
目录

(五十一期·下)

摆脱接口工具束缚.....	01
使用测试工具解决产品问题.....	05
一篇文章带你搞定 Java 数组.....	11
TestNG 数据驱动的袖里乾坤.....	23
转变你的移动端测试方法.....	34
聚焦回归测试.....	37
如何更有效地进行自我管理？	41

如果您也想分享您的测试历经和学习心得，欢迎加入我们~(*^▽^*)

 投稿邮箱：editor@51testing.com

摆脱接口工具束缚

◆ 作者：邵君兰

我司是从 2013 年开始做接口自动化，那时候选用了测试接口工具--SoapUI。当时觉得非常好用，简单易上手，即使是代码小白，也能通过几次培训课，上手该工具。非常适用于代码基础薄弱，仅仅懂得业务的测试团队。

随着业务的发展，一个项目的接口从几十个扩大到几百个，继续使用接口工具，无疑对后期脚本的维护产生了非常大的工作量。虽然说提供界面化管理，让做接口变的简单，但是却因为无法做接口分离，导致代码耦合性太高，大家都不愿意去维护老的接口。出现了老接口的 case 因为调试不通，而被无情删除的情况出现。

事实上，只要接口仍然被软件调用的情况下，无论是老接口还是新接口。我们都是不能随意删除的。不然所谓的可持续集成，也就没有意义了。对于我来说，既然接口做了自动化，就必须是 100%全覆盖。这样我才对服务器的正确性，胸有成竹。发布前以及发布后，动动手指轻点一键跑一下接口自动化，短短几分钟内就能知道发布是否宣成功。

那么接口工具到底有哪些弊端，想让我摆脱它的束缚呢？接下来就来盘点下它的弱点。

用过 SoapUI 的伙伴，可能遇到过这样的情况，比如 case 一多，文件一大，就出现 out of memory 的情况。如果接口改了，那就要到非常多的 case 里面去找这个接口，如果多个 case 调用到的话，我们还要一一做调整和修改。还有在想复用断言代码的时候，为了隔离环境，想在容器里面跑的时候.....工具的一些弊端就体现出来了。

假如说测试团队的小伙伴有一些代码基础了，强烈建议用 python 去写接口自动化，当然用其他语言也 ok。Python 相对学起来更简单，上手更容易，代码量也少。很适合想要将接口工具切换到代码来管理和维护自动化的团队。



当然如果想通过代码方式去实现接口自动化，就要很好的去做代码层级的规划。如果揉在一起的代码，无疑也是不可维护的，与其这样还不如用接口测试工具来管理。所以考虑接口分层必不可少。

如何做到分层?就要从接口的构成开始分。接口组成要素 1) url 2) 请求体 3) 响应。所以我们可以创建 3 个文件。首先是创建管理接口 url 的文件，然后是创建管理请求参数的文件，最后就是创建管理响应的文件。

举个例子:

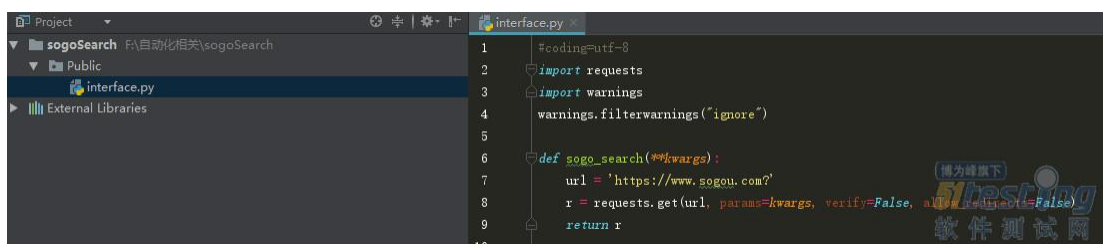
```
#coding=utf-8
import requests
import warnings
warnings.filterwarnings("ignore")

def test_sogo_search():
    url = 'https://www.sogou.com?'
    para = {'query': 'tesla'}
    r = requests.get(url, params=para, verify=False, allow_redirects=False)
    assert r.status_code == 200
```

这段代码是我们把整个 url，请求参数，断言都是揉在一起的一个用例。接下来我们进行拆分。

拆分一:

通过一个 interface 的文件，将 url 以及接口的请求 method 来进行封装，并返回 response。

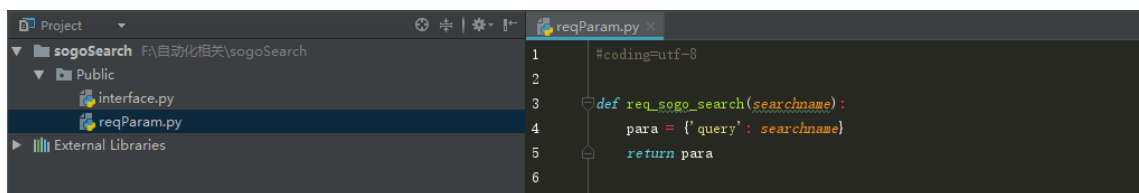


```
1 #coding=utf-8
2 import requests
3 import warnings
4 warnings.filterwarnings("ignore")
5
6 def gogo_search(**kwargs):
7     url = 'https://www.sogou.com?'
8     r = requests.get(url, params=kwargs, verify=False, allow_redirects=False)
9     return r
10
```

拆分二:

通过一个 reqParam 的文件来管理请求的参数





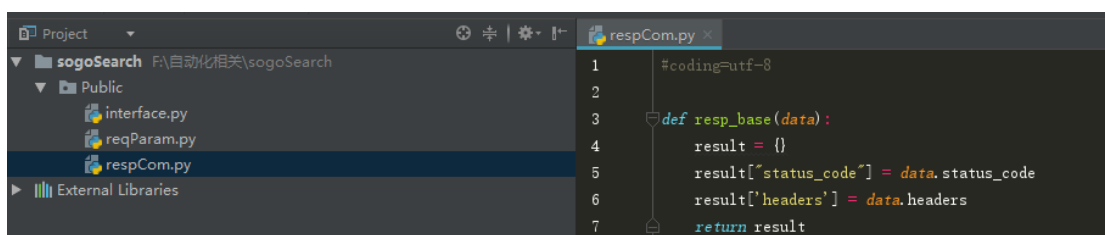
```

1 #coding=utf-8
2
3 def req_sogo_search(searchname):
4     para = {'query': searchname}
5     return para
6

```

拆分三:

通过一个 respCom 的文件来管理返回的响应

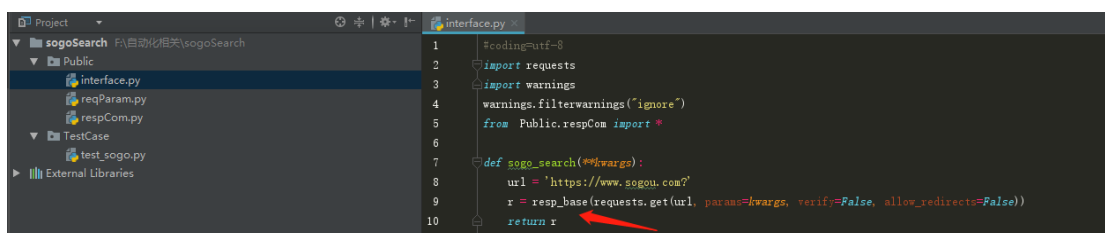


```

1 #coding=utf-8
2
3 def resp_base(data):
4     result = {}
5     result['status_code'] = data.status_code
6     result['headers'] = data.headers
7     return result
8

```

这个时候我们其实可以在 interface 这个文件中, 进行一些改造, 将 return 的响应结果通过 resp_base 再包一层, 变成如下



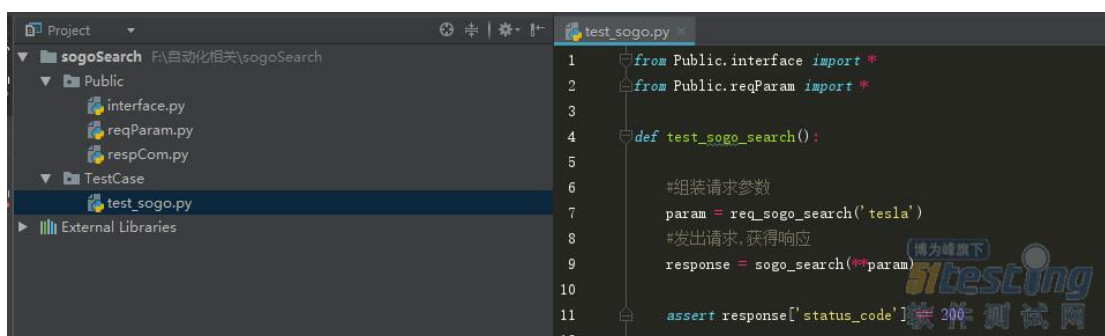
```

1 #coding=utf-8
2 import requests
3 import warnings
4 warnings.filterwarnings('ignore')
5 from Public.respCom import *
6
7 def sogo_search(**kwargs):
8     url = 'https://www.sogou.com?'
9     r = resp_base(requests.get(url, params=kwargs, verify=False, allow_redirects=False))
10    return r
11

```

写 case:

这下有了接口必要的部件后, 我们就可以开始写 case, 将这些部件整合, case 看起来更加简洁。



```

1 from Public.interface import *
2 from Public.reqParam import *
3
4 def test_sogo_search():
5
6     #组装请求参数
7     param = req_sogo_search('tesla')
8     #发出请求, 获得响应
9     response = sogo_search(**param)
10
11    assert response['status_code'] == 200
12

```

从代码量和文件数量上来说, 由于这样的拆分貌似是多了很多, 但是对于后期维护来说确实是非常的清晰, 比如说接口改了一个 url 的地址时, 大家就不再需要去看 case, 只需要修改 interface 里面该接口的 url 即可。请求参数如果是一个非常复杂的 dict 时, 通过这种写法, case 仍然只有了了几句。如果不拆, 那么 case 里面将是一大段一大



段的 dict，无论是修改和阅读都是非常费劲的。

当然上面的拆分还不是最细的，我们还可以将断言封装，将请求 method 封装，将 url 主域名放到一个 config 文件里面进行读取以及将传入的参数单独写文件等等。

尝试一下，你会体会到自己写代码所带来的自由自在，不再受接口工具的束缚。

❖ 拓展学习

■ 直播 | 学透接口自动化，精通企业级项目：<http://www.atstudy.com/course/937>



使用测试工具解决产品问题

◆ 作者：枫叶

标准的网站监控工具能接通网页并证明他们正在响应，而他们不会向你警告一个问题。但是你能使用压力测试技术去监测你的网站，通过跑一个交互脚本能检测出问题并生成必要的邮件。它像一位安静的哨兵持续运行，从来不睡觉或者休一个假，提升了你的网站可靠性。

我们的网站包含了一个用户登录，用户认证过程偶尔被停止。我们标准的网站监控工具能接通主页并验证网页在响应，但是可能与不在我们适当放置的工具外的网页交互。当客户向我们警告一个真实的问题时我们只能了解它。这是不可接受的，我们不得不找到一个更好的办法。

我们之前使用一个压力测试工具开发并执行一系列的测试，允许我们运行很大数量的用户在测试网站上做很多不同的动作。但是我们需要一种方法在重复的基础上去运行一个简单的用户去做简单的脚本，24/7，在它在我们的产品系统上影响我们真实客户前警告我们一个问题。我们的压力测试工具会作为一个单一用户做这种测试，但是当一个问题被检测出来时，它缺少一种生成警告的方法。

与我们的供应商一起工作，我们发现他们提供了一个简单的解决方案：作为一个单一用户以重复的流程并有某些出错时发出警告的方式使用一个不同的应用去执行压力测试脚本。我们现在有这种适当的流程有3年了，并且它有一个极好的解决方案。这儿是我们如何操作它。

设计测试

第一步是做一些业务分析去决定什么被测试和失败看起来像什么。当与压力测试类似，这次测试专注于不仅是网页的压力时间以及脚本运行的结果。你也需要以重复的基础上使用已知的用户名/密码组合登录产品系统的能力。



这次测试的目的是为了简单地验证网站是活跃的并且准备使用。我们的测试不包含事务（销售订单），但是你能包含这个操作；它只要求更多的工作。



我们的检查有这些：

- 每一页需要在少于 5000 毫秒内加载（5 秒）
- 每一页需要正确地加载
- 每一页需要通过文本检查（验证页面加载预期的内容）

编写脚本

现在你已有设计，你可以创建你的脚本。

首先，我们选择一个合法的用户账号能用于这个流程。（它需要存在于产品中但是被看做一个测试账号。）

我们也能增加一些我们局域网站的监控，被授权用户专门使用。这个授权通过使用安全套接层控制，所以对那些网站，我们不得不增加一些特殊的代码去支持安全套接和端口映射。

我们使用我们普通的压力测试脚本设计工具创建脚本，衡量任意被监控工具需要的规则被包含在这个设计里。如果你被支持，你的支持供应商可能有一些有用的在这领域的信息。

一旦你有被设计和正确运行的脚本，现在你能推动它到监控的应用程序里。

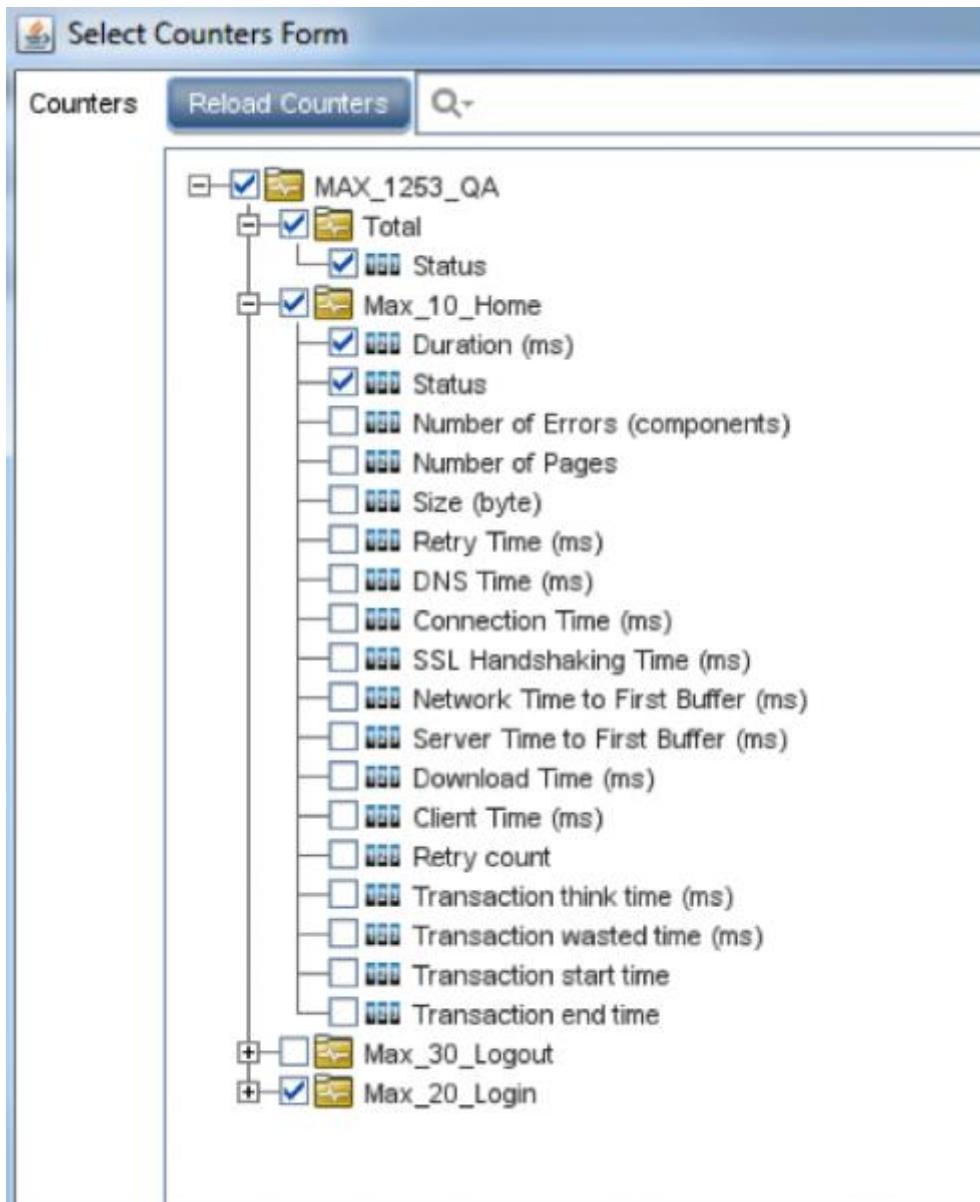
创建监测器

跟你的供应商核实什么脚本元素需要被覆盖，因为在某些情况下你需要所有的实时文件而不仅仅是脚本。好消息是这些文件是小的。

当创建你的监测器，首先决定什么网页元素被追踪。取决于你的解决方案，每一个你选择监测的元素能使用你的协议容量的部分，所以你可能选择限制监测元素的数量。在我们的案例里，我们只需要监测页面加载结果和它需要多长时间去加载，但是你可能



同时选择其他元素，就像下面任意一些。



当你保存监测器，你能看到最初的在监测仪表盘上创建的结果。

Counters (8 out of 8)			
counters in error	✓		0
MAX_1253/Max_10_Home/Duration (ms)	?		90
MAX_1253/Max_10_Home/Status	?		0
MAX_1253/Max_20_Login/Duration (ms)	?		266
MAX_1253/Max_20_Login/Status	?		0
MAX_1253/Max_30_Logout/Duration (ms)	?		30
MAX_1253/Max_30_Logout/Status	?		0
MAX_1253/Total/Status	?		0

创建监测原则

一旦你创建了监测器，你能为什么样的系统用来决定通过或失败的条件去创建规则。我们只选择监测页面状态和加载持续时间。



Threshold Settings

If unavailable:

Set monitor status according to Thresholds

Default status:

Good

On internal error:

Set monitor status according to Thresholds

Add Default Thresholds

Remove Default Thresholds

Error if

Condition	Operator	Value	Schedule
countersInError	>	0	every day, all day
MAX_1253/Max_10_Home/Status	>	0	every day, all day
MAX_1253/Max_20_Login/Status	>	0	every day, all day
MAX_1253/Max_30_Logout/Status	>	0	every day, all day
MAX_1253/Max_10_Home/Duration (ms)	>=	5000	every day, all day
MAX_1253/Max_20_Login/Duration (ms)	>=	5000	every day, all day
MAX_1253/Max_30_Logout/Duration (ms)	>=	5000	every day, all day
MAX_1253/Total/Status	>	0	every day, all day

当你有通过/失败条件的标准集时，仪表盘反应了状态。

Name	Status	Summary
Selected node		
Maximo		W ... MAX_1253/Total/Status=0,
Counters (8 out of 8)		
counters in error		0
MAX_1253/Max_10_Home/Duration (ms)		95
MAX_1253/Max_10_Home/Status		0
MAX_1253/Max_20_Login/Duration (ms)		390
MAX_1253/Max_20_Login/Status		0
MAX_1253/Max_30_Logout/Duration (ms)		24
MAX_1253/Max_30_Logout/Status		0
MAX_1253/Total/Status		0

Status: Good

你也需要决定你多久需要测试执行。当第一次测试仍然运行时假如第二次测试尝试开始，你能得到错误的警告，所以我们在每次测试间允许 3 分钟。

创建警报规则

现在你创建测试并定义什么是一次失败，你需要创建可能被检测失败时告诉谁的规则。

我们的工具允许多于 10 次不同的警报活动，但是我们为我们的警报流程选择邮件。我们使用明显的邮件主题去反应网站问题。我们能发送一封邮件或者短信息，假如单元载体支持邮件地址——比如，5055551212@vtext.com。

其中一件事是区分技术是否易犯错误，偶尔事件不发生在你的网站上。为了减少错误警报数，除非测试在一行里失败了 3 次，否则我们不生成警报。我们也建立了规则，



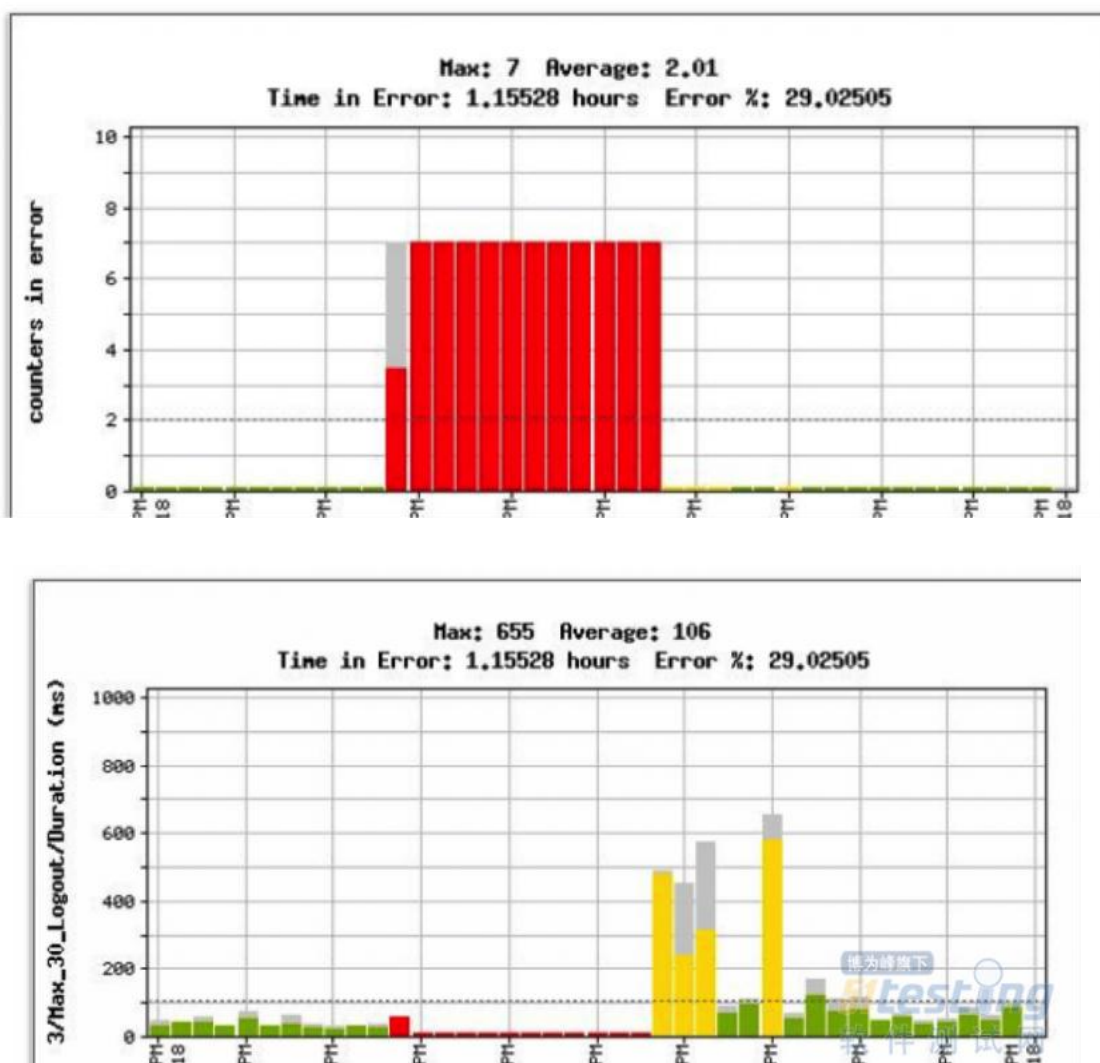
每一小时只生成一封重复的邮件（每 20 次），当物价管制局正处理一个事件时，他们不需要一堆邮件告诉他们已经寻址的事件。

我们也能参与并使警报根据需要不可用，以防止计划的停机时间省城错误的邮件警报。

生成报告

这些监测工具提供了一个好的检查性能超时的方法，并生成有用的报告。这些能被剪切复制或者导出为 HTML。（我发现它更易于被剪切和复制相关的信息而不是尝试解释报告生成的所有数据。）

这些是我们质量实例来的示例报告，关于奔溃和需要及时改变的登录流程：



维护可靠性

我们的工具也包括一个仪表盘视图，提供简单的地方可以快速检查所有的监测器并



看到是否有一个或多个问题，那可能一个更大的问题。

最初，这个工具有些强卖给美国物价管制局团队，因为使用虚拟用户测试的想法是对某些人来讲是一个新的概念。但是现在它执行得就像一个安静的哨兵，从不睡觉或者休个假，我们实际上有一个新的应用程序到来，询问我们检测他们的网站确保用户可靠性是能维护的。

这改进了我们的操作可靠性——并且这难道不是质量保证的角色吗？



一篇文章带你搞定 Java 数组

◆作者：千里

Java 数组

Java 数组在学习 Java 过程中属于比较重要的一个章节也是比较难的一个章节，作业带领大家讲解 Java 数组的一些相关操作。

数组的概念

数组就是一组数据，我们定义变量会比较多，如果使用传统的方式，会出现的问题是变量很多，代码也很多，而且使用起来不方便。

```
int t0 = 0 ;  
int t1 = 1 ;  
int t2 = 2 ;  
int t3 = 3 ;  
int t4 = 4 ;  
int t5 = 5 ;  
int t6 = 6 ;  
int t7 = 7 ;  
int t8 = 8 ;  
int t9 = 9 ;
```

以上代码是通过传统方式定义了 10 个整数变量，如果使用数组，在定义上可以更为简单。

```
int t[] = {0,1,2,3,4,5,6,7,8,9}
```

从变量的操作上，变量的操作一般有：声明、赋值、修改值、输出、变量的处理

数组的声明

传统的变量声明：

```
int t0,t1,t2,t3,t4,t5,t6,t7,t8,t9 ;
```

数组的声明：

```
int t[] = new int[10];    //是通过 new int[10]来生成了 10 个“变量”
```



这个数组生成的变量为：

t[0]、t[1]、t[2]、t[3]、t[4]、t[5]、t[6]、t[7]、t[8]、t[9]

数组的赋值

传统的变量赋值需要分个赋值，数组的赋值也可以分个赋值，也可以一次性进行赋值，但是如果是一次性赋值需要在声明的时候同时赋值。

以下为分个赋值：

//变量的赋值

t0 = 0 ;

//数组的赋值

t[0] = 0 ;

数组可以实现一次性全部赋值

int t[] = {0,1,2,3,4,5,6,7,8,9} ; //它只能够在声明的时候进行全部赋值

我现在要将以上数组 t[] 中的 10 个元素全部值修改为 0

我们可以发现数组有一个规律，就是下标是连续的，如果我使用 for 语句就可以通过循环方式进行有规律的赋值了，这个是普通变量所做不到的地方

```
for(int i=0 ;i<9 ;i++){  
    t[i] = 0 ;  
}
```

数组赋值小练习

声明一个包含 10 个元素的整数数组

使用随机整数对这 10 个元素进行赋值

输出数组的元素

我们不能使用 System.out.println(array) ;，假设 array 是数组。这个输出结果有值，但不是我们想要的。

输出数组元素的方法有两种：

方法一：



//array.length 是获取数组的长度，数组的最后一个元素的下标是数组的长度-1

```
for(int i=0; i<array.length;i++){
    System.out.println(array[i]);
}
```

方法二:

我们可以使用 foreach 增加型 for 循环来实现将数组的元素进行输出，我们经常会处理数组，所以 for 循环专门给我们设计一个输出数组元素的方法。

```
for(int num:array){
    System.out.print(num+"\t");
}
```

在 for 循环中，我们要输出的值为 num，所以我们定义了一个 num 变量来依次接收数组的每一个元素。而这个数组元素数据类型是 int 型的，所以我们就使用 int 来作为 num 的数据类型。后面跟的是冒号，数组名称。

数组的初始值

我们可以声明一个数组，但是不赋初始值，这个数组的每一个元素是存在初始值的。

```
int int_array[] = new int[5];           //初始值为: 0
float f_array[] = new float[5];         //初始值为: 0.0
char c_array[] = new char[5];           //初始值为: ASCII 的第一个值\u0000 空
boolean b_array[] = new boolean[5];     //初始值为: false
String s_array[] = new String[];        //初始值为: null
```

数组的操作

我们可以对数组的元素进行操作，也可以对整个数组进行操作。操作主要是对数组的下标和数组元素及值进行操作，也是通过我们之前所学习的 if、for 循环进行操作的。

1.我们可以将所有元素的值相加

```
//我们以前做过 1-100，有 100 个数依次相加
//这回是我们有下标 0-99 的 100 个元素依次相加

int sum = 0;
```



//以前是数字 0 累加到数字 100

```
for(int i=0;i<=100;i++){
```

```
    sum = sum+i ;
```

```
}
```

//现在是元素 0 累加到元素 100

```
for(int i=0;i<100;i++){
```

```
    sum = sum + array[i] ;
```

```
}
```

//求元素的平均值

```
int avg = sum / array.length ;
```

2.通过下标获取元素，通过元素可以做四则运算，比较

//获取元素中的最大值，最小值

//将一个临时变量，对每一个元素进行比较，如果元素的值更大，则将元素的值赋值给到临时变量就可以得到这个数组的最大值。相反就可以得到最小值

```
int max=array[0],min=array[0] ;
```

```
for(int i=0;i<array.length;i++){
```

```
    if(max<array[i]){
```

```
        max = array[i] ;
```

```
    }
```

```
    if(min>array[i]){
```

```
        min = array[i] ;
```

```
    }
```

```
}
```

3.无序数组的排序（冒泡排序）

//定义一个数组，包含若干个元素。为了效果的明显，元素的值为：

```
int array[] = {9,8,7,6,5,4,3,2,1} ;
```

```
int temp ;
```

```
for(int circle=0;circle<array.length;circle++){
```

```
    for(int count=0;count<array.length-circle-1;count++){
```



```

        if(array[count]>array[count+1]){

            temp = array[count] ;

            array[count] = array[count+1] ;

            array[count+1] = temp ;

        }

    }

}

```

4.二分查找法又叫折半查找法

在一个有序数组中，找到一个我要元素的，看它在哪个位置。

```

int array[] = { 1,2,3,4,5,6,7,8,9} ;

int find = 8 ;

for(int i=0;i<array.length;i++){

    if(find==array[i]){

        System.out.println("这个元素的下标为: "+i);

    }

}

```

二分查找法解析:

先要定义开始位置和结束位置，默认的开始位置就是 array[0]，结束位置是 array[array.length-1]

再将要查找的内容与正中间那个元素进行比较 array[(start+end)/2]

如果要查找的数小于正中间那个元素，就将 end=置为正中间。反之，就将开始位置调到正中间。

一直到查找到我要的数据为止（循环）

```

int array[] = { 1,2,3,4,5,6,7,8,9} ;

int find = 8 ;

int start = 0,end = array.length-1 ;

int middle ;    //作为中间数

while(true){

```



```

middle = (start+end)/2 ;

if(find==array[middle]){

    System.out.println("要查找的数据下标为: "+middle);

    break ;

}else if(find > array[middle]){

    start = middle ;

}else if(find < array[middle]){

    end = middle ;

}

}

```

Java 数组函数

Java 提供了一些关于数组的函数，像 Java 的排序和 Java 数组中查找数据，只需要使用一个函数就能够实现了，已经可以不知道写这样的代码了。

在 Java api 中提供了 Arrays 类可以直接操作 Java 数组。例如要使用二分查找法来查找一个数所在数组的位置：

```

import Java.util.Arrays;

public class Search1 {

    public static void main(String[] args) {

        int array[] = new int[500] ;

        int find = 400 ;

        for(int i=0;i<array.length;i++){

            array[i] = i+5 ;

        }

        int result = Arrays.binarySearch(array,find) ; //核心代码

        System.out.println(result);

    }

}

```

数组中也提供了，对数组进行排序的功能，也可以直接使用 Arrays 类的排序方法进行排序。其操作为：Arrays.sort(数组名)，提供了对数组进行排序，但没有提供反向排



序，不过可以通过程序对数组实现反序排序。其代码为：

```
int array[] = {1,2,3,4,5,6,7,8,9};

for(int i=0;i<(0+array.length)/2;i++){

    int temp = array[i];

    array[i] = array[array.length-1-i];

    array[array.length-1-i] = temp;

}
```

关于 main(String args[])

我们知道在 Java main 方法中有一个 String args[]也是一个数组，那么它是怎么使用的呢？我们先看一段程序：

```
public class Search1 {

    public static void main(String args[]) {

        System.out.println(args[0]);

    }

}
```

它的执行结果为：

Exception in thread "main" Java.lang.ArrayIndexOutOfBoundsException: 0

出现数组越界异常，我们如果让它不越界，那么就需要给这个数组赋值让其用起来。

二维数组

在 Java 中，二维数组用得并不是很多。在 C 语言会涉及到二维数组，因为在 C 语言中没有字符串的概念，所以用到字符串必须是字符数组，然后要使用到字符串数组，这时候在 C 语言中就成了二维数组。其实二维数组就是数组中包含数组，相当于一栋大楼包含若干个单元，每个单元包含若干个房间，每个房间若干个柜子，每个柜子包含若干个抽屉。

二维数组的表示：

```
char array[][] = {{ 'a','b'},{ 'c'},{ 'd','e','f' }};
```

就是一个大的数组 array[]中包含了三个小的数组，其中 array[0]元素又是一个数



组，包含两个数据；array[1]元素也是一个数组，包含 1 个数据；array[3]也是一个数组包含 3 个数据。如果要表示'a'这个数据，则是：array[0][0]来表示，表示第 0 个数组中的第 0 号元素。如果要表示'e'这个元素，则是：array[2][1]表示第 2 个数组中的第 1 号元素。

二维数据也可以使用 for 循环来操作它，操作的也是下标。也可以使用增强型 foreach 循环来查看元素。

```
char array[][] = {{ 'a','b'},{ 'c'},{ 'd','e','f' }};
```

```
//外层数组
```

```
for(char out[]:array){
```

```
    //内层数组
```

```
    for(char in:out[]){
```

```
        System.out.println(in);
```

```
    }
```

```
}
```

以下是使用双重 for 循环来输出二维数组的所有元素

```
public class ArrayTest {
```

```
    public static void main(String args[]) {
```

```
        char array[][] = {{ 'a','b'},{ 'c'},{ 'd','e','f' }};
```

```
        //外层数组，得到 3 个数组，分别为： array[0],array[1],array[2]
```

```
        for(int i=0;i<array.length;i++){
```

```
            //内层数组，是针对于每一个里面的数组而言的
```

```
            //这个 for 循环跟一维数组的操作是一样的
```

```
            for(int in=0;in<array[i].length;in++){
```

```
                System.out.print(array[i][in]+" ");
```

```
            }
```

```
            System.out.println();
```

```
        }
```

```
    }
```

```
}
```



Java api 文档的使用

首先要确定我们的操作对象是属于哪个类的，这个类提供了三大内容：属性、构造方法、方法。属性是本身所拥有的内容，可以直接使用。例如：数组提供了一个叫做 `length` 的属性，构造方法是用来构造一个对象的，例如：`Date d = new Date()`，这个 `Date()` 就是一个构造方法，通过 `new` 关键字来完成构造，把构造的结果保存到对象 `d` 中，这个对象 `d` 是 `Date` 类型的。

方法的使用：直接用对象名.方法名即可，例如：`d1.getTime()`，在这里 `getTime()` 就是方法名，在 `api` 文档中，会告诉我们使用这个方法会返回一个什么类型，我们就可以使用对应类型的变量来存储它的操作结果，如果是 `void` 则不需要使用变量来存储它的操作结果。在 `api` 结果列中，有时候还会出现 `static` 关键字，说明这个方法不需要构造对象，使用类名称.方法名称可以直接使用。例如：`Arrays.binarySearch(array, 400)`

附：交换两个变量的值，一般我们都会通过设置临时变量 `temp` 来存储要交换的值，其实也可以不用临时变量，可以通过其他方式来实现。

//有两个变量 `a` 和 `b`，分别设置了不同的值。

//通过以下三行代码，可以实现两个值的交换

```
int a=10,b=20;
```

```
a = a+b;
```

```
b = a-b;
```

```
a = a-b;
```

String 类的使用

`String` 不是一种基本数据类型，在 C 语言是没有 `String` 类型的，所以在 Java 中 `String` 的首字母是大写的，基本数据类型的首字母是小写的。

正是因为 `String` 数据是大写的，所以它是一种类，就具有对应的对象，使用其构造方法和方法。

String 类可以通过构造方法生成对象：

`String str = new String("hello world");` //这种写法是可行的，这种方式会生成一个新的字符串

当然，也可以通过 `String str = "hello world";` 来生成一个字符串对象。



问题:

上述提供了两种方法, 到底哪种比较好。

```
String str1 = "hello world" ;
String str2 = "hello world" ;
String str3 = new String("hello world") ;
String str4 = new String("hello world") ;

//以下是用来判断是否为同一个数据, 变量是在内存中, 是否为同一块内存数据
System.out.println(str1==str2) ;           //
System.out.println(str3==str4) ;           //
System.out.println(str2==str3) ;           //
```

答案是: true、false、false, 这说明 str1 和 str2 是同一块内存数据, 也就是说定义 str1 的时候因为没有"hello world"在内存中存在, 所以会开辟一块空间存储"hello world"。但是在定义 str2 的时候, 因为系统已经有了"hello world", 所以系统不再开辟空间, 让其直接指向 str1 所在内存空间, 节约了内存。而 str3 因为使用了 new 强行开辟一块内存空间, 所以这时候系统必须开辟一块内存空间, 造成了两个都包含"hello world"的数据内容。str4 也是因为使用 new 关键字, 强行开辟了一块空间。

我们还发现一个问题, 虽然 str1、str2 和 str3 以及 str4 的值都是"hello world", 但是它们并不相等(==)。所以我们用等于去判断两个值是不是一样, 是不可行的。那么 String 提供了一个方法 equals()来判断值是否相等。

String 类的常见方法

将数组转化为字符串 new String(char[] arr), 注意这是一个构造方法, 所以代码实现为:

```
char array[] = {'1','2','3','4','5','6','7','8','9'} ;
String charArray = new String(array) ; //构造方法的使用
System.out.println(charArray);
```

将字符串转化为数组, 对应的方法为: toCharArray() , 其得到的结果为: char[]数组

例如: 有一个字符串, Hello world 123, 统计大写, 小写, 数字各有多少个



```
String str = "Hello world 123" ;
```

//将 str 打散为普通的字符数组，因为只有打散成为了普通数组才有办法对每一个字符分别进行比较

```
char temp[] = str.toCharArray() ;
```

//得到了数组之后，就可以通过 for 循环从第 0 个元素到最后一个元素进行遍历

//遍历元素的时候，进行 if 判断，进行大写判断，小写判断，数字判断

查看某个位置的字符，charAt(int index)

例如：通过身份证查看性别，我们知道是身份证的第 17 位，如果是双数则为女，单数为男

```
String idCard = "430121198801022345" ;
```

```
char s = idCard.charAt(16) ;
```

```
if(s%2==0){
```

```
    System.out.println("该身份证为女");
```

```
}else{
```

```
    System.out.println("该身份证为男");
```

```
}
```

比较两个字符串的值是否相等：equals()

boolean result = str2.equals(str3) ; //equals 是 str2 的方法，所以带点。equals()括号包含要比较的内容

```
boolean result = str2.equals("hello world") ;
```

转字符串转化为 byte[] 数组，以后在 Java 文件读写中用到

```
byte something[] = str1.getBytes() ;
```

```
byte anything[] = "hello world".getBytes() ;
```

字符串第一次出现某字符的位置，结果为索引地址

//因为 str 字符串中没有'a'，所以 index 的结果为-1

```
String str = "hello world" ;
```

```
int index = str.indexOf('a') ;
```

```
System.out.println(index);
```

求字符串的长度，是 length()方法，所以有括号。需要与数组区分开来。



得到字符串的一部分 `substring(int start,int end)`，例如求身份证的出生日期

忽略字符串的前后空格 `trim()`，这个在开发中用得非常多。一般在输入框中获取用户输入内容，都需要进行 `trim()` 处理

```
String newstr = oldstr.trim();
```

附：在测试的时候，不用分别使用前面有空格，后面有空格，前后都有空格的数据进行测试。

其他基本数据类型转为字符串，使用 `valueOf(数据)`，因为是 `static` 方法，所以使用 `valueOf()` 方法的时候，是直接使用 `String.valueOf(数据)` 来实现

```
String str = String.valueOf(3.14); //这时候 3.14 就成了一个字符串了
```

然而有时候，我们遇到了是字符串型的数字，要转化为普通数字，又该如何处理？这时候要使用到基本数据类型所包装的类方法来解决。

```
String num = "3.14";
```

```
double pi = Double.parseDouble(num);
```

//如果要 `String` 对象转为整数，则用：

```
int a = Integer.parseInt(num);
```

前面讲到了基本数据类型的包装类，通过包装类我们可以获取他们对应的最大值，最小值

```
System.out.println(Integer.MAX_VALUE);
```

```
System.out.println(Byte.MAX_VALUE);
```

```
System.out.println(Byte.SIZE);
```



TestNG 数据驱动的袖里乾坤

◆ 作者：王 练

摘要

TestNG 是当前非常流行的自动化测试框架，提供了非常强大的功能，其中数据驱动是应用非常广泛的特性，没有之一。通过 `DataProvider` 注解和 `Test` 注解的配合，可以轻松的实现数据驱动的框架。

本文从简单的数据驱动实现说起，介绍通过封装基类的方式统一数据驱动流程。这些是很常用，也很容易理解的 TestNG 用法。之后本文提出一个需求：如何在 `BeforeMethod`、`AfterMethod` 完成测试数据的准备和清理，也就是手工测试中的预置条件和环境恢复。通过该需求的实现，引出了 TestNG 在层次设计和数据驱动的联动关系，揭秘了在数据驱动上的依赖注入玄机，进而给出了一种更符合逻辑、更合理的数据驱动实现方式。

TestNG 的数据驱动通过 `DataProvider` 注解实现，在 TestNG 官网以及各种书籍、资料中很容易找到实现方式。下面是最简单的实现方法。

通过 `@DataProvider` 注解定义数据驱动来源，代码如下

```
@DataProvider(name="Test-Data")
public Object[][] getTestData(){
    TestDataProducer e=new TestDataProducer("data/DataProviderDemo1.xls");
    return e.getExcelData();
}
```

`@DataProvider` 注解需要的是一个返回 `Object[][]` 二维数组的方法，示例中通过读取 excel 返回行列数据，属于通用处理 excel 的方法。同时需要指定该数据源的名称。

其中的测试数据文件 `DataProviderDemo1.xls` 内容如下：

name	passwd	
name1	passwd1	
name2	passwd2	
name3	passwd3	



在@Test 注解的测试方法中使用 dataProvider 进行修饰，修饰的 value 为 1) 步中定义的数据源名称

```
@Test(dataProvider="Test-Data")
public void testAddUser(HashMap<String, String> data) {
    System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
}
```

修饰 dataProvider 数据源名称要与 1) 定义的一致。这里模拟的测试场景是待测系统的添加用户功能。

编写 TestNG 配置文件如下。

```
<suite name="demo" parallel="tests" thread-count="1">
  <test name="demo" preserve-order="true" verbose="1">
    <classes>
      <class name="com.demo.DataProviderDemo1" />
    </classes>
  </test>
</suite>
```

执行结果如下

```
测试如下测试数据(name-passwd):name1-passwd1
测试如下测试数据(name-passwd):name2-passwd2
测试如下测试数据(name-passwd):name3-passwd3

=====
demo
Total tests run: 3, Failures: 0, Skips: 0
=====
```

统一数据驱动方式

当测试用例数量增加的时候，我们一方面希望通过统一的操作对 TestNG 的流程进行管理，比如在 WebUI 自动化中，希望在 BeforeSuite 和 AfterSuite 中对 WebDriver 进行统一的实例化和销毁；另一方面，数据驱动的方式都是一致的，需要封装减少代码量，比如上述@DataProvider 注解的数据源获取函数，就是一致的从 excel 获取数据。

统一流程的方式就是使用测试基类进行封装。代码如下。

```
public abstract class DemoBaseTester{
    @BeforeSuite
    public void onBeforeSuite(){System.out.println("#-onBeforeSuite");}
    @AfterSuite
    public void onAfterSuite() {System.out.println("#-onAfterSuite");}
    @BeforeTest
    public void onBeforeTest() {System.out.println("#-onBeforeTest");}
    @AfterTest()
    public void onAfterTest() { System.out.println("#-onAfterTest");}
    @DataProvider(name="Test-Data")
    public Object[][] getTestData(){
        TestDataGenerator testDataGenerator = new TestDataGenerator(this.getClass().getResource("").toString(),this.getClass().getName());
        TestDataProducer e=new TestDataProducer(testDataGenerator.getTestDataFilePath());
        return e.getExcelData();
    }
}
```



BeforeSuite、AfterSuite、BeforeTest、AfterTest 在基类中统一处理，子类继承后无需关心细节。当然对统一的 BeforeClass 和 AfterClass 也可以进行封装。数据源的获取方式一致，通过测试类所在路径和名称获取 excel 的全路径，提供数据驱动。

以两个测试类进行说明，代码分别如下

```
public class DataProviderDemo1 extends DemoBaseTester{
    @Test(dataProvider="Test-Data")
    public void testAddUser(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @BeforeMethod
    public void onBeforeMethod() {
        System.out.println(this.getClass().getName()+"::onBeforeMethod");
    }
    @AfterMethod
    public void onAfterMethod(ITestResult result) {
        System.out.println(this.getClass().getName()+"::onAfterMethod");
    }
}

public class DataProviderDemo2 extends DemoBaseTester{
    @Test(dataProvider="Test-Data")
    public void testModifyUser(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(old_desc-new_desc):"+data.get("old_desc")+"-"+data.get("new_desc"));
    }
    @BeforeMethod
    public void onBeforeMethod() {
        System.out.println(this.getClass().getName()+"::onBeforeMethod");
    }
    @AfterMethod
    public void onAfterMethod(ITestResult result) {
        System.out.println(this.getClass().getName()+"::onAfterMethod");
    }
}
```

两个测试分别对应待测系统的添加用户和修改用户测试用例。测试数据文件名分别为 DataProviderDemo1.xls 和 DataProviderDemo2.xls，数据分别如下。

name	passwd
name1	passwd1
name2	passwd2
name3	passwd3

old_desc	new_desc
old-desc1	new-desc1
old-desc2	new-desc2
old-desc3	new-desc3

TestNG 配置文件修改如下

```
<suite name="demo" parallel="tests" thread-count="1">
    <test name="demo" preserve-order="true" verbose="1">
        <classes>
            <class name="com.demo.DataProviderDemo1" />
            <class name="com.demo.DataProviderDemo2" />
        </classes>
    </test>
</suite>
```

修饰执行结果如下。




```

统一-onBeforeSuite
统一-onBeforeTest
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name1-passwd1
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name2-passwd2
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name3-passwd3
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc1-new-desc1
com.demo.DataProviderDemo2自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc2-new-desc2
com.demo.DataProviderDemo2自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc3-new-desc3
com.demo.DataProviderDemo2自己的onAfterMethod
统一-onAfterTest
统一-onAfterSuite

=====
demo
Total tests run: 6, Failures: 0, Skips: 0
=====

```

Suite 和 Test 通过统一的流程处理，每个测试方法有自己的 Method 处理。数据驱动统一给出，测试类无需关心。

新的需求：数据准备

自动化测试实际是手工测试的脚本化，手工测试的所有过程在自动化测试中都不会省略，手工测试遇到的问题，自动化测试也会遇到，也需要解决。我们在手工测试的时候，每个测试用例，都会有一个预置条件和环境清理的过程，自动化测试也存在。所以现在提出一个新的需求：如何在测试方法调用之前和之后进行数据准备和数据清理。

TestNG 通过 Before 和 After 的方式完成，本节针对 BeforeMethod 和 AfterMethod 作说明。如果想进行数据准备和清理，需要将测试数据传递到 BeforeMethod 和 AfterMethod 方法中。直观的考虑，通过 dataProvider 修饰 BeforeMethod 和 AfterMethod 注解，尝试如下。

```

@BeforeMethod(dataProvider="Test-Data")
public void onBeforeMethod() {
    System.out.println("Before Method");
}

@AfterMethod
public void onAfterMethod() {
    System.out.println("After Method");
}

```

The attribute dataProvider is undefined for the annotation type BeforeMethod

9 quick fixes available:

- Change to 'alwaysRun'
- Change to 'dependsOnGroups'
- Change to 'dependsOnMethods'
- Change to 'description'
- Change to 'enabled'
- Change to 'firstTimeOnly'
- Change to 'groups'



通过 **dataProvider** 属性修饰 **BeforeMethod** 注解没有定义，即 **TestNG** 不支持这种做法。有三种替代方案：

通过 **@Parameters** 传递数据给 **BeforeMethod** 方法。语法肯定可以编译通过，但是 **Parameter** 需要在 **TestNG** 配置文件中编写，维护不方便，而且不能太过丰富。所以不是完美的解决方案。

在测试方法前后调用自定义的 **BeforeMethod** 和 **AfterMethod** 方法。实际与 **TestNG** 没有关系，是测试方法的调用顺序决定。一旦测试方法中出现失败，**AfterMethod** 是无法被执行的。

在 **BeforeMethod** 中写固定的数据，进行数据准备；在测试方法中将测试数据赋值给测试类的成员变量，在 **AfterMethod** 中使用成员变量进行数据清理。固定数据不够灵活，实际没有与测试方法的测试数据关联；测试方法对成员变量赋值实际也是依赖于测试成功的，一旦失败，赋值不能保证正确，所以数据清理也会出现问题。

在 **TestNG** 官网中，找到如下说明。

5.19.1 - Native dependency injection

TestNG lets you declare additional parameters in your methods. When this happens, TestNG will automatically fill these parameters with the right value. Dependency injection can be used in the following places:

- Any **@Before** method or **@Test** method can declare a parameter of type **ITestContext**.
- Any **@AfterMethod** method can declare a parameter of type **ITestResult**, which will reflect the result of the test method that was just run.
- Any **@Before** and **@After** methods (except **@BeforeSuite** and **@AfterSuite**) can declare a parameter of type **XmlTest**, which contain the current **<test>** tag.
- Any **@BeforeMethod** (and **@AfterMethod**) can declare a parameter of type **java.lang.reflect.Method**. This parameter will receive the test method that will be called once this **@BeforeMethod** finishes (or after the method as run for **@AfterMethod**).
- Any **@BeforeMethod** can declare a parameter of type **Object[]**. This parameter will receive the list of parameters that are about to be fed to the upcoming test method, which could be either injected by TestNG, such as **java.lang.reflect.Method** or come from a **@DataProvider**.
- Any **@DataProvider** can declare a parameter of type **ITestContext** or **java.lang.reflect.Method**. The latter parameter will receive the test method that is about to be invoked.

选中内容意思为：任何 **@BeforeMethod** 方法都可以定义一个 **Object[]** 参数。该参数传递的就是即将执行的测试方法的参数列表，这个参数或者是 **TestNG** 为我们注入的（就像 **Method** 一样），或者是来自 **@DataProvider** 的。

这段实际是 **TestNG** 依赖注入特性提供的功能，实际验证表明通过这个参数能够获取测试数据，同时 **AfterMethod** 也支持该参数。修改测试类 **DataProviderDemo1** 如下。

```
public class DataProviderDemo1 extends DemoBaseTester{
    @Test(dataProvider="Test-Data")
    public void testAddUser(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @BeforeMethod
    public void onBeforeMethod(Object[] data) {
        System.out.println("进行如下测试数据准备:"+data[0].toString());
    }
    @AfterMethod
    public void onAfterMethod(ITestResult result, Object[] data) {
        System.out.println("进行如下测试数据清理:"+data[0].toString());
    }
}
```



修改 TestNG 配置文件，只运行该测试类，结果如下。

```

统一-onBeforeSuite
统一-onBeforeTest
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name1-passwd1
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name2-passwd2
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo1自己的onBeforeMethod
测试如下测试数据(name-passwd):name3-passwd3
com.demo.DataProviderDemo1自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc1-new-desc1
com.demo.DataProviderDemo2自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc2-new-desc2
com.demo.DataProviderDemo2自己的onAfterMethod
com.demo.DataProviderDemo2自己的onBeforeMethod
测试如下测试数据(old_desc-new_desc):old-desc3-new-desc3
com.demo.DataProviderDemo2自己的onAfterMethod
统一-onAfterTest
统一-onAfterSuite

```

```

=====
demo
Total tests run: 6, Failures: 0, Skips: 0
=====

```

如果希望通过与测试方法一样的 `data.get("key")` 的方式获取测试数据，可以编写一个 `String--->HashMap` 的转化方法，就可以与测试方法一样的进行 `data.get("key")` 获取测试数据了。实际上，`AfterMethod` 的另一个参数 `ITestResult result` 也可以获取测试数据，代码如下：

```

HashMap<String, String> data =
string2Map((result.getParameters()[0].toString()));

```

其中 `getParameters` 是 `ITestResult` 提供的 API，`string2Map` 与测试就是将特定格式（“{key1=value1, key2=value2}”）的 `String` 转化为 `HashMap` 的方法。

袖里乾坤揭秘：层次的设计

上述方案实现了数据准备和数据清理的需求，在 TestNG 官网的说明中可以看到，所有的 `Before`、`After` 都能注入特定的参数，具体如下。



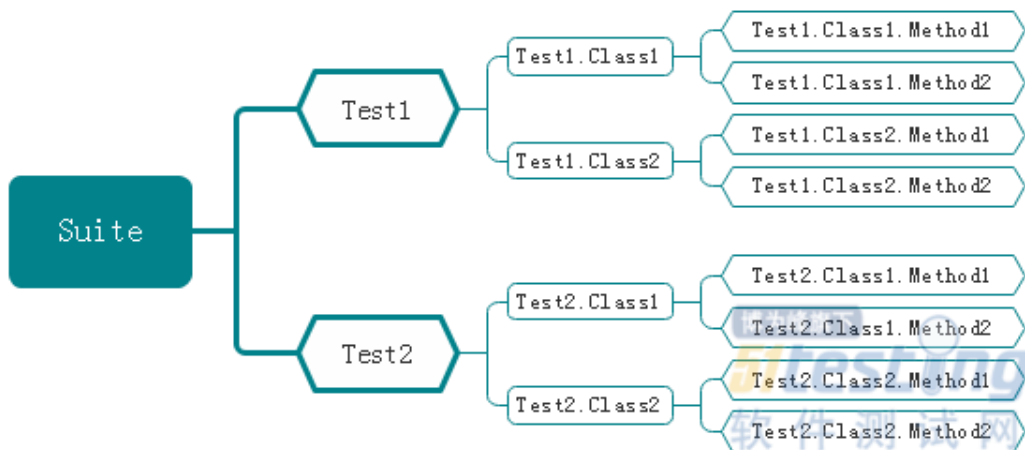
Annotation	ITestContext	XmlTest	Method	Object[]	ITestResult
BeforeSuite	Yes	No	No	No	No
BeforeTest	Yes	Yes	No	No	No
BeforeGroups	Yes	Yes	No	No	No
BeforeClass	Yes	Yes	No	No	No
BeforeMethod	Yes	Yes	Yes	Yes	Yes
Test	Yes	No	No	No	No
DataProvider	Yes	No	Yes	No	No
AfterMethod	Yes	Yes	Yes	Yes	Yes
AfterClass	Yes	Yes	No	No	No
AfterGroups	Yes	Yes	No	No	No
AfterTest	Yes	Yes	No	No	No
AfterSuite	Yes	No	No	No	No

这是 TestNG 依赖注入的特性提供给使用者的功能。考虑在 Suite、Test 和 Class 层次上能否进行测试数据的获取和处理呢？经过尝试，发现如下结果：

Suite、Test 和 Class 层次上无法获取 DataProvider 提供的测试数据；

Suite、Test 和 Class 层次都能够通过 ITestContext 获取 Parameter 提供的测试数据，具体方法为：context.getSuite().getXmlSuite().getParameters()，当然通过 XMLTest 也可以获取。

Suite、Test 和 Class 层次上为什么不能获取 DataProvider 提供的测试数据呢？要揭秘这个玄机，需要先看一下 TestNG 的层次设计，一个典型的 TestNG 测试如下图所示。



由于测试方法与 Class 不是一一对应的，所以通过 DataProvider 提供的测试数据是无法提供给测试类的。Suite、Test 自然就更不能获取了。所以说在 Suite、Test 和 Class



层次上获取 DataProvider 的测试数据本身就是一个伪命题。

如果将 TestNG 的层次设计，与手工测试的测试设计进行类比的话，一个可行的方案是：Suite 对应测试场景，Test 对应功能模块，Class 对应子模块，Method 对应测试用例。测试设计中的预置条件、环境清理，都是针对测试用例设计的，所以测试数据是与测试用例一一对应的。相似的，在 TestNG 管理下的测试层次来说，DataProvider 提供的测试数据是与 Method 一一对应的。此时，DataProvider 提供的测试数据不能，也不应被 Class 获取，更不应被 Test、Suite 获取，这是 TestNG 做不到的。

在上面说明的代码示例中，可以做如下对比。Suite 对应某个版本的测试场景，Test 对应系统功能模块，DataProviderDemo1、DataProviderDemo2 分别对应用户添加和用户修改子模块，testAddUser、testModifyUser 分别对应一个添加用户用例和修改用户用例。添加用户用例 testAddUser 需要全权负责该用例的预置条件和环境清理，而测试类 DataProviderDemo1 是不需要控制测试数据的。

改善的统一数据驱动方式

通过上述分析再考虑之前的统一数据驱动方式，是不是需要调整呢？将 DataProviderDemo1 修改为有两个测试方法的测试类，具体代码如下。

```
public class DataProviderDemo1 extends DemoBaseTester{
    @Test(dataProvider="Test-Data")
    public void testAddUserAdmin(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @Test(dataProvider="Test-Data")
    public void testAddUserGuest(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @BeforeMethod
    public void onBeforeMethod(Method method, Object[] data) {
        System.out.println(method.getName()+"，进行如下测试数据准备:"+data[0].toString());
    }
    @AfterMethod
    public void onAfterMethod(Method method, ITestResult result, Object[] data) {
        System.out.println(method.getName()+"，进行如下测试数据清理:"+data[0].toString());
    }
}
```

此时的场景是待测系统的 Admin 用户和 Guest 用户，预置条件和清理环境方式一致，但测试流程有差别。设计测试数据如下。

name	passwd	
admin1	passwd1	
admin2	passwd2	
guest1	passwd3	
guest2	passwd4	



用之前的数据驱动方式，执行用例结果如下。

```
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd1, name=admin1}
测试如下测试数据(name-passwd): admin1-passwd1
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd1, name=admin1}
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd2, name=admin2}
测试如下测试数据(name-passwd): admin2-passwd2
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd2, name=admin2}
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd3, name=guest1}
测试如下测试数据(name-passwd): guest1-passwd3
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd3, name=guest1}
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd4, name=guest2}
测试如下测试数据(name-passwd): guest2-passwd4
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd4, name=guest2}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd1, name=admin1}
测试如下测试数据(name-passwd): admin1-passwd1
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd1, name=admin1}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd2, name=admin2}
测试如下测试数据(name-passwd): admin2-passwd2
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd2, name=admin2}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd3, name=guest1}
测试如下测试数据(name-passwd): guest1-passwd3
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd3, name=guest1}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd4, name=guest2}
测试如下测试数据(name-passwd): guest2-passwd4
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd4, name=guest2}

=====
demo
Total tests run: 8, Failures: 0, Skips: 0
=====
```

考虑选中的测试，对于与 testAddUserAdmin 无关的测试数据 {passwd=passwd3, name=guest1} 也进行了测试，从测试角度，多测试自然可以接受，但从逻辑上分析，就不合理了。一方面，与该测试无关的测试数据，是不应该被执行的，手工测试的时候也不会执行与该测试无关的测试数据；另一方面，会导致最终的通过率计算出现很大误差，并且增加了测试时间。所以这种数据驱动的方式是需要改进的。

改进的目的是每个 Method 拥有自己的 DataProvider。思路还是利用 TestNG 的依赖注入，将 Method 注入到测试逻辑中。修改后的测试类 DataProviderDemo1 如下。




```
public class DataProviderDemo1 extends DemoBaseTester{
    @DataProvider(name="testAddUserAdmin-Data")
    public Object[][] getTestDataAdmin(Method method){
        return super.getTestData(method);
    }
    @Test(dataProvider="testAddUserAdmin-Data")
    public void testAddUserAdmin(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @DataProvider(name="testAddUserGuest-Data")
    public Object[][] getTestDataGuest(Method method){
        return super.getTestData(method);
    }
    @Test(dataProvider="testAddUserGuest-Data")
    public void testAddUserGuest(HashMap<String, String> data) {
        System.out.println("测试如下测试数据(name-passwd):"+data.get("name")+"-"+data.get("passwd"));
    }
    @BeforeMethod
    public void onBeforeMethod(Method method, Object[] data) {
        System.out.println(method.getName()+" , 进行如下测试数据准备:"+data[0].toString());
    }
    @AfterMethod
    public void onAfterMethod(Method method, ITestResult result, Object[] data) {
        System.out.println(method.getName()+" , 进行如下测试数据清理:"+data[0].toString());
    }
}
```

测试基类修改如下

```
public abstract class DemoBaseTester{
    @BeforeSuite
    public void onBeforeSuite() { /*System.out.println("#-onBeforeSuite");*/ }
    @AfterSuite
    public void onAfterSuite() { /*System.out.println("#-onAfterSuite");*/ }
    @BeforeTest
    public void onBeforeTest() { /*System.out.println("#-onBeforeTest");*/ }
    @AfterTest
    public void onAfterTest() { /*System.out.println("#-onAfterTest");*/ }
    @DataProvider(name="Test-Data")
    public Object[][] getTestData(Method method){
        TestDataGenerator testDataGenerator = new TestDataGenerator(this.getClass().getResource("").toString(), this.getClass().getName(), method.getName());
        TestDataProducer e = new TestDataProducer(testDataGenerator.getTestDataFilePath());
        return e.getExcelData();
    }
}
```

运行结果如下:

```
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd1, name=admin1}
测试如下测试数据(name-passwd): admin1-passwd1
testAddUserAdmin, 进行如下测试数据清理: {passwd=passwd1, name=admin1}
testAddUserAdmin, 进行如下测试数据准备: {passwd=passwd2, name=admin2}
测试如下测试数据(name-passwd): admin2-passwd2
testAddUserAdmin, 进行如下测试数据清理: {passwd=passwd2, name=admin2}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd3, name=guest1}
测试如下测试数据(name-passwd): guest1-passwd3
testAddUserGuest, 进行如下测试数据清理: {passwd=passwd3, name=guest1}
testAddUserGuest, 进行如下测试数据准备: {passwd=passwd4, name=guest2}
测试如下测试数据(name-passwd): guest2-passwd4
testAddUserGuest, 进行如下测试数据清理: {passwd=passwd4, name=guest2}

=====
demo
Total tests run: 4, Failures: 0, Skips: 0
=====
```

可以看到, 每个 Method (测试用例) 只测试了与自己相关的测试数据, 在逻辑上是正确的, 也不会有多余的测试出现, 精确了报告计算, 缩短了测试时间。

这里面隐含的意思是, 如果测试数据没有个性的需要, 使用基类的 DataProvider 即



可，如果需要区分对待同一个测试类的不同测试方法，需要自己定义 `DataProvider`。其中需要指定的是测试方法名称，具体获取测试数据的方法，还是依靠基类提供的统一获取流程。

总结

TestNG 的设计是很巧妙的，简单的使用，很容易就能够上手，但真正研究起来其中有很多玄机。本文从简单的数据驱动实现，进而通过一个实际的 `BeforeMethod` 和 `AfterMethod` 的测试数据需求说明了，在简单的数据驱动后面背后的玄机。在感叹 TestNG 设计巧妙的同时，对后续的使用做出如下总结：

通过 TestNG 的依赖注入，能够将测试数据提供给 `BeforeMethod` 和 `AfterMethod` 使用，完成数据准备和清理的需要。

TestNG 的依赖注入，提供给 `Before*`、`After*` 以及 `Test`、`DataProvider` 注解很多的注入对象，通过这些对象能够完成很丰富的需求。

一切的自动化测试管理，与手工测试的逻辑都是相通的。

每当有自动化测试中想不通的问题时，参考第 3 点总结，会有所启发。

本文使用的 TestNG 版本：6.9.9。



转变你的移动端测试方法

译者：于 芳

概述：

今天的移动端应用和期望值已经快速地改变了，正是持续不断演变的移动端技术和无数用户现在能够跟移动手机互动的方式融合的结果。因为这种技术和用户行为的前进，测试机构必须也能够提升其移动端测试方案来保证他们能尽可能地推送最直观的、最新的经验。

尽管听起来可能有些奇怪，移动端测试是当今移动端企业一直表现不佳的一个关键领域。这主要是因为测试人员被逼着执行那些最小量或太基础的层级的测试，大部分常常是源于缺少测试资源，缺乏经验，或者没有一套健壮的移动端测试策略和计划。

目前大多数测试机构所使用的移动端测试惯例，方法和工具是不够的。公司们要追求一种更严格的测试策略，这非常关键。

我们应该多花时间投入的一个主要测试方法是可用性测试。尽管它经常被忽视，这实际上却是机构们需要多加关注和注意的最重要的移动端测试领域。

我曾经在不同种行业的公司里做过十年，我发现大部分测试机构都使用标准的可用性测试方法：焦点组和 APP 监控，伴随有传统的数量上的分析。正当这种方式可能从前构成了测试最佳管理，今天的移动端应用和期望已经快速地改变了。由于移动端技术和无数用户现在能够与移动手机互动的新方式持续融合的结果。因为这种技术和用户行为的前进，测试机构也必须改进其移动端测试方案来确保他们可以持续输送最直观、最新的经验。

因为我的团队和我已经帮助客户输送游戏变化的移动端解决方案和很好的用户期望，我们注意到那些传统的可用性测试方法不再足够。

首先，测试的黄金法则依然有用：真实的人在真实的设备上仍然是最好的测试方法。但是，我们的经验揭露了一个问题。一次又一次，当你叫一个测试一个移动端 APP 时，他们不会使用你的目标客户会使用的那种方式来测。当分配给他们一个测试一个 APP 的任务时，测试人员的大脑思维和行为变了，变得与使用该 APP 的普通用户的行为有显著地偏差。这包括在每个屏幕上花费的时间长度，他们所遵循的流程步骤，甚至他们用来导航 APP 的姿势。所以，从技术的角度来看，你必须意识到你的移动端 APP 在一个被控制的测试下与其在典型的工作环境里的常规用户条件下使用可能产生不同的结果。



另一个问题：如果你的 APP 不直观、也不易使用，移动用户就会直接不用它。他们会卸载或者就是遗弃掉它。从个人经验上来讲，我在把很多 APP 下载下来的数分钟内就删掉了他们，因为他们会冻结当我想使用其主要功能时，或者其用户界面很拥挤每次我想去点触或者选择我想要的功能时，我得以点击到另外的功能来结束因为那些按钮或文本链接彼此之间太紧凑了。

所以，现在的移动企业要做什么呢？机构们对其可用性测试方法要超越的一个方法是在基于用户触摸地来追踪和产生热量地图的新型数量分析工具的使用上。这些工具聚合了所有的用来与 APP 互动的手势数据--轻触，滑动，捏，等等。用户互动被捕捉而且视觉地呈现为一副透明的热量地图层放置在移动 APP 上方。这使得测试人员能精确地看到用户在哪里以及如何与该移动 APP 互动了。

互动的频率是使用一个坡度色彩图描述的，其中蓝色描述频率最低的互动，红色代表互动最好频区。下面是来自 APPsee 和 HeatData 的触碰热量地图的例子。



任何一个移动方案的主要目标都应当是使其直观和易用。那就是说，一个机构在运行可用性测试时需要多注意的关键区域是无响应姿势：当移动用户与你的 APP 互动但是他们的手指没有得到相应的时刻。

很多因素都会导致这个情况。也许屏幕油腻，APP 有缺陷，或者用户正在尝试滑动而需要一个触击。重点是没有得到相应的收拾不应当被忽略，因为他们代表了用户挫败感的其他源头，以及用户可能会删掉你的移动 APP 的另外一个原因（并且在 APP 应用商店里或通过社交媒体发布差评。）

另外一个超过传统的可用性测试方法并且融合类似触碰热量地图这样的工具以此来跟踪移动端互动情况是很多不同的移动设备当今会使用的方式因素，例如手表，电话和平板电脑。补充资金到这些新型可用性测试方法上将会帮助开发者识别移动 APP 元素是否是看起来是离开屏幕还是不同的方式因素导致了糟糕的 UI 期望。



作为一个移动端策略专家，我看到越来越多地公司在运行移动应用程序开发平台方案来使他们能在多个平台和设备（原生的，混合的和 web 的）部署一个单一的代码库。这已经越来越成为移动 APP 开发的规范标准而且是无可争议的完好的策略，部署到不同的平台和设备上创造了额外的复杂性当遇到移动端测试时，尤其是可用性测试时。

我最近在与一个有成千上百名员工的客户一起工作，他们在该领域发展且使用不同类型的移动设备，包括加强坚固的平板电脑和只能手机。一些员工被要求穿戴保护性装备或一副来执行其工作，所以当帮他们测试他们的移动 APP 时，我们理解每个用户将要使用这些设备和移动解决方案的实际环境和方式，这点很重要。我们试着模拟相同的用例和环境，尤其是那些不常见的，例如在带着保护性手套时操作设备。

移动 APP 开发和持续演变的移动技术的前进十年后，我们绝对确认的一件事是移动用户期望和行为也将继续演变。

为了确保你的用户对你的移动解决方案有良好的体验，你需要一个规律评审的完好定义的移动测试策略和帮你超越的工具来保证你的 APP 被用户接纳--并不断使用。

❖ 拓展学习

■ 全栈式测试开发，高级实战通关：<http://h.atstudy.com/atstudy/wow/pc/>



聚焦回归测试

作者：高上明

最近，从生产上反馈了一些产品缺陷，开发、测试一起对缺陷进行了多维度的原因分析，缺陷的逃逸是由产品生产过程中多种因素造成的，其中有部分原因是回归测试的策略选用不当。在此，我们有必要聚焦一下回归测试。

什么是回归测试

回归测试的英文名称：**Regression Testing**，从字面上看，是“倒退测试”。这就表明，软件有当前状态和原来状态之分。回归测试就是对软件的原来状态重新进行功能和非功能的测试，用以确保先前开发并测试过的软件在缺陷修复、配置改变、软件更新等等这些变化之后，仍能符合要求的运行（即：软件当前状态中那些没有被修改的部分的功能和非功能与原来状态保持一致）。

这里所说的软件当前状态和原来状态的概念，可以简单的认为，每一个当前提交测试的版本就是该软件的当前状态，上一个版本就是原来状态。或者，当前基线就是当前状态，上一次基线就是原来状态。具体如何区分，可以根据管理要求来制定。

回归测试的时机

不论是基于新产品的开发还是老产品的维护升级，回归测试都是一个完整测试必不可少的一部分。通常，需要进行回归测试的时机是：

➤ 缺陷修复

当开发人员完成对当前状态的软件进行缺陷修复之后，并且提交新的测试版本或进行了代码基线；

➤ 软件更新

开发人员完成新增功能和代码重构之后，并且提交测试版本或进行了代码基线。其中，新增功能往往对应于维护类的产品，对其要进行完整的测试，即对新功能的验证测



试，以及回归测试。对于代码重构，应该本着运行质量永远高于代码质量的原则慎重对待，如果一旦决定重构，则必须进行全量回归测试；

➤ 配置改变

出于软件参数配置灵活性以及运维高效的需要，对配置文件、脚本、批处理命令、存储过程进行修改并正式通知测试部门之后。

回归测试的技术

回归测试的技术通常有以下三种：

➤ 全量测试用例重复测试

对当前状态的软件进行全部测试用例的执行和结果检查。这种技术由于进行的是全量测试用例的重新执行，能将缺陷遗漏的风险降至最低水平，但是测试成本（时间成本、人力成本、财务成本）很高；

➤ 选择部分测试用例重复测试

与上述不同的是，只对当前状态的软件进行部分测试用例的执行和检查。这种技术的最大好处是测试成本（时间成本、人力成本、财务成本）低于全量测试用例重复测试；

➤ 测试用例优化

测试用例的优化技术是指，从测试用例的范围即有效和效率角度出发，优化、提取对后续软件版本或特定版本有用的测试用例（集合）。

回归测试的策略

回归测试技术用以服务于回归测试策略。通常情况下，我们在日常工作中会在以下策略中进行选择。

➤ 全量回归测试策略

在一些特殊行业，比如金融、通信行业，对生产环境中软件产品质量的要求通常是严重以上级别的逃逸缺陷是 0，在此情况下，我们必须用安全系数最高、风险等级最低的策略进行回归测试，以达到质量要求。这时，我们可以采用全量测试策略，使用全量测试用例重复测试的技术进行回归测试。



全量回归测试策略可以适用任何情况，它是一种不是策略的策略，虽然有效，却不高效。

➤ 非全量回归测试策略

实际工作中，测试用例会越来越多，工作量也会越来越大，特别是维护更新类的产品。当质量要求不变，又有上线时间以及其它条件制约时，我们会选择非全量回归测试策略，使用部分测试用例重复测试技术进行回归测试。这也是我们最常使用的策略。

这种策略的核心思想就是使用测试用例优化技术，提高测试效率，减少资源投入。

但是这种策略的漏测风险也是显而易见的，比如，测试人员对代码的相依性分析能力较弱，这很容易导致忽略了可能会发现缺陷的测试用例。为了规避这个风险，除了采用测试用例优化技术之外，还应该结合基于业务风险驱动的分析方法来进行。

总体上，测试人员需要通过与开发人员、业务人员的讨论，对被测试软件进行风险分析和操作分析，了解被测软件最重要、最频繁使用的功能，也就是了解被测软件的业务关键度与故障可能性，然后根据这两个维度的风险分析结果采用相应的测试用例（集合）优化技术进行回归测试。

具体的，在测试行业里业务影响测试是一个公理，因此，要了解被测软件的业务关键度是首要任务。其次，软件及其中的模块都有故障可能性，这二者的相关指标及示例可以用以下二表所示：

功能模块	业务关键度指标						业务关键度
	功能类型	故障影响类型	使用频率	受影响用户数	受影响范围	接口使用	
账号登录	业务处理	业务无法使用	高	高	监察部	使用	高
静态界面	业务处理	提示信息不能显示	中	中	监察部	不使用	中

表 1：业务关键度指标与示例

功能模块	故障可能性指标					故障可能性
	功能复杂度	缺陷检出情况	严重级别情况	变更情况	技术成熟度情况	
账号登录失败	中	高	高	高	稳定	高
静态界面	中	低	低	低	稳定	低



验证						
----	--	--	--	--	--	--

表 2: 故障可能性指标与示例

通过以上二表，结合业务关键度与故障可能性评估得出被测软件功能模型的风险等级，如下表所示：

功能模块	业务关键度	故障可能性	功能模块风险
账号登录	高	高	高
静态界面	中	低	低

通过上表，为非全量回归测试策略中缩减测试用例集合，集中力量做最需要的测试，提供了依据，从而降低漏测风险。当然，不论表 1 还是表 2，其中的等级划分需要测试人员的经验积累，也需要业务人员、开发人员的有力支持。

自动化回归测试策略

由于回归测试是一个反复的过程，通常我们一提到回归测试的时候，就会考虑使用自动化的方式进行。因此，可以通过使用测试用例优化技术将提取出的测试用例进行按维护、复用、自动化的步骤进行自动化回归测试平台的建设，这种方式对维护类、升级类软件（比如：市场监察、结算系统）的回归测试可以做到即有效且高效。

但是，若要达到使用自动化回归测试策略的目的，需要专门人员进行实施，从业务流驱动、数据与用例分离、组件技术、统一的平台多维度出发进行长期建设。

我们都知道，测试不可能做到完全穷尽，但我们可以通过在软件的不同阶段采取不同的回归测试策略，减少直至避免在生产上出现严重以上级别的逃逸缺陷。当然，如果从软件生产的全生命周期入手严格进行代码与缺陷管理可以更直接的降低线上产生缺陷的风险了，毕竟回归测试也是可以用于单元测试的。



如何更有效地进行自我管理？

◆作者：晴空

开篇语：时间，计划，目标，压力如何管理？如何锻炼自己的创新思维？怎样进行高效的沟通？职业规划仅凭自己的兴趣来吗？如何超出个人视角来分析问题？好吧，这篇总结就是和小伙伴们探讨怎样面对上述问题的，是否能解决问题，请小伙伴们自行拿捏领学活用，至少我相信，这篇文章中我们会 get 到新的技能点~

一：时间管理

我的时间都去哪儿了？

原因很多，有些是客观原因，比如被打断，去找其他小伙伴面对面沟通，喝水，上厕所，闲聊(我感觉有一种“编程 5 分钟扯淡 2 小时”的既视感，请自行脑补~)；

主观原因大部分时候是如下的场景：

- 做事目标不明确(我们也会聊到目标管理的，莫慌)；
- 做事拖泥带水；
- 缺乏有限顺序，抓不住重点；
- 过于注重细节；
- 做事虎头蛇尾；
- 没条理，不简洁，简单事情复杂化；
- 事必躬亲，不懂授权；
- 不会拒绝别人的请求；
- 消极思考。

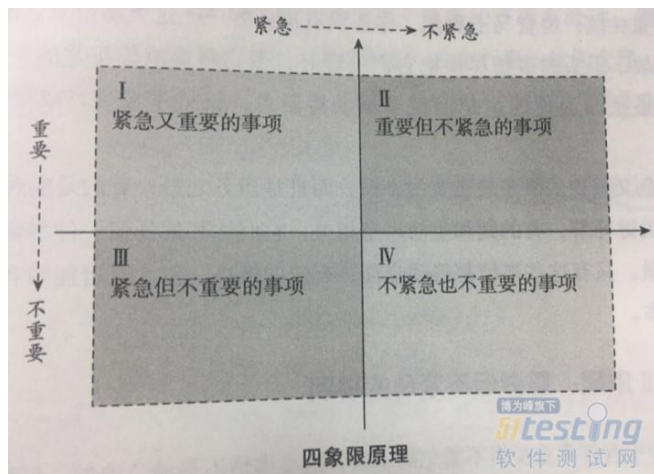
有一种忙，叫“瞎忙”！这个梗相信有小伙伴体会很深，但是小伙伴们有没想过自己的时间浪费在哪儿了？我们如何才能在有限的时间资源下把各项事务有条不紊或者说游刃有余地解决？不同的小伙伴们在相同的时间长度和环境下其效能会有很大，这就说明时间可以被更好地管理。通过时间管理，我们可以提高单位时间的效率，做更多的



事，更高效地做事，这也是我们要进行时间管理的原因。

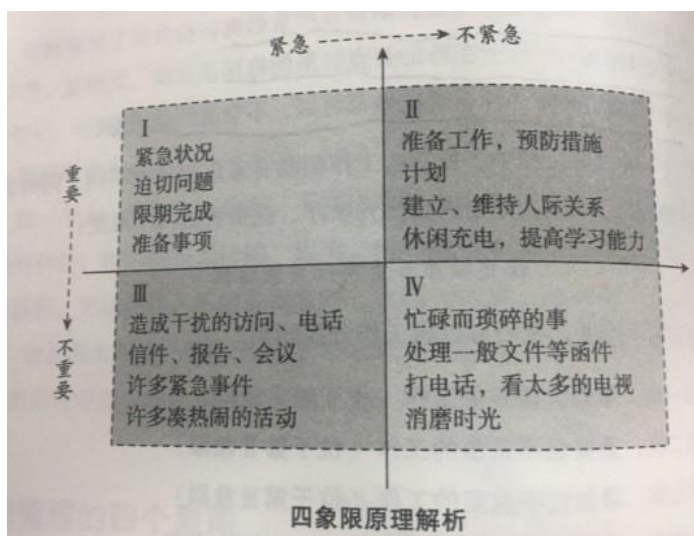
“四象限原理”规划时间

所有的事务不是同等紧急，重要的。我们根据重要性&紧迫性两个维度把事情分文别类。



- 第 I 象限：紧急又重要的工作。
- 第 II 象限：重要但不紧急的事项。
- 第 III 象限：紧急但不重要的事项。
- 第 IV 象限：不紧急也不重要的事项。

如何具体将事项归类呢，我们深挖下四象限原理：



I 类：“紧急”是必须马上需要做的事项，“重要”是对团队或个人有重大影响的事项。



(ps: 只有这些事情得到合理有效的解决, 我们才有可能顺利第进行别的工作)

II 类: 这类事情不是最急需要做的, 但是关系到长远发展。

(ps: 重要的事情一般有较为充足的时间安排, 都是可以在一定的时间内完成的。但是如何每天忙于琐碎的次要的事, 那么这类事情将会变成第 I 类事项)

III 类: 这些事情, 紧急但不重要, 应列入次有限的事项中。

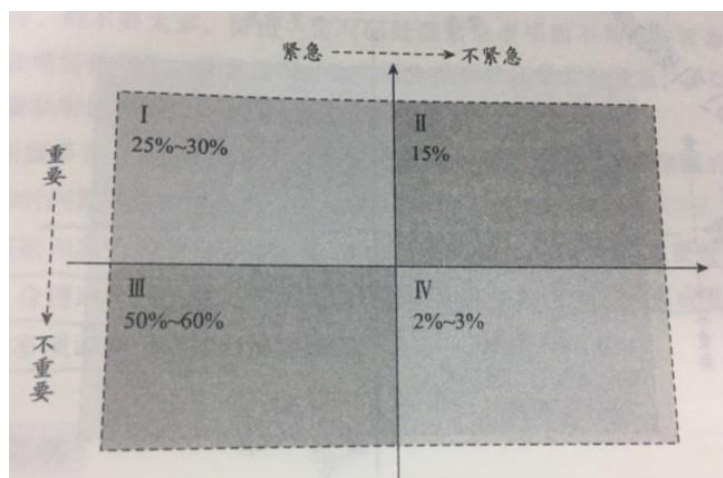
(ps: 如果没安排好优先次序, 可能会把一些紧急的事项当成了重要的事项来处理, 颠倒了主次。)

很多时候, 我们习惯按照事情的“紧急程度”决定行事的优先次序, 而不是首先衡量事情的“重要程度”, 如果每天花费 80% 的时间和精力在“紧迫的事”上, 无疑会导致效能的降低。

要避免把大部分时间和精力花在紧迫但不重要事情上, 可以兼顾紧急&重要程度。

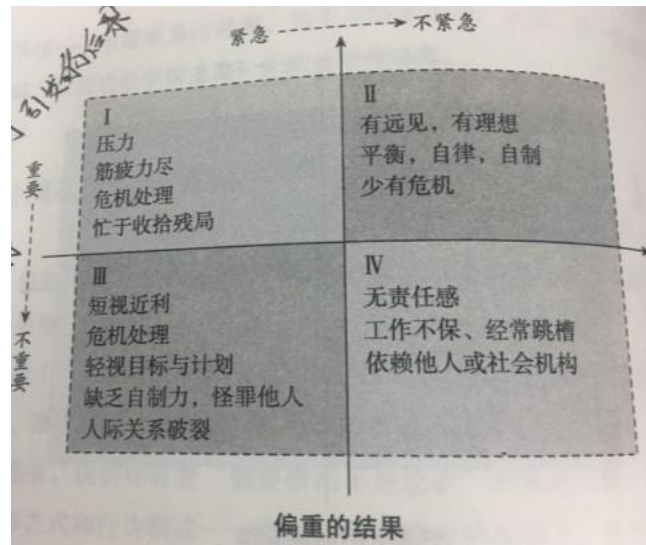
IV 类: 既不紧急也不重要的事项, 可做可不做。

我们大多数同学的时间安排可能是下面这样的: 我们的大部分时间被紧急但不重要的事项所耗费!



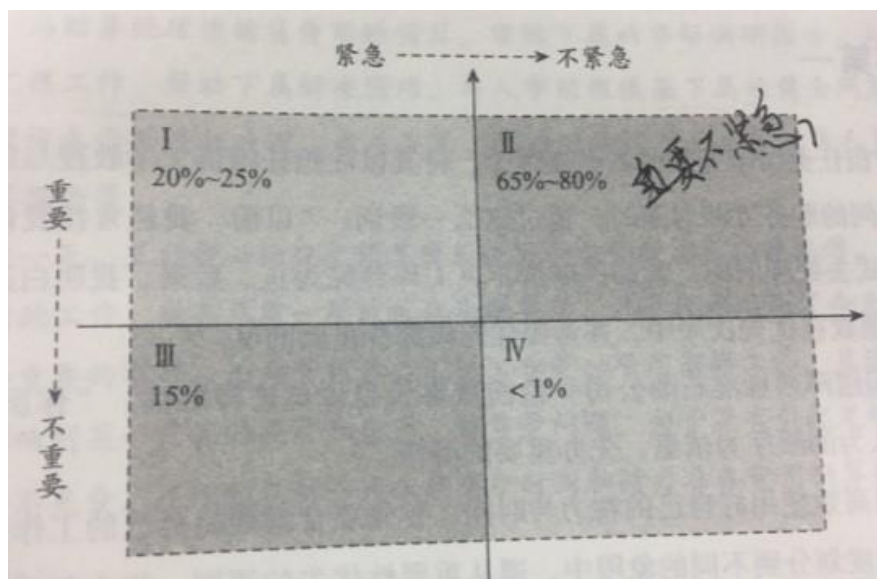
来对比下这样不合理的事项偏重所引发后果吧





(细思极恐！有木有？有木有？)

我们理性的时间安排应该是下面这样的：



敲黑板：根据四象限的原则安排工作时，一定要遵从重要性优先的原则！！

我们首先需要将自己的工作按重要和紧急程度划分为到不同的象限中，遵从要事第一也即重要性优先的原则，将大部分时间和精力用于重要而不紧急的第 II 象限事务中去！

值得注意的是：很多第 I 象限的事务实际上是因为没及时处理好第 II 象限的工作而产生的。

~~~手动划重点~~~ 不要把重要的事项最后都推到第 I 象限，也不要整天集中精力在第 III 象限上去做看上去非常紧急却不那么重要的事项。要做好第 I 象限的工作，但不需要太多。留出一定时间处理紧急事项而不是陷在紧急事项中。注意区分第 I，第 III 象限的工作，确保是不是仅仅是紧急而



不那么重要，不要被紧急的假象所迷惑！

## 番茄工作法

番茄工作法（Pomodoro Technique），是一种最为简单易用的工作方法。

番茄工作法的应用过程如下：

从今天的番茄记录表中选出一件事情，然后开始执行。

在执行前，先将计时器——计时器可以是手机闹钟，或专门的番茄计时器，设置到 25 分钟，然后便开始专注的工作。

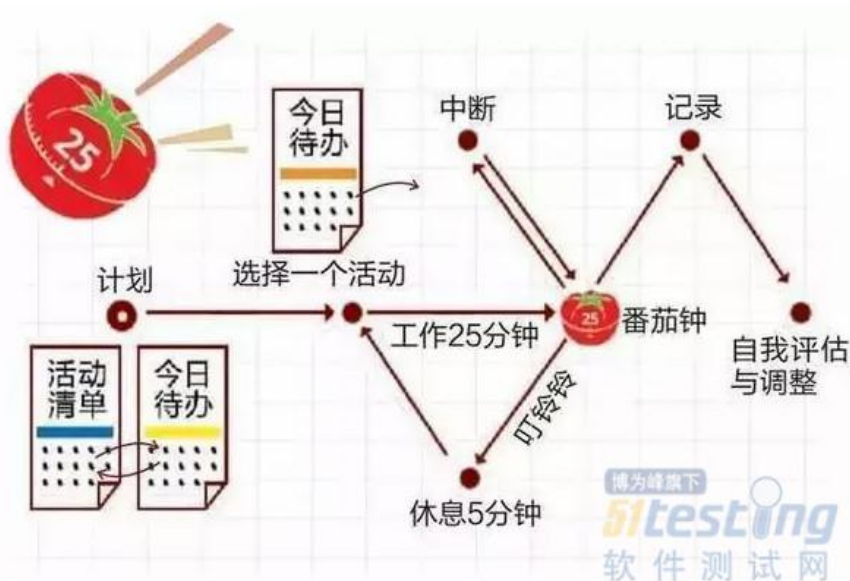
25 分钟就是一个番茄时间，在这 25 分钟里，不能被打断，如果你突然想接个水、聊个天等等，都可以记录到番茄记录表的“计划外事项”清单中，记录完毕，马上回到专注的工作中来。

当 25 分钟结束，闹钟响起，一个番茄时间结束，立即停下工作，在番茄记录表的该事项后面做一个完成标记，并开始 5 分钟的休息，在休息期间，不要想与刚才工作相关的任何内容，这样有利于保持大脑的清醒。

注意：在番茄时间中遇到非打断不可的情况，比如小伙伴们有急事找你，你只得放下你的番茄时间，那么这个时候，这个番茄时间作废，哪怕你已经专注了 24 分钟，只剩下 1 分钟。

5 分钟的休息时间结束，便开始你的下一个番茄时间。

当执行完四个番茄时间时，来一轮大的休息，长度为 30 分钟。图示如下：





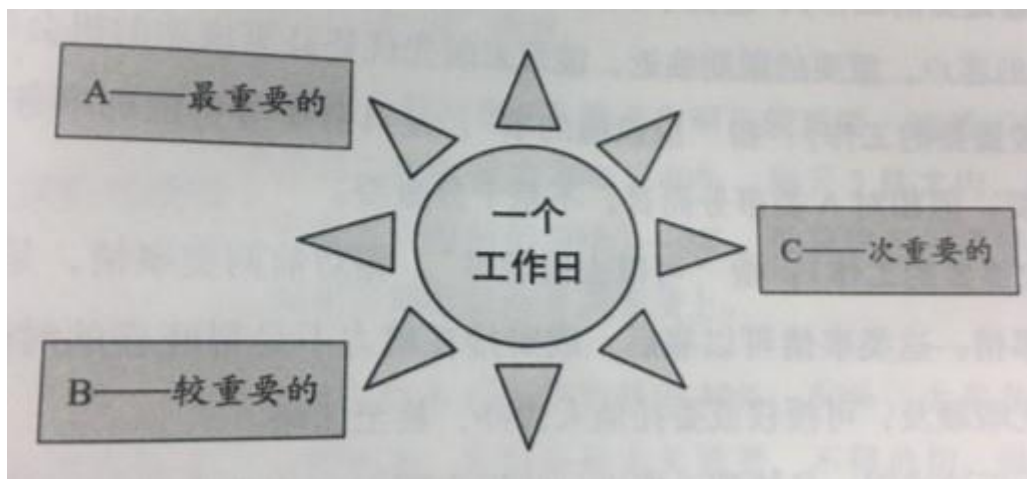
### 番茄工作法的特殊说明:

一个番茄时间, 即 25 分钟, 不可分割, 不存在半个番茄时间的概念, 一个番茄时间被打断, 就应作废。

当一个任务所花的番茄时间大于 5 个时, 将这个任务当成一个大的 task, 在执行前, 分解为多个小任务。

### “ABC”控制法

“ABC 控制法”是根据事务在工作中的重要和紧急程度按照最重要, 重要, 不重要 3 种情况分成 A,B,C 类, 有区别第管理时间的是一种分析方法。

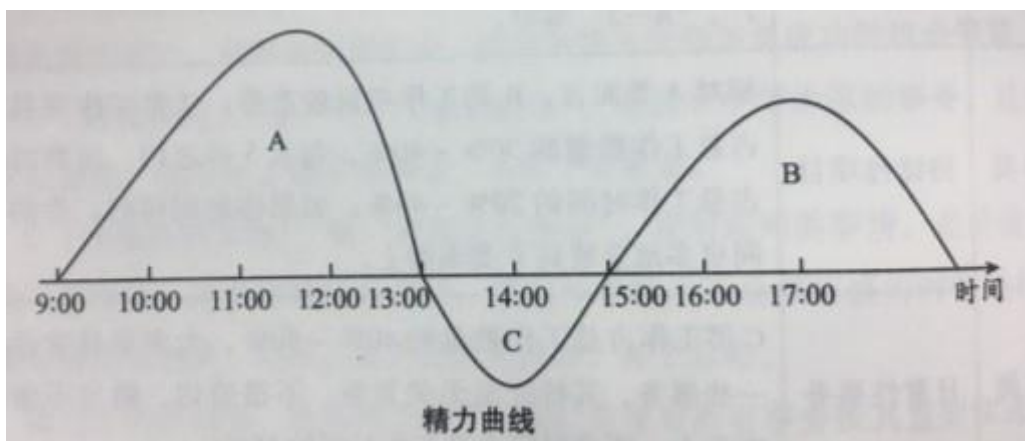


### “ABC 控制法”具体说明:

- A(最重要的工作): 指“必须做的事”, 是关键事务。
- B(较重要的工作): 指“应该做的事”, 具有中等价值的事务。
- C(次重要的工作): “可以去做的事”, 如果无暇顾及, 可授权/委托他人待办或者忽略。

根据精力变化, 合理安排“ABC”类工作





根据生理活动周期性变化的规律，人分为“百灵鸟”型，“猫头鹰”型和“混合型”。

“百灵鸟”型：黎明即起，情绪高涨，思维活跃，他们喜欢在凌晨进行最复杂的创造性工作。

“猫头鹰”型：夜晚大脑兴奋，精神饱满，毫无倦意，他们乐意在深夜工作。

“混合型”：这类同学全天用脑效率差不多，相对而言上午8~10点，下午3~5点左右效率最高。

就整个人类群体来说，绝大多数属于“混合型”。

随精力不同安排工作时应遵守以下的原则：

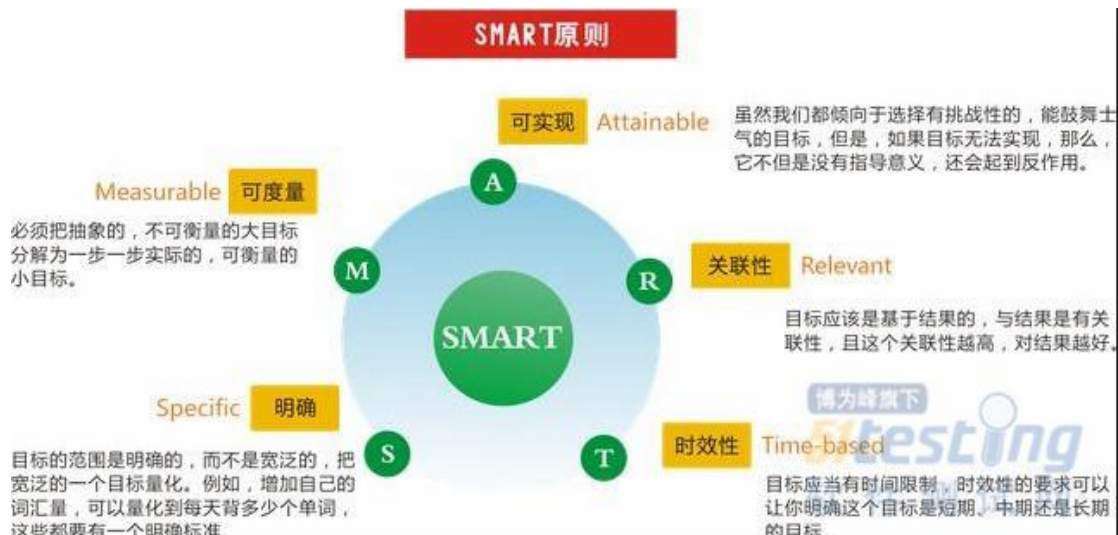
- 1: 以重要的，关键的工作项目为中心，制定一天的工作日程。
- 2: 以当天必须首先要做而且必须做完的工作为中心，制定一天的工作日程。
- 3: 使工作日程和自己的身体状况，能力曲线相适应，在精力最充沛时从事那些最富有创造和挑战性的工作。

## 二：目标管理

人的行为特点本身是有目的性的，目标作为一个人或者一个团队实现目的明确声明，给人的行为设定了明确的方向，人可以把自己的行为与目标加以对比，清晰地评估每个阶段的进展，检讨每个行为的效率，进而清晰地知道自己或团队与目标的距离。

### 2.1 用“SMART”确定目标





## 何为“SMART”？

**S: 可确定的(Specific)。**

目标必须的明确的，具体的。明确就是实现规定目标的工作量，达成日期，责任人，资源等。

**M: 可衡量的(Measurable)。**

目标必须可以将其表现形态用数字化指标来补充描述。

Ps: 如果目标无法衡量，就无法告诉执行者要到哪里去；如果没有一个衡量标准，具体的执行者就会少做工作，尽量减少自己的工作量和为此付出的努力，因为他们认为没有具体的指标要求，也没有约束他们的工作必须要做到什么地步，只要似是而非地做些工作就可以了。

**A: 可接受的(Acceptable)。**

目标必须是可以被目标执行人所接受的，这里所说的接受是执行真正愿意接受这一目标，认同这一目标。

**R: 现实可行的(Realistic)。**

目标在现实条件下不可行，常常由于乐观地估计了当前的形势。一方面可能过高估计了达成目标所需要的各种条件，制订了不恰当的工作目标；另一方面可能是错误地理解了目标，主观地认为执行人能够完成目标。

Ps: 从最基本的出发点来说，一个目标无法实现就无法保证目标管理进行下去。

**T: 有时间限制的(Time Indication)。**

如果没事先约定的时间限制，每个人都会对一项工作的完成时间各有各的理解。



利用“SMART”制订目标的步骤:

Step1: 列出符合 SMART 标准的目标。

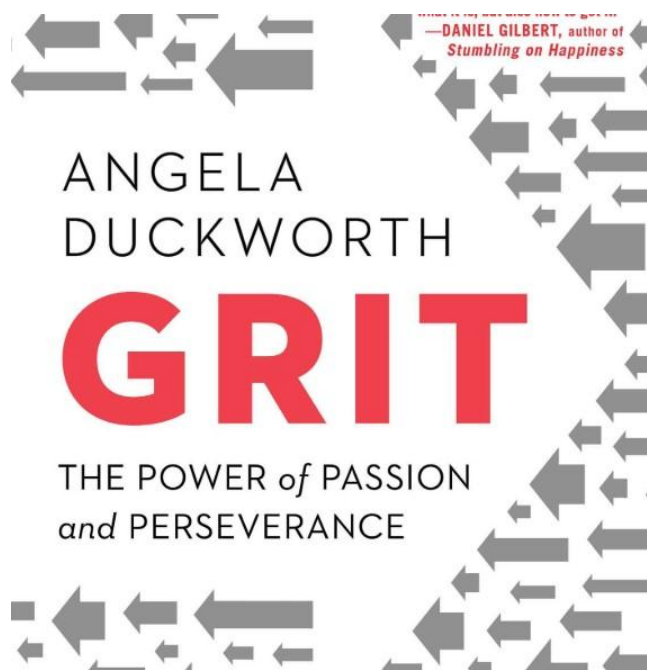
Step2: 列出上述目标带来的好处。

Step3: 可能的困难与阻碍, 以及相应的解决方案。

Step4: 所需的技能和知识。

Step5: 为达成目标必须合作的对象

Step6: 目标完成日期。



目标固然重要, 但实现目标过程中我们不可避免地会遇到各种挫折困难。

坚毅(Grit)The Power Of Passion And Perseverance 是最重要的品性!!!

## 2.2 用“目标多权树”分解目标

制订清晰有效的目标是我们提高工作效能的重要方式, 但是如果不会分解目标, 也许, 我们的目标会是海市蜃楼。

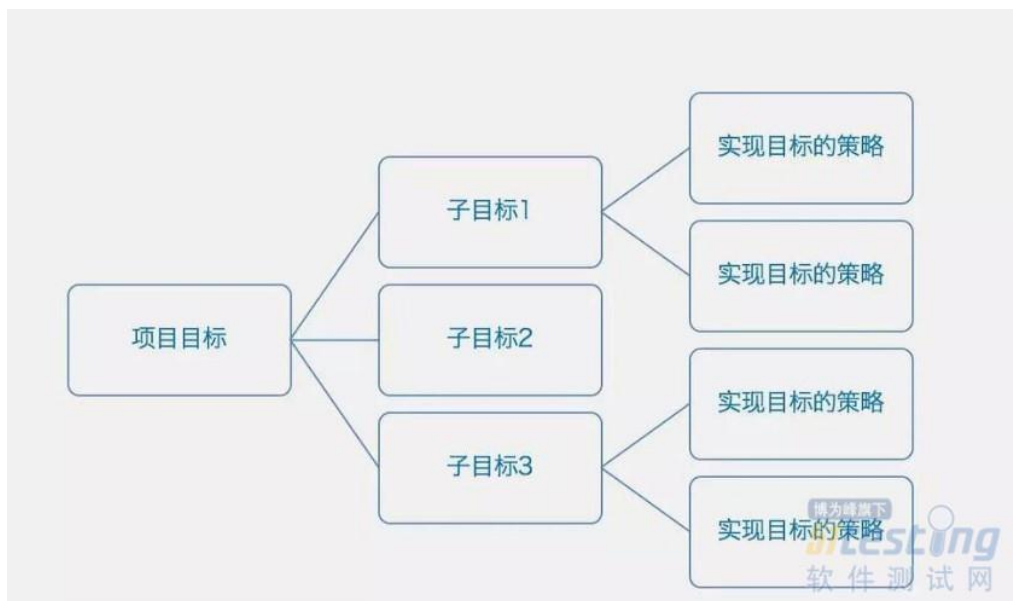
许多事之所以半途而废, 也许并不是因为困难重重无法逾越, 而是目标迷失!!!

什么是目标多权树?

目标多权树是专业的目标分解方法, 用树干代表大目标, 用每一根树枝代表小目标, 用叶子代表即时的目标也就是陷在要去做的一件事。也叫“计划多权树”。



## 如何运用目标多权树？



在目标多权树中，大目标和子目标的关系：

- 子目标是大目标的策略和条件。
- 大目标是小目标的结果。
- 子目标实现之“和”一定是大目标的实现。

Step1: 写下一个大目标。

Step2: 写出实现该目标所有的必要条件及充分条件作为子目标，使其成为第一级树杈。

Step3: 写出实现每个小目标所需的必要条件及充分条件，变成第二级树杈。

Step4: 以此类推，直到画出所有的树叶----即时目标为止。

Step5: 检查多权树分解是否充分，即反向从叶子到树枝再到树干，不断检查如果小目标均达成，大目标是否一定会达成。如果是则表示分解已完成，如果不是则表明所列的条件还不够充分，继续补充被忽略的树杈。

Step6: 评估目标。从目标合理性和计划可行性评估。

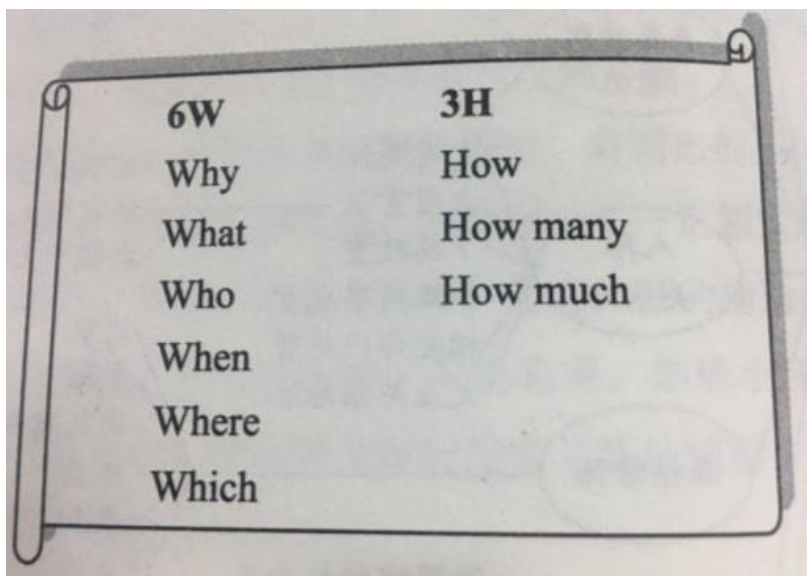
评判准则 1: 目标分解完成后，单位时间无法完成“树叶”所需工作量，表示目标太大。

评判准则 2: 目标分解完成后，单位时间可轻松完成，表示目标太小。



Ps: 为自己的每个目标做系统的评估, 避免因目标太大最终无法实现, 亦避免因目标太小而浪费自己的潜能。

## 2.3 用“6W3H”细化目标



### 6W3H 详解:

- 为什么(Why)

Why 指做事的理由, 目的, 根据。问 Why 可以让我们更清楚地洞悉我们这样做的缘由。

- 什么(What)

What 指要做的是什么及描述达成指令事项后的状态。问 What 以确认目标一致。

- 谁(Who)

Who 指达成目标要接触或关联的对象。

- 何时(When)

When 指达成目标所允许的时间期限或者子任务的 dead line。

- 何地(Where)

Where 泛指各项活动发生的场所。

- 哪个(Which)

Which 指各种选择及优先顺序。





- 怎样(How)

How 指方法，手段，怎么做。

- 多大/少(How Many)

How Many 指需求大小，让目标量化。

- 多少费用(How Much)

How Much 指多少，同 How Many。

### 6W3H 法的实施步骤:

Step1: 明确了解工作进行的目的及理由(Why)

Step2: 明确要做哪些事项(What)

Step3: 明确团队成员(Who)

Step4: 明确截至日期(When)

Step5: 明确任务完成地点(Where)

Step6: 确定各项工作优先顺序，找到解决问题的重点对策(Which)

Step7: 明确各项任务进行的步骤(How)

1: 找到问题的真正原因。

2: 确定解决问题的重点对策。

3: 制定解决问题的行动计划。

Step8: 明确工作量(How Many)

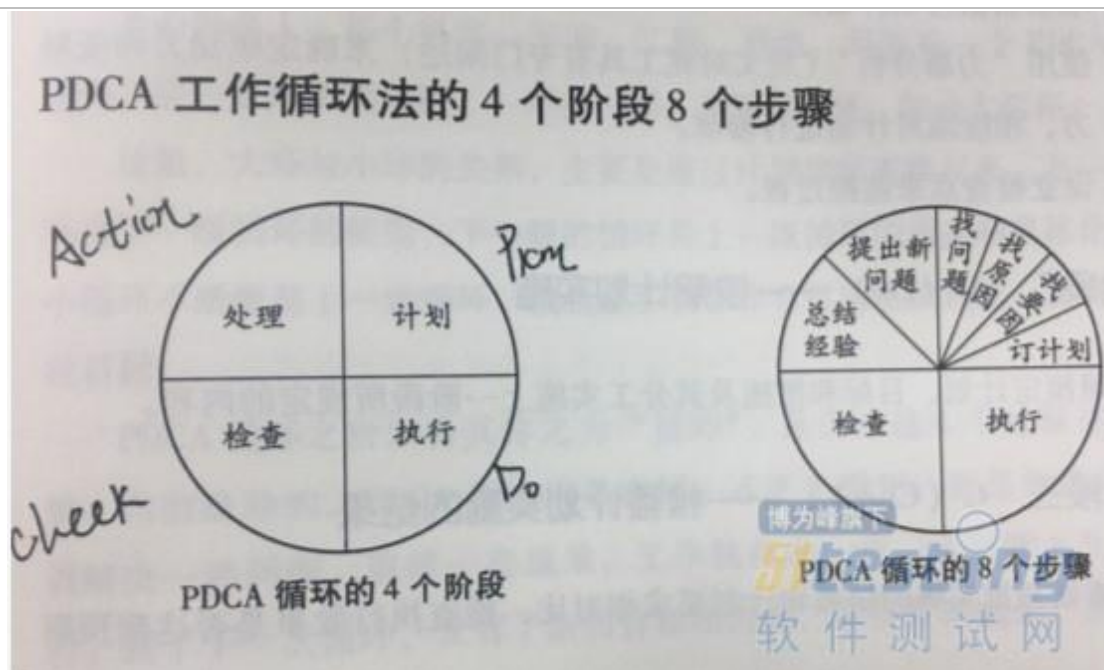
Step9: 明确预算(How Much)

### 三：计划管理

#### 3.1 用“PDCA”戴明环实施计划







“戴明环”解释:

P: Plan(计划)

D: Do(执行)

C: Check(检查)

A: Action(处理)

PDCA 闭环的实施:

阶段一: **Plan** 制定详细的工作计划

Step1: 找问题

Step2: 找原因

Step3: 找要因, 找出主要原因, 明确问题最根本的原因。

Step4: 拟定计划

阶段二: **Do** 按照计划实施

Step5: 按照预定计划, 目标和策略及其分工, 实施上一阶段的内容。

阶段三: **Check** 检验实施效果

Step6: 对比计划的预期结果和实际结果。



阶段四：Action 跟进实际结果总结经验，制定下一步的计划。

Step7: 总结经验。分析成功的方案和做的不好的地方，针对不足调整计划。

Step8: 提出问题。反思过程，通过一个工作循环，学到了什么？找到遗留问题并转入下一个循环解决，并为下一个循环提供资料和依据。

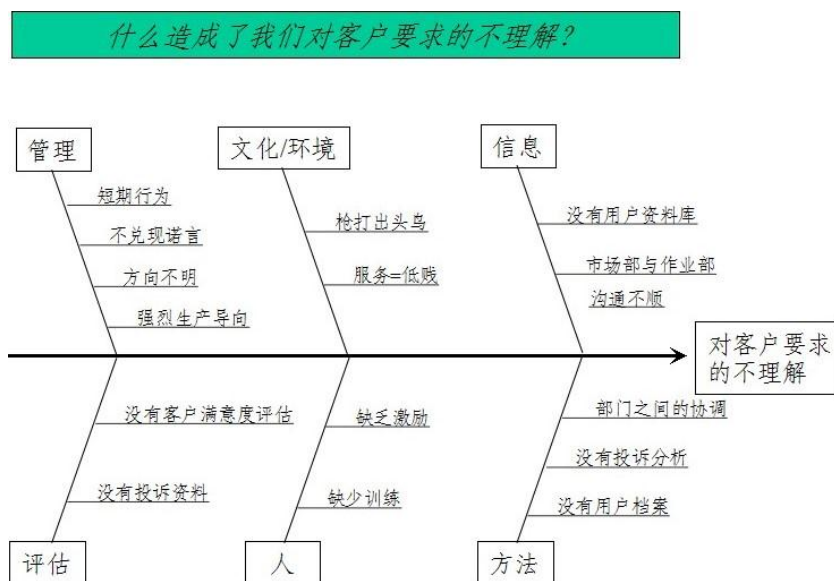
**PDCA 是一个闭环，很多时候我们在工作必须要坚持两个原则：**

1: 闭环原则。所谓闭环原则是指凡事要有始有终都要根据 PDCA 来循环，而且是螺旋上升。

2: 优化原则。根据木桶原理，找到短板即时整改提高全局水平。

## 四：创新管理

### 4.1 鱼骨因果图



所谓鱼刺因果图法，是在工作中寻找关键问题产生原因的一种图示法，它揭示结果和各种影响因素间的关系，有助于我们理清思路，明确面临的问题以及解决问题过程中所处的位置。

### 如何应用“鱼骨因果图”法

#### Step1: 选择问题

选择一个具体的问题或结果，把问题写在白板上，要保证问题范围足够小，参与人员可以理解要分析的问题和内容。



## Step2: 头脑风暴

对问题或结果所有可能原因进行一次头脑风暴，记下每条分析意见。

## Step3: 绘制鱼骨图

将问题或结果置于头部，把各种原因分类，顺着鱼骨向后写出主要原因类别，把每一类原因都标记在相应的鱼骨或鱼脊上，简而言之，将头脑风暴得出的潜在原因已鱼骨图形式展现出来。

## Step4: 分析根本原因

以下的方法在分析原因时可以有

- 1: 通过参与者之间的公开讨论来分享看法和经验。
- 2: 寻找重复原因或鱼特定类别有关的原因数目。
- 3: 使用检查表收集资料。

示例:

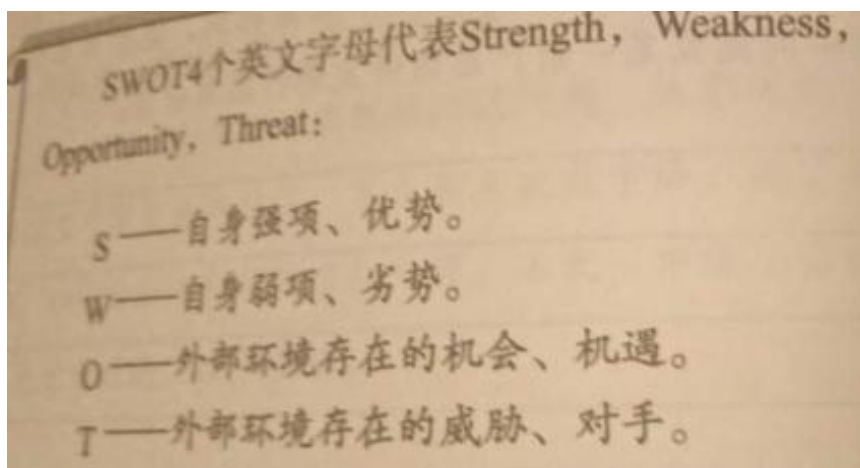


## 4.2 SWOT 分析法

什么是 SWOT 分析法



SWOT 分析法是指在分析时, 将与我们密切相关的各种主要内部优势因素(Strength), 弱点因素(Weakness), 机会因素(Opportunity)和威胁因素(Threat)通过调查罗列出来并依照一定的次序按矩阵形式排列起来, 然后运用系统分析的思想, 把各种因素相互匹配起来加以分析, 从中得出一系列的结论或对策。



S: 自身强项, 优势。

W: 自身弱项, 劣势。

O: 外部环境存在的机会。

T: 外部环境存在的威胁, 对手。

#### 实施步骤:

Step1: 确定分析的问题, 构造 SWOT 矩阵

1.1 画一个方格, 对 4 个领域进行头脑风暴。

1.2 将每一项都列在图表的响应象限内。

在此过程中将直接的, 重要的, 迫切的, 长期的影响因素有限排列出来, 而间接的, 次要的, 不急的, 短暂的影响因素排在后面。

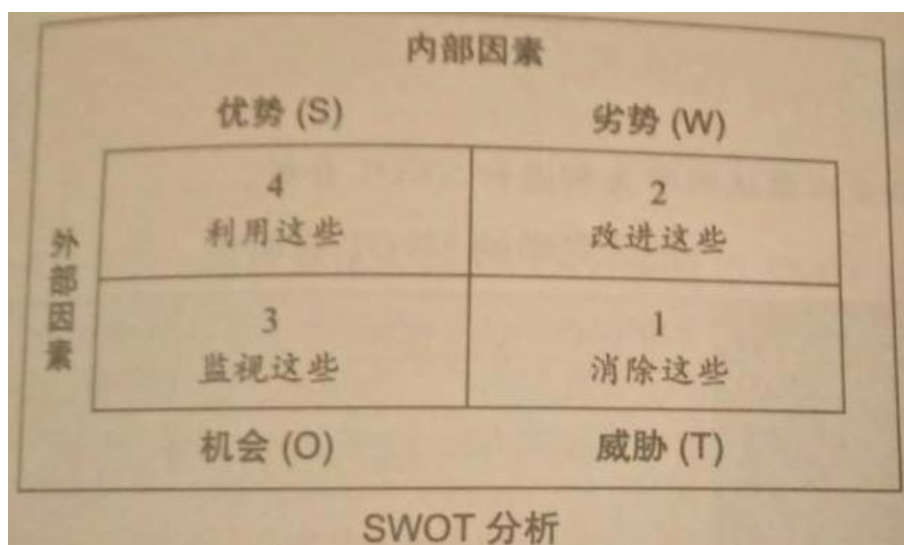


|                                                                                                                                                                                                          |                                                                                                                                                                                                   |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>关键优势 (S)</b> <ul style="list-style-type: none"> <li>你擅长什么?</li> <li>你有什么新技能?</li> <li>能做什么别人不能做的事?</li> <li>如何能够重复最近的一次成功?</li> <li>什么使你与众不同?</li> <li>你的顾客为什么要来你这里?</li> </ul>                       | <b>关键劣势 (W)</b> <ul style="list-style-type: none"> <li>你不擅长什么?</li> <li>缺乏什么新技能?</li> <li>别人在什么事情上比你做得好?</li> <li>你最近的一次失败是什么? 为什么?</li> <li>你尚未完全满足哪一个顾客群体?</li> <li>你最近失去了哪些顾客? 为什么?</li> </ul> |
| <b>关键机遇 (O)</b> <ul style="list-style-type: none"> <li>是否发生了你希望的变化?</li> <li>你学会了什么技能?</li> <li>你能够提供什么新产品或服务?</li> <li>你能够接触到哪一个新的顾客群体?</li> <li>你如何能够使你与众不同?</li> <li>你的组织在未来 5~10 年中的情况怎样?</li> </ul> | <b>关键威胁 (T)</b> <ul style="list-style-type: none"> <li>是否发生了不利于你的变化?</li> <li>你的竞争对手正在做什么?</li> <li>是否发生了任何会伤害你的变化?</li> <li>是否存在威胁你所在组织的情况?</li> </ul>                                           |

## Step2: 制定策略

为了掌控全局, 我们应该考虑到图表中的每种组合, 创造出更多的可选择结果, 并对这些结果进行评估。

|                            |                            |                            |
|----------------------------|----------------------------|----------------------------|
|                            | <b>关键优势 S</b><br>(按程度顺序排列) | <b>关键劣势 W</b><br>(按程度顺序排列) |
| <b>关键机遇 O</b><br>(按程度顺序排列) | 机遇与优势组合<br>会出现什么样的选择       | 机遇与劣势组合<br>会出现什么样的选择       |
| <b>关键威胁 T</b><br>(按程度顺序排列) | 优势与威胁组合<br>会出现什么样的选择       | 劣势与威胁组合<br>会出现什么样的选择       |





整体上看, SWOT 可以分为两部分, 分别是 SW(内部条件)和 OT(外部条件)

我们分析时可以综合各方面的因素得出适合自己的策略

1: 最小与最小策略(WT 策略)

即: 考虑弱点因素和威胁因素, 目的是努力使这些因素趋于最小。

2: 最小与最大策略(WO 策略)

即: 着重考虑弱点因素和机会因素, 目的是使弱点趋于最小, 使机会趋于最大。

3: 最大与最小策略(ST 对策)

即: 重点考虑优势因素和威胁因素, 目的是努力使优势因素趋于最大。

4: 最大与最大策略(SO 策略)

即: 考虑优势因素和机会因素, 目的是努力使这两种因素都趋于最大。

利用 SWOT 方法可以很明显地找出对自己有利的值得发扬的因素, 以及对自己不利的和该去避开的因素, 发现存在的问题, 可以将问题按轻重缓急分类, 找出解决方案, 并明确后续的发展方向。

## 五: 员工管理

### 5.1 根据意愿和能力对小伙伴分类

对一个渴望成功的小伙伴来说, 最严格的表现标准应该是自己设定的, 而不是他人要求的!

我们姑且将成功定义为达成预期目标, 那么目标达成的 3 大核心要素是: 意愿, 方法, 行动。

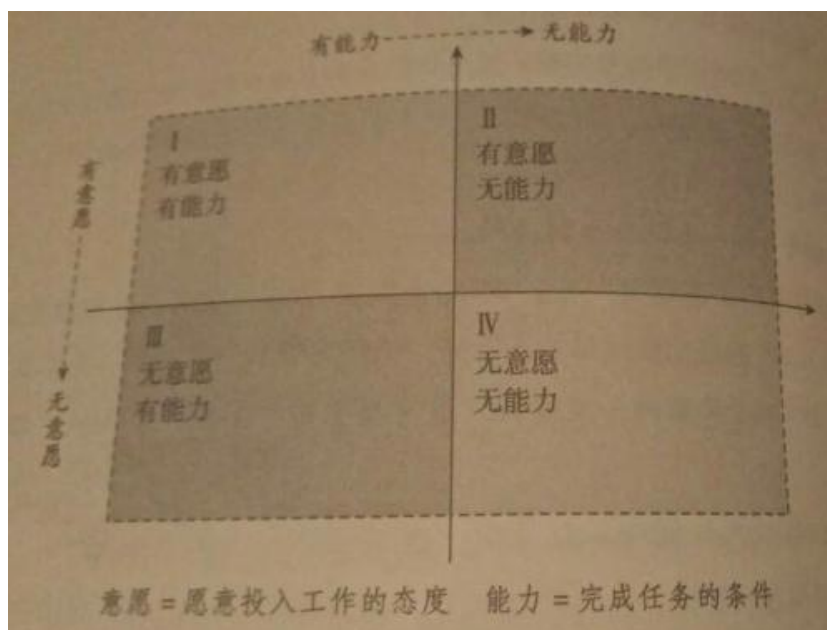
这 3 者之间存在逻辑顺序, 第一要素不是“行动”, 因为 100%的行动并不能保证 100%的成功! 若是方法不得当, 执行的越好离目标越远, 所以方法也不是第一要素!

目标达成的条件叫能力, 愿意投入的态度叫意愿, 成功的第一要素是“意愿”! 因为 100%的意愿一定会催生 100%的方法和 100%的行动!

根据意愿和能力, 我们可以把身边的小伙伴们分为 4 类:







- 1: 有目标达成意愿和能力。
- 2: 有目标达成意愿但没能力。
- 3: 有能力但无达成目标的意愿。
- 4: 既没有能力又无达成目标的意愿。

#### 如何分别对待这 4 类小伙伴呢？

- 1: 对既有意愿又有能力的小伙伴，应尽量授权，将权力赋予给他们，让他们去完成目标。
- 2: 对有意愿没能力的小伙伴，应尽量培养，培训，提升能力。
- 3: 对有能力但无意愿的小伙伴，应尽量激励，让他们产生达成目标的意愿。
- 4: 对既无能力又无意愿小伙伴，可以放弃，至少难堪重用！当然最好是促使他们转变为其他类型的小伙伴。

#### 一般来说，小伙伴们发挥的主动性分为 5 个层次：

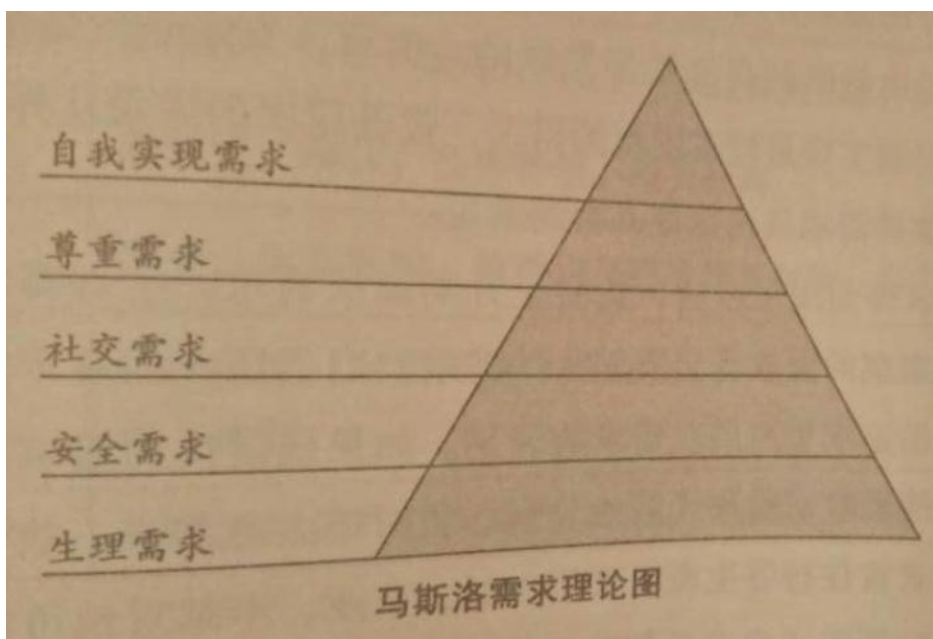
- 1: 等待他人的吩咐(最低层的主动性)
- 2: 询问该做些什么。
- 3: 提出建议，然后就采取相应的行动。
- 4: 采取行动，但立即提出建议。



5: 自己主动行事, 然后定期汇报(最高层次的主动性)。

一个成功的小伙伴应该是一个主动性很强的人! 即使不优秀, 但是有足够的主动性, 依然可以做好不能完全胜任的任务! 因为主动性会使你逐渐提升到合格深知优秀的高度, 而且, 主动性能够提供更多的机会, 促使自身能力的提高和发挥, 不断增强的能力和源源不断的主动性结合让我们的效率提高, 逐步让我们卓然超群~

## 5.2 马斯洛需求模型



心理学家们在研究人的行为时, 深刻剖析了需求, 动机和行为之间的关系。

一般来说, 需求引发动机并带来行为。

著名的马斯洛需求模型核心如下:

- 1) 生理需求: 生存, 衣食住行等行为的满足。
- 2) 安全需求: 保护自己免受生理和心理伤害的需求。
- 3) 社交需求: 对比人的认同, 接纳, 友谊, 人际关系, 团队归属等的追求。
- 4) 尊重需求: 自尊, 自主, 成就, 地位, 权力, 责任, 任何和关注。
- 5) 自我实现需求: 追求个人能力极限的内驱力, 包括成长, 发挥自己的潜能和自我实现。

一个人在每个时期都只有一种需求占据主导地位, 其他需求处于相对从属地位。人



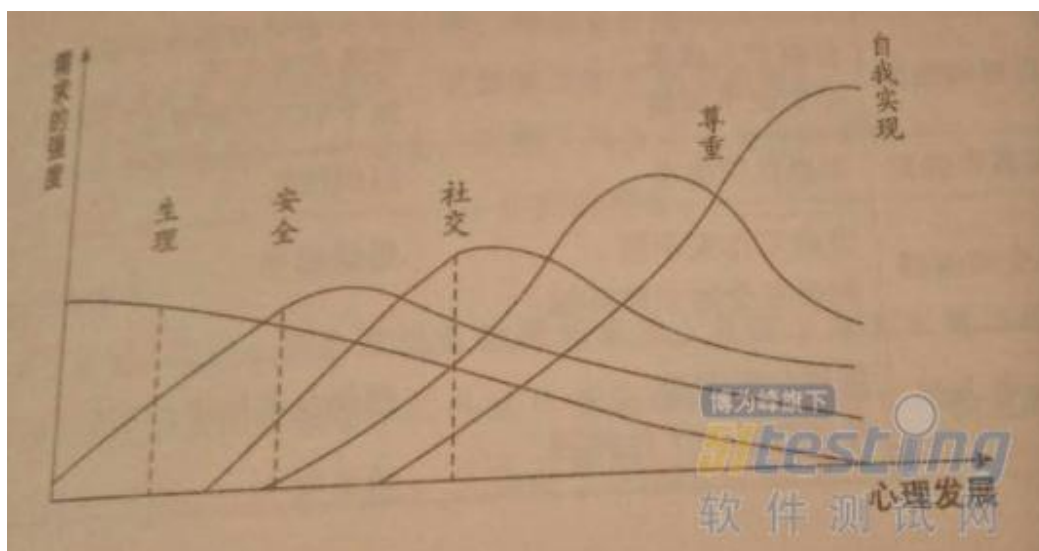
们所处的层次不同，最期望满足的需求也不同。当一个主导需求被满足时，这个需求不再具有很好的激励作用。

(PS:让我想起和之前老大对话，总是感觉被各种吊打！现在想想确实如此，我们的需求差别太大！我关注的重点对人家早已不具有吸引力了~)

### 马斯洛需求模型激励理论的应用：

#### 1: 对个人的应用

|      |                              |
|------|------------------------------|
| 需求层次 | 个人                           |
| 自我实现 | 责任感，与上下级充分沟通，有挑战性的工作，参与高级决策。 |
| 尊重需求 | 成就感，承认，公平待遇，他人崇拜。            |
| 社交需求 | 聚会，生日礼物，团建。                  |
| 安全需求 | 医疗保险，定期体检，安全的工作环境，稳定的工作，休假。  |
| 生理需求 | 高薪，独立的工作空间，班车，不加班，便宜的住房。     |



(PS:嘿嘿~ 我个人目前的主导需求是生理需求，想来测试小伙伴也大同小异吧)

#### 2: 对团队的应用

|      |                     |
|------|---------------------|
| 需求层次 | 组织                  |
| 自我实现 | 给予事业成长机会，鼓励创造，鼓励成就。 |



自尊和地位 公布个人成就赞扬良好表现，经常给予回馈，给予更大工作责任。

归属和社交 团建

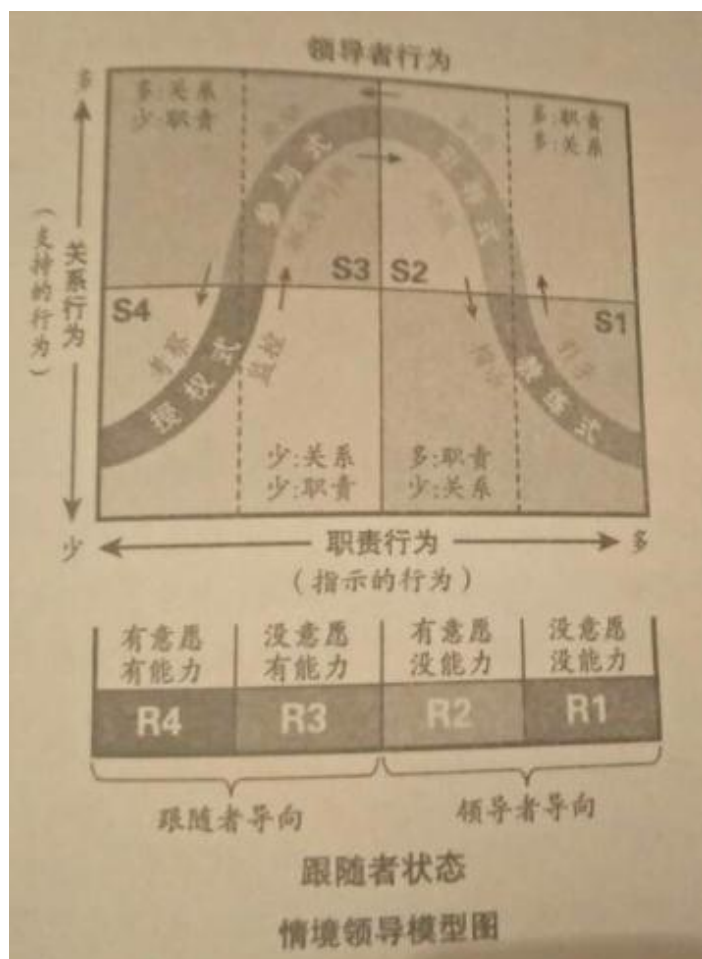
安全和保障 提供福利，提供安全的工作环境，营造工作安全感。

基本生活 提供公平薪资和足够的休息时间，提供舒适的工作环境。

### 5.3 情景领导模型

“情景领导”模型是赫塞博士发明并推广的，其核心思想是：

根据情景的不同及员工准备度的判断，领导者适时调整自己的领导风格，并根据权力基础来适时领导，从而实施有效的管理和领导。



这一管理模型被取得成功并被迅速接收的原因之一是因为它将关注点从领导者的态度改变成了领导者的行为。由于侧重点在于行为方面，情景管理者的任务就从心里疗法转变为了行为训练。关于领导者的争论也从性格转到了行为！



情景领导模型的实施步骤:

#### Step1: 绩效计划

这一步主要侧重了解员工所负责的工作范围。这里主要内容是目标设定, 特别强调对于所要求或所期望达到的绩效水平达成共识。

#### Step2: 发展分析

这一步的目的在于: 就员工在其任务或者所负责人的领域中所处的发展水平达成共识。

可划分为 4 个阶段:

2.1 低竞争力和高参与性。      2.2 始终的竞争力和低参与性。

2.3 高竞争力和不确定的参与性      2.4 高竞争力和高参与性。

#### Step3: 匹配领导模式

作为绩效计划的会谈的一部分而进行时, 匹配领导方式显得最为有效。

管理者必须使对话得以顺利进行, 这样员工和管理者对于所需的指导和支持行为的结合体的理解才能一致。

#### Ste4: 管理方式的实现

这一步根据员工的具体表现而定。管理者需要连续不断地监督并评价员工的绩效: 如果绩效尚可, 那么可以进一步减少指导, 并同时逐渐减少支持; 如果其绩效欠佳, 那么根据员工的发展阶段, 管理者要加强支持和指导。

### 六: 控制好自己的压力

#### 什么是压力?

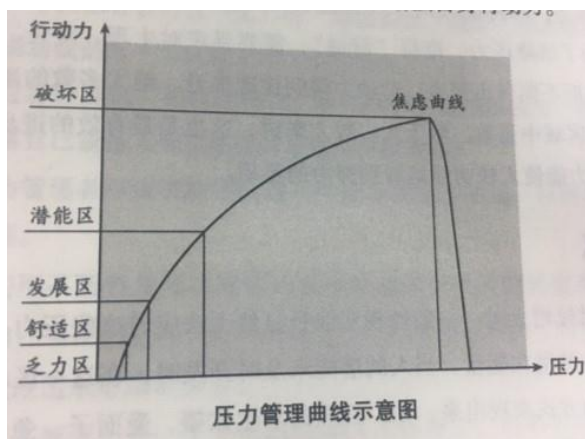
从心理学角度来看, 压力是指个体觉得某种状况超出可应对的能力范围, 而感受到威胁的心理体验。

很多时候, 我们想到压力就会联想到压力负面, 其实, 压力可以产生非常积极的作用, 它可以激发人们的积极性和创造力, 前提是我们必须先熟悉压力然后把控好它!

心理学家们统计了一份有意思的压力管理曲线, 嗯, 值得一看







压力会带来心理上的焦虑，随着压力的变化，心理的焦虑程度也会发生变化(如上图)，心理学家们称为“焦虑曲线”或者压力曲线。

那么问题来了，面对心理重压时，我们该如何释放压力呢？

应对压力常见的方式：



然而，上面这些并不是应对压力的好方法。

正确的姿势应该是下面这样：







除此之外，推荐减压的 2 个方法：

### 1: 凯利魔术方程式

Step1: 问自己可能发生的最坏情况是什么。

Step2: 准备接受最坏的状况。

Step3: 设法改善最坏状态。

在不能迅速摆脱负面情绪对我们的影响时，应用凯利魔术方程式可以帮忙我们用理性战胜负面情绪的影响。

### 2: “3R” 减压原则

即：放松(Relaxation) 减少(Reduction) 重整(Reorientation)

将放松自己，减少压力源和重整期望值三者结合起来，在已有的正面压力，自发压力与过度压力之间寻找一个平衡点。

## 七：高效沟通

### 何谓沟通？

沟通是为了一个设定的目标，将信息，思想和情感在个人或团队中传递，并且达成共同认知的过程。对职业人士来说，沟通分为下向沟通，上向沟通和平行沟通。

1: 下向沟通 上级对下属提供指导，激励，授权，建议，反馈业绩情况，解释政策和程序等信息和感情的交流。

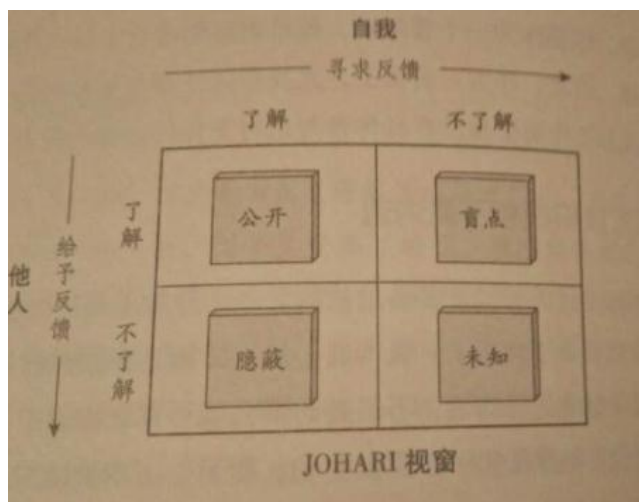
2: 上向沟通 下属对上级提供反馈，建议或进行信息和感情交流。

3: 平行沟通 与同级别的同事进行交流，反馈等。

### 7.1 “JOHARI” 视窗



根据沟通中的信息内容分类，信息可如下图被分为 4 个区域。



**公开区域：**所有信息都是公开的，也就是自己知道这些信息，他人也知道这些信息。这个区域是良好沟通的结果。

**隐蔽区域：**这里是我知而他人不知的区域。由于在沟通中信息不对称，许多自我了解的信息他人不了解，这会造成沟通中的误解和障碍。

**盲点区域：**这是他人知而我不知的区域。就沟通双方来说，这是一个“自我”不了解他人了解的区域，盲点区域和隐蔽区域一样会造成沟通障碍。

**未知区域：**这是沟通双方都不了解的信息内容，未知区域是沟通中最糟糕的情况！

### 我们该如何利用”JOHARI“视窗？

#### 1: 扩大”公开区域“

积极寻求和给予反馈，尽量扩大公开区域。

#### 2: 积极地”寻求反馈“

主动寻求对方给予自己更多的信息。

#### 3: 积极地”给予反馈“

积极给予他人的信息

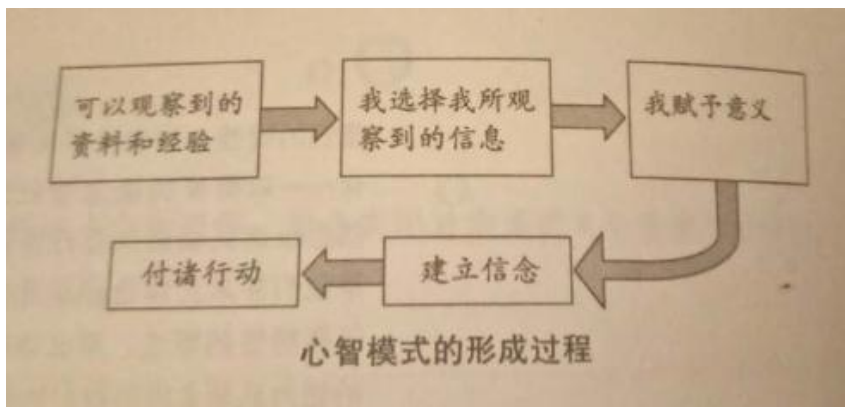
#### 4: 既积极”寻求反馈“又积极”给予反馈”

这种沟通方式可以有效地减少屏蔽，未知和盲点区域，公开区域会越来越大，所以说这是最佳的沟通反馈方式。



## 7.2 沟通反思环

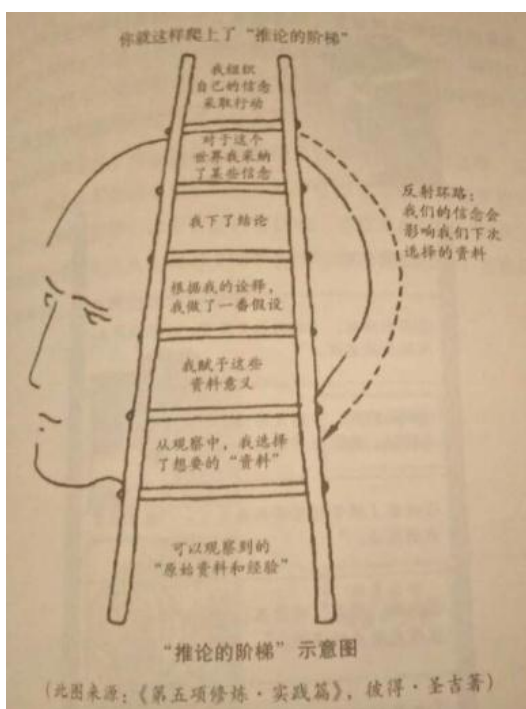
我们的理性心智常常将具体事项概念化---以简单的概念替代许多细节，然后以这些概念来进行推论。但是如果我们并未察觉自己从具体事项跳跃到概括性的概念，那么以概念来推论的能力反而会限制我们的学习。



我们多数的信念是自创的和未经验证的，我们信奉这些信念是因为它们以结论为基础，而这些结论是根据我们的所见和过去的经验推断出来的。跳跃式的推论将假设视为理所当然而不需要再加以验证的定论，常常隐含着我们将顽固持有的，未经验证的信念，有碍学习，所以我们必须持续验证，反思自己的心智模式。

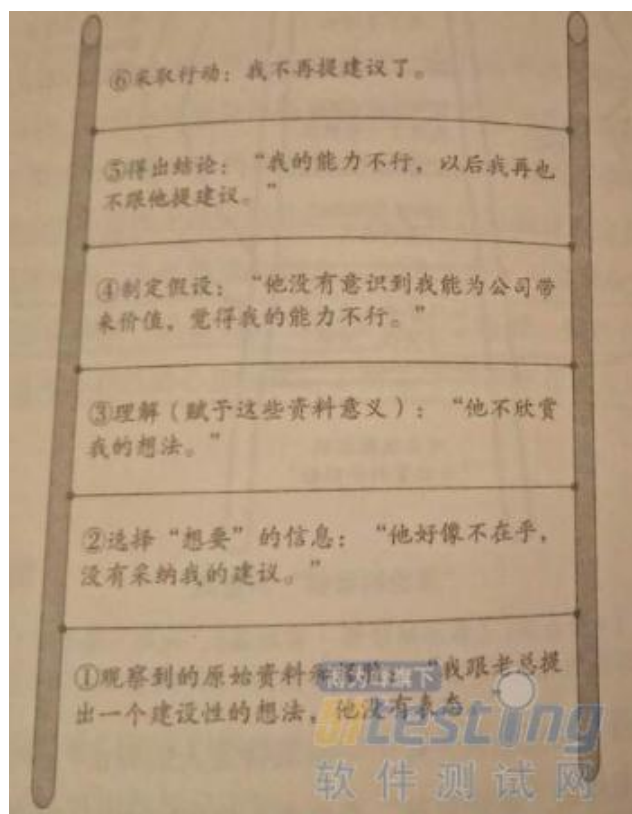
反思验证我的心智模式，前提是摊开，理解我们的心智模式形成过程。

“推论的阶梯”---正是我们摊开自己心智模式有效的透明化工具。



“推论阶梯”使我们的推论过程“透明化”，并可以仔细加以分析。在这样的“阶梯推论”中，我们首先要问自己对周边的事物持有什么样的看法或信念，质问自己某项概括性的看法所依据的“原始材料”是什么，然后问自己：我是否愿意再想想看，这个看法是否不够精确或有误导性？诚实地回答这些问题非常重要，如果答案是不愿意，再继续下去是没多大意义的。

下面这个例子，活生生地说明了，跳跃式思维坏处，恰好也证明：要让他人理解自己的决策结果是非常困难的。



推论的阶梯显示了我们聚集，吸收信息以及得出结论的过程，它帮助我们更好地了解自己复杂的心理过程，了解并仔细思考阶梯模型有助于理解那些没有公开说明的假设，以便于进行有目的，有意义的讨论！

### 7.3 如何处理沟通过程中出现的问题

大多数时候，沟通不畅的因素是下面这些：

- 1: 不会倾听(知觉过滤，鸡尾酒会效应)
- 2: 缺乏换位思考的能力
- 3: 认知风格差异



- 4: 价值观念差异
- 5: 沟通风格差异
- 6: 不恰当的身体语言
- 7: 未考虑沟通的背景(时间, 地点, 场合)

为了避免或解决沟通问题, 下面的技巧还是不错的~

### 7.3.1 阻止不健康的争论

如果沟通过程中情绪过于激动, 分歧偏离太大, 沟通就有可能变成争论!

这时候我们可以采用下面的方式来有效回避沟通不畅引发的问题

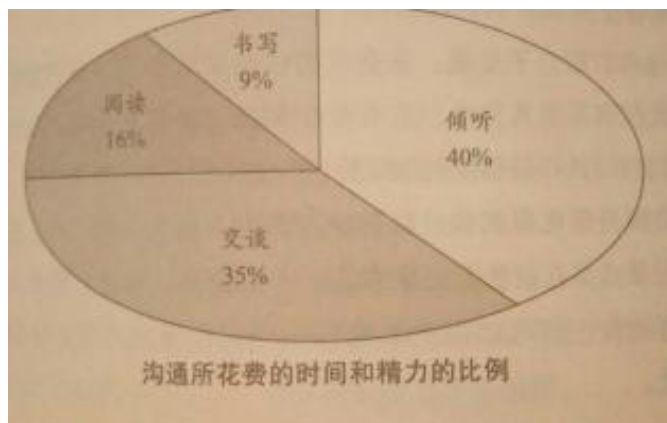
- 1: 休息片刻。短暂的休息对停止个人间的争执非常必要。给参会者一个机会以他们自己的方式专业地表达自己的想法。
- 2: 改变参会者的态度。升级过快的分歧是由于缺乏双赢的视角, 通常都是以获胜的心理为特征的, 或是输赢的心理, 这样的心态常常导致争执升级并攻击对方。
- 3: 发泄情绪。如果发现某些小伙伴处于沮丧或愤怒状态, 那么让他表达他的沮丧或愤怒, 并把这种情绪释放出来。
- 4: 与分歧方秘密会谈。在封闭环境下和分歧方私下进行快速且善解人意的交谈, 让其冷静下来, 这是将争论双方拉回到共同基础和动机上的一种极为有效的手段。
- 5: 借用幽默来缓解压力。

### 7.3.2 避免某个人或小集体控制

- 1: 实施公正原则。确保每个参会者都有机会表达自己的见解。
- 2: 劝请每个人都参与。
- 3: 进行提醒。可以语言提醒, 也可以使用肢体语言。

### 7.3.3 排除倾听障碍





沟通是双向的，更多时候我们需要倾听也就是接受对方传递的信息。

最有价值的人不一定是最能说的人，我们两只耳朵一张嘴巴也恰恰说明进化过程中我们更多是要聆听，善于聆听才是成熟的人最基本的素养。

#### 7.3.4 如何应对喜欢打断他人的人

不管是否承认，生活工作总会或多或少遇到在沟通过程中喜欢打断他人的小伙伴。

一个成功的会议组织者或协调人需要让炭火始终围绕着共同关心的问题，而又不能使人为难或侮辱任何人。

对这类喜欢打算别人的小伙伴我们可以采用下面 2 中策略：

##### 1：单独和打断分子交流。

在会议前或会议后私下交谈，而不是在会上和他们对抗。尝试理解他们的动机并获得他们的支持。

##### 2：让打断分子忙碌起来。

给喜欢打断他人的小伙伴一项任务让他们忙碌起来，比如：让他们做会议记录或在白板上记录意见。

对不同的打断行为，我们具体采用下面的方式

打断分子的行为

协调人应对的反应

1) 敌对的：“这样永远没用”  
法？

其他人对这个想法有什么看

或”这是我们能采取的最好

你可能对，但是我们需要回顾





下事实的方法吗？”

2) 叽里呱啦的

能总结下你的主要思想吗？

经常脱口而出很多想法

很不错的提议，现在我们听听其他同学的意见或想法

3) 打断分子

xx 请等一分钟，我们要坚持基本原则

在别人还没说完就开始说

则，让 y 先说完

4) 沉默的干扰者

请等一分钟，让 xx(干扰者)先说

玩手机，干其他事儿

或者在会议间隙单独谈话

东张西望，小声私聊

### 7.3.5 避免小团体思想

小团体思想可能出现在过于内聚的团队中，在这种团队中讨论被中止并非是因为某个人控制了讨论，而是因为整个团队都回避健康的争论。

下面几种因素通常是导致团队思考的主要因素

1: 不犯错心理。

有些团队因为幸运或者自认为可以不犯错，因此就没有动力去进行健康的多方位的思考。

2: 正义理由。

有时候，一个团队会理所当然地认为自己做的是件”好事“，因此他们不争论他们做的是对错。

3: 集体合理化。

有时候团队偏偏忽视和既定状况相反的事实，而且他们总是自欺欺人地认为已经认可的状况是正确的。

4: 自我压抑。

团队成员为了避免产生不受欢迎的状况或是不愿意被拒绝或惩罚的危险，就有可能什么也不说。

5: 一致通过的错觉。



当团队成员压抑自己，那么下一个误区就是难以避免的一致通过的错觉。没人不同意，那么我们都一致通过。

#### 6: 对持异议者施加压力。

如果所有压力都施加到全体成员上，而某个人还是把不同意见说出来，那么其他的团队成员就会对这个人施加直接的压力(社会惩罚)。

我们可以通过尊重主动的讨论，健康的反对以及广泛的想法和视角来与小团体思想做斗争。除此之外，我们还可以采用下面的方式。

对议程上每个不同的主题，分配不同的人担任吹毛求疵的角色。

分成小组，然后让每个小组都提出建议和意见。

做出决策后，举行第二次会议并强烈鼓励成员表达的保留意见。

好啦，我们对很多问题阐述是原因，实施步骤和应对策略，小伙伴们可以在工作中多多尝试或者和自己的风格多对比然后加以矫正。学而不思则罔，思而不学则殆。孔先生很早就告诉我们了大方向，希望有机会看到此文的小伙伴们有些许收获~

### 《51 测试天地》（五十一）上篇 精彩预览

- 测试女巫自动化进化论之"由猴子进化到原始人篇"
- Python 爬虫初探
- 从一个常见的面试题目说起
- TestNG 关键字 dependsOnMethods 踩坑记
- 掌握测试驱动开发的 3 个关键因素
- 如何在 Selenium 项目中使用 GeckoDriver
- 当我们谈隐私安全时，我们在谈些什么

马上阅读

