 而且这样的工作效率也更高。

验证软件是验证软件是“工作的”, 以正向思

维, 针对软件系统的所有功能点, 逐个验证其正确性。这是传统工业的质检工作在软件业的自然延伸。

但仅仅这样理解软件测试还不够, 需要更全面的理解软件测试, 就可以更好地做好工作, 也可以适应不同的软件开发过程所带来的挑战, 包括敏捷方法带给软件测试的极大挑战。

将近 30 年前, 在 G. J. Myers 的经典著作《软件测试之艺术》(The Art of Software Testing) 中, 给出了测试的定义: “程序测试是为了发现错误而执行程序的过程”。那时对软件测试的认识还非常具有局限性, 这也是受软件开发瀑布模型的影响, 认为软件测试是编程之后的一个阶段。只有等待代码开发出来之后, 通过执行程序, 像用户那样操作软件发现问题, 这就是“动态测试”。

如果在此时发现功能设计不合理或性能不好, 就需要修改需求或修改设计, 那就不得不返工到需求定义或系统设计阶段, 造成很大的代价。所以, 有必要将软件测试延伸到需求、设计阶段, 对需求、设计进行验证, 因为需求定义文档、设计的技术文档也都是产品的组成部分, 即阶段性的成果输出, 只要有输出就需要验证。通过需求评审、设计审查活动, 可以更早地发现问题, 及时地修正问题。这就是静态测试, 即对软件开发过程中阶段性产品进行互为评审、走读或会议评审, 发现其中的问题。通过静态测试, 可以更早地发现问题, 把问题消灭在萌芽之中, 将每个阶段产生的缺陷及时清除。所以, 通过需求评审、设计评审和代码评审等, 能够提高产品的质量, 并能大大降低企业的成本。

软件测试是由“静态测试”和“动态测试”构成，不仅仅对软件的主题——程序进行检验，还有对阶段性半产品也要进行检验，确保软件开发过程中的输出都能满足要求。所以，软件测试是贯穿整个软件开发生命的质量保证活动，是软件质量保证的重要手段之一。从本质上看，软件测试还是事后检验，是对产品（包括阶段性产品和最终产品）的检验，也就是说对产品质量的评估。通过对软件产品的各个质量属性（功能、性能、安全性、兼容性、易用性等）进行检查，从而了解软件产品在这些方

面满足事先设定要求的程度。从这个意义上看，软件测试是对软件质量的全面评估，以决定软件产品能否发布出去。而且，通过对软件质量的全面评估，了解、确定开发人员容易犯的错误，为软件开发过程的改进提供依据。贯穿整个软件开发生命的软件测试活动，就可以看作对软件产品质量的持续评估。所以，在敏捷方法中，不仅提倡持续集成，而且提倡持续测试，就是持续的质量评估，如图 1 所示。更重要的是，持续集成是为了持续测试、持续的质量评估。

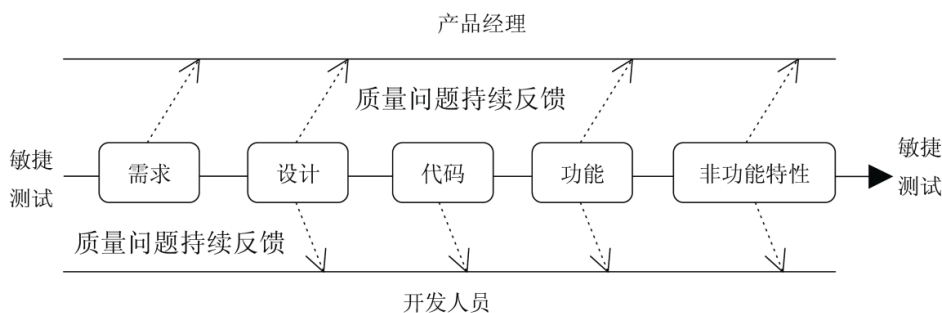


图 1 软件测试是持续的质量评估

同时，我们知道“不能穷举测试”，即对复杂的或大规模的软件系统，测试的覆盖度不能做到 100%，即使能够百分之百地覆盖代码、分支和条件，也不能百分之百地覆盖路径组合、不能覆盖各种输入数据、各种使用条件和环境的组合等。所以，软件测试总是存在风险的，软件的风险还来源于需求不明确、标准不够充分。从风险角度看，软件测试是找到最大的风险、主要的风险，从而消除这些风险。举一个例子，当你的经理给你安排一个测试任务时，按照实际情况和高质量的要求，要完成这个任务需要两周时间，现在你的经理只给你一周时间。这时，你当然可以争取更多的测试时间，但是如果你的经理说，时间不能商量，只能给一周时间，而且也没有更多的人可以给。这时，你也无需拒绝任务，也无需加班，而只是设法缩小测试范围，按

照测试优先级来决定测试的工作，并将更大的风险告诉经理。也就是说，两周的测试会消除更多的风险，包括非常小的风险，达到非常高的质量水准。而一周的测试会增加风险，但我们可以消除绝大部分的主要风险，只是承担一些比较小的风险，也能达到较高的质量水准。在基于风险的测试概念下，首先，我们要评估测试的风险，功能出问题的概率有多大？哪些是用户最常用的 20% 功能——Pareto 原则（也叫 80/20 原则）？如果某个功能出问题，其对用户的影响有多大？然后根据风险大小确定测试的优先级。优先级高的测试，优先得到执行，一般来讲，针对用户最常用的 20% 功能（优先级高）的测试会得到完全执行，而低优先级的测试（另外用户不经常用的 80% 功能）就不是必要的，如果时间或经费不够，就暂时不做或少做。

“一个好的测试用例是在于它能发现至今未发现的错误”，体现了软件测试的经济学观点。实际上，软件测试经济学问题至今仍是业界关注的问题之一。经济学的核心就是要盈利，盈利的基础就是要有一个清楚的商业性目标。同样，商业性目标



是否正确，直接决定了企业是否盈利的结果。多数情况下，软件测试是在公司内的执行。正是公司的行为目的，决定了软件测试含义或定义的经济性一面。正如，对软件质量的定义不仅仅局限于“和客户需求的一致性、适用性”，而且要增加其它的要求——“预算内、按时发布、易于维护”。软件测试所花费的成本可以看作是保障性成本，而由于软件缺陷引起的成本是劣质成本，从经济观点来看，当保障性成本小于或远远小于劣质成本，软件测试的工作才有意义。所以，软件测试也不可能无止境地进行下去，也是适可而止，即软件测试的目标是“足够的好（good enough）”，而不追求完美。这一点，在敏捷开发中、在互联网软件行业体现得更为充分。

从标准论来看软件测试，可以定义为软件测试

就是“验证（Verification）”和“有效性确认（Validation）”活动构成的整体，即软件测试 = V&V。

“验证”是检验软件是否已正确地实现了产品规格书所定义的系统功能和特性。验证过程提供证据表明软件相关产品与所有生命周期活动的要求（如正确性、完整性、一致性、准确性等）相一致。相当于，以 Spec 为标准进行软件测试活动，验证软件产品和 Spec 的一致性。

“有效性确认”是确认所开发的软件是否满足用户真正需求的活动。相当于，保持对软件需求定义、设计的怀疑，一切从客户出发，理解客户的需求，发现需求定义和产品设计中的问题。这主要通过各种软件评审活动来实现。

究竟什么是软件测试呢？综上所述，软件测试是贯穿整个软件开发生命周期、对软件产品（包括阶段性产品）进行验证和确认的活动过程，是持续的质量评估过程，其目的是尽快尽早地发现在软件产品中所存在的各种问题，尽最大可能消除软件质量问题所带来的风险，使企业获得良好的竞争力和效益。

## 测试有效性评估的优化

**作者：郑文强** CSTQB 专家工作组成员；ISTQB 高级测试经理证书获得者，著作：《软件测试管理》等。先后在中兴通讯、阿尔卡特、朗讯等大型通讯/软件企业从事软件测试、软件测试管理、软件质量管理以及软件测试过程改进等工作。精通软件测试生命周期各个测试阶段的控制和管理，熟悉各种软件测试技术和方法，以及对测试能力改进有深入的研究和丰富的实践。

联系方式：E-mail/MSN: zwqwuy@hotmail.com

缺陷检测百分比 DDP (Defect Detection Percentage) 常常可以用来评判软件测试生命周期内某个阶段的测试有效性。但是，在实际评判测试有效性过程中，DDP 也存在某些方面的欠缺。本文通过分析 DDP 的基本原理，笔者提出了更适合测试有效性评估的优化 DDP (简称 O-DDP)。

### 1) DDP

穷尽测试是不可能的，因此，不可避免有一些缺陷会遗漏到客户的使用现场，从而触发软件产品产生令用户不满意的失效或者各种问题。通常来说，在测试过程中判断测试人员的测试有效性是很困难的。但是通过用户反馈的缺陷数目，却可以直观的说明测试人员是否遗漏了比较多的问题，从而反映测试人员的测试有效性。缺陷检测百分比 DDP (Defect Detection Percentage) 就是基于这样的目的进行定义的，它可以用来评判软件测试生命周期内某个阶段的测试有效性，它以百分比的形式进行计算[1]。其计算公式为：

$$DDP = [R1 / (R1 + R2)] * 100\%$$

其中：

- ° R1 指的是被评估阶段发现所发现的缺陷数目；
- ° R2 是被评估阶段之后所发现的所有的缺陷数目；

DDP 计算公式的分母并不是发现的总的缺陷数目，而是指该阶段之后所有发现的缺陷数目（包括该阶段的缺陷数目）。表 1 是计算 DDP 的一个例子（主要针对动态测试而言的）：

表 1 DDP 计算的例子

测试阶段	发现的缺陷数目	DDP
集成测试	376	51.58%
系统测试	251	71.10%
验收测试	65	63.73%
产品发布（3 个月之内）	37	NA



采用 DDP 作为测试有效性的评估指标，主要存在两个问题：第一个是 DDP 无法在测试过程中对测试有效性进行评估，必须要等待产品发布一段时间之后才能进行，例如：产品发布之后 3 个月；另一个问题是 DDP 计算过程中只考虑了用户反馈的缺陷的数目，而没有考虑每个缺陷的严重程度；单纯的缺陷的数目，并不能正确的反映遗漏到客户的缺陷对用户造成的影响程度。而测试的有效性，要求测试人员尽早发现尽量多的严重的问题，而不仅仅是尽量多的缺陷数目。

对于 DDP 的第一个问题，由于 DDP 本身的定义要求，无法提供好的建议来解决该指标对测试有效性评估的延后性。而对于第二个问题，本文将综合考虑用户反馈的缺陷数目和严重程度，对 DDP 计算实现优化，从而更好的评估测试有效性。

## 2) O-DDP

O-DDP 的基本原理和 DDP 是一样的，只是在计算过程中，同时考虑了缺陷的数目和缺陷的严重程度。根据笔者所在组织和项目的特点，其定义的缺陷严重程度如下[2]：

**严重程度 1（致命的）：**产品在正常的运行环境下无法给用户提供服务，并且没有其他的工作方式可以补救；或者软件失效会造成人身伤害或危及人身安全；

**严重程度 2（严重的）：**极大地影响系统提供给用户的服务，或者严重影响系统要求或者基本功能的实现；

**严重程度 3（一般的）：**系统功能需要增强或存在缺陷，但有相应的补救方法解决这个缺陷；

**严重程度 4（轻微的）：**细小的问题，不需要补救方法或对功能进行增强；或者操作不方便，容易使用户误操作；

**不同的缺陷严重程度，**其定义的权重分别为  $w_1 = 10$ 、 $w_2 = 4$ 、 $w_3 = 2$  和  $w_4 = 1$ ，分别对应严重程度 1（致命的）、严重程度 2（严重的）、严重程度 3（一般的）和严重程度 4（轻微的）。

下面通过案例的阐述，说明 DDP 和 O-DDP 计算得到的不同结果，以及根据结果分析得到的不同结论。在该案例中，评估的目标是系统测试的有效性，因此采用的缺陷数目分别来自系统测试发现的缺陷和在产品发布 3 个月之内用户反馈的缺陷。表 1 是根据 DDP 得到的结果：

表 1 DDP

阶段	缺陷数目	DDP
系统测试	351	89.54%
产品发布（3 个月之内）	41	NA

我们从表 1 中得到的  $DDP = 89.54\%$  结果来看，系统测试的有效性应该是非常不错的，因为遗留到用户现场的缺陷数目相对比较少。但是，在我们进行具体详细的分析用户反馈的缺陷的时候，发现用户对该产品的质量严重不满意。他们所反馈的缺陷，很多都是严重影响他们正常使用的问题。因此，上表中的 DDP 结果并不是让人信服的。



而后我们引入 O-DDP，重新进行测试有效性的评估，其计算公式和 DDP 的公式一样，只是其中每个参数的计算进行了优化：

$$O-DDP = [R1 / (R1 + R2)] * 100\%$$

其中：

R1 是系统测试阶段发现的缺陷和它们不同严重程度权重乘积之和，具体计算公式是  $R1 = m1*w1 + m2*w2 + m3*w3 + m4*w4$ ，其中 m#指的是系统测试阶段不同严重程度的缺陷的数目；

R2 是产品发布之后在用户现场发现的缺陷和它们不同严重程度权重乘积之和，具体计算公式是  $R2 = n1*w1 + n2*w2 + n3*w3 + n4*w4$ ，其中 n#指的是系统测试阶段不同严重程度的缺陷的数目

得到的详细数据如表 2 所示：

表 2 O-DDP

阶段	严重程度 1	严重程度 2	严重程度 3	严重程度 4	O-DDP
权重	10	4	2	1	
系统测试	21	57	211	62	80.31%
产品发布（3 个月之内）	11	28	2	0	NA

### 3) O-DDP 和 DDP 比较

DDP 的计算仅仅考虑了遗留到用户现场的缺陷的数目，完全没有考虑不同缺陷对用户的影响程度。而 O-DDP 的计算兼顾了缺陷的数目和缺陷的严重程度，相对于 DDP 而言，在判断测试人员的测试有效性方面 O-DDP 更加全面。

根据表 1 得到的 DDP = 89.54%，而表 2 计算得到 O-DDP = 80.31%，其中的差距差不多是 10%。这里面说明了什么？我们可以对此进行简单的分析：

第一个可能是测试人员进行系统测试的时候，发现的严重程度为 1 和 2 的缺陷占的比重较低；

第二个可能是在用户现场发现的缺陷严重程度为 1 和 2 的缺陷占比比较高；

不管是哪种情况，都应该引起测试人员的注意：测试人员在系统测试过程中遗漏了某些重要的缺陷，特别是用户特别关注的受到影响的缺陷。

参考资料：

[1] Dorothy Graham, Grove Consultants, [www.grove.co.uk](http://www.grove.co.uk)

[2] 郑文强，马均飞，软件测试管理，电子工业出版社，2010-7

## 测试管理艺术的未来

**作者：熊晓虹** CSTQB 专家工作组成员；上海滔瑞信息技术有限公司技术总监 CSTQB 专家组成员二十多年通讯业测试和管理经验专注于软件测试管理的理论与方法。

电子邮箱：[xiongxxh988@gmail.com](mailto:xiongxxh988@gmail.com) 微博：[weibo.com/xiongxiaohong](http://weibo.com/xiongxiaohong)

概述：什么是测试管理的艺术？在物联网、云技术、移动互联网的兴起发展，三网合一成为大趋势的未来，测试管理艺术又将何去何从呢？本文旨在与对测试管理感兴趣的同仁进行探讨。



名家名言

艺术不是你所看到的东西，而是你让别人看到的东西。

——埃德加 德加 (Edgar Hilaire Germain de Gas)

### 什么是测试管理的艺术？

一提到艺术我们马上就会想到绘画、雕塑、戏剧、建筑、舞蹈、诗歌等等，但在这里我们要讨论的，是关于测试管理的艺术。首先我们来看，什么是测试？ISTQB®为测试做了如下定义：

测试是一个过程，它包括了软件生命周期的所有活动，有静态的也有动态的。它涉及到计划、准备和对软件及其相关工作产品的评估，目的是

- 判定软件或软件的工作产品是否满足特定需求；
- 证明它们是否符合目标；
- 发现缺陷。

但是什么时候做测试？是在产品将要完成的时候来做还是从产品需求定义的时候就开始做？实际经验又告诉我们，如果在产品将要完成的时候再做测试那么就太晚了，预防缺陷远比发现缺陷耗费的费用和时间少的多。所以，测试的目的应该是：

- 预防缺陷；
- 提供与产品质量相关的信息和信心；
- 发现缺陷。

### 什么是管理？

管：为了达成某一目的，行使一定的权力，组织分配人员执行任务。

理：在目标实现的过程中，控制过程，使其条理化、有序进行。

测试管理(manage)就是制定计划、执行计划、检查和改进过程从而达到测试目的的一切方法和活动。制定计划（或规定、规范、标准、法规等）是设计达到目标的路径，将整体的大目标分成一个个阶段性的小目标，确定实现阶段性目标所需要采取的战略措施，部署相应的人力、物力、规定走向目标时应该遵循的规范、标准、法规和过程等；执行就是按照计划去做，即实施；检查就是将执行的过程或结果与计划进行对比，总结出经验，找出差距；改进首先是推广通过检查总结出的经验，将经验转变为长效机制或新的规定；再次是针对检查发现的问题进行纠正，制定纠正、预防措施，以持续改进。

测试管理的艺术就是创造管理方法和技巧，创造性的运用管理方法和技巧实现测试的目的。它应该是基于实践的，与时俱进的，同时也是感性的，反映人类内心的情感和诉求，反映对理想的追求。因为只有这样的艺术才会有生命力。

回顾国际上的管理学艺术之路，我们可以看到管理学经历了两大阶段：

## 第一阶段：从行为科学到战略管理

从个体行为到组织行为(1956—1965)

从组织中的人到人的组织(1966—1975)

从过程管理到战略管理(1976—1985)

## 第二阶段：从组织变革到知识管理

从职能组织到变革组织(1986—1995)

从组织管理到知识管理(1996—2005)

回顾软件测试的目的演变，我们可以看到如下的脉络：

以调试为主（从有软件开始-1956）

证明程序是正确的（1957 - 1978）

证明程序中有错误（1979 - 1982）

评估产品能力（1983 - 1987）

预防缺陷（1988 - 1992）

预防缺陷，发现缺陷，评估质量（1992 - ）



管理理念方法和技巧都是以目标为导向的。当我们的目标发生了变化的时候，管理的艺术也随之得到了发展。我们可以清楚的看到管理艺术随着测试目的的变化而变化，在有一个时间上一一对应（或者略带滞后）的关系。

比如在测试目的从“调试”转换到“证明程序是正确的”时，也是管理艺术从个体行为到组织行为转变的过程。再比如，当测试的目的从“证明程序中有错误”改变为“评估产品能力”时，管理艺术也经历了从过程管理到战略管理的转换。

随着物联网、云技术、移动互联网的兴起和发展，测试管理也受到了空前未有的挑战，因为测试对象的开发规模，组织形式，应用范围以及对人类生活的影响都产生了前所未有的革命。如何创造管理方法和技巧，创造性地运用管理方法和技巧来适应这场革命成为我们必须面对的课题。

笔者以为，未来的测试组织和测试过程应该体现：效率 Performance，安全 Security，随时可取 Availability，灵活缩放 Scalability 的特性。

未来的测试管理应该是

- 多种软件生命周期的组合 - V 模型和敏捷开发敏捷测试的一体化；
- 多种测试组织形式的组合 - 内包、外包、研发测试人员角色互换，独立测试团队，第三方测试多种测试组织形式的一体化；
- 多种文化交融，超越地域分布，集目标管理、知识管理、人才管理、信息化管理为一体。

只有这样，我们才能与时俱进，适应新的形势发展，创造出新的测试管理艺术。

让我们听从内心的直觉，听从内心对美的呼唤和追求，一起去探索寻求 21 世纪新的测试管理艺术，并将这些艺术表现出来。因为正如法国古典印象主义画家埃德加·德加（Edgar Hilaire Germain de Gas）所说的：

“艺术不是你所看到的东西，而是你让别人看到的东西。”

**参考书目：**

**管理学图谱， 作者：陈悦，出版社：大连理工大学出版社，出版日期：2010-09-03**

**ISTQB 术语表 2007**



# 千里之行，始于足下

## —自动化测试 ROI 实践

作者：李岳梅

自动化测试是一项“一旦开始，就需要持续投入”的工作，所以它一直是测试领域的一块鸡肋。不做吧，好像手工测试重复得让人有些厌倦，而且手工测试时间也缩短不了。做吧，害怕投入的比回报要多。

没实施自动化的团队有各种各样的困扰。有的说：“项目有太多的老代码需要补充自动化测试脚本，补不起！”有的说：“项目开发太紧张，如果同时还要自动化，等不起！”还有的说：“自动化测试工具太贵了！买不起！”确实，各种各样的“伤不起”使得大量的组织在“要不要自动化”这个问题上总在了解和观望，踌躇不前。

我们阅读了一些关于自动化测试 ROI 的文章，发现大多都是介绍各种不同的计算方法，但来自实际的数据分享比较少。所以，2011 年当我们组织想推行自动化测试的时候，为了打消大家（尤其是管理层）对于自动化测试的投入和产出方面的疑虑，计算我们自己的自动化测试投资回报率 ROI (Return on Investment) 成了我们启动时就考虑的问题。本文将分为四部分介绍我们的实践方法和结果。

### 第一部分：业界计算自动化测试 ROI 的方法

简言之， $ROI = \text{收益} / \text{投入}$ 。但收益如何计算，投入包括哪些，众说纷纭，并没有一个定论。

在 Dion Johnson 的“test automation ROI”中给出了三种计算自动化测试 ROI 的方法。第一种方法“简单 ROI”着重从“钱”的方面去看。它考虑了工具、培训、机器等各种费用，并把测试时间的投入通过单位时间的工资转化成为钱。第二种方法“效率 ROI”与第一种方法不同的是从测试效率的角度，只考虑了时间投入所产生的收益，而没有考虑其它如购买工具方面的投入。这个方法比较适合测试人员计算收益。第三种方法“降低风险 ROI”着重计算自动化测试与手工测试相比在降低风险方面的收益。它会假设不做某种自动化测试，相关的风险一旦成为事实所带来的损失，从而计算 ROI。这个方法比较适合管理人员从整体考量自动化的收益。

那么，目前我们的团队期望自动化测试能带来哪些收益，尤其是哪些收益是目前不能奢望的？我们的经理愿意提供多少资源投入自动化测试呢？带着这些问题，我们开始了自己对自动化测试 ROI 的定义和度量。

### 第二部分：我们计算自动化测试 ROI 的方法

在度量自动化测试的收益方面，角度很多。我们选择的是从“多、快、好、省”四个方面去看。

#### 更多

鉴于我们处于自动化测试的初级阶段，我们打算暂时先不去追求“更多”。即我们不奢望一年之内整个项目组在一个版本里做更多的工作，因为在自动化投入初期难以提高团队的生产力。我们也不奢望测试人员马上能有更多时间去做更有价值的工作（相对于一次测试的多次重复执行）。因为测试人员通过自动化测



试从测试执行上节约出来的时间需要投入到自动化工具和技能的学习上去。

## 更快

在时间维度上，我们希望能够更快地发现和修复稳定的主流程上的明显的严重缺陷。如果一个测试人员手工测试多个功能，那么测试执行的并行度总有个上限。而多个并行执行的自动化测试脚本可以更快速地验证版本，一次性地报告问题。这尤其在测试初期版本不稳定，或者是每日构建的时候有用。有时，甚至是在我们不觉得有测试必要的时候，自动化测试可以及时报告刚引入的问题。另一方面，更快地发现缺陷也意味着可能可以更快地修复缺陷。

## 更好

我们希望自动化测试可以帮助我们实现对“更好”的追求，包括质量、信心、士气三个方面。

### 1. 更好的质量

更好的质量最容易被理解成为更少的缺陷。但这里需要强调的是“更少的缺陷个数并不仅仅能依靠我们基于界面的自动化测试来达到”。我们这里希望自动化测试能够帮助我们减少生产环境中某种特定类型的缺陷。这些缺陷包括环境或者配置相关的缺陷、在主流程上本来正常但因为后期修改影响到的功能、以及容易被忽略的地方（如：同一功能的多个入口、不常使用的功能）等。

### 2. 更强的质量信心

在内部测试中，我们希望借助自动化测试来提升的是对质量的信心。这主要体现在：（1）对于小版本和并行版本的质量更好地把关。小版本通常要求更快速的响应。并行版本通常要求测试人员频繁切换环境和被测对象。而人在压力下也更容易犯错。所以，我们常碰到的是匆忙中由于疏忽，一些比较重要或者明显的问题没有被及时发现。（2）对缺陷修复的质量更好地把握。根据统计，大约 7% 的缺陷修复会产生新的缺陷，而这些新缺陷有时会出现在前面已经测试过并且不会再手工测试的地方。对于如上两种情况，重复利用自动化测试脚本可以不需要额外的投入，快速得到关于整个版本稳定性的信息和质量信心。

### 3. 更高的士气

对于测试团队，我们希望自动化测试可以唤起更高的工作热情。这一方面来自于可以部分地将测试人员从大量重复的测试执行中解放出来，另一方面来自于新技术、新工具带来的新鲜感。开发团队和终端用户会是自动化测试的间接受益者，因为开发团队能感到问题会更快地暴露出来，终端用户会感到应用程序更稳定了。甚至在不远的将来，如果测试时间可以借力自动化而缩短，那么用户希望的功能也能更快地交付使用了。

## 更省

有了自动化测试，我们希望能省去以下工作：1、在每日构建后不需要手工验证版本的可测试性；2、在非需求（硬件、其它软件）变更的时候，尽量少的（甚至没有）手工主流程测试；3、在上线支持方面的不需要手工批量操作。

从上面的“多快好省”的分析中，我们明确了目前这个阶段我们希望从自动化测试中获得的主要收益，也发现了其中有些收益并不好度量。简单起见，我们决定记录可以量化的收益如下：

节省的测试人力：如果需要手工执行自动化测试案例覆盖的功能，那么需要多少人力。这个数据乘以自动化测试执行的次数，代表节约的手工测试人力。

发现缺陷的收益：对于自动化测试发现的缺陷，根据其发现的阶段设定不同的权重，并折算成它的风险收益。根据“持续交付”一书提到的理念，持续集成中“常红”或者“常绿”都是不正常的状态。类似地，我们认为自动化测试应该在验证版本基本正确性（绿）的基础上增加一些可能失败（红）的脚本/数据。因此我们将发现内部缺陷当作我们希望的自动化测试收益。



自动化测试的投入这一方面，因为测试工具已经购买而且是共享的，硬件方面也是利用已有资源，我们选择只考虑人力方面的投入，包括测试人员和开发人员一起投入的人力。因为开发人员有时会和测试人员一起解决自动化脚本的技术问题以及环境问题，如果其投入超过一定数量，我们将纳入计算。当然，测试人员的投入占绝大部分。为此，我们设计了一个表格，要求测试人员如实填写自动化测试的相关时间投入。除了时间，还需要记录其对应的类别，如团队学习（开会、培训）、个人学习（学习、研究）、测试用例设计、脚本开发与维护、环境等。做类别的区分主要是想看看剥离掉前期学习部分，每个版本在脚本维护方面的平均开销是多少。

### 第三部分：我们的结果

在首个半年的实施中，我们多个项目都实现了基于 QTP 的主要业务流程的自动化。我们的投入和收益实际情况如下：

	QA 人数	自动化测试投入 (man*hour)	自动化测试带来的 收益 (man*hour)
项目 1	3	160	40
项目 2	3	190	32
项目 3	2	161	33

从上述数据中我们可以看到自动化测试的收益并不高。这迫使让我们思考下一步如何才能获得更多的收益。而我们也马上产生了许多具体的想法。1、提高执行的次数。这可能需要我们把自动化测试和每日构建集成起来。2、在增加发现缺陷可能性方面，可以（1）利用现有的自动化测试脚本，但增加数据的多样性，这样脚本方面投入不大，但能增强发现缺陷的可能；（2）增加现有脚本的检查点，发现更多可能的缺陷；（3）分析缺陷，增加对容易聚集缺陷的相关功能的覆盖。3、优化脚本：对脚本的结构进行优化，提高复用性、灵活性、易维护性；加强脚本的稳定性和健壮性，提高其正确执行的概率。

接下来，我们尝试了自动化测试脚本和版本构建的持续集成，增加了测试数据的多样性，并随着项目的变化对原有脚本进行了必要的维护。与此同时，我们的项目也意外地碰到了多次硬件设备迁移，软件（操作系统、数据库、底层构架、第三方控件等）版本更新，以及小版本和并行版本的测试。此时我们都借助于自动化测试脚本，迅速地验证了版本，发现了一些缺陷，在项目组面临巨大的时间压力的时候提升了大家对质量的信心，项目经理开始纷纷表示对自动化测试的支持！自动化测试如同零存整取，平时挤一些时间去做，到了紧急需要的时候，那种雪中送炭的感觉真的很棒！

### 第四部分：结语

我们的自动化测试刚刚起步，度量的 ROI 结果也并不漂亮，但我们相信只要跨出了第一步，自动化测试的千里之行始于足下。



# 移动互联网页面转换系统抓取子系统一次性能测试总结

**作者: 王磊** 百度质量部高级架构师。拥有 10 年测试工作经验, 擅长性能测试、库函数测试以及网络测试。在 2005 年 5 月加入百度质量部后, 负责过百度众多产品的测试工作, 后组建通用组件测试组和基础架构测试部, 目前主要领域为技术改进以及敏捷测试相关工作。

微博: <http://weibo.com/cowleywang>

## 背景说明

手机应用越来越广, 其中使用手机浏览互联网是需求量较大的一类。手机浏览器因为手机资源的限制, 处理能力还达不到 PC 版本, 因此一些大的网站会有移动互联网专门的页面。而大部分网站只提供 html 代码的页面。为了可以在手机上也能够方便地看到网站, 提供了一个从 html 代码到 xhtml 代码转换的系统, 帮助手机用户浏览互联网资源。本次测试就是针对其中一个子系统进行性能优化测试。

## 测试目的

无线页面转换系统的抓取子系统, 可以简单的描述为 2 个程序, 一个类似于调度功能的程序 (简称为 sf), 完成接收上游模块下发的抓取具体 url 任务, 将该真正抓取任务下发给下游模块 (简称 bc), sf 程序还要提取 url 对应 html 代码中的外链 (比如图片、css 等), 发起二次抓取。bc 程序主要功能就是实际抓取页面, 抓取任务来源由 sf 模块下发。

上线之后, 随着业务量的增加, 抓取子系统不能满足线上的要求。但不能确定性能瓶颈是在 sf, 还是 bc。这次测试目的就要对比出性能瓶颈是在 sf 还是在 bc, 如果在 bc, 还需要定位一下具体的问题。

## 测试指标

性能测试指标的确定是和测试目的相关的, 针对本次测试目的, 是为了对比 sf 和 bc 的性能, 并找出可能存在性能差的原因。

对比性能差, 哪个指标最能代表这两个程序的性能值呢? 线上给人的感官是速度慢, 也就是抓取速度慢, 代表抓取速度的指标是每秒抓取网页数量。本次关注指标就是每秒抓取网页数量。为了消除页面大小对结果的干扰, 测试用的页面大小取线上平均页面大小。

## 测试分析

定位性能差的位置：

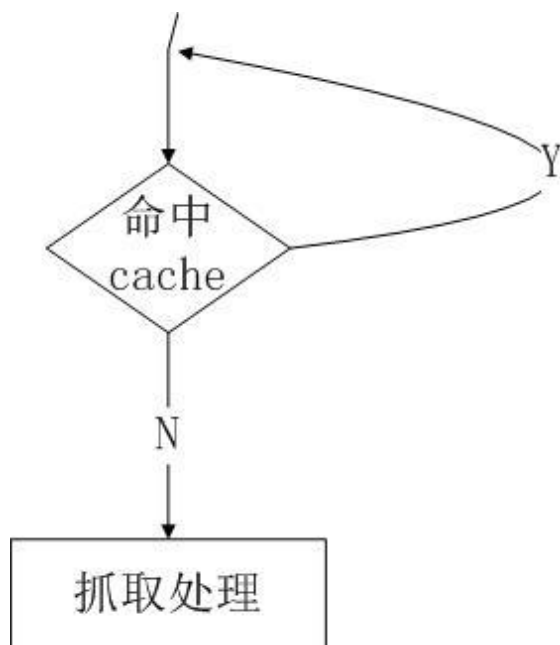
页面 cache：线上使用 2 层 cache 策略，但该 cache 使用 3 层会更有效。cache 使用方式可能会导致性能差原因。作为本次测试关注点。

配置项：长短连接、词表、线程数等，和线上保持一致。这个成为性能差的可能性很小，如果是通过一些配置项修改就应该能改善。配置项不作为测试关注点。

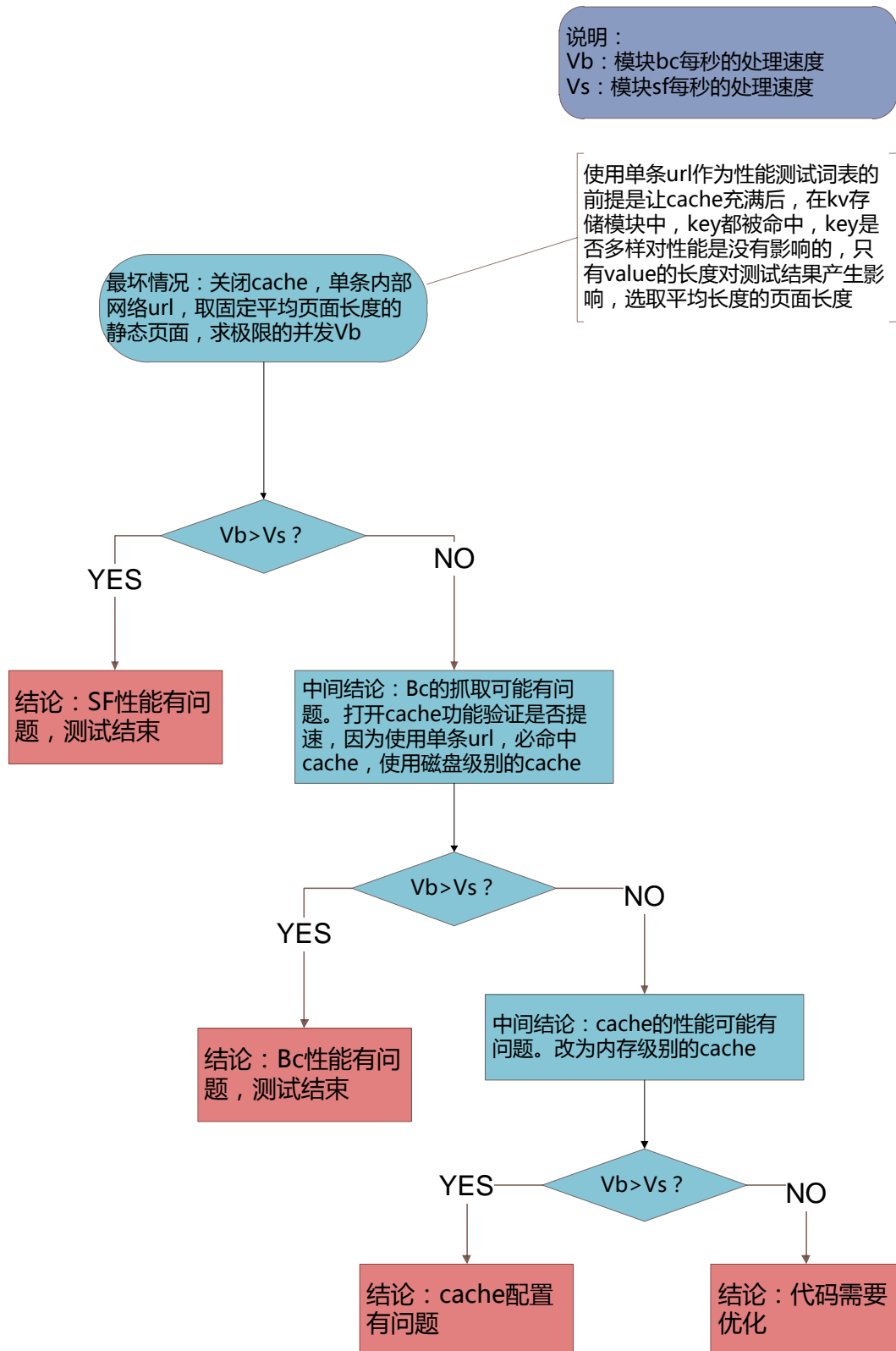
处理逻辑：程序在处理抓取流程上存在问题，从而影响性能。但这个太广泛，还无法在进一步细化原因。如果问题定位在这里，则需要另行分析。网络模型的问题，也归到此类。

网络环境：抓取速度也依赖于网络状况，这个在线上不是程序能控制的。本次测试不作为关注点。

与上述影响因素相关的程序处理图如下：



根据处理图做出测试分析如下：





## 测试环境与数据

网络环境：为了降低网络环境对测试结果的影响，抓取的页面置于内网环境。

压力词表：压力 url 中只含有一条 url。之所以考虑用 1 条 url，是因为 cache 的查找使用 hash 方式，随着 cache 容量的增长，查找速度不会变长。得出的性能结果与实际结果不会发生偏差。

## 总结

明确测试目的。测试目的发生偏差，就会使测试内容发生偏差，从而导致最终测试结果不能达到要求，性能测试结果被质疑，测试目的分析不准确也是原因之一。

通过对内部结构的抽象，得出性能测试的基础模型，针对模型产生用例，并且直接能够定位性能问题。

“在测试之前就可以确定测试执行次数，做性能测试中有一个经常被测试工程师提及的问题就是性能测试执行的次数太多，

反反复复的在做，也不清楚该做多少次。通过这个分析图可以看出来，在本次测试过程中，最多只做 3 次执行。

而且任何一个判断条件为 yes 的时候，都可以停止继续测试。”

## 探索式测试实践之缺陷大扫除和结对测试

**作者：高翔**（花名季哥） 淘宝软件有限公司资深测试工程师，曾任职于华为南京研究所和群硕软件有限公司。有着通讯、ERP、互联网等多种行业的测试经验，对需求分析、测试流程、测试设计方法、风险分析有较深的理解，擅长于测试模型的建立、用例架构的设计、公共组件功能的抽象和应用、探索式测试流程和方法实践。

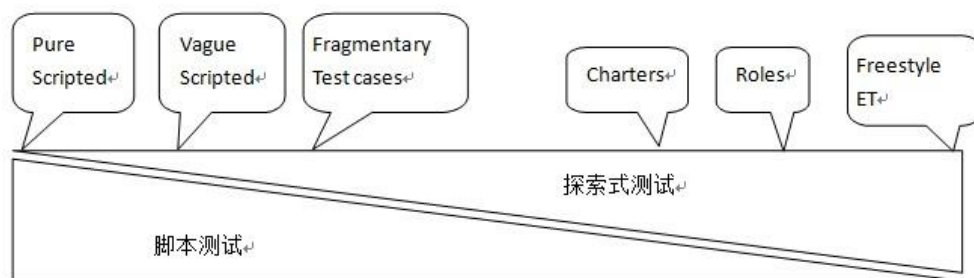
探索式测试的定义在我的 blog 都做了较多说明，其中也谈到了探索式测试在项目的实践方式，接下来会详细的说明其中两个实践方式的具体实施过程。

### 探索式测试四象限

探索式测试是一种测试风格和思考方式，它强调的是学习在测试过程中的作用。无论测试人员在做功能测试、性能测试、安全测试或其他类型的测试，都可以使用探索式测试的思维方法，来帮助自己找到初始测试设计未考虑到的危险区域。

探索式测试不只是在脚本测试后才开始，它可以应用于软件测试的各个阶段。作为一种测试风格，探索式测试可以使用适合当前情景的任何测试技术（包括脚本测试常应用的测试技术）。那为什么我们还要把探索式测试和脚本测试分开呢？这是因为探索式测试的优势来源于并行地实施测试学习、设计、执行和评估，来源于测试人员在此过程中的积极探索和主动调整，如果一再强调把探索式测试融合在脚本测试过程中，将不利于观察到探索式测试的内涵和潜在效果。

测试专家 James Bach 曾经对探索式测试 (ET) 和脚本测试 (ST) 在项目实践过程的变化进行了对比，请看下图



从严格的脚本测试到自由式的探索式测试，James Bach 做了如下解释：

最左边的 Pure Scripted，是严格的按照测试用例来执行测试，而且测试用例非常详细，在项目测试过程中较少存在这种现象。

最右边的 Freestyle ET，是自由式的探索式测试，在测试执行的时候不依赖于测试文档，只记录产品缺陷和风险除外；测试执行之前不需要任何特别的准备，比如测试设计。





这两种测试方式都是不常见的,有些走极端的倾向。在项目测试过程中,不可能完全采用 Pure Scripted 或 Freestyle ET。比较好的办法是在项目中混合脚本测试和探索式测试,并在不同的项目中采取不同的混合方式来制定项目测试计划。在大部分的项目测试过程中,综合脚本测试和探索式测试的优点可以得到较好的效果。

目前许多测试团队以脚本测试为主导,偶尔在测试执行的时候发散性地去测试有疑惑的地方,但该发散性测试受经验、时间、功能特性、测试任务等众多因素影响,其结果无法跟踪,且经验不能传承。为了更好的组合脚本测试和探索式测试的优点,可以考虑尽量减少编写文档的时间,也可以考虑增加在测试执行时学习产品和技术的时间,从而带来更强的思维扩展性。

下面简单介绍如何组合脚本测试和探索式测试过程中需要用到的几个关键性的因素:

**Vague Scripted:** 比较简要的测试用例,是对脚本测试(ST)的初步简化,可以理解为测试人员需要编写测试用例,(但不必编写详细的预期结果)。测试用例可以包含操作步骤,但描述比较简单,其目的是留下更多的空间给测试人员在测试执行的时候自由发挥。

**Fragmentary test cases:** 使用一句话或几个词语描述的测试用例,类似于脚本测试中的测试用例标题。

**Charters:** 探索式测试过程中使用到的一个非常清晰的任务列表,指出了要测试什么、怎么测试(强调策略,不是详细测试步骤)、寻找什么样的 Bug、有哪些风险、要去检查什么文档等。

**Roles:** 测试过程中确定每个测试人员一个独立的角色去测试产品的某个部分。由测试人员自己掌控测试任务的进度和质量。

接下来将说明在实际的项目测试过程中,如何组合探索式测试和脚本测试。下图展示了不同目的的探索式测试象限。



## 探索式测试象限

象限编号的顺序与不同形式的探索式测试实践没有直接关系。例如,在我们进行“Freestyle ET”形式的探索式测试实践时,完全可以采纳结对测试(Pair Testing)的形式,或进行探索式测试之后采用全民分享(All Sharing)方式来分享更多的测试经验。探索式测试属于软件测试的上下文驱动学派(Context-Driven School),具体采用何种方式来进行探索式测试实践,或如何组合多种实践方式都是依赖于项目当时的情境,也就是项目本身的上下文环境。

由于本文只对第四象限的两个方式进行详细说明，则其他象限的具体实践方式不在此说明。特别说明的是象限一和象限四是两个面向探索精神的实践方式，象限二和象限三是两个偏向于流程和系统性的实践方式。这里需要强调的是 ET 主导的实践方式也是有一定的项目流程的基础和系统性的思维路径的，本文不做详细的说明。

象限四中的测试实践方式是由三个非常具有探索性的小活动(接下来称为趣味性的实践方式)组成。Bug 大扫除 (Bug Bash)在微软使用得非常普遍，且效果非常好。结对测试(Pair Testing)是与结对编程相对应，由两名或多名测试人员在一起同时测试一个 SUT，并相互给出建议和指令，共享通过使用系统得到的(隐含)信息，从而不断地积累系统知识和测试经验。全民分享(All Sharing)是测试执行前或执行后所进行团队分享的一个活动，具有不同背景和特长的测试人员分享自己发现 Bug 的思路，分享自己如何去测试软件，分享自己在某种测试类型上的策略。

## 缺陷大扫除

熟悉微软测试流程的测试人员应该都听说过缺陷大扫除 (Bug Bash)。它是一项短期的全员测试活动。在微软，许多开发团队会在里程碑 (Milestone) 的末期执行缺陷大扫除。程序员、测试员、程序经理、用户代表、市场人员在 1~3 天的窗口中，运用各自的技能和职业背景，集中精力来搜寻软件的缺陷。通常，每位参与者会获得一个小礼品，发现缺陷数目最多的冠军会获得一份大奖。

一般缺陷大扫除的组织者需要准备如下的内容：

- 活动对象
- 活动时间
- 测试环境
- 如何访问
- 测试数据
- 项目功能
- 问题反馈
- 奖项设置
- 已发现但在处理中的 bug 列表

缺陷大扫除是常规测试的有效补充。测试团队将各个子系统连成业务系统，执行端到端 (end-to-end) 的系统测试，能够发现个人在子系统测试中难以发现的缺陷。此外，测试人员在测试不熟悉的子系统时，没有任何先入为主的“偏见”，往往能立即发现那些被“熟视无睹”的缺陷。而资深测试人员还可能发现一些初学者难以察觉的隐蔽问题。不过，相比找到的缺陷，我认为缺陷大扫除在以下两个方面更有价值：

团队建设。在日常工作中，测试人员更多的时间在独立地工作，彼此之间的联系并不紧密。在缺陷大扫除中，测试员进行渗透式交流，互通情报，一起嘲笑那些拙劣的设计、滑稽的缺陷，甚至说一些无关的笑话以相互逗乐。全部这些“小事”都在潜移默化中逐步凝聚一个团队。

团队学习。团队举行“缺陷检讨”会议，总结缺陷模式 (bug pattern)，完善测试策略，补充测试检查列表 (check list)。这是一种积极的集体学习行为。在此过程中，测试人员可以积累经验、分享技能，测试团队可以沉淀知识、凝聚士气。

2011 年度淘宝网的商品线也举办了几次缺陷大扫除活动,效果是比较显著的。对于互联网的测试行业,存在多个复杂的运行环境。用户群拥有的不同的浏览器、浏览器的不同版本、不同的硬件配置、不同的网络速度,这些都可能会影响网络应用的页面的展示和功能特性之间的交互。由于涉及到内部信息,本文不展示缺陷大扫除活动的具体数据。

我以前的 blog 也提到过国外有一个专门从事探索式测试的团队,有着非常高的单位时间缺陷发现率。我们可以通过多次开展缺陷大扫除活动,从而观察出每次活动缺陷发现前三名的测试人员。我认为这些测试人员可以是探索式测试团队的储备人员,他们还需要接收更全面更深入的培训,包括测试技术、产品知识、系统思维、测试方法等。这样的一个专门团队需要在实战和学习中不断的提高,让自己的处事效率更加专业和正确。一旦公司团队紧急需要(任务复杂度,紧急程度等),这些储备人员将以最高效的测试手段进行测试,最快速的反馈产品的质量,就像特种部队一样。但是公司领导需要明白的是,探索式测试团队测试产品后,仍有存在重要缺陷的风险。另外公司成立特种部队的主要目的是让他们传播经验、培训员工。让大多数员工都成长,才是团队的努力方向。

总体来说,Bug Bash 不仅仅能给项目的质量带来新的思考方法和参考指标(质量和用户体验上都有所提升),而且还会驱使 Bug Bash 的参与者更加有激情和热度去从事测试工作。至于 Bug Bash 活动的相关细则可以根据自己团队的特点来定制化,目标就是提升项目的质量和提高测试人员的测试敏感度、测试关注度、创造性思维、团队分享精神。

## 结对测试

当开发人员都在关注结对编程的时候,测试人员应该关注什么呢,如何解决单人测试带来的测试遗漏这个问题呢,结对测试行不行呢?结对测试是软件开发中的一种技术,它允许团队中两个人一起测试某个产品,一个测试人员实际操作测试产品,另一个测试人员分析或评审测试过程和测试结果。

结对测试也未必一定是两个人,多个测试人员合作共同测试的时候就称为结队测试。测试人员都是很敏感的,在自己执行测试的时候,程序稍有一点反常,都会意识到可能是缺陷,一定会去究根追底,去确认到底是缺陷还是操作问题引起的。当测试人员互相交流发现的缺陷的时候,可能会互相启发去发现更多的缺陷。一个测试人员发现了一个缺陷,另一个测试人员可能发现和这个缺陷类似的更多缺陷,特别是在复杂环境下的测试。

两个测试人员工作在一起,在一个固定的时间段内一起产生测试思路,且持续的交换测试思路。假设我们需要进行测试设计,则一个成功的结对测试实践需要三个具体的条件:

- (1) 至少有一个测试人员可以被信任且能在没有指导的情况下进行测试;
- (2) 另一个测试人员需要参与到测试设计过程中;
- (3) 两个测试人员必须要有一起合作的能力和心态;

在结对测试过程中,实践者需要注意以下关键因素。

## 交换测试想法

在很多活动过程中,将你的想法解释给其他人听是一个负担,但是在测试活动中,这将是一种收益。因为测试的过程就是一个测试想法生成的过程,解释和质疑的过程有助于培养出更多新的测试想法。尤其当某个测试人员的知识远少于另外一个测试人员时更是这样。很多情况下,一个测试人员单独的测试容易陷入到一个错误的结论,除非另一个测试人员质疑该结论,否则他也不会重新审视该结论。

## 关注个人和社会因素

我们不知道每个测试人员的个人特质，如脾气、技能、经验对结对测试的效果有多大的影响。但是结对测试的实践表明该活动非常有趣，无论你的经验是丰富还是欠缺。当然，测试人员必须是友好相处的，且在过程中有一定的承诺，如果某一个测试人员在工作过程中感觉到被攻击、失去自主权、或变得沮丧都会影响结对测试的效率。结对测试是一个比团队测试更加特殊的组织方式，结对意味着最大化每个测试人员的贡献。在某个主管和其下属的结对情况下，通过结对的提问和测试的演示可以让主管更加信任测试的质量。

## 与测程相结合

我们在结对测试的过程中，可以在早期选择好需要结对测试的测程，测程的概念在我之前 blog 中已经做了详细描述。在一个测程中，结对的测试人员有一个清晰的目标和测试策略。当然，由于结对测试充满着乐趣，则结对测试的时间段可以是测试人员的测程的间隙。

通过结对测试，对初级测试人员来说，特别是和资深的测试工程师结对的时候，是一个非常好的学习机会，学习测试思路的转变、业务需求的分析、测试时的策略等。结对的人员是搭档，而初级测试人员一般是在计算机上操作相关功能，他/她应该得到充分的尊重，是可以根据自己的想法去测试。

原则上，结对测试时的两个测试人员需要对被测需求负有共同的责任，但是某些人员存在自己的一些个性和处事方式，可能会影响结对测试的效率。此时应和资深测试人员或测试领导一起来沟通解决此问题，让大家对结对测试有一个明确的认识，一起提高合作效率。

总体来说，有经验的测试人员和新测试人员一起结对测试会带来双赢的收益，这里我鼓励大家多去参与其他测试人员的测试过程中去。我需要提醒大家的是，结对的测试人员需要把握一个度，那就是测试人员给出的意见或建议需要有一定的价值（这样才能让测试执行人员更换测试步骤关注其他的测试点，若更换步骤后发现了 bug 就更有说服力），而不能频繁抛出自己的想法或思路（让测试执行人员产生不耐烦的心态）。





# 思维成就测试

## ——探索式测试实践篇

**作者:** *Tiger*, 某大型跨国公司 资深测试工程师, 主攻性能测试、异常测试、安全测试等, 兴趣在探索式测试。(需要联系者, 由杂志俱乐部代为转达)

探索式测试是一种软件测试手段, 不是一种具体的软件测试技术(如等价类划分、边界值分析、组合测试等), 它需要测试人员同时开展测试学习、测试设计、测试执行和测试结果评估等活动, 以持续优化测试工作。怎么样在日常的测试流程中实施探索式测试, 同时不与公司的现有研发流程产生冲突呢? SMART 原则为探索式测试提供了很好的建议。

- Specific (具体的): 测试需要一个具体的目标。
- Measurable (可度量的): 有明确的度量可以评估目标是否达成。
- Achievable (可实现的): 当前的目标应该是可实现的。这潜在地要求将一个大的目标分解为多个小目标, 每个小目标也是具体的、可度量的。此外, 跟踪小目标的完成情况也提供了整体进度的可度量性。
- Relevant (相关的): 目标要切合当前语境, 符合团队利益, 且不忘企业愿景 (vision)。
- Time-boxed (有时间限制的): 为每个目标设定一个合理的最后期限。这是帮助测试人员在固定的时间窗口 (time window) 中排除不相关干扰、专注工作

下面引入几个代表性 Backend Server Bug 的发现过程来解释一下探索式测试的理论, 以及如何在我们的日常测试中加入这个测试手段, 提高发现 bug 的效率。

**【场景 1】**输入参数探索式测试 Bug:非法的用户 Id 发送到了备份的数据中心导致了服务器出现宕机

通过日常 Review 服务器端软件 Bug 列表, QA 测试发现输入参数导致服务器出现异常行为的 Bug, 引入对同类型问题的思考, 输入参数会导致 Server 提供的 API 产生异常, 那么其他应用类型的 Server 会有这种类似问题? Server 内部是怎么样的一个处理逻辑才会导致 crush?





那么先从自己负责的 **Server** 下手，通过静态扫描 **Code** 发现“可疑 code”

```
CmResult CMs****Gsb****RequestPdu::解码(CmMessageBlock &aData)
{
    .....
    size_t nPos = strKey.find('^');//可疑 code
    if (nPos != string::npos)
    {
        strKey.resize(nPos);
    }
    m_user.用户 ID = strKey;
    .....
    return CM_OK;
22}
```

看来 DEV 下一个逻辑用到了 Decode 代码解析出来的 m\_user. 用户 ID, 但是这个值却是依赖于特殊字符'^' 作为分隔符, 看来我们测试的重点就是让解析的值出错, 那么肯定会造成程序的出错, 到底会出什么错呢? 拭目以待了! 接下来为了测试出这个 logical , 需要 design Case 走进这个分支的代码, 通过背景知识了解, 这个解码的方法输入参数 CCmMessageBlock &aData, 是通过 Primary 数据中心发到另一个数据中心(GSB)。Case 就可以写出来了, 重点是在 Primary 数据中心构造一个\*\*Id 带有多'^' InputValue. 测试结果看到了这个 Bug 大家应该知道了, 通过探索式的测试方法构造输入参数, 针对性的找到了这个隐藏的 Bug, 显然这种针对性的构造输入参数, 比模糊式的测试发现的 bug 效率会有一定提高。

接下来我们来看一下开发如何 **Fix** 这个问题的,

```
CmResult CMs****Gsb****RequestPdu::解码(CmMessageBlock &aData)
{
    .....
    CmResult rv = CMs****Util::GetFromUniqueKey(strUniqueKey, strKey, strSubKey);//Fix code
    m_user.用户 ID = strKey;
    .....
}
```

UniqueKey code logic 简单在这里说明一下: not use the specail char to generate the Key use the each selfkey and selfKey length 作为一个单元; 显然这种编程风格非常的规范, 特别是针对 **Server**, 这样企图通过输入参数破坏程序逻辑就变得很困难了。

最后, 通过对这个 bug 的学习, 采用探索式的思想, 需要对 Server 另外 4 个 APP (App1, App2, App3, App4) 进行统一的静态扫描 code; 发现 App2 存在 Null Key issues , 这里就不再累述了。

【场景 2】线程安全性探索式测试 Bug Run Two MultiThreadTool with two \*\*\*\*. so 1000 thread cause



crush

首先熟悉线程安全性的概念：当对一个复杂对象进行某种操作时，从操作开始到操作结束，被操作的对象往往会经历若干非法的中间状态。调用一个函数（假设该函数是正确的）操作某对象常常会使该对象暂时陷入不可用的状态（通常称为不稳定状态），等到操作完全结束，该对象才会重新回到完全可用的状态。如果其他线程企图访问一个处于不可用状态的对象，该对象将不能正确响应从而产生无法预料的结果。

那么 QA 怎么测试一个程序是否具备了线程的安全性呢？需要通过 reviewcode？显然这是一件很困难的事情，而且效率不一定会高。因为个人感觉多线程的编程本来就比一般的编程要难很多，需要注意的细节多很多。既然不能用白盒测试发现问题，那么怎么去测试呢？黑盒？

下面是我在多线程测试中的一些探索，结合这个 bug 的发现过程和大家分享一下。\*\*\*\*.so 一个 C++ NativeClient，提供给 Nginx \*\*\*\* Process 使用，考虑到这个 NativeClient 可以启动多个连接，多个实例，那么对于测试者来说直接把这个 NativeClient 想象成一个黑色的盒子，测试预期是肯定这个“黑盒子”支持多线程。通过在这个 NativeClient 加上一层多线程的应用场景，通过多线程模式测试“黑盒子”的内部的线程安全性，看是否和 QA 预期结果一致。测试方案确定，用多线程测试多线程，这就需要 QA 写的多线程自身要具备线程安全性，才能正确达到测试的预期结果。



以下是我当时为了测试开发\*\*\*\*.so 写的一个 C++ 多线程的测试程序。下面摘录控制线程的几个主要方法，相信大家都不会陌生。（线程实体就不在这里摘录了）

```
#include "pthread.h"
class Mutex
{
public:
    static void pthreadMutexInit(pthread_mutex_t* aMutex_t);
    static void pthreadCondInit(pthread_cond_t * aCond_t);
    static void pthreadCondWait(pthread_mutex_t * aMutex_t, pthread_cond_t * aCond_t);
    static void pthreadCondSignal(pthread_mutex_t * aMutex_t, pthread_cond_t * aCond_t);
    static void pthreadCondBroadcast(pthread_cond_t * aCond_t);
}
```

写好测试程序和 makefile 文件，在 linux 平台编译成了 MultThreadTool 进行测试,通过测试发现上述 bug, 通过和 Dev Review Code 发现由于回调线程与发送线程(网络线程)在 request timeout Condition 重复多次调用了 removeClnetTransaction 对象导致了 crush。

接下来看一下开发解决方案，把计划定时器，取消定时器, onTimeout 放到发送线程, instead of 回调线程。这样做的好处简化回调线程要处理的逻辑，让网络线程处理这些异常的条件，比如刚刚由于 TimeOut 导致的 crush 问题。通过这种测试方法，测试线程安全会更高效率，在提高 QA 测试技术的同时，也最大程度的提高发现这种线程安全性的可能性。刚刚讲的只是由于线程安全的一个方面，用同样的方法实际上还测试了线程安全性典型的另一个 bug,线程间抢占资源导致的死锁。Bug: A add B as Friend then UpdateB Del A same time Cause Thread Lock together

这是个人探索式测试的第一次亲密接触，结合正开展的项目把理论变成了实践。最后总结一下“探索式测试是一种软件测试手段，不是一种具体的软件测试技术，但是它需要多种测试技术手段的支持”希望这篇文章能帮助 QA 提高日常的工作效率,同时帮助 DEV 减少同类型错误的二次发生。



## 测试的心理定势

**作者：邵晓梅** CSTQB 工作组成员；华为公司 10 年测试工作经验。擅长测试分析设计、测试管理、敏捷测试、探索性测试、基于模型的测试、测试过程改进、基于需求的测试、缺陷根因分析等方面。由她提出的 MFQ&PPDCS 软件测试分析设计方法和 T-RCA 缺陷根因分析方法在华为多个产品得到实践应用。近期研究“测试人员的思维”课题，在测试人员的培养方面有独到见解。

新浪微博：MiniStarClub001-邵晓梅，邮箱：testcomt@gmail.com

岳晓东博士在中国版幸福课《幸福在我心》中，拿出一副达·芬奇的画作《蒙娜丽莎》让听众观察画里面还有什么。有的看出画里面有一只猫，有的看出画里面还有一只狗，有的人除了蒙娜丽莎怎么也看不出来画里面还有什么……如果你看不出有什么动物，当休息一会儿，再看这幅画时，也许就能看到了。

这在心理学上称为“心理定势”，指的是一个人看问题一旦形成某种模式后，就不再思考了；但是如果暮然回首再回头看时，会觉得原来不是没得可为、不能再做什么事了，而是看待问题又有了新的突破。

这种心理现象在测试中是普遍存在的。

前一段时间，我和另外几个人对同一被测对象各自单独进行了探索性测试。被测对象是 Word2007 的插入页码功能。在 1 小时的探索中，我测试了插入页码在页面顶端、底端、页边距、当前位置，以及设置不同的页码格式、删除页码功能，发现缺陷集中在“在页面上编辑页码格式”（比如放大页码格式的形狀、移动其位置等等）以及对超大文件插入页码方面；由于页码删除功能比较简单，只测试了少许时间，认为其风险比较低。总体而言，我对

这次探索性测试比较满意。

可是，当我观察另外一个人的探索性测试时，我惊奇地发现，还有这样一个重要的功能完全不在我的视线范围之内：编辑某种页码格式的属性。按如下的操作步骤：插入一页码一页面顶端一普通数字 1—右键点击鼠标，选择“编辑属性……”，则弹出“修改构建基块”的对话框，用户可以编辑修改此种页码格式相关的属性。同时我发现，她花费了较多的时间测试“删除页码”的功能，并且发现了一些问题。

为什么会出现如此的差异呢？这些功能和问题为什么我没有发现呢？

一个直接的原因就是，我把注意力集中在每种页码格式可能存在的问题上，每次选择一种页码后，直接点击鼠标左键，然后急于观察这种页码格式显示是否正确、当我改变其形状移动其位置后表现是否正确，从来没有想过点击鼠标右键。这么做，好的方面是，我找到一个点后，对这个点的测试会比较深入（Test in Depth）；不好的方面是，测试宽度上（Test in Width）容易有所遗漏。

这里面有几个有趣的点值得深入探讨一下。

### （1）心理定势

“总是直接点击鼠标左键、从来没有想过点击鼠标右键”，这虽然描述的是我的动作，但是是一个人的行为取决于他/她（下文统一用“他”代替了）的思想和意识。

我的心理定势是：我认为插入页码的主要功能就是“插入各种格式的页码”，主要的风险也在这里，所以我的注意力也集中在“每种页码格式可能

存在的问题上”。当我把这个集中的点进行了较充分的测试，并且真的发现了其中的一些问题，并且对这些问题进行了进一步的调查后，我就会认为：这次测试还是有些收获的、差不多可以结束了、主要的点基本都覆盖过了、应该没有什么大问题了，对那些可能被遗漏的点（比如“编辑属性”功能）和那些未知的缺陷（删除页码相关的缺陷）却浑然

不觉。

当然，如果我再进行一轮测试，很有可能会突破前一轮测试的思维定势，找到新的测试点、找到新的缺陷。所以，从测试的心理定势我想到的，为

了保证更好的测试效果，使不同的测试人员对同一个被测对象展开测试、或者同一个测试人员在不同时间对同一被测对象展开不只一次的测试，是很有必要的。

## (2) 深度测试与广度测试

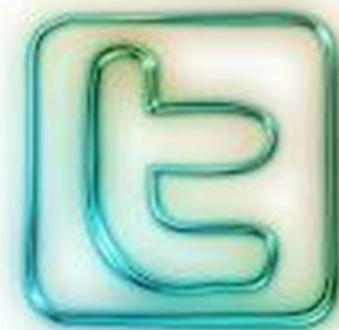
注意，在上面的分析中，我把“并且真的发现了其中的一些问题”重点强调了一下。如果我把注意力集中在“每种页码格式可能存在的问题上”，却什么问题也没有发现，会怎么样呢？也许，我会怀疑之前自己对风险的假设，从而关注其他值得深入测试的点，也许就能发现上述的测试盲区。这说明，发现的缺陷往往会阻止缺陷的发现。

这句话的意思是，当我发现了一个缺陷，我就会花费时间去调查这个缺陷，它真的是个缺陷吗？还是我测试过程出现了问题？这个缺陷有可能导致什么问题？到底如何触发这个缺陷的？有没有其他值得进一步挖掘的缺陷？对于一个优秀的测试人员来讲，解答这些问题的过程才是真正有趣的测试过程，就像一名猎手闻到猎物后开始的一系列追踪的过程，充满希望和挑战！但测试也是充满矛盾的过程：测试需要这种深入的测试过程，深入的测试才能发现有价值的缺陷，可是发现了缺陷就会占用你的测试时间、放缓你的测试脚步、“占用”你的测试思维，让你没有更多的时间、也无法避免地忽视其他的缺陷，从而导致测试的遗漏，所以总是要在深度测试和广度测试之间把握一定的平衡。

本文中提到的探索性测试的例子，我更偏向于深度测试，而另外一个人更偏向于广度测试。如果用那个沿着一条路挖水井的例子做类比，我挖了很多坑，大部分都挖了 100 米，最终找到了 4 口井；她挖了更多的坑，都不超过 50 米，找到了 2 口井。其实，更好地平衡深度测试与广度测试的话，我们都可以挖掘出更多的更有价值的井。

由深度测试与广度测试，还可以联想到另外一个测试中的矛盾点：你可以相信测试人员提供的对被测对象质量的评价，但你又不能完全相信这个评价。深度测试更多与降低风险相关，广度测试更多

与增加覆盖相关。无论测试人员做到怎样的平衡，都不能同时做到 100% 的深度测试和 100% 的广度测试，也就是说，不可能既降低了所有的质量风险、又覆盖了所有应该覆盖的东西。我们知道，每个测试人员都是基于他的测试情况给出他对被测系统质量的评价。比如，基于我的测试过程，我认为 Word2007 的插入页码功能在页数较多的大文件插入页码时存在质量风险、系统对不同种类页码格式的处理有所不同（我的感觉）因而需要分别测试、对页码形状进行编辑等操作容易引起问题；而另外一名人员经过他的测试过程，认为 Word2007 的插入页码功能质量不错，只有一些小的 GUI 界面提示性问题，没有什么大的问题，无需对每一种页码格式进行单独测试。作为测试管理者，你应该相信每一位测试人员提供的质量评价，因为这个测试结论是测试人员基于他的测试过程对质量的真实理解，但是你显然不能尽信这个观点，因为这个真实的理解并不代表真实的质量信息，只是目前测试人员能够觉察到的质量信息（Perceived Quality），你还要去了解这个观点是在什么样的测试深度和测试广度下得出的结论。





### (3) 交替思维

怎么样克服测试的心理定势、避免测试的盲区呢？

也许你会说，我不用探索性测试，我用脚本化测试（Scripted Testing，这里对脚本化测试的理解与 Cem Kaner 在 <http://www.kaner.com/pdfs/ValueOfChecklists.pdf> 里对 Scripted Testing 的描述是一致的）方法，测试前仔细阅读需求规格说明书，这样我就可以事先知道一共有多少个点需要测试了，就不会遗漏。这样做其实并不能从根本上解决问题。

首先，需求一旦写成文档，就是不完整或不够细致的需求了。你并不能依赖需求文档的完整性来避免漏测。而且，就算在测试执行之前，你已经了解了所有的需求，测试的心理定势以及其他一些因素仍然会致使你漏掉一些缺陷。

其次，如果过于依赖脚本化测试方法，你的测试思路会不知不觉地受文档描述的影响。我们有另外一个人对插入页码功能也进行了测试。与我们的测试过程不同，他先阅读插入页码的帮助，然后再开展测试。结果，我发现他的测试过程有很多“受文档影响的痕迹”。对于文档中没有提到的部分，他很少会关注。比如“对页码的形状进行编辑”、“对过大的文件”来测试等等这样的操作”；而对于文档中提到的部分，他会关注较多，比如帮助中有这么一句话：“您可以将页码添加到文档的顶部、底部或页边距。保存在页眉和页脚或页边距中的信息显示为灰色，并且不能与文档正文信息同时进行更改”，很显然这句话给他留下了深刻的印象，所以他在测试中时不时地就会验证一下页码信息和正文信息是否可以同时修改。是的，我这里用的词是“验证”，也可以叫“检查”，可以对应到英文的 checking，我是说他的测试过程更偏向于 checking，而不是 testing。关于 checking 和 testing 的区别，可以参考 Michael Bolton 的博文，

<http://www.developsense.com/blog/2009/08/testing-vs-checking/>。更偏重于 checking 的人，测试执行时会更倾向于去验证一下他之前认为的一个结果是否正确；而更偏重于 testing 的人，测试执行时会更多地想发现一些之前没有想到的情况、发现一些新的信息、获得对真实的系统更深一些的理解。

克服心理定势、避免测试盲区，可能有很多种方法，比如运用系统性思维（Systems Thinking），散焦思维（Defocused Thinking），交替思维（Alternation Thinking）等，关于这些概念的更多信息，可以参考 James Bach 的书《Scerets of a Buccaneer-Scholar》，也可以参考这本书《经典思维 50 法》，里面有不少有趣的例子。限于篇幅，这里只分享一下我对交替思维的理解。

你在测试时，是否有这样的时候，绞尽脑汁也发现不了什么问题、缺乏新的测试思路，或者感觉当前有点混乱、测试效率很低；或者当你读一本书的时候，读了一段时间，发现大脑反应有点迟钝、阅读效果极差、感觉有点烦躁。这个时候，并不是你不专心高效测试了、不爱读书了，而是你的大脑给你传来一个信号：我已经接受了太多的信息，请您休息片刻，让我消化消化，再继续工作。此时，是你运用交替思维的绝佳时机，你可以做些其他的事情转移当前的注意力，比如随便点击鼠标试试其他的功能、比如整理一下你的测试记录、或者干脆休息一下。虽然你在休息，但你的大脑并没有休息，它在消化吸收刚才的信息，当你再次开始测试时，你会发现测试效率提高了、测试思路打开了，往往会有意想不到的收获。

所以，下次测试的时候，不妨注意一下你是否也存在某种心理定势？这种心理定势是如何影响了你的测试效果？你的测试深度与测试广度又是如何平衡的？有意识地运用一下交替思维、系统性思维等思维方式，让你的测试更加高效起来！

## 从医学检测来看软件测试

**作者: Ricky** 自 2005 年开始从事软件测试工作, 参与过多个大型企业级产品的测试, 目前从事互联网产品相关的测试。

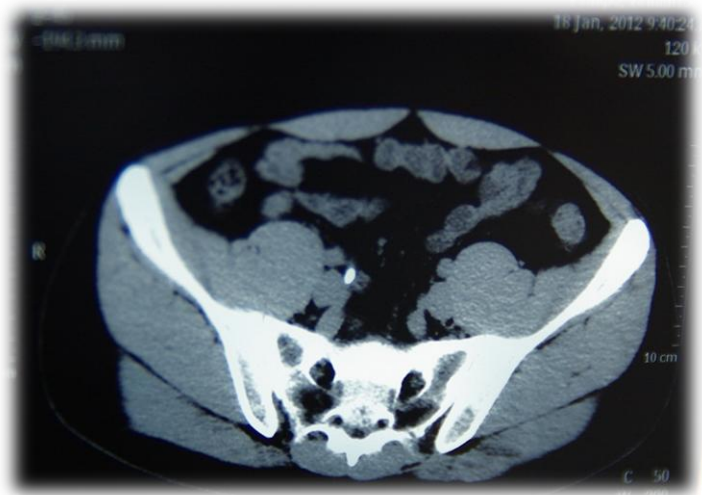
个人 blog 地址: <http://blog.csdn.net/superqa>

年前的时候被体内一颗短径 4.8mm 的小石头折磨, 还提前休了几天病假, 并第一次被要求去做了个 CT。以前只是听说过这个检查, 第二天拿到报告, 看不懂, 对我们这些外行来说看起来比较高深。拿到的报告是很大的一张片子, 里面密密麻麻的很多小图, 其中的一张翻拍如下。给医生看的时候才知道中间的那个小白点就是结石, 准确的说应该是它的横截面。

咨询了当外科医生的高中好友, 他打了一个简单又有点恶心的比方, 就像在自助餐厅里服务员拿一大块牛肉用机器切成几毫米的薄片, 每一张图片就是一片, 大概间隔 3mm 左右。虽然比方不太合适, 但是意思能懂了。

想想其实这是一个很强大的检测技术, 发现人体内那么小的一个石头的确切位置, 而且关键是这个检查本身不需要放入任何物体到体内, 也没有对人体造成直接的创伤(当然照多了还是有害的), 就可以得到比较直观清晰的图像, 帮助医生来定位问题, 并作为后续治疗的参考。

因为好奇, 后来在网上查了下 CT 相关的技术, 觉得很有意思, 有兴趣的可以看看。



<http://zh.wikipedia.org/wiki/CT>

<http://wenku.baidu.com/view/0e4fa922bcd126fff7050bc5.html>

这篇文章也比较适合于技术爱好者

<http://yx.yangning.net/showtopic-627.aspx>

联想到自己一直从事的软件测试工作, 觉得 CT 有很多的启发。

姑且我们把检测作为测试的一个部分, 简单来说其目的大致有两类。

- 检查有没有问题, 比如体检
- 试图确定问题在哪儿, 比如不舒服之后去做某些项目的检查

其实我们的测试工作很多时候也是做这样的事情。拿到一个新的版本或者改动, 去检查有没有问题, 觉得有些地方不太对劲的时候, 试图去获得更多信息来帮助判断问题可能出在哪里。



和软件测试一样，CT 并没有直接的改变被测对象，问题还是在哪儿；另一方面，它也没有直接给出如何修复或者说是治疗的方法。但是它们有一个共同点，那就是能发现问题，并且提供有价值的信息！

当然，做到这些还有一些重要的前提，比如：

### 1. 结果的准确性。

这一点毋庸置疑，错误的结果会误导，甚至延误重要的时机。这个也是一项新技术出现的根本。

### 2. 侵入性

一项测试手段对被测的对象的影响。这也是 CT 技术超越动脉造影、气脑造影、开颅探查等传统检测手段的一个重要原因，它对人体的伤害和影响要小很多，当然 X 光本身也是有伤害的，只不过不常做的话影响不大。

对于软件测试也是一样，如果一项测试技术会干扰到产品本身的运行，就会影响到测试的效果或者适用范围。例如 **coverage test** 是帮助测试人员判断测试的范围和程度的良好的手段之一，但是因为它具有一定的侵入性，不仅影响性能，有时甚至会影响到功能，所以使用受到限制，一般也不会放入 **production** 的代码中。

### 3. 检测的速度和效率

在前面关于 CT 的文章中也可以看到，从第一代的机器到后面的改进，很重要的一个指标就是扫描的速度，这其实就是检测效率的问题。能及时地判断有没有问题或者发现问题是很重要的，对于软件测试也是一样，特别是对于互联网产品，发布的周期非常短，很多产品平均每周都有好几次发布，如果一项测试需要花到几天的事情，基本就很难广泛使用。而速度和效率很多时候需要依靠新的更先进的技术。比如 CT 的扫描速度的依赖于更多的探测器和更强的计算机处理能力。

### 4. 成本

成本也是一个很重要的因素，因为太昂贵的东西难以普及。可以参见上面第三个 link 中 CT 早期因为成本也是一个政治话题。成本包括很多方面，初次购买设备、运营维护（耗材，电力等）、工作人员的成本。对于一个新的测试手段也是，需要的工具和平台（软硬件）、初次开发的成本、日常维护的成本等。

除了 CT 以外，还有很多的医学检测的种类。比如：基本的身高、体重、心率、血压等。

- 1) 血常规，尿常规等生化方面的检查；
- 2) B 超，心电图, X 光等更细致的一些检查；
- 3) 还有很多针对特定器官（如肝和胃）和部位的针对性的检查。

所有这些检查，只要大家能说得出来的，你会发现有一个特点，那就是在每家（有些可能要大点的）医院都能做，而且正常情况下操作相似，用的设备类似，结果不会偏差很多。还有一点就是这些可能是由不同的人来操作的，大家都能做出正确的结果。某种程度上，已经变成了比较广泛使用的标准的做法。

这里顺便谈一下关于自动化的思考，还是来自于医学检查的启发。

医学的检查有很多的方面，包含最简单的身高、体重和血压。这些通常是用手工的方法，后来有了简单的自动化的工具，用来提高效率，如果条件不够的地方，还是可以退回到人工的方式来做。但是很多检测的手段，手工是没有办法做的，必须借助于先进的技术和工具，比如这里提到的 CT，除了弄一台 CT 机通常也没什么办法。

这就好像如果计算机只是用来算数字，那么它起到的作用就只是加快速度，提高效率。理论上讲在这种情况下还是可以手动来做的，只是时间可能长得无法接受。但是如果它用来做视频的解码和播放，那

就有质的变化了，这个时候已经完全无法用手工来做了。

我想对于我们的测试手段也是一样，有些模拟用户操作的手工测试可以被自动化，但是有时也可以退回来手工，但是有些测试，比如性能测试，压力测试以及 **fuzzing test** 等则不得不借助工具来完成。

对比医学检查，目前的软件测试显然还没有达到这样的程度，粗浅地想了一下，可能下面的一些原因：

### 1. 人类身体各器官的构造，即便考虑年龄、种族等因素，其差别也是很小的。

但是软件的差别则很大，包含所基于的平台，所用的开发语言，使用的主要组件，不同开发人员写出的同样功能的代码也差异很大。不过我在想，对于同一类的软件是不是有一些共性，比如 Windows 桌面版的应用、一个用来处理大量客户端请求的 Linux daemon、一个 Android App？针对这些共性的地方，能否做一些提炼？

### 2. 比较容易判断有没有问题

其实和第一条也有一些关系，因为已经有大量的经验值，比如大家看到的体检报告上常会给出某个检测指标的参考值，比如白细胞计数，血压和血糖的正常范围。有了这些数据之后就比较容易判断正常和异常。而对于软件而言，这些相对比较困难，或者说目前只能做到一些相对外围的比较粗略的，比如 **crash**、**memory leak**、CPU 使用过高等等。但是对于内部逻辑是否正确、当前状态是否正常就比较难以判断了。

就目前的状况来看，还需要一些针对性的定制来解决这个问题，就好比自动化中常用到的断言。

### 3. 标准的检测手段

CT 借助了 X 光来进行成像。X 光是一个核心的检测方法。那么对于软件，有没有什么比较标准的检测方法？一些性能监控和调试工具中可能会借用一些处理器、操作系统，或者编译器提供的底层功能来监控应用的行为，类似 **vtune**，**oprofile** 这样的工具。

经过这么多年的发展，其实软件测试行业已经有了很多的积累。比如对于性能的检测有很多比较标准的指标，然后有很多对应的工具来检测这些指标。这些指标可以通过独立的监控工具，或者在线的监测系统来完成。还有一个例子就是遇到 **crash**，大家会立刻想到去做 **coredump** 的分析，这其实有点像上面提到的医学的某项标准检测，比如发烧比较厉害了医生可能让去做个血液的检查，看有没有什么感染。但是相比医学上比较成熟的成体系的通用的检测手段，我们测试人员的还显得比较少。

那么另一个方面，像医学检测一样将很多技术、方法和经验慢慢固化下来有什么好处呢？我初步想了一下，可能有如下的一些方面：

#### 1. 有利于持续的改进

没有衡量的标准改进就难以进行，因为难以判断结果的好坏。如果有一套比较针对性的标准的检测方法，其实也是一个改进的指导。改进的结果也能很清晰的看到，也能体现到具体的东西上面。

#### 2. 树立了一个基本的产品质量的门槛：

因为不满足这些检测标准的产品很快能被排查出来。从这一点引申出来，其实就涉及到我们测试团队在日常工作中经常提及的一个问题，那就是如何推动开发自测。这是一个提了很久的话题，除了协作和配合以及人的观念的问题，我觉得具体的可操作的方法也值得考虑。

如果能有一套简洁高效的测试方法，比如最简单的一套稳定高效的自动化测试集，那么推动开发自测就不是一个问题。能高效快速的知道有没有问题，并在失败时给出必要的信息。如果这一条做到了，而且自动化程度很高，这个时候就可以把这个作为一个内部的标准，开发人员做出来的东西如果没有 **pass** 这些测试，就不能提交给测试人员做进一步的测试。这样，基本的验收测试慢慢就不需要测试人员来做了，相

当于提高了测试的门槛。

在曾经做过的一个项目中，这一点让我印象很深。

当时我们将大概一半测试用例（约几千个）自动化了，但是由于各种原因，**daily build** 来用的时候总是有部分用例失败，要去确定是用例的问题还是产品的问题需要不少时间，每天这样做也耗不起。后来我们抽取了其中涉及主要功能点的重点用例，约 200 个左右，花力气弄得比较稳定了。能达到的结果就是，只要有用例 **fail** 了，基本都是产品的 **bug** 或者一些未知的改动导致的（如果我们不能做到如此凭什么要求产品也稳定？）。这样运行了一段时间后，比较有信心了。我们帮助开发人员把这一套自动化的用例集在他们环境里部署起来，作为 **daily build** 的验收测试，每天的结果由邮件通知大家，不全 **pass** 的 **build** 不接受做进一步测试。

开始我们想这样会增加开发人员的工作量，估计会比较反感。但是后来我们发现其实并非如此，因为这套测试集比较稳定，能及时的发现问题，开发人员很积极，第一时间就会去看，而且觉得对他们很有帮助，因为他们有时也不确定改动有没有问题。

对比我这里举的例子，也不难理解，我们大概也很少听到医生讨厌医学检测，因为这些东西确实是在帮助他们发现问题，定位问题，或者至少提供有用的信息。

我想从这个意义上，才能实现开发和测试良好合作，因为双方都为同一个目标贡献有价值的东西。而另一个方面，测试人员要赢得别人发自内心的尊重，一定是因为专业，而不是口头呼吁重视测试。

如果能达到类似的程度，我相信就不会有人质疑测试这项工作本身的价值和技术含量，因为“1979 年 10 月，由于 Hounsfield 和 Cormack 在 CT 领域的开拓作用，获得了当年的诺贝尔医学奖”。就好比会有病人或者医生质疑 CT 和 MRI 等检测手段的价值和技术含量吗？

手工的测试是有价值的，但是个人的观点，我觉得测试作为一个专业领域，方向还是应该逐渐的把一些通用的技术和方法通过工具和系统的方式固化下来，变成可以更便利和更大范围收益的东西。让更多的人，组织内或者组织以外的，能用到日常的实际工作中。而另一些人，去改进现有的东西，或者再去探索新的技术和方法。所以我这篇文章的意思并不是强调工具流程而贬低人在测试工作中的价值，相反，有经验、技术和新想法的人是这些的推动力。

但是我不认同将经验和技能停留在经验和技能。这里拿中医和西医做个对比，无意冒犯，只是做个类比。

在传统中医（现在大的中医院里，西医的各种器和术都有）里看病，讲究望闻问切。比如看舌苔，摸脉搏，看气色等等。我不否认有很高明的中医，基于自己高深的技术和多年的经验，能很好的运用这些，来进行准确的诊断和对症的治疗。但是现实中，我们都依赖于这些含糊的说不清楚的经验，好像难以应付那么多人的看病的需求，而且高明的中医似乎太少。他们的这些经验都是靠日积月累，难以传承。

这是一种困境，我们看到的现实还是西医的思维起主导作用，而且生病的时候大部分还是靠的西医。因为一些标准化和定量的东西更容易普及和推广，而很多时候这些体现在具体的设备和工具。大家看西医的方法，有很大一部分的时间（包括患者花的钱）都是在前期的检测，其中的监测和后面的复查。这其实也说明了测试的价值。

说了这么多，我想我的 **point** 不在于否认测试人员的技能和经验的重要性，或者要把一些东西僵化。但是我感觉到我们软件测试行业这种被广泛认可的专业性的东西还是太少，不能每次说到我们对比开发或其他软件研发的工种的独特价值就是我们对测试的理解、测试的策略和方法论等等，而应该是一些能拿得出来的、被普遍认可价值的、有明确结果的东西。

最后想提醒大家要多喝水，多锻炼，多保重身体！



## 代码评审三步走

**作者:** 夏江平 混迹民企, 历经合资, 心水外企。坐山空食老本, 胡乱摸索野球拳。白头方识学院派, 亡羊补牢幸未迟。爱旅游、爱摄影、爱游戏、爱幻想, 也有测试职业病。

联系方式: [gypsy.x@gmail.com](mailto:gypsy.x@gmail.com)

微博: [MiniStarClub084](#)

### 序言

在工作中参加过不同产品的多次代码检视活动, 但在过程中观察到的一些现象引起了我的注意: 不管是被检视者还是检视者, 似乎都没有什么思路, 只是对着代码发呆。即便能够发现一些问题, 多数也是影响甚微的小问题。少数专家虽然能够发现有价值的问题, 但也是凭经验+直觉。要问他们怎么做才能发现这些问题, 却又没有答案。

为什么会有这样的现象呢? 这种情况有没有办法加以改善呢?

我个人感觉, 这是缺乏“套路”所致。研发活动, 无论是架构设计, 还是代码编写, 或者测试活动, 都有各自的方法或“套路”。这些“套路”能够指导相关人员有序地开展活动, 并保证活动的有效性。那么代码检视是不是也应该有这样的“套路”可循呢?

经过一段时间的反复思考, 我将自己的一些想法整理成此文, 希望能够对代码检视活动提供一些帮助。

### 正文

下面将结合示例代码进行代码检视三步走的实战演示。

```
bool EnemyChecker::IsValidMonster(
    NPCManagement::HordeIdType hrdId,
    const NPCManagement::MonsterInfoSeq& mnsInfos,
    NPCManagement::MonsterInfoSeq& vmnsInfos,
    NPCManagement::FailedMonsterInfoSeq& ivmnsInfos)
{
    bool bRet = true;
    ivmnsInfos.length(mnsInfos.length());
    vmnsInfos.length(mnsInfos.length());
    unsigned int iip = 0; // index of invalid horde
    unsigned int ivp = 0; // index of valid horde
    //检查怪物是否属于部落群
    for (CORBA::ULong i = 0; i < mnsInfos.length(); ++i)
    {
        const NPCManagement::MonsterStatusInfoSeq& mnsStatusSeq
            = mnsInfos[i].monsterInfos;
```



```
for (CORBA::ULong i = 0; i < mnsInfos.length(); ++i)
{
    const NPCManagement::MonsterStatusInfoSeq& mnsStatusSeq
        = mnsInfos[i].monsterInfos;
    IntSet monsterIds;
    for (CORBA::ULong j = 0; j < mnsStatusSeq.length(); ++j)
    {
        monsterIds.insert((int)mnsStatusSeq[j].monsterId);
    }

    if (monsterIds.empty())
    {
        vmnsInfos[iip].horde = mnsInfos[i].horde;
        vmnsInfos[iip].monsterInfos.length(0);
        ++iip;
        continue;
    }

    std::map<int, short> mnsTypes;
    NPCManagement::MonsterLoader::getMonsterType(hrdId, -1, monsterIds, mnsTypes);

    MonsterVec invalidMnsIds;
    getInvalidMnsIds(mnsStatusSeq, mnsTypes, invalidMnsIds);
    //存在部落群和怪物不一致或者怪物类型错误的情况
    if ((monsterIds.size() != mnsTypes.size())
        || !invalidMnsIds.empty())
    {
        if (mnsTypes.empty())//一个有效怪物都没有时，记录无效怪物
        {
            ivmnsInfos[iip].horde = mnsInfos[i].horde;
            ivmnsInfos[iip].monsters.length(mnsStatusSeq.length());

            for (CORBA::ULong pos = 0; pos < mnsStatusSeq.length(); pos++)
            {
                ivmnsInfos[iip].monsters[pos].monsterId = mnsStatusSeq[pos].monsterId;
                ivmnsInfos[iip].monsters[pos].monsterName
                    = mnsStatusSeq[pos].monsterName;
                ivmnsInfos[iip].monsters[pos].reason =
                    NPC_STR2WSTR(WORLD::Message(getCvt(), "import_invalid_monster"));
            }
            iip++;
        }
    }
}
```



```
else
{
    MonsterVec validMonsters;
    MonsterVec invalidMonsters;
    validMonsters.reserve(monsterIds.size());
    for (IntSet::iterator it = monsterIds.begin(); it != monsterIds.end(); ++it)
    {
        MonsterVec::iterator invalidIt
            = std::find(invalidMnsIds.begin(), invalidMnsIds.end(), *it);
        //数据库中不存在, 且类型也有效
        if ( invalidIt == invalidMnsIds.end()
            && mnsTypes.find(*it) != mnsTypes.end())
        {
            validMonsters.push_back(*it);
        }
        else // 怪物Id和对应的怪物类型不一致或怪物不属于部落群
        {
            invalidMonsters.push_back(*it);
        }
    }
    vmnsInfos[ivp].horde = mnsInfos[i].horde;
    vmnsInfos[ivp].monsterInfos.length((unsigned long)validMonsters.size());

    for (unsigned int k = 0; k < validMonsters.size(); k++)
    {
        for (CORBA::ULong pos = 0; pos < mnsStatusSeq.length(); pos++)
        {
            if ((int)mnsStatusSeq[pos].monsterId == validMonsters[k])
            {
                vmnsInfos[ivp].monsterInfos[k] = mnsStatusSeq[pos];
                break;
            }
        }
    }
}

ivmnsInfos[iip].horde = mnsInfos[i].horde;
ivmnsInfos[iip].monsters.length((unsigned long)invalidMonsters.size());

for (unsigned int k = 0; k < invalidMonsters.size(); k++)
{
    for (CORBA::ULong pos = 0; pos < mnsStatusSeq.length(); pos++)
    {
```



```
        if ((int)mnsStatusSeq[pos].monsterId == invalidMonsters[k])
        {
            ivmnsInfos[iip].monsters[k].monsterId = (unsignedint)invalidMonsters[k];
            ivmnsInfos[iip].monsters[k].monsterName =
                mnsStatusSeq[pos].monsterName;
            ivmnsInfos[iip].monsters[k].reason =
                NPC_STR2WSTR(WORLD::Message(getCvt(), "import_invalid_monster"));
            break;
        }
    }

    ivp++;
    iip++;
}

else//部落群和怪物一致，且对应怪物类型一致
{
    vmnsInfos[ivp].horde = mnsInfos[i].horde;
    vmnsInfos[ivp].monsterInfos = mnsStatusSeq;
    ivp++;
}

ivmnsInfos.length(iip);
vmnsInfos.length(ivp);
return bRet;
}
```

#### 示例代码段一：

##### 第一步：

代码检视要做的首先就是评审代码的逻辑正确性，即代码是否完整、准确地实现了业务逻辑。

这一步只需对代码进行抽象，将其“翻译”为“注释”就可阅读其实现逻辑，并与需求逻辑（设计逻辑）相比较。

这第一步很重要，而且实践难度其实不高。对于有“注释驱动编码”习惯的开发人员来说更是容易。

对示例代码段一，“翻译”成简略的“注释”（有省略），可以如下文（以大括号{之后紧接的为第一行）：

```
// (1-5 行) 初始化输出值

// (6-102 行) 超大循环, 遍历输入的 mnsInfos, 对每一项执行以下的操作:

// (8-14 行) 首先从 mnsInfos 中取出一组数据并记录其中的所有怪物 ID

// (15-21 行) 如果该组数据为空, 则记入输出参数 vmnsInfos 并跳过

// (22-25 行) (如果该组数据不为空) 根据这一组怪物 ID 获取其对应的怪物类型 (mnsTypes) 及
无效怪物的集合

// (26-95 行) 如果存在部落群和怪物不一致或者怪物类型错误的情况, 那么:

// a、(29-42 行) 若一个有效怪物都没有时, 记录无效怪物

// b、(43-94 行) 否则遍历这一组所有怪物, 并分别记录下有效值和无效值, 当

// 1、数据库中存在怪物信息且类型也有效时, 记为有效怪物, 并存入输出参数 vmnsInfos

// 2、怪物 ID 和对应的怪物类型不一致或怪物不属于部落群, 记为无效怪物, 并存入输出参数
ivmnsInfos

// (96-101 行) 否则如果部落群和怪物一致, 且对应的怪物类型也一致, 则将信息都存入 vmnsInfos

// (103-105 行) 循环结束, 返回执行成功标识
```

从以上的“注释”中, 可以看出几点:

- 1、虽然该函数有 100 多行, 但是其实逻辑并没有那么复杂。
- 2、逻辑并不复杂的代码却写出 100 多行, 其中大量代码都是在做基本的 sequence、vector 遍历、拷贝的原始动作, 这些完全可以剥离出去, 使主体逻辑清晰化。
- 2、返回值不能指示任何情况, 因为只要函数执行完就必定返回 true; 如果中途抛出异常, 该函数也没处理, 自然也不会有返回值。该设计不是很合理。

如果有足够的把握, 上面的“翻译”过程也可以在脑中进行, 无需写出来。

当代码“翻译”为“注释”后, 就可将提取出的逻辑与设计 and 需求进行比对, 检查其是否忠实体现了设计, 是否完整、准确地实现了需求。

需求和设计文档这里就不给出了, 对抽取出的逻辑参考相关文档进行检查不是太困难的事, 在此不展开。

## 第二步:

接下来要做的就是针对“常规输入”的可能取值进行核查。所谓“常规输入”就是指函数的参数, 这是函数的显式输入。

我们知道, 任何一段程序都可以用 IPO (Input-Process-Output) 模型来分析、检查。所以代码检视要做的也是针对所有输入的可能取值, 检查代码段是否有正确的处理, 能否产生正确的输出。

函数参数正是最显明明白的输入项, 对它的取值采取等价类划分和取边界值, 就能够检视出代码中是否能够覆盖到各类的输入组合。

示例代码段一中，函数有两个输入参数 hrdId 和 mnsInfos（余下两个是输出参数），我们一个个地看。

hrdId 在代码中仅有一处使用，即

```
NPCManagement::MonsterLoader::getMonsterType(hrdId, -1, monsterIds, mnsTypes);
```

这是一个外部调用。

第一条、检视规则：输入参数仅用于外部调用的情况，在代码检视中可不必关注。

因为通常情况下这样的输入参数不影响被检视函数走向，而是否影响被调用的外部函数、对象，则是应当在其他的检视活动中覆盖的内容。

该规则，我们在“常规输入”检查阶段跳过这一行。

第二个参数 mnsInfos 在整个函数中很多地方都有使用，因此要详细分析。

第一处用到是在初始化语句中：`ivmnsInfos.length(mnsInfos.length());`

我们说过对输入参数仅用于外部调用的情况可以不关注，因此这句略过。

第二处是作为循环终了判断条件出现：

```
for (CORBA::ULong i = 0; i < mnsInfos.length(); ++i)
```

这里提出第二条检视规则：输入参数参与循环判定或条件判定时，需对该参数取值进行分析，检查程序是否存在条件遗漏。

这里的取值情况比较明显，只需检查当 `mnsInfos.length()` 取值为 0 时程序是否有正常处理即可。从代码看出，取值为 0 时，跳过整个循环，输出参数（序列）长度都置为 0，如果这样的处理符合设计模型，就没有问题。

请注意斜体字部分，对代码实现是否有问题不应简单、孤立地判定，而应结合设计与需求进行检查，切记。

接下来是一个赋值语句，将 `mnsInfos[i].monsterInfos` 赋值给局部变量 `mnsStatusSeq`。

这里提出第三条检视规则：变量以某种方式与输入参数建立关联的，等同于输入参数对待。

这条规则的意思就是，变量与输入参数关联后，就同样适用于第一、第二两条规则。这里有点递归的意思。

因此，对 `mnsStatusSeq` 的后续使用，同样要与输入参数 `mnsInfos` 一样进行分析，由于分析原则一样，就不作重复说明。

另外需要指出的是，在分析 `hrdId` 时，实际上有一个变量 `mnsTypes` 与 `hrdId` 建立了关联——尽管不是赋值，但是通过函数调用产生了关联（`mnsTypes` 是函数 `getMonsterType` 的出参），因此对 `mnsTypes` 变量同样要进行相应的分析。

在此之后直接出现 `mnsInfos` 的地方全都是赋值，同样遵循规则三处理。

对全部的（2 个）直接输入参数分析完成后，就可以递归地对所有的“间接输入参数”（即前面提及的，通过某种方式与输入参数建立关联的局部变量，如 `mnsStatusSeq`、`mnsTypes` 等）进行分析。

所有输入参数分析完毕后，第二步就算是完成了。

### 第三步：

接下来需要覆盖的，就是真正的“祸患之源”，即“非常规输入”，或者叫“隐性输入”。

我们来看第二段代码：



## 示例代码段二

```
virtual ::std::auto_ptr<World::Quest::TaskResult>
run(World::QuestMgr::RoleProxy& roleProxy) {

    QuestPos notExistQuests;
    try
    {
        World::StateMgr::StateManager& stateMgr =
        Frame::kernelModule<World::StateMgr::StateManager>(owner_.provider().kernel());

        //获取 Quest
        CORBA::ULong processPos = 0;
        for(; (!cancel_) && processPos < questIDs_.length(); processPos++)
        {
            if(notExistQuests.find(processPos) != notExistQuests.end()) {
                continue;
            }

            try
            {
                World::Model::Quest questObj =
                Frame::kernelModule<World::Model::QuestCvt::QuestConvertMgr>(
                    owner_.provider().kernel()).cvtQuestById(
                    World::Model::Basic::Quest(questIDs_[processPos].in()));

                owner_.appendQuestState(
                    stateMgr.getQuest(
                        stateName_,
                        questObj));
            }
            catch(...) {
                STATEMGR_ERROR("acquire quest state catch exception");
            }
        }
    }
    STATEMGR_REPORT_EXCEPTION_PERISH("StateQueryOp::run catch exception");

    owner_.setIsDone(true);
    return World::Quest::TaskResult::_success();
}
```

在以上的代码段里，你能找到哪些“隐性输入”？

我个人将“隐性输入”划分为以下四个层次：

层次一，我们可以很容易找到：

`questIDs_.length()`、`quest` 的取值、`notExistQuests.find()` 的返回值和 `stateMgr.getRequest()` 的返回值。从宏观上说，凡是来自于本程序段之外的数据都应视为本程序段的输入，在进行程序走向和处理分析时，都需要予以关注。

如上所述的四个数值都属于该代码段的“非常规输入”，因此对它们可能的取值，都应该分别分析是否影响程序走向，是否有正确的处理。

`questIDs_.length()` 的处理可以参考常规输入的处理。

`questObj` 的取值则需要参考 `cvtQuestById` 函数的定义（或向该函数的编写者询问相应信息），要考虑有效 `id` 和无效 `id` 的情况（即对象找不到）。

同样 `notExistQuests.find()` 也会返回两种结果。

而 `getRequest()` 的结果要更复杂一些，包括正常的对象列表，空对象列表（没找到）和异常三种（即输入参数无效，`questObj` 的取值就是无效对象）。当然，在处理输入的 `questObj` 无效时，`getRequest()` 的返回可能是空对象列表，也可能是抛出异常，具体情况还是要再对 `getRequest()` 进行分析才能确定。

特别地，对于调用外部函数获取的输入值，如果代码编写者自己并不清楚其可能的取值范围，就需要从设计文档或该外部函数设计者处获取信息。

层次二，还有其他的“非常规输入”吗？结合对“输入”的定义，我们能够发现 `cancel_` 也是个输入值。尽管它是当前类定义的成员变量，对“类”这个单元来说是属于内部数据，但是对当前被检视的 `run` 函数而言还是个“外来户”。当然，根据这个标准，那些“下划线一族”也都是“外来户”，包括 `questIDs_`、`owner_`。

如果了解函数所在类的调用上下文，可以看出 `run` 函数是以线程方式运行的，`cancel_` 的取值在该函数运行过程中是可以改变的——相比于函数启动就固定的输入值而言，在过程中可变的输入值，更值得注意。

关注被检查对象运行过程中输入值的变化，是多线程模式下进行代码检视和单元测试的重点内容。

在这段代码中，`cancel_` 的取值尽管只有 `true`

和 `false` 两种，但是结合运行中的动态变化，就可以看出能够影响程序走向和处理的有四种状态：初始值为 `false`，且全程不变；初始值为 `true`，且全程不变；初始值为 `false`，且中段变为 `true`；初始值为 `true`，且中段变为 `false`。

最后一种状态对程序走向没有影响，而且结合程序其他代码段来看，也没有提供从 `true`→`false` 的转换，因此归为无效状态，不用测试/检查。

前两种状态属于静态输入，与“常规输入”进行同样检查即可。

而第三种状态属于“动态输入”，需要检查在输入发生变化时，程序是否能够按照预期进行处理。例如在 `cancel` 后是否能够正常销毁对象，是否能够正常处理对象队列而不出现越界现象……在这段简单的示例代码中没有出现这些问题，但稍复杂一些的代码就很有可能出现以上疏漏。

另外，在“层次一”中提到的 `questIDs_.length()` 其实与 `cancel_` 是同一性质的输入。从理论上说，它在循环中也是可以变化的——譬如在某次进入循环体后，由于外部动作导致 `questIDs_` 中的部分内容被删除，此时的 `questIDs_` 及其 `length()` 都会发生变化。因此 `questIDs_[processPos]` 这个访问就很可能引发一个越界异常。

当然，以上情况是基于 `questIDs_` 成员变量可以改变而言的，结合整个类的完整代码看，该变量是对象初始化时生成的，且在对象生命周期内没有提供改变的接口，因此不会发生以上异常情况。

第三个层次，是识别所谓的“共享资源”，这点在以上示例代码中没有直接体现。

所谓“共享资源”，指的是部分或全部代码共同分享的资源，包括但不限于全局变量（对象）、数据库连接、数据表内容、文件、内存块、线程锁或信号量……等等。当代码中出现数据库读写、文件读写等内容时，实际上都是对外部的“共享资源”发生了依赖。这些“共享资源”的取值范围或状态，就可能是代码中需要考虑处理的。同样的情况也发生在并发程序获取锁或信号量时，程序代码如果没有充分考虑到会影响锁或信号量的其他代码，就可能会导致死锁的发生。

在检视涉及“共享资源”的代码时，除了检查

代码对这些“共享资源”的取值是否有合适处理外，还要进一步检查当前检视代码与其他相关代码是否对“共享资源”的竞争进行正确处理，它们对“共享资源”的依赖，是否构成“依赖环”。

要将所有的共享依赖都梳理出来不是件简单的事，篇幅所限，这里就不展开讨论了。

第四个层次，是对程序中所有异常也作为“非常规输入”处理。

异常，即 **Exception**，是程序在遭遇非预期或不能处理的状况时，用于通知外层程序或调用者出现状况的一种机制。

从最简单的  $a=b/c$  可能发生的除零异常，到复杂的函数调用可以抛出多种类型的异常，这些都是程序中常见的情形。

在调用函数或进行数据处理时发生的异常，都是由外部传入/返回给当前函数（被检视函数）的，因此对于当前函数而言，异常 **Exception** 自然也应当作为一种“非常规输入”进行处理。

由于异常在代码段中通常是以“隐身”的方式存在着，除了函数声明时可能有的 **throw** 指示（有时候也没有）之外，难以寻觅它的踪影，因而它也是最容易遗漏的“非常规输入”，应当作为一个重点的代码检视对象。

在示例代码段二中，对 **cvtQuestById**、**appendQuestState**、**getQuest** 等方法的调用，都有可能遭遇它们抛出的异常。而从代码中也可以看到，该段程序对可能遭遇的异常确实进行了处理。

再看示例代码段一，它的逻辑简单，看起来信心满满，没有任何异常保护。

但是它真的不会遭遇任何异常吗？仔细检查可以看到其中有这样一条语句：

```
NPCManagement::MonsterLoader::getMonsterType(hrdId, -1, monsterIds, mnsTypes);
```

循着这条语句顺藤摸瓜，会看到 **getMonsterType** 方法是通过数据库查询实现的，且此处的数据库查询是通过新建数据库连接进行的。

在其中建立数据库连接的部分（以一个 **DBConnection** 构造函数的面目出现），明明白白地写出了有可能抛出各种类型的异常。换言之，当无法获取数据库连接时，**getMonsterType** 就会抛出异常，而示例代码段一中没有捕获该异常。

这个未捕获的异常是否会引发未定义的问题，就完全要看调用该方法的外层调用者的处理了一—生存还是死亡，这是一个问题。

即便现存的所有调用者都注意到这点并进行了异常保护，也无法保证今后新增加的调用者也都会遵循这个“传统”。最好的措施还是自我保护。

在进行代码检视时，对调用的外部函数、系统 **API** 等，都要仔细检查其是否可能抛出异常，并检查程序代码中对可能发生的异常，是否按照规范进行了适当处理。

对异常的处理分为两个阶段：捕获与处置。按捕获情况可分为三类：捕获特定异常，捕获所有异常，不捕获异常。

捕获异常后的处置方法可以分为两类：1、记录而不处理（吞噬异常），程序继续流转；2、记录并中断程序运行，重新抛出异常。无论是哪一种捕获和处置方法，是否“正确”都要看是否符合业务逻辑，是否符合设计规范，不能一概而论。

一般而言，对可能存在的异常不捕获是比较危险的。因为除非明确知道外围程序有异常保护机制，否则本程序对异常不处理会导致异常扩散，很有可能导致产品崩溃、错乱，发生无法预知的后果。

以上就是我所提倡的“代码检视三步走”的全部内容。受个人能力和当前工作重点所限，以上给出的内容既不全面也可能存在许多错漏之处。希望能够起到抛砖引玉的作用，引出更多专家的宝贵经验。





# 接口验证模式

**作者：原草莫莫** 10 余年的软件测试从业经验，获得 ISTQB 高级（测试分析）证书，高级工程师。对测试设计、性能测试、可靠性测试、需求的静态测试、可测试性等技术有一定的研究，对于测试技术应用、如何帮助产品进行测试技术改进有一定的理解。目前主要关注错误模式和人因工程等领域。

**摘要：**接口验证是软件测试中一个重要的方面。本文按被测对象与周边实体的消息处理关系将接口验证方式抽象成几种模式：C 模式、S 模式、C&S 模式、分发模式、异步模式等。然后按模式从接口契约定义、请求和响应配合等方面，给出接口验证的一般要求。

**关键词：**接口验证 测试模式 协议一致性

## 1. 相关概念

### 1.1 接口

这里所说的接口主要是指的是消息接口，是二个部件之间的通信契约，有发送方、接收方等方面的属性，同 GUI 接口、文件接口一样，它本质上属于一种输入、输出方式，只是它涉及到 2 个不同部件/实体，有请求/响应、有连接通道要求，由此带来超时、重发、重连等方面的一系列要求。

### 1.2 接口、流程、处理的关系

一个流程由一系列的处理、接口调用组成。

一个流程可能涉及多个不同部件，涉及多个不同的接口调用。

一个接口可能服务于多个流程，多个流程共用同一个接口。由此，接口验证里需要对同一个接口遍历不同的流程调用场景。

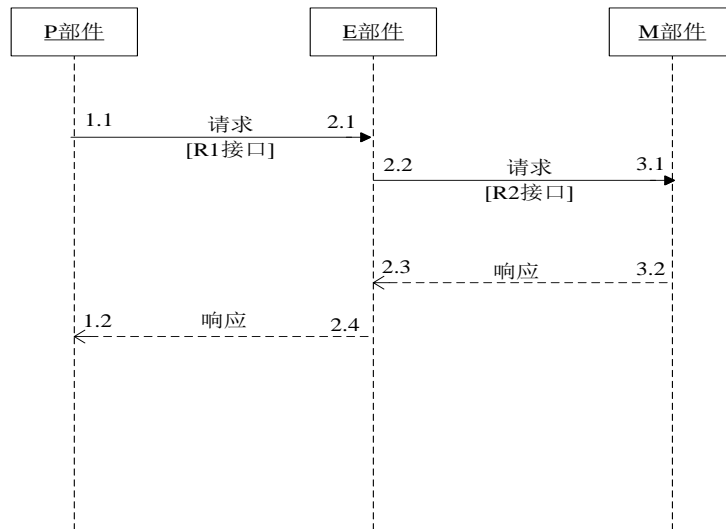
接口作为数据的一种形式，它影响流程的走向。

接口作为数据的一种形式，它影响流程的结果。

有些接口处理可能是纯接口的、只做中转、协议转换等。例：下面例子中的 E 部件接口；有些接口处理可能有较强的功能逻辑，根据需要可能还会进一步细化成内部接口。由此，接口验证可能需要针对接口处理作进一步的功能逻辑验证。

## 2. 一个例子

以下为某个处理的简化流程。P 部件发出请求，E 部件协议转换后转发给 M 部件，M 部件进业务逻辑处理后返回响应给 E 部件。



## 接口的测试设计思路:

- 列出与每个部件的交互点。包括：与 P 部件的交互点 1.1~1.2；与 E 部件的交互点 2.1~2.4；与 M 部件的交互点 3.1~3.2
- 对每个部件的每个交互点进行正常与异常方面的验证。

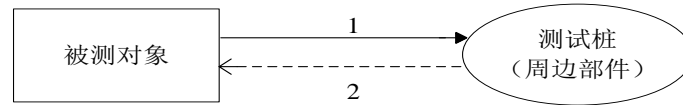
部件	对象	测试项	测试目的	说明
P 部件	R1 接口	1. 1-正常	发送 R1 请求消息正确性验证	依据接口契约定义，包括： <ul style="list-style-type: none"> <li>• 消息整体：协议、消息格式</li> <li>• 消息各参数格式遍历：是否可选、唯一性、类型、取值范围、长度及字段间一些其它取值约束等</li> </ul>
		1. 1-异常	发送 R1 请求消息异常	发送消息失败、失败重发
		1. 2-正常	接受 R1 响应消息处理正确性	响应消息格式正确性 响应消息返回码遍历验证，对 E 部件不同响应码，P 部件后续处理正确，并给出用户合适的错误提示
		1. 2-异常	接受 R1 消息响应异常	没有响应、响应超时、超时重发、收到重复响应

部件	对象	测试项	测试目的	说明
E 部件	R1/R2 接口	2.1-正常	收到 R1 请求消息参数合法性校验	<ul style="list-style-type: none"> <li>依据接口契约定义，同 1.1-正常</li> <li>收到重复消息</li> </ul>
		2.2-正常	发送 R2 请求消息正确性验证	同 1.1-正常
		2.2-异常	发送 R2 请求消息异常	同 1.1-异常
		2.3-正常	接受 R2 响应消息处理正确性验证	响应消息格式正确性 根据 M 部件不同响应码，E 部件给出相应的处理码转换
		2.3-异常	接受 R2 消息响应异常	没有响应、响应超时、超时重发、收到重复响应 根据 M 部件不同响应异常，E 部件给出相应的处理码转换供 P 部件决策
		2.4-正常	发送 R1 响应消息内容正确性验证	同 1.1-正常
		2.4-异常	发送 R1 响应消息异常	同 1.1-异常
M 部件	R2 接口	ALL	参数一致性	收到 R1 请求消息和发送 R2 请求消息中相关参数 收到 R2 响应消息和发送 R1 响应响应中相关参数
		3.1-正常	收到 R2 请求消息参数合法性校验	同 1.1-正常
		3.1-正常	收到 R2 请求消息参数取值遍历、逻辑验证	通过不同输入取值触发 M 部件不同处理逻辑
		3.2-正常	发送 R2 响应消息正确性验证	同 1.1-正常
		3.2-异常	发送 R2 响应消息异常	同 1.1-异常

### 3. 接口验证模式

#### 3.1 基本模式

- **C 模式：**被测对象作为客户端发送请求消息。一般来说，流程起点的接口（例子中的 P 部件接口）多数为 C 模式。



C模式

## 基本验证要求：

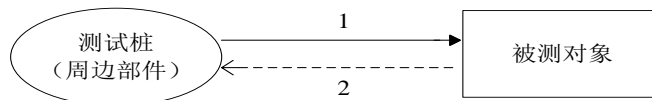
- 发送请求消息正确性。包括：协议、消息格式、各参数验证。
- 响应消息字段、错误码遍历。确认根据对端不同响应作了相应的正确处理。比如：根据错误码展示

正确的错误提示也为一种正确处理方式。

## 进一步验证要求：

- 考虑接口请求和响应配合上的异常，包括：
  - 请求发送异常：发送失败、失败重发。
  - 响应接受异常：无响应、响应超时、超时重发、收到重复请求

- **S 模式：**被测对象作为服务端接收请求，一般来说，流程终点的接口（例子中的 M 部件）多数为 S 模式。



S模式

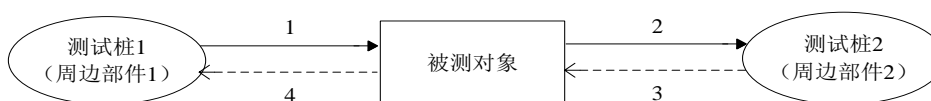
## 基本验证要求：

- 收到的请求消息参数合法性校验。包括：
  - 协议、消息格式的验证、非系统识别消息、存在非法字段、收到重复消息
  - 遍历各字段进行参数合法性校验：是否可选、唯一性、类型、取值范围、长度(<、=、>)等
- 遍历请求消息的各字段取值及组合，确认根据不同输入返回了不同的结果（可以等价）
- 发出响应消息正确性：协议、消息格式、各参数验证等。

S&C 模式：被测对象既作为服务端接收请求又作为服务端发送请求。一般来说，流程中点（例子是的 E 部件）多数为 S&C 模式。

如果将周边部件 1 作为被测对象一部分，它即是 C

如果将周边部件 2 作为被测对象一部分，它即是 S



S&C模式

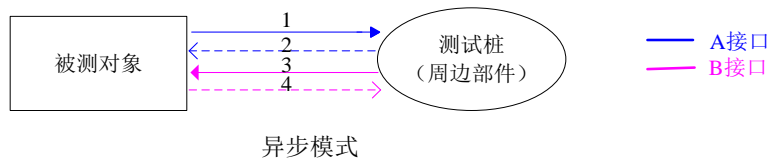
**基本验证要求：**除了 C 模式和 S 模式的基本验证要求，考虑对不同消息间相关参数一致性进行校验。

例：R1 接口中 X 参数取值为 1-255，经过转换后的 R2 接口中相应的 X 参数取值也应为 1-255。

**进一步验证要求：**参见 C 模式和 S 模式中的进一步验证要求。

## 3.2 复合模式

- **异步模式：**被测对象发出消息后，对端立即响应，对端在处理结束后再发送回执消息给部件，部件根据对端所给出的消息作出相应的处理，流程结束。一般来说，如果对端处理较为复杂、为避免被测对象长时间被阻塞，会采用此通信方式。



对于异步模式，可以拆分为 2 对消息，但这 2 对消息是基于事务、有状态的。因此，对这类消息的验证除了基本模式 C 和 S 的验证要求外，还需要考虑 2 对消息关系的配合对被测对象的状态影响验证。

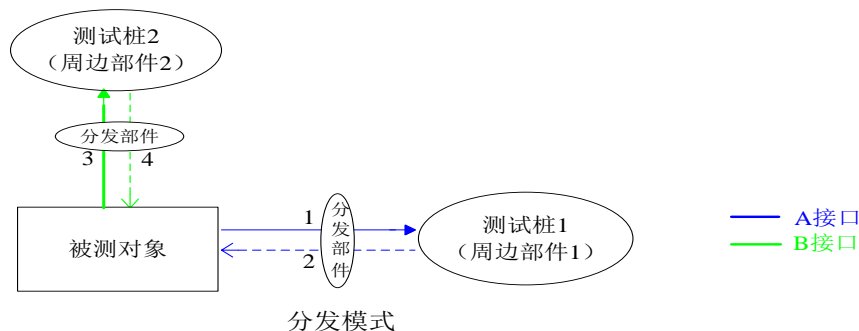
以图示为例，被测对象的验证内容包括：

- 对 A 接口的验证。参见 C 模式
- 对 B 接口的验证。参见 S 模式
- A 和 B 接口的配合：

条件：A 接口处理失败、未收到 B 接口消息、B 接口处理失败、B 接口处理成功

结果：被测对象的状态、数据

- **分发模式：**需要将消息采用同步方式向其它多个部件进行分发，待消息收齐后才能决定自身的最终状态。例：被测对象通过分发部件将数据同步分发给不同的部件。需要说明的是：图示中的分发部件，这时从物理上来说，可能看到的只是一个部件，由它统一接受和分发消息，但从逻辑上来说，它是代表了不同部件的接口处理的。



对于分发模式一般也是基于事务、有状态的，但由于涉及到了 2 个以上的周边部件，还需要考虑对不同部件的接口消息处理结果进行结合。

以图示为例，被测对象的验证内容一般包括：

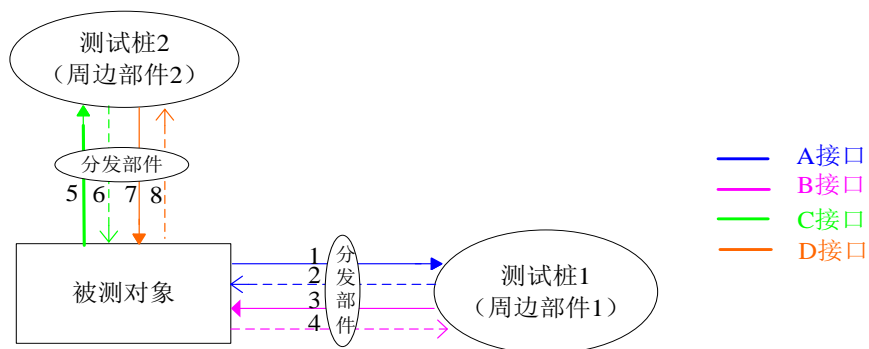
- 对 A 接口的验证。（参见 C 模式）
- 对 B 接口的验证。（参见 C 模式）

• 对部件 1 和部件 2 处理结果结合验证：

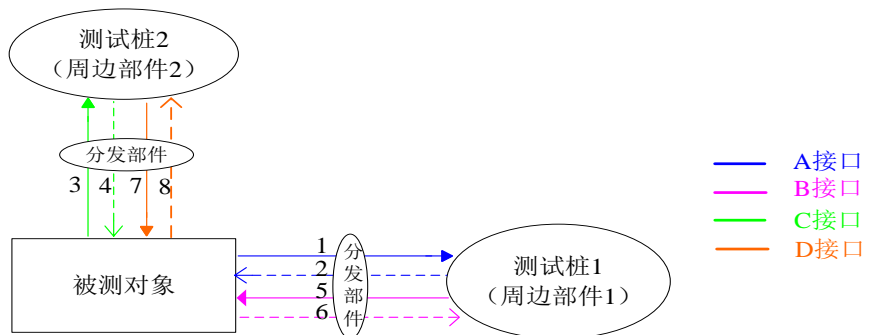
条件：1 成功 2 成功；1 成功 2 失败；1 失败 2 成功；1 失败 2 失败

结果：被测对象的状态、数据

- **异步分发模式：**即采用异步方式进行消息分发，为异步和分发模式的结合。比较典型的是数据同步异步接口。被测对象 通过分发部件 同时将数据同步消息通知分发给不同的部件，各个不同部件收到通知后再向被测对象请求获取同步数据。如果通知有优先级，例：部件 1> 部件 2，待部件 1 处理完再通知部件 2，即为异步分发模式 1。如果多个部件的分发并行执行（一般来说，部件 1 和部件 2 可能代表的是同类部件的不同物理实例），即为异步分发模式 2。



异步分发模式1



异步分发模式2

对于异步分发模式，也即异步+分发模式的组合。此时被测对象涉及到 2 种类型的消息配合：同一个部件的通知和回执的组合；不同部件间的消息处理结果的配合。由此，被测对象的状态迁移会更为复杂些。

以图示为例，被测对象的验证内容包括：

- 对 A 接口的验证。（参见 C 模式）
- 对 B 接口的验证。（参见 S 模式）
- 对 C 接口的验证。（参见 C 模式）
- 对 D 接口的验证。（参见 D 模式）
- 对 A 和 B 接口的配合验证。（参见异步模式）
- 对 C 和 D 接口的配合验证。（参见异步模式）
- 对部件 1 和部件 2 处理结果组合验证。（参见分发模式）



## 4. 相关说明

### ➤ 参数合法性检验策略

如果业务流程涉及多次转发，原则上由逻辑处理部件进行接口参数的强校验；其它转发部件（例：E 部件）进行弱校验。

#### 消息序列验证

如果不同的接口消息之间是基于事务、有状态的，则还需要考虑消息序列异常的问题，无论是何种模式。其验证点包括：消息乱序、少传消息包、多传消息包、传重复消息包、事务超时后收到消息等。

#### 接口可靠性保证

- 对于重发的验证，一般来说，重发机制中需要有重发策略、重发次数方面的考虑，不能出现消息反复重发引发消息风暴的问题。
- 对于超时的验证，需要考虑各部件超时配置不一致的问题。
- 对于处理失败造成双方数据不一致问题，需要有事务号、回滚或补偿机制等方面的设计考虑。

### ➤ 接口验证的不同阶段

对于接口验证在单部件测试、点-点接口联调、E2E 联合测试等不同阶段都有所涉及。一般来说：

**单部件测试：**理论上通过测试桩可以模拟对端各种情况，对于真实实体只能通过系统状态预置、输入数据从外部触发。所以，能在单部件测试考虑的尽可能放到单部件去做，至少保证单部件自身是 OK 的。

**点-点接口联调：**如果将 2 个部件看作一个整体的话，则相当于单部件测试。对于部件-部件间的接口无法通过测试桩来模拟，需要通过外部驱动输入。另外还需要关注部件-部件间的网络连接，包括：是否可正常建立连接、连接中断后是否会重连、连接吊死与释放、时断时续等。

**E2E 联合测试：**所有内部部件均为真实实体，对于接口间配合的问题（例：事务或数据一致性问题）可以考虑放到此考虑。除此还需要关注与外部部件间的接口对接测试。



## Call For Paper

### 会议愿景：

发挥 CSTQB 优势，搭建一个国内一流的软件测试技术交流平台

### 会议内容：

CSTQB 软件测试高峰论坛一直致力于打造一个高端的专注于软件测试与质量及其相关领域的大会。在 2-4 天的会议期间，大会将以产业交流与学术讨论并重。不仅包括软件测试领域的经验交流、技术和方法的探讨，还包括学术标准和前沿趋势的讨论。



### 会议目标听众：

IT 经理、IT 研发经理、项目经理、测试工程师、测试经理、高校讲师等。

### 会议日期：2012 年 9 月

### 会议地点：

主会场—上海

分会场—北京、广州、江苏、黑龙江、成都等。

### 参与方式：详情可联系 [info@cstqb.cn](mailto:info@cstqb.cn)

- 研讨会 (Tutorial / Workshop)
- 主题演讲 (Conference Presentation)
- 产品展示 (Products / Services Exhibition)

### 往届会议：<http://www.cstqbforum.com/>



去年的第二届 CSTQB 国际软件测试高峰论坛主要围绕“分享经验，推进国际软件测试技术交流”主题展开，吸引了来自中国、日本、韩国、德国、挪威、以色列等国权威专家进行主题演讲。来自同济大学、华为、宝马、HP、Intel、eBay、IBM、Sony 等国内各大企业、高等院校、科研院所的近 150 位软件测试专家们参加了本次高峰论坛。参会代表们在大会的专题讨论中纷纷上台分享测试经验和案例，并就云测试、敏捷测试、游戏测试、汽车领域测试、测试标准、测试团队建设等热点议题展开了热烈的讨论。



## 上海滔瑞信息技术有限公司 (imbus Shanghai)

### ISTQB 认证培训专业机构

#### 一、公司简介

上海滔瑞信息技术有限公司 (imbus Shanghai) 是德国知名企业 imbus AG 的中国子公司, 作为德国软件测试专家, 我们致力于为客户提供高质量的**软件测试服务(imbus testing service)**、**软件测试培训(imbus Academy)**和**测试管理工具与解决方案(imbus TestBench)**。

滔瑞公司对不同行业的客户提供了软件测试外包、测试人员外派、测试流程改进、测试团队建设、测试人员培训、软件测试管理工具及解决方案等全方位的软件测试相关解决方案。我们的客户涉及 IT、通讯、电信、金融、自动化控制、电子商务、医疗、教育、传媒、工业制造、能源等多种行业。



#### 二、软件测试学院简介(imbus Academy)

上海滔瑞信息技术有限公司 (imbus Shanghai) 是 ISTQB 中国分会 CSTQB 的核心发起单位和副理事长单位, 国内首家 ISTQB 授权培训机构, 公司拥有国际化的讲师团队, 和挪威、美国、英国、德国、以色列、日本等多个国家的测试专家保持密切的合作关系, 可为客户提供最为专业的软件测试培训和咨询服务。

截止到 2011 年 11 月份, 上海滔瑞信息技术有限公司 (imbus Shanghai) 已经为全国 100 多家知名企业进行测试培训和咨询, 超过 60 家世界 500 强企业客户, ISTQB 培训参与人数超过 2000 人, ISTQB 认证人数超过 1500 人。客户遍布上海、北京、广州、大连、深圳、珠海、武汉、成都、无锡、南京、杭州、昆山和香港等全国各地, 二次培训率超过 90%, 培训满意度超过 95%!



#### 三、2012 年测试培训课程计划

2012 年 ISTQB 认证培训公开课计划			
课程名称	课程简称	培训天数	公开课时间
ISTQB Foundation Level	CTFL	4 天	4 月 19 日—22 日
			7 月 19 日—22 日
			8 月 23 日—26 日
			10 月 25 日—28 日
			12 月 20 日—23 日
ISTQB Advanced Level -Test Manager	CTAL-TM	5 天	3 月 21 日—25 日
			9 月 19 日—23 日
ISTQB Advanced Level -Test Analyst	CTAL-TA	5 天	5 月 23 日—27 日
			11 月 21 日—25 日

#### 联系方式:

上海滔瑞信息技术有限公司 (imbus Shanghai) 培训部:

联系人: 胡小姐 咨询热线: (+86) 21- 50274732-19

联系人: 王小姐 咨询热线: (+86) 21- 50274732-21

E-mail: [training@imbus.cn](mailto:training@imbus.cn)

中国网址: [www.imbus.cn](http://www.imbus.cn)

德国网站: [www.imbus.de](http://www.imbus.de)

# 测试人



Software Testing Club

软件测试俱乐部 出品

投稿联系: [SWTC@CSTQB.CN](mailto:SWTC@CSTQB.CN)

地址: 中国 上海杨浦区彰武路1号同济大厦A座812室