泽众软件测试网
电子期刊

SPASVO

第十五期

2013年3月

最好的职业生涯规划的文章

从1.5k到18k，一个程序员的5年成长之路

职场新人“七宗罪”

为什么集成测试比单元测试更重要

软件测试人员的华丽转身

测试驱动开发上的五大错误

自动化测试开发人员的十八般武器

上海泽众软件电子期刊

2013 年 3 月 第十五期

主办单位：上海泽众软件科技有限公司

联系电话：021-61079698 传真：021-61079698 转 8017

意见反馈：fangmh@spasvo.com 投稿：wangmf@spasvo.com

公司地址：上海市普陀区曹杨路 450 号绿地和创大厦 18 楼 1801 室

邮政编码：200063

公司主页：www.spasvo.com 论坛：bbs.spasvo.com

目录

泽众软件——专业软件测试门户
2013 年 3 月 第十五期 电子杂志期刊

最好的职业生涯规划的文章

职业的选择，总的来说，无非就是销售、市场、客服、物流、行政、人事、财务、技术、管理几大类，有趣的现象是，500强的CEO当中最多的是销售出 身，第二多的人是财务出身，这两者加起来大概超过95%。现代IT行业也有技术出身成为老板的，但实际上，后来他们还是从事了很多销售和市场的工作，并且 表现出 色，公司才获得了成功，完全靠技术能力成为公司老板的，几乎没有。这是有原因的，因为销售就是一门跟人打交道的学问，而管理其实也是跟人打交道的学问，这 两者之中有很多相通的东西，他们的共同目标就是“让别人去做某件特定的事情。”而财务则是从数字的层面了解生意的本质，从宏观上看待生意的本质，对于一个 生意是否挣钱，是否可以正常运作有着最深刻的认识。

公司小的时候是销售主导公司，而公司大的时候是财务主导公司，销售的局限性在于只看人情不看数字，财务的局限性在于只看数字不看人情。公司初期，运营 成本低，有订单就活得下去，跟客户也没有什么谈判的条件，别人肯给生意做已经谢天谢地了，这个时候订单压倒一切，客户的要求压倒一切，所以当然要顾人情。 公司大了以后，一切都要规范化，免得因为不规范引起一些不必要的风险，同时运营成本也变高，必须提高利润率，把有限的资金放到最有产出的地方。对于上市公 司来说，股东才不管你客户是不是最近出国，最近是不是那个省又在搞严打，到了时候就要把业绩拿出来，拿不出来就抛股票，这个时候就是数字压倒一切。

前两天听到有人说一句话觉得很有道理，开始的时候我们想“能做什么?”，等到公司做大了有规模了，我们想“不能做什么。”很多人在工作中觉得为什 么领导这么保守，这也不行那也不行，错过很多机会。很多时候是因为，你还年轻，你想的是“能做什么”，而作为公司领导要考虑的方面很多，他比较关心“不能 做什么”。

我并非鼓吹大家都去做销售或者财务，究竟选择什么样的职业，和你究竟要选择什么样的人生有关系，有些人就喜欢下班按时回家，看看书听听音乐，那也 挺好，但就不适合找个销售的工作了，否则会是折磨自己。有些人就喜欢出风头，喜欢成为一群人的中心，如果选择做财务工作，大概也干不久，因为一般老板不喜 欢财务太积极，也不喜欢财务话太多。先想好自己要过怎样的人生，再决定要找什么样的职业。有很多的不快乐，其实是源自不满足，而不满足，很多时候是源自于 心不定，而心不定则是因为不清楚究竟自己要什么，不清楚要什么的结果就是什么都想要，结果什么都没得到。

我想，我们还是因为生活而工作，不是因为工作而生活，生活是最要紧的，工作只是生活中的一部分。我总是觉得生活的各方方面都是相互影响的，如果生 活本身一团乱麻，工作也不会顺利。所以要有娱乐、要有社交、要锻炼身体，要有和睦的家庭……最要紧的，要开心，我的两个销售找我聊天，一肚子苦水，我问他 们，2年以前，你什么都没有，工资不高，没有客户关系，没有业绩，处于被开的边缘，现在的你比那时条件好了很多，为什么现在却更加不开心了?如果你做得越 好越不开心，那你为什么还要工作?首先的首先，人还是要让自己高兴起来，让自己心态好起来，这种发自内心的改变会让你更有耐心，更有信心，更有气质，更能 包容……否则，看看镜子里的你，你满意么?

有人会说，你说得容易，我每天加班，不加班老板就会把我炒掉，每天累得要死，哪有时间娱乐、社交、锻炼?那是人们把目标设定太高的缘故，如果你还 在动不动就会被老板炒掉的边缘，那么你当然不能设立太高的目标，难道你还想每天去打高尔夫?你没时间去健身房锻炼身体，但是上下班的时候多走几步可以吧， 有楼梯的时候走走楼梯不走电梯可以吧?办公的间隙扭扭脖子拉拉肩膀做做俯卧撑可以吧?谁规定锻炼就一定要拿出每天2个小时去健身房?你没时间社交，每月参 加郊游一次可以吧，周末去参加个什么音乐班，绘画班之类的可以吧，去尝试认识一些同行，和他们找机会交流交流可以吧?开始

的时候总是有些难的，但迈出这一 步就会向良性循环的方向发展。而每天工作得很苦闷，剩下的时间用来咀嚼苦闷，只会陷入恶性循环，让生活更加糟糕。

跳槽与积累

首先要说明，工作是一件需要理智的事情，所以不要在工作上耍个性，天涯上或许会有人觉得你很有个性而叫好，煤气公司电话公司不会因为觉得你很有个 性而免了你的帐单。当你很帅地炒掉了你的老板，当你很酷地挖苦了一番招聘的 HR，账单还是要照付，只是你赚钱的时间更少了，除了你自己，没人受损失。

我并不反对跳槽，但跳槽决不是解决问题的办法，而且频繁跳槽的后果是让人觉得没有忠诚度可言，而且不能安心工作。现 在很多人从网上找工作，很多找工作的网站常常给人出些馊主意，要知道他们是盈利性企业，当然要从自身盈利的角度来考虑，大家越是频繁跳槽频繁找工作他们越 是生意兴隆，所以鼓动人们跳槽是他们的工作。所以他们会常常告诉你，你拿的薪水少了，你享受的福利待遇差了，又是“薪情快报”又是“赞叹自由奔放的灵 魂”。至于是否会因此让你不能安心，你跳了槽是否解决问题，是否更加开心，那个，他们管不着。

要跳槽肯定是有问题，一般来说问题发生了，躲是躲不开的，很多人跳槽是因为这样或者那样的不开心，如果这种不开心，在现在这个公司不能解决，那么在下 一个公司多半也解决不掉。你 必须相信，90%的情况下，你所在的公司并没有那么烂，你认为不错的公司也没有那么好。就像围城里说的，“城里的人拼命想冲出来，而城外的人拼命想冲进 去。”每个公司都有每个公司的问题，没有问题的公司是不存在的。换个环境你都不知道会碰到什么问题，与其如此，不如就在当下把问题解决掉。很多问题当你真 的想要去解决的时候，或许并没有那么难。有的时候你觉得问题无法解决，事实上，那只是“你觉得”。

人生的曲线应该是曲折向上的，偶尔会遇到低谷但大趋势总归是曲折向上的，而不是象脉冲波一样每每回到起点，我见过不少面试者，30多岁了，四五份工作 经历，每次多则3年，少则1年，30多岁的时候回到起点从一个初级职位开始干起，拿基本初级的薪水，和20多岁的年轻人一起竞争，不觉得有点辛苦么?这种 日子好过么？不少面试者，30多岁了，四五份工作经历，每次多则3年，少则1年，30多岁的时候回到起点从一个初级职位开始干起，拿基本初级的薪水，和 20多岁的年轻 人一起竞争，不觉得有点辛苦么?这种日子好过么?

资本靠的就是积累，这种积累包括人际关系，经验，人脉，口碑……如果常常更换行业，代表几年的积累付之东流，一切从头开始，如果换了两次行 业，35岁的时候大概只有5年以下的积累，而一个没有换过行业的人至少有了10年的积累，谁会占优势?工作到2-3年的时候，很多人觉得工作不顺利，好像到了一个瓶颈，心情烦闷，就想辞职，乃至换一个行业，觉得这样所有一切烦恼都可以抛开，会好很多。其实这样做只是让你从头开始，到了时候还是会发生和原来 行业一样的困难，熬过去就向上跨了一大步，要知道每个人都会经历这个过程，每个人的职业生涯中都会碰到几个瓶颈，你熬过去了而别人没有熬过去你就领先了。 跑长跑的人会知道，开始的时候很轻松，但是很快会有第一次的难受，但过了这一段又能跑很长一段，接下来会碰到第二次的难受，坚持过了以后又能跑一段，如此 往复，难受一次比一次厉害，直到坚持不下去了。大多数人第一次就坚持不了了，一些人能坚持到第二次，第三次虽然大家都坚持不住了，可是跑到这里的人也没几 个了，这点资本足够你安稳活这一辈子了。

一份工作到两三年的时候，大部分人都会变成熟手，这个时候往往会陷入不断的重复，有很多人会觉得厌倦，有些人会觉得自己已经搞懂了一切，从而懒得 去寻求进步了。很多时候的跳槽是因为觉得失去兴趣了，觉得自己已经完成比赛了。其实这个时候比赛才刚刚开始，工作两三年的人，无论是客户关系，人脉，手 下，和领导的关系，在业内的名气……还都是远远不够的，但稍有成绩的人总是会自我感觉良好的，每个人都觉得自己跟客户关系铁得要命，觉得自己在业界的口碑 好得很。其实可以肯

定地说，一定不是，这个时候，还是要拿出前两年的干劲来，稳扎稳打，积累才刚刚开始。

你足够了解你的客户吗?你知道他最大的烦恼是什么吗?你足够了解你的老板么?你知道他最大的烦恼是什么吗?你足够了解你的手下么?你知道他最大的烦恼 是什么吗?如果你不知道，你凭什么觉得自己已经积累够了?如果你都不了解，你怎么能让他们帮你的忙，做你想让他们做的事情?如果他们不做你想让他们做的事 情，你又何来的成功?

等待

这是个浮躁的人们最不喜欢的话题，本来不想说这个话题，因为会引起太多的争论，而我又无意和人争论这些，但是考虑到对于职业生涯的长久规划，这是一个躲避不了的话题，还是决定写一写，不爱看的请离开吧。

并不是每次穿红灯都会被汽车撞，并不是每个罪犯都会被抓到，并不是每个错误都会被惩罚，并不是每个贪官都会被枪毙，并不是你的每一份努力都会得到 回报，并不是你的每一次坚持都会有人看到，并不是你每一点付出都能得到公正的回报，并不是你的每一个善意都能被理解……这个，就是世道。好吧，世道不够 好，可是，你有推翻世道的勇气么?如果没有，你有更好的解决办法么?有很多时候，人需要一点耐心，一点信心。每个人总会轮到几次不公平的事情，而通常，安 心等待是最好的办法。

有很多时候我们需要等待，需要耐得住寂寞，等待属于你的那一刻。周润发等待过，刘德华等待过，周星驰等待过，王菲等待过，张艺谋也等待过……看到 了他们如今的功成名就的人，你可曾看到当初他们的等待和耐心?你可曾看到金马奖影帝在街边摆地摊?你可曾看到德云社一群人在剧场里给一位观众说相声?你可 曾看到周星驰的角色甚至连一句台词都没有?每一个成功者都有一段低沉苦闷的日子，我几乎能想象得出来他们借酒浇愁的样子，我也能想象得出他们为了生存而挣 扎的窘迫。在他们一生最中灿烂美好的日子里，他们渴望成功，但却两手空空，一如现在的你。没有人保证他们将来一定会成功，而他们的选择是耐住寂寞。如果当 时的他们总念叨着“成功只是属于特权阶级的”，你觉得他们今天会怎样?

曾经我也不明白有些人为什么并不比我有能力却要坐在我的头上，年纪比我大就一定要当我的领导么?为什么有些烂人不需要努力就能赚钱?为什么刚刚改 革开放的时候的人能那么容易赚钱，而轮到我们的时候，什么事情都要正规化了?有一天我突然想，我还在上学的时候他们就在社会里挣扎奋斗了，他们在社会上奋 斗积累了十几二十年，我们新人来了，他们有的我都想要，我这不是在要公平，我这是在要抢劫。因为我要得太急，因为我忍不住寂寞。二十多岁的男人，没有钱， 没有事业，却有蓬勃的欲望。

人总是会遇到挫折的，人总是会有低潮的，人总是会有不被人理解的时候的，人总是有要低声下气的时候，这些时候恰恰是人生最关键的时候，因为大家都 会碰到挫折，而大多数人过不了这个门槛，你能过，你就成功了。在这样的时刻，我们需要耐心等待，满怀信心地去等待，相信，生活不会放弃你，机会总会来的。 至少，你还年轻，你没有坐牢，没有生治不了的病，没有欠还不起的债。比你不幸的人远远多过比你幸运的人，你还怕什么?路要一步步走，虽然到达终点的那一步 很激动人心，但大部分的脚步是平凡甚至枯燥的，但没有这些脚步，或者耐不住这些平凡枯燥，你终归是无法迎来最后的那些激动人心。

逆境，是上帝帮你淘汰竞争者的地方。要知道，你不好受，别人也不好受，你坚持不下去了，别人也一样，千万不要告诉别人你坚持不住了，那只能让别人获得坚持的信心，让竞争者看着你微笑的面孔，失去信心，退出比赛。胜利属于那些有耐心的人。

在最绝望的时候，我会去看电影《The Pursuit of Happyness》《Jerry Maguire》，让自己重新鼓起勇气，

因为，无论什么时候，我们总还是有希望。当所有的人离开的时候，我不失去希望，我不放弃。每天下班坐在车里，我喜 欢哼着《隐形的翅膀》看着窗外，我知道，我在静静等待，等待属于我的那一刻。

**网友的话我很喜欢，抄录在这里：

每个人都希望，自己是独一无二的特殊者

含着金匙出生、投胎到好家庭、工作安排到电力局拿1w 月薪这样的小概率事件，当然最好轮到自己

红军长征两万五、打成右派反革命、胼手胝足牺牲尊严去奋斗，最好留给祖辈父辈和别人

自然，不是每个吃过苦的人都会得到回报

但是，任何时代，每一个既得利益者身后，都有他的祖辈父辈奋斗挣扎乃至流血付出生命的身影

羡慕别人有个好爸爸，没什么不可以

问题是，你的下一代，会有一个好爸爸吗?

至于问到为什么不能有同样的赢面概率?我只能问：为什么物种竞争中，人和猴子不能有同样的赢面概率?

物竞天择。猴子的灵魂不一定比你卑微，但你身后有几十万年的类人猿进化积淀。

入对行跟对人

在中国，大概很少有人是一份职业做到底的，虽然如此，第一份工作还是有些需要注意的地方，有两件事情格外重要，第一件是入行，第二件事情是跟人。 第一份工作对人最大的影响就是入行，现代的职业分工已经很细，我们基本上只能在一个行业里成为专家，不可能在多个行业里成为专家。很多案例也证明即使一个 人在一个行业非常成功，到另外一个行业，往往完全不是那么回事情，“你想改变世界，还是想卖一辈子汽水?”是乔布斯邀请百事可乐总裁约翰?斯考利加盟苹果 时所说的话，结果这位在百事非常成功的约翰，到了苹果表现平平。其实没有哪个行业特别好，也没有哪个行业特别差，或许有报道说哪个行业的平均薪资比较高， 但是他们没说的是，那个行业的平均压力也比较大。看上去很美的行业一旦进入才发现很多地方其实并不那么完美，只是外人看不见。

说实话，我自己都没有发大财，所以我的建议只是让人快乐工作的建议，不是如何发大财的建议，我们只讨论一般普通打工者的情况。我认为选择什么行业 并没有太大关系，看问题不能只看眼前。比如，从前年开始，国家开始整顿医疗行业，很多医药公司开不下去，很多医药行业的销售开始转行。其实医药行业的不景 气是针对所有公司的，并非针对一家公司，大家的日子都不好过，这个时候跑掉是非常不划算的，大多数正规的医药公司即使不做新生意撑个两三年总是能撑的，大 多数医药销售靠工资撑个两三年也是可以撑的，国家不可能永远捏着医药行业不放的，两三年以后光景总归还会好起来的，那个时候别人都跑了而你没跑，那时的日 子应该会好过很多。有的时候觉得自己这个行业不行了，问题是，再不行的行业，做得人少了也变成了好行业，当大家都觉得不好的时候，往往却是最好的时候。大 家都觉得金融行业好，金融行业门槛高不说，有多少人削尖脑袋要钻进去，竞争激励，进去以后还要时时提防，一个疏忽，就被后来的人给挤掉了，压力巨大，又如 何谈得上快乐?也就未必是“好”工作了。

太阳能这个东西至今还不能进入实际应用的阶段，但是中国已经有7家和太阳能有关的公司在纽交所上市了，国美苏宁永乐其实是贸易型企业，也能上市， 鲁泰纺织连续10年利润增长超过50%，卖茶的一茶一座，卖衣服的海澜之家都能上市……其实选什么行业真的不重要，关键是怎么做。事情都是人做出来的，关 键是人。

有一点是需要记住的，这个世界上，有史以来直到我们能够预见得到的未来，成功的人总是少数，有钱的人总是少数，大多数人是一般的，普通的，不太成 功的。因此，大多数人的做法和看法，往往都不是距离成功最近的做法和看法。因此大多数人说好的东西不见得好，大多数人说不好的东西不见得不好。大多数人都 去炒股的时候说明跌只是时间问题，大家越是热情高涨的时候，跌的日子越近。大多数人买房子的时候，房价不会涨，而房价涨的差不多的时候，大多数人才开始买 房子。不会有这样一件事情让大家都变成功，发了财，历史上不曾有过，将来也不会发生。有些东西即使一时运气好得到了，还是会在别的时候别的地方失去的。

年轻人在职业生涯的刚开始，尤其要注意的是，要做对的事情，不要让自己今后几十年的人生总是提心吊胆，更不值得为了一份工作赔上自己的青春年华。我 的公司是个不行贿的公司，以前很多人不理解，甚至自己的员工也不理解，不过如今，我们是同行中最大的企业，客户乐意和我们打交道，尤其是在国家打击腐败的 时候，每个人都知道我们做生意不给钱的名声，都敢于和我们做生意。而勇于给钱的公司，不是倒了，就是跑了，要不就是每天睡不好觉，人还是要看长远一点。很 多时候，看起来最近的路，其实是最远的路，看起来最远的路，其实是最近的路。

跟对人是说，入行后要跟个好领导好老师，刚进社会的人做事情往往没有经验，需要有人言传身教。对于一个人的发展来说，一个好领导是非常重要的。所谓“好”的标准，不是他让你少干活多拿钱，而是以下三个。

首先，好领导要有宽广的心胸，如果一个领导每天都会发脾气，那几乎可以肯定他不是个心胸宽广的人，能发脾气的时候却不发脾气的领导，多半是非常厉 害的领导。中国人当领导最大的毛病是容忍不了能力比自己强的人，所以常常可以看到的一个现象是，领导很有能力，手下一群庸才或者手下一群闲人。如果看到这 样的环境，还是不要去的好。

其次，领导要愿意从下属的角度来思考问题，这一点其实是从面试的时候就能发现的，如果这位领导总是从自己的角度来考虑问题，几乎不听你说什么，这 就危险了。从下属的角度来考虑问题并不代表同意下属的说法，但他必须了解下属的立场，下属为什么要这么想，然后他才有办法说服你，只关心自己怎么想的领导 往往难以获得下属的信服。

第三，领导敢于承担责任，如果出了问题就把责任往下推，有了功劳就往自己身上揽，这样的领导不跟也罢。选择领导，要选择关键时刻能抗得住的领导，能够为下属的错误买单的领导，因为这是他作为领导的责任。

有可能，你碰不到好领导，因为，中国的领导往往是屁股决定脑袋的领导，因为他坐领导的位置，所以他的话就比较有道理，这是传统观念官本位的误区， 可能有大量的这种无知无能的领导，只是，这对于你其实是好事，如果将来有一天你要超过他，你希望他比较聪明还是比较笨?相对来说这样的领导其实不难搞定， 只是你要把自己的身段放下来而已。多认识一些人，多和比自己强的人打交道，同样能找到好的老师，不要和一群同样郁闷的人一起控诉社会，控诉老板，这帮不上 你，只会让你更消极。和那些比你强的人打交道，看他们是怎么想的，怎么做的，学习他们，然后跟更强的人打交道。

选择

我们每天做的最多的事情，其实是选择，因此在谈职业生涯的时候不得不提到这个话题。

我始终认为，在很大的范围内，我们究竟会成为一个什么样的人，决定权在我们自己，每天我们都在做各种各样的选择，我 可以不去写这篇文章，去别人的帖子拍拍砖头，也可以写下这些文字，帮助别人的同时也整理自己的思路，我可以多注意下格式让别人易于阅读，也可以写成一堆， 我可以就这样发上来，也可以在发以前再看几遍，你可以选择不刮胡子就去面试，也可以选择出门前照照镜子……每天，每一刻我们都在做这样那样的决定，我们可 以漫不经心，也可以多花些心思，成千上万的小选择累计起来，就决定了最终我们是个什么样的人。

从某种意义上来说我们的未来不是别人给的，是我们自己选择的，很多人会说我命苦啊，没得选择阿，如果你认为“去微软还是去 IBM”“上清华还是上 北大”“当销售副总还是当厂长”这种才叫选择的话，的确你没有什么选择，大多数人都没有什么选择。但每天你都可以选择是否为客户服务更周到一些，是否对同 事更耐心一些，是否把工作做得更细致一些，是否把情况了解得更清楚一些，是否把不清楚的问题再弄清楚一些……你也可以选择在是否在痛苦中继续坚持，是否抛 弃掉自己的那些负面的想法，是否原谅一个人的错误，是否相信我在这里写下的这些话，是否不要再犯同样的错误……生活每天都在给你选择的机会，每天都在给你 改变自己人生的机会，你可以选择赖在地上撒泼打滚，也可以选择咬牙站起来。你永远都有选择。有些选择不是立杆见影的，需要累积，比如农民可以选择自己常常 去浇地，也可以选择让老天去浇地，诚然你今天浇水下去苗不见得今天马上就长出来，但常常浇水，大部分苗终究会长出来的，如果你不浇，收成一定很糟糕。

每天生活都在给你机会，他不会给你一叠现金也不会拱手送你个好工作，但实际上，他还是在给你机会。我的家庭是一个普通的家庭，没有任何了不起的社 会关系，我的父亲在大学毕业以后就被分配到了边疆，那个小县城只有一条马路，他们那一代人其实比我们更有理由抱怨，他们什么也没得到，年轻的时候文化大革 命，书都没得读，支援边疆插队落户，等到老了，却要给年轻人机会了。他有足够的理由象成千上万那样的青年一样坐在那里抱怨生不逢时，怨气冲天。然而在分配 到边疆的十年之后，国家恢复招研究生，他考回了原来的学校。研究生毕业，他被分配到了安徽一家小单位里，又是3年以后，国家第一届招收博士生，他又考回了 原来的学校，成为中国第一代博士，那时的他比现在的我年纪还大。生活并没有放弃他，他也没有放弃生活。10年的等待，他做了他自己的选择，他没有放弃，他没有破罐子破摔，所以时机到来的时候，他改变了自己的人生。你最终会成为什么样的人，就决定在你的每个小小的选择之间。

你选择相信什么?你选择和谁交朋友?你选择做什么?你选择怎么做?……我们面临太多的选择，而这些选择当中，意识形态层面的选择又远比客观条件的选择 来得重要得多，比 如选择做什么产品其实并不那么重要，而选择怎么做才重要。选择用什么人并不重要，而选择怎么带这些人才重要。大多数时候选择客观条件并不要紧，大多数关于 客观条件的选择并没有对错之分，要紧的是选择怎么做。一个大学生毕业了，他要去微软也好，他要卖猪肉也好，他要创业也好，他要做游戏代练也好，只要不犯 法，不害人，都没有什么关系，要紧的是，选择了以后，怎么把事情做好。

除了这些，你还可以选择时间和环境，比如，你可以选择把这辈子最大的困难放在最有体力最有精力的时候，也可以走一步看一步，等到了40岁再说，只 是到了40多岁，那正是一辈子最脆弱的时候，上有老下有小，如果在那个时候碰上了职业危机，实在是一件很苦恼的事情。与其如此不如在20多岁30多岁的时 候吃点苦，好让自己脆弱的时候活得从容一些。你可以选择在温室里成长，也可以选择到野外磨砺，你可以选择在办公室吹冷气的工作，也可以选择40度的酷热 下，去见你的客户，只是，这一切最终会累积起来，引导你到你应得的未来。

我不敢说所有的事情你都有得选择，但是绝大部分事情你有选择，只是往往你不把这当作一种选择。认真对待每一次选择，才会有比较好的未来。

从 1.5k 到 18k, 一个程序员的 5 年成长之路

昨天收到了心仪企业的口头 offer， 回首当初什么都不会开始学编程， 到现在恰好五年. 整天在社区晃悠， 看了不少的总结， 在这个时间点， 我也写一份自己的总结吧.

我一直在社区分享，所以，这篇总结也是本着一种分享的态度，希望相比我还年轻的同学们，可以从中找到一些让自己成长更快的文字.

先介绍下背景:

1. 2008 年 3 月开始学习编程， 目前 2013 年 3 月;

2. 2009 年 6 月计算机专业本科毕业;

3. 大学期间， 基本稳拿班级倒数第一， 高考英语 49 分， 大学英语除了补考没及格过.

接着， 是一份总结:

1. 5 年间 60%以上的时间， 每天凌晨 2-4 点睡觉;

2. 为学习编程花费的总时间超过 6000 小时;

3. 手写了超过 50 万行代码;

4. 记录了超过 100 万字学习笔记;

5. 录制了两份视频教程;

6. 翻译了小型技术文档 5 份以上， 5 个 php 扩展的官方文档， jqueryui 官方文档一份， 书籍

7. 供职过 4 家公司;

8. 获得两次优秀员工;

下面是这几年的流水，本人没有过硬的文采，只是以流水的方式记录，希望可以激励到别人，仅此而已:

2008 年 03 月 -- 2008 年 10 月， 一个偶然的机会， 看到了北京尚学堂的 java 视频教程， 以此为起点， 我开始了自己的编程学习之路. 5 月份汶川地震， 我在甘肃， 学校给通宵电， 在这段时间， 我就基本很少去学校上课了， 每天晚上学习到凌晨 5-6 点， 接着睡到 10-11 点， 每天两袋 1.3 元的方便面， 其余所有的时间都用来学习. 这段时间， 我学会了基本的程序设计， 更重要的是， 视频中， 马士兵老是给我灌输了影响我后来最重要的两个观念: 1) 不要怕英文， 出错信息， 文档， 都是学习英

文的机会; 2) 不要怕出错， 出错就是学习的机会. 在后来的 5 年中， 我一直坚持， 我是从基本每个英文单词都要查翻译软件， 到现在能翻译一些东西的. 同样， 我在碰到问题的时候， 都是自己跟踪源代码去解决. 这两个观念直接决定了我今天可以进入自己心仪的企业.

2008 年 10 月 -- 2009 年 1 月， 这段时间， 宿舍搬到了校本部， 没有通宵电了. 学校一位老师找我们帮他做项目， 他为我们提供了他的宿舍， 很烂， 但我依然搬进去了， 就为了通宵电， 我住进了这个阴森森的宿舍， 恐惧缠绕着我， 但我依旧坚持. 白天帮老师做项目， 晚上继续自己的学习. 同样， 我也很感谢这位老师， 虽然他只给了我们很少的报酬， 但是， 我知道， 我得到的远远不是这些报酬可以比拟的. 这里这种很容易满足的心里也是很重要的， 我奉劝各位职场新人， 刚开始不要期望什么， 放低身段， 去做自己的积累就好了. (想起从社区里看到的一句话: 现在的你， 凭什么翘着二郎腿， 你应该放下你的腿， 身体前倾， 时刻保持战斗姿态)

2009 年 02 月， 一个小插曲， 我独自一人来到北京， 开始了第一次真正意义的独立， 我来找工作. 我开始在网上投了 10 多份简历， 没有回信. 接着我就急了， 直接从网上找招聘的公司， 查地图，直接去公司面试， 一般都还是会给面试机会的. 最好的成绩是一家表示能给到 3000 左右， 但后来也没信了. 这段时间， 我有两个收获: 1) 我打印了一整本的， 20 天的时间， 把基础的数据结构与算法读了一遍; 2) 在一个完全陌生的城市， 我独自一人， 查招聘信息， 查地图， 找公司， 厚颜无耻的霸王面， 这一切都是对我处世能力的极大提升.

2009 年 03 月 -- 2009 年 06 月， 回到学校， 仍然没有工作. 我继续在老师的小黑屋， 实现了所有我找工作时学习的数据结构与算法. 接着， 就进入了毕业季， 每天都泡在酒精中， 浑浑噩噩. 这段时间， 老婆把工作签到了山东淄博， 我也联系了一家淄博的公司， 准备毕业后去面试. 这段时间， 和高中的毕业季一样， 是值得怀念的， 放松， 惬意.

2009 年 06 月 -- 2010 年 03 月， 我来到了淄博， 找到了之前联系的公司， 面试没有通过. 淄博是一个小城市， 做软件开发的没有几家， 还好， 山东人好， 虽然没有面试通过， 但我依然可以借宿在公司宿舍找工作， 就这样， 我放下了所有的尊严， 在别人的宿舍借住， 15 天后， 我找到了我的第一家公司. 公司有 30 人左右， 做国家电网的项目， 老板人很好， 我还得到了一台笔记本电脑， 并且可以带回家. 我每天加班到 9 点， 然后回家继续学习大凌晨 2 点左右. 公司的工作相对轻松， 当然，工资也很少， 只有 1.5k. 老婆每周末都过来， 我们虽然没有钱， 经常要靠吃方便面度日， 但我们很幸福. 这段时间， 我学习了 python， javascript， 翻译了 jqueryui 的文档， 录制了一套 python 的视频教程， 录制了一份 fullcalendar 的视频教程， 我的生活非常的充实， 当然， 我还有另一个收获: 我学会了吃苦. 冬天， 我住的小屋窗户基本起不到保暖作用， 买个小电暖也起不了多少作用， 屋里水龙头都已经结冰， 我很长时间每天需要吃两包方便面， 但我依然坚持学习， 因为我知道有一天我将不再这样.

2010 年 03 月 -- 2011 年 03 月， 从后来很多次建议来看， 不得不说老婆很有眼光. 她建议我去北京找工作， 我请了 3 天假， 再一次来到了北京， 不同于上一次， 我现在有大半年经验， 我翻译过文档， 录制过视频教程， 有一定资本了. 然而， 事情并不是那么顺利， 当时期望的用友并没有通过，在用友面试完后， 我就觉得自己一无是处. 无奈， 只能退而求其次， 来到一家刚创业的游戏公司. 跟这家公司谈完薪水后， 我先给妈妈打了电话， 5.5k， 已经到了妈妈不能相信的地步. 就这样， 我正式的踏入了北京的土地. 一年中， 我更多的是为公司付出， 自己以 javascript 研发进入， 后来页面制作也的我搞， php 后来我也要做， 服务器我还要搞， 不得不说， 非常锻炼人. 还是前面的态度， 我觉得这种锻炼就是我最大的收获， 从这里开始， 我正式的转向 php 开发.到 2011 年 3 月的时候， 已经觉得公司很不行了， 又碰巧老婆怀孕， 不得不考虑结婚的事情， 因此， 我就辞职， 回家结婚.

2011 年 04 月 -- 2012 年 03 月， 婚后的生活很好， 我又找到了一份工作， 这家公司相对比较大，

2000+的规模. 不过有一些体制内的特质， 我进入的薪资是 6k， 全年能拿到 18 薪以上， 公司的福利待遇都很完善， 工作也相对轻松， 我的领导人也很好， 技术也很好， 就这样， 在这家公司， 我又开始了自己的学习之路. 我继续每天凌晨 2-3 点睡觉，完成了这几年最重要的积累: unix 环境高级编程，unix 网络编程， php 内核和扩展， shell/awk/sed 等等最基础的东西. 同样， 在大公司中， 我也学会了更多的团队协作， 同事关系方面的东西. 2012 年 03 月， 公司一位副总跳槽， 就这样， 我们小组集体来到了下一家公司.

2012 年 03 月 -- 2013 年 03 月， 新的公司， 项目还没有上线， 已有的东西问题诸多， 我们被委以重任， 6.1 日上线. 旧的团队存在诸多问题， 士气不振， 技术水平不足. 我们在 6.1 之前完成了很重要的几件事: 1) 提升团队士气; 2) 整理旧的框架不好的东西; 3) 修改大量的 bug; 4) 规范工作流程; 5) 规划未来的技术框架. 虽然后来看做的东西一般，但在这么短的时间完成这一切，我觉得实在难得. 虽然我一直自我感觉对自己更多的提升是自己业余时间的学习， 但不可否认， 在这家公司的一年中， 我在团队建设， 团队管理， 团队协作方面也有了质的变化. 在这家公司， 我的薪资得到了很大的提升，达到了 18K， 这是我从来都没有想过的事情. 所以， 我想给诸位职场新人说， 你不要一开始就想要这要那， 只要你做的够好， 终有一天， 你会发现你得到的远比失去的要多.

后面这两家公司中， 我觉得更重要的是我学会了一种处世方式， 首先学会听别人说话， 然后自己做感悟， 做提升.

流水的结束， 就是在昨天， 我收到了自己心仪公司的口头 offer， 并且也已经给现在的公司提出了离职. 和 leader 聊了一会儿， 大家都还挺开心.

在前 5 年中， 我用的网名是 selfimpr， 是 self-improvement 简写而来， 含义是: 自强不息. 5 年中，我从所有可能激励我的地方去激励自己， 让自己可以一直坚持走到今天. 这几天， 我在考虑， 用一个词总结我的前 5 年. 我想， 这个词就是"积累".

这也正是我想给这个行业的后来者说的， "积累"， 并且要是不计回报的积累， 因为你一旦太过计较回报， 你的心就很难平静， 往往就会半途而废.

此外， 还有一点要说的是， 比我基础更差的同学估计也很难找了， 所以， 能不能学有所成， 关键不在你是否有基础， 而在于你付出了多少.

接下来， 我已经更换了自己的网名 goosman， 是从 swan goose 演化而来， 我不知道"鸿鹄"用英语怎么说， 就用了这个单词， 希望我可以像书中所说， 利用自己前 5 年的积累， 一展鸿鹄之志.

职场新人“七宗罪”

在工作中，每个人都是从新人阶段开始的。我们在新人阶段难免会碰到各种各样的问题，相信大家说起自己的“青葱岁月”时，一定有着自己特殊的回忆。我从学校毕业到进入 ThoughtWorks 工作一年半里，同样也经历了自己的“青葱岁月”，从一个对技术“依然懵懂”的学生开始快速成长起来，其中的挫折与收获依然历历在目。说起新人该如何度过自己的这段特殊阶段，除了网上各种各样“职场新人指南”之类的文章之外，我也有自己独特的视角。“如何成为一个合格的新人”，答案可能很多很发散，然而我今天想从另外一个角度去看这个问题，那就是以“如何成为一个不合格的新人”为主题，说说我自己曾经犯下的，也是大多数新人最容易犯下的“七宗罪”。

1. 不会用 Google 搜索

搜索，我想大家一定不会陌生，“Google 是什么?不知道，Google 一下吧。”在我们日常工作中，几乎每时每刻都离不开搜索。但有很多人忽略了一点，那就是选择搜索引擎。我认为搜索引擎的不同会直接影响搜索的结果，所以我强烈推荐 Google，这是我工作时的唯一选择。在这里我不是说非 Google 就一定不行，主要看我们搜索什么。我个人的经验，在平时的工作中，搜索最新技术，某技术的疑难杂症时，Google 总让我满意。而百度，必应甚至谷歌香港之类的总会差一些，然而在非工作的部分他们表现的也不错的。所以我总是建议身边的人，要根据自己的搜索内容选择合适的搜索引擎。使用 Google 时一定要使用 Google 英文主站，不是谷歌香港，不知道怎么访问 Google 英文主站的同学们，那现在就开始尝试用 Google 来搜索一下这个问题吧(使用 Google 英文主站需要特殊的配置，这也是一个好程序员的必备技能之一)。

// 伯乐在线插播推荐文章：《如何使用搜索技巧来成为一名高效的程序员》

2. 不追求高效率

“要高效率，不要拖沓，我就是我，我是程序员”。想要提高自己的工作效率，不光是年年升级机器硬件这么简单，最重要的是“升级”自己的工作方式。明明有更好更快的方法，却依然“刀耕火种”式的埋头苦干，这样就有点自讨苦吃了。举个例子，在写程序时有人会使用鼠标进行某些操作，殊不知当手离开键盘操作鼠标的时候，实际上已经在浪费时间了，我们应该尽可能的将双手放在键盘上，利用快捷键完成所有可能的操作(需要工具支持)。这样既提高的效率，也能保证思维的连贯性。除了写程序之外，工作时也经常会浏览网页，那么我们能否做到在浏览网页时也键盘流呢?只要你想，不是不可能的。其实利用“键盘流”提高效率可不是我发明的，这可是著名编辑器 VI 的设计理念呢。再举个例子，我们是否还在桌面上放满了各种程序的快捷方式，又或者在“开始”目录里翻天覆地的找要启动的程序，这都是很低效的方式。我们可以在 windows 下使用 launchy，或在 mac 下使用 Spotlight，这样的话想启动 word 程序，只要敲入“word”这四个字母，word 程序就打开了(“So easy!妈妈再也不用担心我学习了~”)。上面说到的两点虽然只是“雕虫小技”，但足以说明问题。作为新人，我们应该留意自己低效的方面，多积累些别人“高明”的工作方式。这样下来，每次“加速”一点点，将来一定妥妥的。

3. 怕丢脸不敢问

一个新人在刚加入团队时，最容易产生一种怕丢脸的想法，掩盖自己一切的问题来保护自己，觉得“不能问这种白痴的问题，否则他们会笑死我的”。产生这种想法的原因很简单：环境不够安全，自信

心不够。先说说新人自己，作为新人，我们应该做的不是极力的掩盖自己的问题，而应该正确评估自己，然后设立正确的期望，利用后天的努力来弥补差距。我们本来就是一张白纸，虚心向老员工学习请教，多和其他同事沟通交流，不要不懂装懂。把问题告诉老员工，多听听大家的建议，往往能够事半功倍。再说说环境因素，如果一个团队中的老员工总是不耐烦的对付新人，或者总是以命令甚至责骂的方式和新人沟通，那新人的日子也肯定不好过了。我想，只有给新人一个安全的环境，才能真正激发新人的积极性，扔掉内心的包袱;反之只能给新人产生巨大的心理压力，导致新人更加难以融入团队。老员工在这方面可以从很多渠道下手，比如帮助新人设置计算机的开发环境，手把手的指导团队中的编程风格，鼓励新人多发问等，一步一步让新人更加活跃起来。

4. 不敢表达想法

“他们都是牛人，哪里能轮到我一个新人指手画脚”，“老大问大家有什么问题，别人都没说话，我也别说话好了，枪打出头鸟呀”。作为新人，如果我们真的抱有这种想法，那我们就可能真的很难真正融入团队了。想要融入团队中，首先要做好的就是应该勇敢的表达自己的想法。我这里有一个典型的测试：你是否有过这样的经历，参加了一个会议却从头至尾没有说一句话。我的同事说过：“如果在开会时可以不说一句话，那也许意味着根本不需要参加这个会议。”这种情况偶尔一次没要紧，可一旦出现多次，我们是否敢于表达“也许我不需要参加这种议会”的想法呢?参加会议如此，那平常工作也会如此。不要隐藏自己，收起自己的戒心，把我们的真实想法告诉大家，每一句话，每一个想法，都应该清楚的表达出来，这样才能真正做到透明，真正融入团队。不要去做默默无闻的人，要勇敢的表达自己。

5. 不参加培训

对新人来讲，除了在工作时间内学到的知识外，还要利用各种培训给自己“充充电”。我相信很多公司都会有培训，培训形式一般有两种，第一种是话题分享，基本上就像上课一样，一个人讲，一堆人听。我认为缺点就是很难深入，毕竟时间有限，不过我们倒可以开拓眼界。另一种就是实战演练，通常组织者会带领大家一起通过一个实战练习来强化某一方面的技能，通常需要很多次课程，这种培训强度大，效果好，毕竟自己能真的动手练习。如果大家真的有机会参加培训，一定不要错过它们，牺牲一些自己看电影，玩游戏的时间吧，积极参加各种培训，学会总结、利用别人的知识，即训练了自己的学习能力，也增强了同事之间的感情，何乐不为呢?

6. 不总结，不分享

学习新知识最好的验收标准是什么?不是整整齐齐的笔记，而是把学到知识自己经过整理再和别人讲一遍。在公司里新人每天都会接触到新鲜的知识，如何处理并保存知识就变得重要起来。首先要做的就是总结，因为这样能帮助自己记忆。总结的方式有很多种，写在纸上，用记录软件如 Evernote，写成博客等。其次，也就是最重要的分享。我相信很多新人都会总结，但不是每个新人都会去分享，原因很简单，“我的知识是不值一提的”这种想法占了绝大多数。其实分享就是我上面提到的“验收标准”，只有通过分享，我们才有机会把自己的理解说出来，如果有错误，那肯定会得到纠正，然后改正再分享，接着再次得到纠正。在这样的良性循环下，通过不断验证的知识才能真正融入自己的血液中。我经常鼓励新人写博客，这样即能总结自己的知识，也能分享出来进行知识的验证。你看，我的同事就已经写出《把知识发表出来》系列的博客，系统的阐述了“为什么要写博客”。那么，2013 年你准备分享什么呢?

7. 不独立思考

从小到大各种各样填鸭式的教育，已经把我们变成了最棒的学习机器，可惜的是也限制了我们独立思考的能力。刚加入公司，我们肯定每天都生活在一大堆自己没听过的名词，没接触过的技术中。在埋头苦学，吸取他人经验的同时，也要多问问自己为什么，不要道听途说，要学会养成独立思考的能力。

每次学到新东西时，除了记录下来之外，一定要有自己的思考，在脑子里问自己问题，并且试着再深入一些，如果大多数问题都不能给出让自己满意的答案，那我们就要开始培养独立思考能力了。有人说过：我听到的我会忘记，我看到的我会记住，我做过的真正明白。没有经过“思考”加工过的知识，就像过眼云烟睡一觉就忘了。培养自己的思考、钻研精神说起来容易做起来难，然而比这更难的就是让新人意识到这个问题。俗话说当局者迷旁观者清，这时我们就需要和老员工多多沟通，让他们多“批评批评”。有了大家的“鞭策”，再加上时不时自己“为难”自己，肯定错不了~

OK，看完了以上的七点，各位新人是不是也莫名其妙的“躺着中枪”了呢?呵呵，大家可千万要 hold 住呀。首先，这“七宗罪”有一定的局限性，可千万不要认为这是“葵花宝典”，充其量只是小弟我身在江湖中的一些“武功心得”而已。其次，知错能改，善莫大焉。大家能做到“有则改之无则加勉”就好，可千万不要被自己的“罪状”吓倒。诚然，我们从“学生系统”升级到“工作系统”难免有些磕磕绊绊，大家可千万不要轻易就“系统崩溃”了。认识到自己的缺点，就努力克服，这样才能更好地享受工作。养成良好的工作习惯肯定能帮助我们更平稳的度过“新人阶段”，希望身为新人的我们都能为自己打下坚实的基础，走出更美好的明天。

为什么集成测试比单元测试更重要

单元测试很棒。在假定一些数据的环境下，能顺利通过测试的系统就可算是一个好系统。

不过，现在可以直连外部资源的集成测试才让程序更有价值。谁知道那些内容商（供应商，vendor）会做出什么傻事来！

很多人一直尝试着让测试达到 100%的代码覆盖率，这是很棒的想法，但我倒觉得它有些基本概念上的问题。LosTechies， Ryan Svihla 提出了"反模式（anti-pattern"， 有个有趣的观点： “多数应用都需要与外部资源交互”。他们有许多不错的论点，比如：

大多数的单元测试都需一个运行着的数据库、网页或应用服务器。

确实如此。

全是同数据和外部系统相关的

我下面将要证明多数 Bug 并不来源于程序本身，而是由从外部输入的数据所引起的。

为什么？ 因为通常 Bug 出现在实际的工作环境中，我们的程序总会处理不好那些外部系统输入的原始数据，或者程序输出到外部系统中的数据。而"单元测试"或 TDD 中所强调的是提供一组假定的数据，检验程序是否能够按照预期的方式运行。也就说整个测试是在一个假定环境下进行的。这就是为什么接口总是如此轻易就成功运行了。在这之上还可以达到自动化，预测试性，以及可重复的测试。但是仍有很多系统无法解决现实的问题。

因为这样的测试方式所能解决的仅是软件开发要面对的问题中的一部分。如何才能发现真正的问题？最终还是要让你的软件与它的外部资源连接起来运行才能发现。

举个略为抽象的例子：

1） 一些来自外部系统的数据.

2） 应用程序开始处理这些数据.

3） 应用程序将处理后的数据发送到外部资源中 （一般是与第 1 步不同的数据）

4） 外部数据拿到第 3 步的数据，在处理后再发送到应用程序.

5） 再次接收到数据，并加以处理.

如此反复。

我们常用 mocks/stubs 或者类似的程序来产生第 1 步的数据来进行第 2 步的测试，而测试第 5 步时

所使用的数据也是使用类似的方式产生的。其中第 1 步和第 4 步是不可靠，也是不可预计的，因为你根本不知道外部系统会给你什么数据。

第 1 步和第 4 步是程序在上线环境下要面对的，所以它们才是最需要关注的。

外部系统都很挑剔（External systems are finicky mean things）

对于一些 e-Commerce 系统，或者财务系统，各式的接口，各式的数据在各个系统间流转。

我们来谈一些高层次的问题：

1）理想情况下，当外部系统更改了要发送的数据，无论是是格式（format）或模式（schema），你希望会提前知道。这有些一厢情愿了。最近我曾与一个电子商务系统，外部数据系统的税务信息增加了一栏，同时需要应用程序调整内部逻辑。就是外部数据系统已经开始发送数据了，我们才知道的。

2）理想情况下，当外部系统声称支持新的 API，你应该改变应用程序的内部逻辑，并且发送新数据。最后，你会发现，他们支持新的 API，要么在他们的 UAT（User Acceptance Test）环境而不在上线（PRODUCTION）的环境，或者只在上线环境中而不在 UAT 环境中。

3）你的程序已使用关于国内资产的采购信息很顺畅了，外部系统和程序的配合也很好。然后开始加入一些国际资产信息时，你可能并不能及时地发现数据已完全变了。

这些都是我曾遇到的场景。我要说就是你在单元测试中根据无法了解到未来面对的环境，只有实际运行时才有办法。更悲哀的是那些在凌晨 3 点把你叫起来的问题通常都这样产生的。

如何完善测试规格（How to setup your Specs）

我做设计也是从规格文档开始的。规格（specification）其实就是另一种测试（well-crafted tests）。我会区分规格中的单元和集成，并写不同的代码。

所谓“单元”的测试规格是测试内部的业务逻辑，看看有没有把东西都串起来了。就是在测试时提供一些场景，确保程序执行正确的逻辑，输出期望的结果。

而集成（Integration）的测试规格则和外部资源有关。直接提供一组外部资源数据，以及要返回的数据。这些是集成测试中实际关心的东西。和外部资源的交互才能真正确定程序是否可以正常工作。

[只译出主要概念，详细内容请阅读原文！]

原文地址：
http://www.blogcoward.com/archive/2010/07/16/More-Reasons-Why-Integration-Tests-Can-Be-More-Important-Than.aspx

本文转载自：http://blog.csdn.net/horkychen/article/details/8685473

软件测试人员的华丽转身

软件测试，是一件非常令人沮丧的事情。为什么这么讲呢？从测试的工作量而言，测试是一件非常消耗人力和时间成本的工作；从测试人员的心理而言，重复的去做同一件看似毫无技术含量的工作，没有成就感。大型软件项目的测试尤甚。测试的痛苦在于，测试的目的在于发现软件 bug，从某种角度上讲，发现的软件 bug 越多，证明了测试的有效性越强，但从另一角度而言，软件的 bug 越多，也就说明软件在设计和实现过程中的问题很多，该软件项目就算是比较失败的项目。有人可能会说，在软件项目中，开发人员应该是最重要的，如果没有开发人员的聪明才智，就不会有成功的软件。没错，但是测试人员的努力工作换来了高质量的软件。

既然测试人员工作非常辛苦，而且有了他们的工作才能保证高质量的软件。那么如何做才能够帮助他们实现华丽的转身，让测试人员有时间回家陪老婆抱孩子，而不是整天面对一堆用例，面对众多 bug 以及不停的回归迭代。

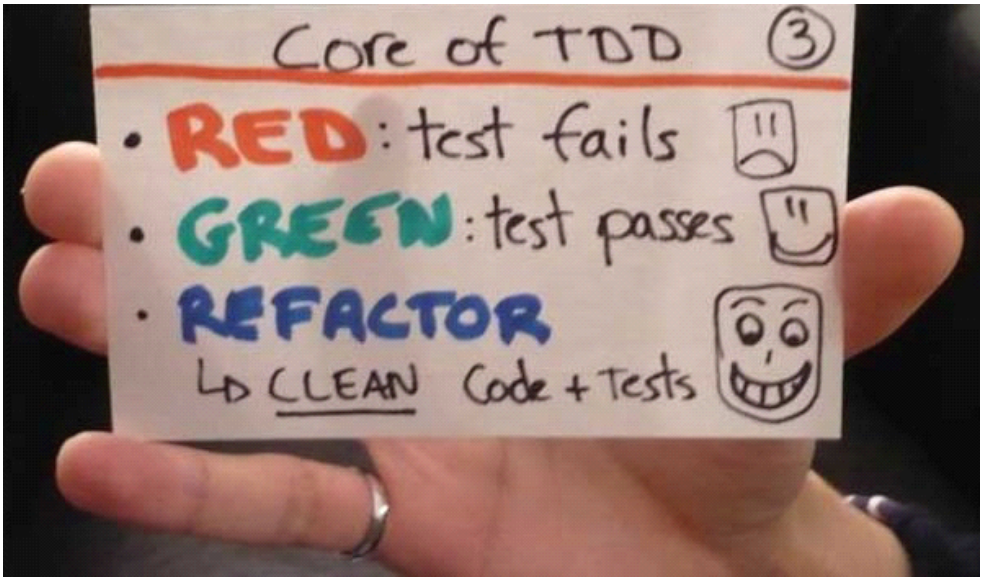
自动化测试是目前看似行之有效的方法（没有之一）。自动化测试的概念就如同它的名字一样清晰，就是通过使用自动化工具将测试人员部分的工作解放出来，使测试人员能够专注于测试设计的本身，而不去考虑测试脚本编写、测试执行等实现问题。自动化的好处当然显而易见，但是自动化的实施以及自动化实现的程度却根据不同的公司、不同的项目而异。一般说来，自动化测试对于已经运行维护的系统进行回归测试的效果较为明显，而对于新开发项目的实施成本较高且效果不够显著。这是因为对已运行的系统，其功能和特性已经较为熟悉，在迭代过程中，能够通过自动化去替代人工对原有的功能进行测试。二新开发的软件项目的功能可能并不清晰，在设计开发中的改动也比较大，如果使用自动化测试工具，需要根据被测对象的改变而不停的维护测试工具。因此代价和成本也较高。自动化测试可以从两个方面来解放测试人员的工作，提高测试效率。一方面是自动化测试管理平台的建立，另一方面是自动测试工具的建立。对于前者，可以采用开源软件 testLink 来实现，主要进行用例的管理维护工作。本文先对后一类型进行说明。目前自动化测试工具用的比较多的有 HP 公司的 QTP。QTP 通过录制/回放的功能，能够自动的进行测试。而且 QTP 不仅支持对象库，而且支持描述性编程，能够适应不停改变的项目。

自动化测试的开展首先必须要考虑到自动化要达成什么样的目标？要达到多高的覆盖率？是不是覆盖率越高的话自动化就越好？其实，自动化的覆盖率和自动化效率是一个抛物线的图，对某一项目而言，在自动化覆盖率达到一个阈值时效率最高，随着覆盖率的进一步加大，维护的成本就会加大，被测对象的一点改变都必须进行自动化测试工具相应的维护和改变。目前来看这个阈值只能是根据开发测试架构的测试开发人员的经验去判断。对于前台 UI 的自动化，有一种说法是这个目标很难实现，因为在设计中 UI 是最容易变更的，这样对自动化工具而言成本就非常大。一次有幸听过 google 中国测试部门老大的讲座，他提到没有见到一例做 UI 自动化成功的。看来 UI 自动化还是任重道远。对于后台的自动化，因为功能部分比较稳定，因此这里是实现自动化的主要方面。但是后台自动化的问题是如何与前台进行交互。有一种方法是在后台和前台之间放一层中间层，后台所有测试的数据最后都放在中间层中，而前台直接调用中间层的数据，避免了前后台直接交互，而只采用了数据交互的方式。

发散的说了这么多，还只是作为一个自动化测试菜鸟的一点学习总结和感悟，自动化测试的愿景是美好的，道路是曲折的。曲折的道路不仅包括技术问题，更重要的是成本的计算、开发测试等项目组人员自动化测试意识的培养、以及整体项目的推进情况决定的。总之，自动化测试就类似于共产主义，测试人员都在期待那天的来临，如果实在太遥远，那就先进行社会主义初级阶段，从可以推进的项目逐渐

进行，将测试人员的部分工作逐渐自动化替代，实现测试人员华丽的转身。

测试驱动开发上的五大错误



我曾经写过很多的糟糕的单元测试程序。很多。但我坚持着写，现在我已经喜欢上了些单元测试。我编写单元测试的速度越来越快，当开发完程序，我现在有更多的信心相信它们能按照设计的预期来运行。我不希望我的程序里有 bug，很多次，单元测试在很多弱智的小 bug 上挽救了我。如果我能这样并带来好处，我相信所有的人都应该写单元测试！

作为一个自由职业者，我经常有机会能看到各种不同的公司内部是如何做开发工作的，我经常吃惊于如此多的公司仍然没有使用测试驱动开发(TDD)。当我问“为什么”，回答通常是归咎于下面的一个或多个常见的错误做法，这些错误是我在实施驱动测试开发中经常遇到的。这样的错误很容易犯，我也是受害者。我曾合作过的很多公司因为这些错误做法而放弃了测试驱动开发，他们会持有这样一种观点：驱动测试开发“增加了不必要的代码维护量”，或“把时间浪费在写测试上是不值得的”。

人们会很合理的推断出这样的结论：

写了单元测试但没有起到任何作用，那还不如不写。

但根据我的经验，我可以很有信心的说：

单元测试能让我的开发更有效率，让我的代码更有保障。

带着这样的认识，下面让我们看看一些我遇到过/犯过的最常见的在测试驱动开发中的错误做法，以及我从中学到的教训。

1、不使用模拟框架

我在驱动测试开发上学到第一件事情就是应该在独立的环境中进行测试。这意味着我们需要对测试中所需要的外部依赖条件进行模拟，伪造，或者进行短路，让测试的过程不依赖外部条件。

假设我们要测试下面这个类中的 GetByID 方法：

```
public class ProductService : IProductService
{
    private readonly IProductRepository _productRepository;

    public ProductService(IProductRepository productRepository)
    {
        this._productRepository = productRepository;
    }

    public Product GetByID(string id)
    {
        Product product =    _productRepository.GetByID(id);

        if (product == null)
        {
            throw new ProductNotFoundException();
        }

        return product;
    }
}
```

为了让测试能够进行，我们需要写一个 IProductRepository 的临时模拟代码，这样 ProductService.GetByID 就能在独立的环境中运行。模拟出的 IProductRepository 临时接口应该是下面这样：

```
[TestMethod]
public void GetProductWithValidIDReturnsProduct()
{
    // Arrange
    IProductRepository productRepository = new StubProductRepository();
    ProductService productService = new ProductService(productRepository);

    // Act
    Product product = productService.GetByID("spr-product");

    // Assert
    Assert.IsNotNull(product);
}

public class StubProductRepository : IProductRepository
{
    public Product GetByID(string id)
    {
```

```
            return new Product()
            {
                ID = "spr-product"，
                Name = "Nice Product"
            };
        }

        public IEnumerable<Product> GetProducts()
        {
            throw new NotImplementedException();
        }
    }
```

现在让我们用一个无效的产品 ID 来测试这个方法的报错效果。

```
    [TestMethod]
    public void GetProductWithInValidIDThrowsException()
    {
        // Arrange
        IProductRepository productRepository = new StubNullProductRepository();
        ProductService productService = new ProductService(productRepository);

        // Act & Assert
        Assert.Throws<ProductNotFoundException>(() => productService.GetByID("invalid-id"));
    }

    public class StubNullProductRepository : IProductRepository
    {
        public Product GetByID(string id)
        {
            return null;
        }

        public IEnumerable<Product> GetProducts()
        {
            throw new NotImplementedException();
        }
    }
```

在这个例子中，我们为每个测试都做了一个独立的 Repository。但我们也可在一个 Repository 上添加额外的逻辑，例如：

```
    public class StubProductRepository : IProductRepository
    {
        public Product GetByID(string id)
        {
            if (id == "spr-product")
```

```
            {
                return new Product()
                {
                    ID = "spr-product"，
                    Name = "Nice Product"
                };
            }

            return null;
        }

        public IEnumerable<Product> GetProducts()
        {
            throw new NotImplementedException();
        }
    }
```

在第一种方法里，我们写了两个不同的 IProductRepository 模拟方法，而在第二种方法里，我们的逻辑变得有些复杂。如果我们在这些逻辑中犯了错，那我们的测试就没法得到正确的结果，这又为我们的调试增加了额外的负担，我们需要找到是业务代码出来错还是测试代码不正确。

你也许还会质疑这些模拟代码中的这个没有任何用处的 GetProducts()方法，它是干什么的？因为 IProductRepository 接口里有这个方法，我们不得不加入这个方法以让程序能编译通过——尽管在我们的测试中这个方法根本不是我们考虑到对象。

使用这样的测试方法，我们不得不写出大量的临时模拟类，这无疑会让我们在维护时愈加头痛。这种时候，使用一个模拟框架，比如 JustMock，将会节省我们大量的工作。

让我们重新看一下之前的这个测试例子，这次我们将使用一个模拟框架：

```
[TestMethod]
public void GetProductWithValidIDReturnsProduct()
{
    // Arrange
    IProductRepository productRepository = Mock.Create<IProductRepository>();
    Mock.Arrange(() => productRepository.GetByID("spr-product")).Returns(new Product());
    ProductService productService = new ProductService(productRepository);

    // Act
    Product product = productService.GetByID("spr-product");

    // Assert
    Assert.IsNotNull(product);
}

[TestMethod]
public void GetProductWithInValidIDThrowsException()
```

```
{
    // Arrange
    IProductRepository productRepository = Mock.Create<IProductRepository>();
    ProductService productService = new ProductService(productRepository);

    // Act & Assert
    Assert.Throws<ProductNotFoundException>(() => productService.GetByID("invalid-id"));
}
```

有没有注意到我们写的代码的减少量？在这个例子中代码量减少 49%，更准确的说，使用模拟框架测试时代码是 28 行，而没有使用时是 57 行。我们还看到了整个测试方法变得可读性更强了！

2、测试代码组织的太松散

模拟框架让我们在模拟测试中的生成某个依赖类的工作变得非常简单，但有时候太轻易实现也容易产生坏处。为了说明这个观点，请观察下面两个单元测试，看看那一个容易理解。这两个测试程序是测试一个相同的功能：

Test #1

```
TestMethod]
public void InitializeWithValidProductIDReturnsView()
{
    // Arrange
    IProductView productView = Mock.Create<IProductView>();
    Mock.Arrange(() => productView.ProductID).Returns("spr-product");

    IProductService productService = Mock.Create<IProductService>();
    Mock.Arrange(() => productService.GetByID("spr-product")).Returns(new
Product()).OccursOnce();

    INavigationService navigationService = Mock.Create<INavigationService>();
    Mock.Arrange(() => navigationService.GoTo("/not-found"));

    IBasketService basketService = Mock.Create<IBasketService>();
    Mock.Arrange(() => basketService.ProductExists("spr-product")).Returns(true);

    var productPresenter = new ProductPresenter(
                                                productView，
                                                navigationService，
                                                productService，
                                                basketService);

    // Act
    productPresenter.Initialize();

    // Assert
```

```
    Assert.IsNotNull(productView.Product);
    Assert.IsTrue(productView.IsInBasket);
}
```

Test #2

```
[TestMethod]
public void InitializeWithValidProductIDReturnsView()
{
    // Arrange
    var view = Mock.Create<IProductView>();
    Mock.Arrange(() => view.ProductID).Returns("spr-product");

    var mock = new MockProductPresenter(view);

    // Act
    mock.Presenter.Initialize();

    // Assert
    Assert.IsNotNull(mock.Presenter.View.Product);
    Assert.IsTrue(mock.Presenter.View.IsInBasket);
}
```

我相信 Test #2 是更容易理解的，不是吗？而 Test #1 的可读性不那么强的原因就是有太多的创建测试的代码。在 Test #2 中，我把复杂的构建测试的逻辑提取到了 ProductPresenter 类里，从而使测试代码可读性更强。

为了把这个概念说的更清楚，让我们来看看测试中引用的方法：

```
public void Initialize()
{
    string productID = View.ProductID;
    Product product = _productService.GetByID(productID);

    if (product != null)
    {
        View.Product = product;
        View.IsInBasket = _basketService.ProductExists(productID);
    }
    else
    {
       NavigationService.GoTo("/not-found");
    }
}
```

这个方法依赖于 View， ProductService， BasketService and NavigationService 等类，这些类都要模拟或临时构造出来。当遇到这样有太多的依赖关系时，这种需要写出准备代码的副作用就会显现出来，

正如上面的例子。

请注意，这还只是个很保守的例子。更多的我看到的是一个类里有模拟一、二十个依赖的情况。

下面就是我在测试中提取出来的模拟 ProductPresenter 的 MockProductPresenter 类：

```
public class MockProductPresenter
{
    public IBasketService BasketService { get; set; }
    public IProductService ProductService { get; set; }
    public ProductPresenter Presenter { get; private set; }

    public MockProductPresenter(IProductView view)
    {
        var productService = Mock.Create<IProductService>();
        var navigationService = Mock.Create<INavigationService>();
        var basketService = Mock.Create<IBasketService>();

        // Setup for private methods
        Mock.Arrange(() => productService.GetByID("spr-product")).Returns(new Product());
        Mock.Arrange(() => basketService.ProductExists("spr-product")).Returns(true);
        Mock.Arrange(() => navigationService.GoTo("/not-found")).OccursOnce();

        Presenter = new ProductPresenter(
                                        view，
                                            navigationService，
                                            productService，
                                            basketService);
    }
}
```

因为 View.ProductID 的属性值决定着这个方法的逻辑走向，我们向 MockProductPresenter 类的构造器里传入了一个模拟的 View 实例。这种做法保证了当产品 ID 改变时自动判断需要模拟的依赖。

我们也可以用这种方法处理测试过程中的细节动作，就像我们在第二个单元测试里的 Initialize 方法里处理 product==null 的情况：

```
[TestMethod]
public void InitializeWithInvalidProductIDRedirectsToNotFound()
{
    // Arrange
    var view = Mock.Create<IProductView>();
    Mock.Arrange(() => view.ProductID).Returns("invalid-product");

    var mock = new MockProductPresenter(view);

    // Act
```

```
        mock.Presenter.Initialize();

        // Assert
        Mock.Assert(mock.Presenter.NavigationService);
    }
```

这隐藏了一些 ProductPresenter 实现上的细节处理，测试方法的可读性是第一重要的。

3、一次测试太多的项目

看看下面的单元测试，请在不使用“和”这个词的情况下描述它：

```
[TestMethod]
public void ProductPriceTests()
{
    // Arrange
    var product = new Product()
    {
        BasePrice = 10m
    };

    // Act
    decimal basePrice = product.CalculatePrice(CalculationRules.None);
    decimal discountPrice = product.CalculatePrice(CalculationRules.Discounted);
    decimal standardPrice = product.CalculatePrice(CalculationRules.Standard);

    // Assert
    Assert.AreEqual(10m，  basePrice);
    Assert.AreEqual(11m，  discountPrice);
    Assert.AreEqual(12m，  standardPrice);
}
```

我只能这样描述这个方法：

“测试中计算基价，打折价和标准价是都能否返回正确的值。”

这是一个简单的方法来判断你是否一次测试了过多的内容。上面这个测试会有三种情况导致它失败。如果测试失败，我们需要去找到那个/哪些出了错。

理想情况下，每一个方法都应该有它自己的测试，例如：

```
[TestMethod]
public void CalculateDiscountedPriceReturnsAmountOf11()
{
    // Arrange
    var product = new Product()
    {
```

```
            BasePrice = 10m
        };

        // Act
        decimal discountPrice = product.CalculatePrice(CalculationRules.Discounted);

        // Assert
        Assert.AreEqual(11m， discountPrice);
    }

    [TestMethod]
    public void CalculateStandardPriceReturnsAmountOf12()
    {
        // Arrange
        var product = new Product()
        {
            BasePrice = 10m
        };

        // Act
        decimal standardPrice = product.CalculatePrice(CalculationRules.Standard);

        // Assert
        Assert.AreEqual(12m， standardPrice);
    }

    [TestMethod]
    public void NoDiscountRuleReturnsBasePrice()
    {
        // Arrange
        var product = new Product()
        {
            BasePrice = 10m
        };

        // Act
        decimal basePrice = product.CalculatePrice(CalculationRules.None);

        // Assert
        Assert.AreEqual(10m， basePrice);
    }
```

注意这些非常具有描述性的测试名称。如果一个项目里有 500 个测试，其中一个失败了，你能根据名称就能知道哪个测试应该为此承担责任。

这样我们可能会有更多的方法，但换来的好处是清晰。我在《代码大全(第 2 版)》里看到了这句经验之谈：

为方法里的每个 IF，And，Or，Case，For，While 等条件写出独立的测试方法。

驱动测试开发纯粹主义者可能会说每个测试里只应该有一个断言。我想这个原则有时候可以灵活处理，就像下面测试一个对象的属性值时：

```
public Product Map(ProductDto productDto)
{
    var product = new Product()
    {
        ID = productDto.ID，
        Name = productDto.ProductName，
        BasePrice = productDto.Price
    };

    return product;
}
```

我不认为为每个属性写一个独立的测试方法进行断言是有必要的。下面是我如何写这个测试方法的：

```
[TestMethod]
public void ProductMapperMapsToExpectedProperties()
{
    // Arrange
    var mapper = new ProductMapper();
    var productDto = new ProductDto()
    {
        ID = "sp-001"，
        Price = 10m，
        ProductName = "Super Product"
    };

    // Act
    Product product = mapper.Map(productDto);

    // Assert
    Assert.AreEqual(10m， product.BasePrice);
    Assert.AreEqual("sp-001"， product.ID);
    Assert.AreEqual("Super Product"， product.Name);
}
```

4、先写程序后写测试

我坚持认为，驱动测试开发的意义远高于测试本身。正确的实施驱动测试开发能巨大的提高开发效率，这是一种良性循环。我看到很多开发人员在开发完某个功能后才去写测试方法，把这当成一种在提交代码前需要完成的行政命令来执行。事实上，补写测试代码只是驱动测试开发的一个内容。

如果不是按照先写测试后写被测试程序的红，绿，重构方法原则，测试编写很可能会变成一种体力劳动。

如果想培养你的单元测试习惯，你可以看一些关于 TDD 的材料，比如 The String Calculator Code Kata。

5、测试的过细

请检查下面的这个方法：

```
public Product GetByID(string id)
{
    return _productRepository.GetByID(id);
}
```

这个方法真的需要测试吗？不，我也认为不需要。

驱动测试纯粹主义者可能会坚持认为所有的代码都应该被测试覆盖，而且有这样的自动化工具能扫描并报告程序的某部分内容没有被测试覆盖，然而，我们要当心，不要落入这种给自己制造工作量的陷阱。

很多我交谈过的反对驱动测试开发的人都会引用这点来作为不写任何测试代码的主要理由。我对他们的回复是：只测试你需要测试的代码。我的观点是，构造器，geter，setter 等方法没必要特意的测试。让我们来加深记忆一下我前面提到的经验论：

为方法里的每个 IF，And，Or，Case，For，While 等条件写出独立的测试方法。

如果一个方法里没有任何一个上面提到的条件语句，那它真的需要测试吗？

祝测试愉快！

自动化测试开发人员的十八般武器

在软件开发中，后台开发人员需要掌握的是后台开发技术，前台开发人员需要掌握的是前台技术。理所应当，测试开发人员也就应该有自己的十八般兵器。在下不才，列出个一二三条，供大家茶余饭后消遣。

第一，至少掌握一种自动化测试框架。无论是开源的自动化测试框架，还是针对具体系统的测试框架，都是进行自动化测试的利器。自动化测试框架通常可以对一类被测对象具有通用性，在框架的基础上可以进行自动化测试的设计、开发。

第二，一种后台编程语言和后台技术。无论是 apache 下使用 cgi 或者是 php。如果你不想使你的自动化测试仅仅停留在前台界面的 UI 测试上（事实上开展 UI 测试是一项非常费事和费时的事），那么掌握后台开发技术是十分必要的。有了后台开发技术，可以对被测对象的后台功能进行验证。

第三，一种前台脚本技术，vbs，js 等。虽说前台 UI 自动化测试费力不讨好，但是还是十分有必要进行的。UI 测试的难点就在于测试用例维护量很大，因为后台功能可能变化不大，但是前台 UI 经常变动，UI 一变化，原有的测试用例就要进行维护。

第四，一种自动化测试工具。无论是商业的，非商业开源的，还是自己开发测试工具。有了自动化工具就像无翼天使长了翅膀，可以通过工具驱动各种脚本执行，或者使用录制/回放等很容易实现自动化测试。

第五，理解测试的能力。前面说了那么多，无非是从设计开发自动化测试的角度来讲的，无论怎么开发自动化测试，其根本都在于替代手工测试，所以切记在自动化测试中不能单一的追求测试覆盖率，不能为了实现自动化而进行自动化。

第六，测试管理工具。能够将对测试进行管理，比如测试用例的管理，测试计划的管理等。

第七，指导、协调测试的能力。这点就不单纯属于测试开发的要求了，对于一名资深测试开发而言，能够明确的分析开发测试的成本，在自动化测试和手工测试之间找到平衡点，并能够指导测试人员进行自动化测试方面的实践。

泽众软件工具使用技术支持

电话：021-61079698

Email：sales@spasvo.com

QQ：1404189128

MSN：spasvo_support@hotmail.com

产品租用		
下载	在线申请	详细

AutoRunner 是一款自动化测试工具。AutoRunner 可以用来执行重复的手工测试。主要用于：功能测试、回归测试的自动化。它采用数据驱动和参数化的理念，通过录制用户对被测系统的操作，生成自动化脚本，然后让计算机执行自动化脚本，达到提高测试效率，降低人工测试成本。

在线体验		产品租用	
企业版	免费版	在线申请	详情

TestCenter 是一款功能强大的测试管理工具，它实现了：测试需求管理、测试用例管理、测试业务组件管理、测试计划管理、测试执行、测试结果日志察看、测试结果分析、缺陷管理，并且支持测试需求和测试用例之间的关联关系，可以通过测试需求索引测试用例。

其他测试工具

Precise Project Management

Terminal AutoRunner

PerformanceRunner

有关培训、产品购买及试用授权方法等事宜

电话：021-61079698

Email： sales@spasvo.com

QQ：1404189128

MSN： jennyding0829@hotmail.com

SPASVO