Java 第一天

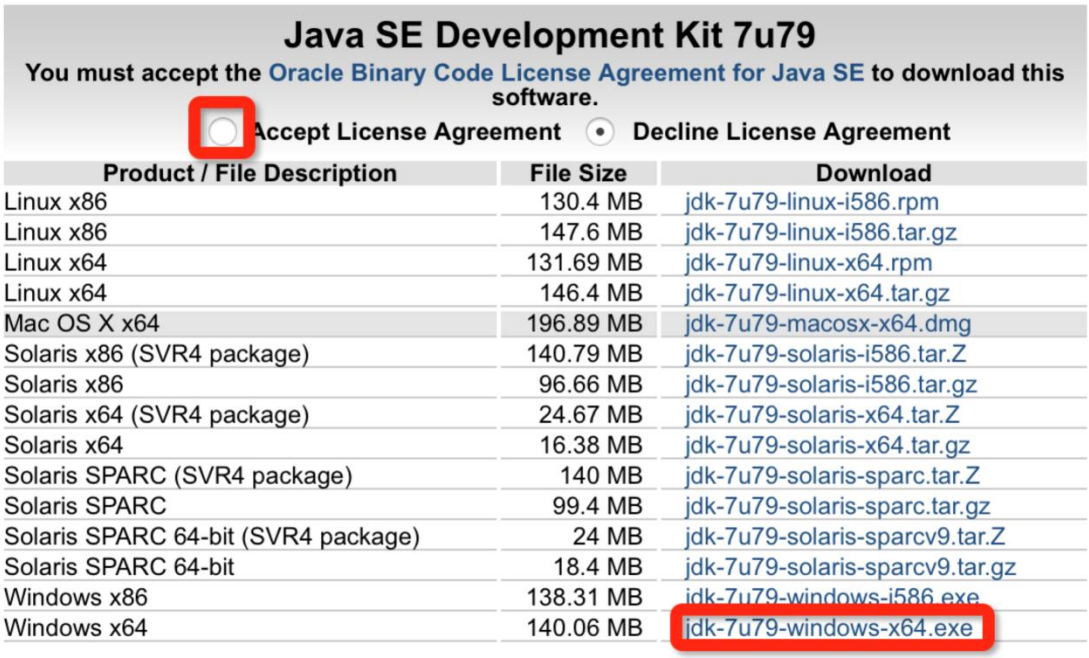
jdk 安装及配置，eclipse 配置

1.1 JDK 下载

Jdk7 的下载地址：

http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html

比如是在 windows 环境下，按下图下载：



Java SE Development Kit 7u79

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

○ Accept License Agreement ● Decline License Agreement

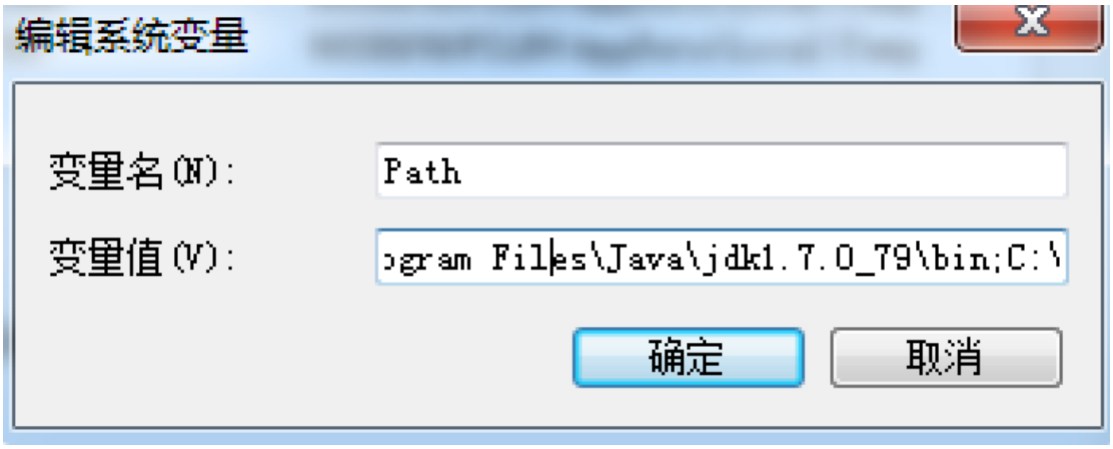
Product / File Description	File Size	Download
Linux x86	130.4 MB	jdk-7u79-linux-i586.rpm
Linux x86	147.6 MB	jdk-7u79-linux-i586.tar.gz
Linux x64	131.69 MB	jdk-7u79-linux-x64.rpm
Linux x64	146.4 MB	jdk-7u79-linux-x64.tar.gz
Mac OS X x64	196.89 MB	jdk-7u79-macosx-x64.dmg
Solaris x86 (SVR4 package)	140.79 MB	jdk-7u79-solaris-i586.tar.Z
Solaris x86	96.66 MB	jdk-7u79-solaris-i586.tar.gz
Solaris x64 (SVR4 package)	24.67 MB	jdk-7u79-solaris-x64.tar.Z
Solaris x64	16.38 MB	jdk-7u79-solaris-x64.tar.gz
Solaris SPARC (SVR4 package)	140 MB	jdk-7u79-solaris-sparc.tar.Z
Solaris SPARC	99.4 MB	jdk-7u79-solaris-sparc.tar.gz
Solaris SPARC 64-bit (SVR4 package)	24 MB	jdk-7u79-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	18.4 MB	jdk-7u79-solaris-sparcv9.tar.gz
Windows x86	138.31 MB	jdk-7u79-windows-i586.exe
Windows x64	140.06 MB	jdk-7u79-windows-x64.exe

1.2 JDK 安装

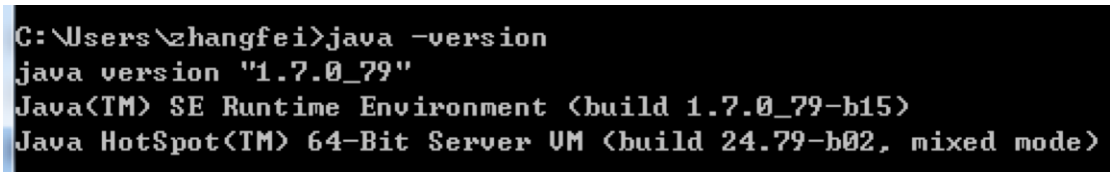
下载完成后，双击进行安装，安装中间过程中，只需要注意一下安装路径，默认是 C:\Program Files\Java，在安装过程中，一般用默认路径即可，如果想改变，注意选择好自己想要存放的路径即可。

1.3 JDK 配置

安装完成后，就是 jdk 的配置了，将 JDK 的安装路径下的 bin 文件加到 path 中即可，比如：



检查 JDK 是否配置成功：



1.4 eclipse 下载安装

下载地址：http://www.eclipse.org/downloads/，选择一种版本，比如 mars 即可。

Eclipse 的安装比较简单，将下载下来的压缩包解压，双击 eclipse.exe 即可打开使用了。

1.5 eclipse 配置

- 关于 workspace

在第一次打开 eclipse 时，会要你选择一个 workspace，当然，已经有一个默认的路径帮你选择好了，但一般我们会把工程放在我们能快速找到的地方，所以，在一个你准备放置代码的文件夹下面，事先新建一个文件夹”workspace”，然后在打开 eclilpse 时，选择你新建好的 worksapce 路径即可。

java 代码规范

对于规范，每个公司不一样，对于我们测试人员而言，不需要讲究太多，列举一下：

- 包名全小写，全由字母组成，不要带有数字，下划线之类的
  比如：com.demo.baixiaosheng
- 类名首字母大写，驼峰式命名全由字母组成，不要带有数字，下划线之类的
  比如：public class UserInfo
- 常量要加上 final 修饰符，常量及变量名最好要有意义。非静态类常量与变量名也是驼峰式命名，最好全由字母组成，不要带有数字，下划线之类
  比如：public final String username
- 静态常量与变量最好全由大写字母组成，非驼峰式，中间用下载线隔开
  比如：public static final String USER_NAME
- 方法名最好全由字母组成，驼峰式命名，首字母小写，不要带有数字，下划线之类的
  比如：public void addUser()
- 方法命名要有一定的意义，比如增加用户，方法名为 addUser()
- 代码要养在加注释的习惯
- java 的大小写是敏感的

这是一些常用的规范，其它的规范大家可以参考各自公司的要求。

著名的"Hello World!"

```
package com.demo.baixiaosheng;

public class HelloWorld {

    public String helloWorld ;

    public String getHelloWorld() {
        return helloWorld;
    }
```

```
    public void setHelloWorld(String helloWorld) {
        this.helloWorld = helloWorld;
    }

    public HelloWorld(String helloWorld) {
        this.helloWorld = helloWorld;
    }

    public HelloWorld() {}

    public void outputHelloWorld(){
        System.out.println(helloWorld);
    }

    public static void main(String[] args) {
        String helloWorld = "Hello World!";
        HelloWorld h = new HelloWorld();
        h.setHelloWorld(helloWorld);
        h.outputHelloWorld();
        HelloWorld w = new HelloWorld(helloWorld);
        w.outputHelloWorld();
    }

}
```

Java 第二天

for while if 的使用规则

1. for 与 while 都是循环，其语法规则分别为：

```
for(initialization; Boolean_expression; update)
{
   //Statements
}

for(declaration : expression)
{
   //Statements
}

while(Boolean_expression)
{
   //Statements
}
```

2. 分别举例：如果有一个数组，我们分别的用这三种循环体来遍历：

```
package com.demo.baixiaosheng;

public class LoopTest {

   private int[] testArray = new int[]{2,4,3,5,1};

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

```
    public void testForLoopEnhance(){
        for (int i : testArray) {
            System.out.println(i);
        }
    }

    public void testWhileLoop(){
        int index = 0;
        while(index<testArray.length){
            System.out.println(testArray[index]);
            index++;
        }
    }

    public static void main(String[] args) {
        LoopTest t = new LoopTest();
        t.testForLoop();
        t.testForLoopEnhance();
        t.testWhileLoop();
    }

}
```

3. 既然提到了循环，有两个关键字就不得不提了: break continue
   1) break:退出当前循环体
   2) continue:退出当次循环并继续下一次循环，即当次循环体中 continue 语句下面的代码不执行，并再继续下一次循环。
4. if 是表示判断的一个关键字。

   第 3 第 4 综合举例：

```
    private int[] testArray = new int[]{2,4,3,5,1};

    public void testForLoop(){
        for (int i = 0; i < testArray.length; i++) {
```

```
            if(testArray[i]==3){
                continue;
            }
            System.out.println(testArray[i]);//最后输出结果为2451
        }
    }

    public void testForLoopEnhance(){
        for (int i : testArray) {
            System.out.println(i);//最后输出结果为243
            if(i==3){
                break;
            }
        }
    }
```

静态变量的使用

1. 概念

静态变量，涉及到的关键字是 static，通俗点说我们可以理解为全局变量，任何对象都可以对其进行修改。但如果加了 final 关键字，就表示是一个静态常量，也就是全局常量，其它对象只能随意拿来用，但不能修改其值（前提是修饰符是 public）。代码如下：

```
package com.demo.baixiaosheng;

public class StaticDemo {

    public static String USER_NAME = "baixiaosheng";

}
```

```
package com.demo.baixiaosheng;
```

```
public class StaticDemoTest {

    public static void main(String[] args) {
        System.out.println(StaticDemo.USER_NAME);
        StaticDemo.USER_NAME = "ceshibaixiaosheng";
        System.out.println(StaticDemo.USER_NAME);
    }

}
```

static 这个关键字，还可用于静态块.

```
package com.demo.baixiaosheng;

public class StaticDemoTest {

    static{
        System.out.println("Hello world!");
    }

    public static void main(String[] args) {
        System.out.println(StaticDemo.USER_NAME);
        StaticDemo.USER_NAME = "ceshibaixiaosheng";
        System.out.println(StaticDemo.USER_NAME);
    }

}
```

2. 说明

静态块可以被称为非静态类中的构造器！即在用到该类时，会被第一时间执行到，且只执行一次，一般用于配置文件的加载。大家可以去自行查询一下 java 的变量加载机制。

String 字符串的基本方法

1. 字符串的常用 API 有：

```
package com.demo.baixiaosheng;

public class StringDemo {

    private String demoString = "ceshibaixiaosheng";

    public void testString(){
        String dsg = demoString.concat(" very good!");//字符串相加，也可用+来表示
        System.out.println(dsg);//输出ceshibaixiaosheng very good!
        int len = demoString.length();//字符串的长度
        System.out.println(len);//输出17
        boolean eq = "baixiaosheng".equals(demoString);//比较两个字符串是否相等
        System.out.println(eq);//输出false
        String sub = demoString.substring(5, 8);//取子字符串,从第5个字符开始，到第8个字符，但不包含第8个字符
        System.out.println(sub);//输出bai
        String subString = demoString.substring(5);//取子字符串，从第5个字符开始一直到字符串尾
        System.out.println(subString);//输出baixiaosheng
        boolean sw = demoString.startsWith("ceshi");//判断是否以某个字符串开头
        System.out.println(sw);//输出true
        boolean ew = demoString.endsWith("baixiaosheng");//判断是否以某个字符串结尾
        System.out.println(ew);//输出true
        int subIndex = demoString.indexOf("bai");//找出子字符串在字符串中第一次出现的index，如果找不到则返回-1
```

```
        System.out.println(subIndex);//输出5
        int lastIndex = demoString.lastIndexOf("e");//找出子字符串在字符串中最后一次出现的index,如果找不到则返回-1
        System.out.println(lastIndex);//输出14
        System.out.println(demoString.toUpperCase());//字符串中的字母全变成大写,CESHIBAIXIAOSHENG
        System.out.println(demoString.toLowerCase());//字符串中的字母全变成小写,ceshibaixiaosheng
        System.out.println(" baixiaosheng ".trim());//将字符串首尾的空格去掉,baixiaosheng
        String subReplace = demoString.replace("ceshi", "");//将字符串中的某段子字符串替换成新的子字符串
        System.out.println(subReplace);//输出baixiaosheng
        String subReplaceF = demoString.replaceFirst("e", "");//将字符串中的某段第一次出现的子字符串替换成新的子字符串,支持正则
        System.out.println(subReplaceF);//输出cshibaixiaosheng
        String subReplaceA = demoString.replaceAll("e", "");//将字符串中的所有出现的子字符串替换成新的子字符串,支持正则
        System.out.println(subReplaceA);//输出cshibaixiaoshng
    }

    public static void main(String[] args) {
        StringDemo s = new StringDemo();
        s.testString();
    }

}
```

基本数据类型与复合数据类型的关系

1. java 有 8 大基本数据类型及其对应的复合数据类型为：

byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean

2. 需要注意的是：

- 基本数据类型的初使化值不能为 null
- 复合数据类型的初使化值允许为 null
- 基本数据类型参数传递时为按值传递
- 复合数据类型参数传递时为地址传递
- 基本数据类型对应的复合数据类型的值一旦赋值后，都被加了 final 关键字，是不允许修改的，在重新赋值时，会产生一个新的数据对象。
- String 也是复合数据类型，其也是地址传递，也是一旦赋值都被加了 final 关键字，不允许修改，在重新赋值时，会产生一个新的字符串对象。

```
package com.demo.baixiaosheng;

public class DataTypeDemo {

    public Byte b = 10;

    public void test(Byte b){
        b = 19;
    }

    public static void main(String[] args) {
        DataTypeDemo d = new DataTypeDemo();
        d.test(d.b);
```

```
        System.out.println(d.b);
    }

}
```

字符串转数字

1. 转换很简单，有 API 提供，但我们先看下面这段代码：

```
package com.demo.baixiaosheng;

public class StringToInt {

    public Integer changeType(String s){
        return Integer.valueOf(s);
    }

    public static void main(String[] args) {
        StringToInt t = new StringToInt();
        System.out.println(t.changeType("s"));
    }
}
```

运行是有报错的，因为显然 s 是不能转为数字的，我们可以修改下代码：

```
package com.demo.baixiaosheng;

public class StringToInt {

    public Integer changeType(String s){
        try{
            return Integer.valueOf(s);
        }catch(Exception e){
            return null;
        }
    }
```

```
    public static void main(String[] args) {
        StringToInt t = new StringToInt();
        System.out.println(t.changeType("s"));
    }
}
```

这样就不会报错啦，因为 Integer 是允许值为 null 的，上面我们刚刚讲过！

数字的加减乘除

1. 代码示例

```
package com.demo.baixiaosheng;

public class CalculateTest {

    private double b = 9.50;

    private double c = 7.80;

    public double testAdd(){
        return b+c;
    }

    public int testDel(){
        return (int) (b-c);//强制取整，将小数点后面的全抹掉，非四舍五入
    }

    public Long testDelRound(){
        return Math.round(b-c);//取四舍五入值
    }

    public double testMul(){
        return b*c;
```

```
    }

    public float testDiv(){
        return (float)(b/c);//除法的正确写法
    }

    public double testDivDouble(){
        return (double)9/2;//除法的正确写法
    }

    public double testDivDou(){
        return (double)(9/2);//整数除以整数，结果会被自动的处理成为整数，请注意
    }

    public static void main(String[] args) {
        CalculateTest t = new CalculateTest();
        System.out.println(t.testAdd());
        System.out.println(t.testDel());
        System.out.println(t.testDelRound());
        System.out.println(t.testMul());
        System.out.println(t.testDiv());
        System.out.println(t.testDivDouble());
        System.out.println(t.testDivDou());
    }

}
```

Java 第三天

数组基本使用

1. 数组特点

数组，用于存储相同类型的元素的一个固定大小的连续集合。数组一旦定义好就不可以修改长度，即数组是定长的，灵活性较差。但是数组在初始化时，会占用连续的内存空间，所以在访问数组中的元素时可以直接根据数组在内存中的起始位置以及下标来计算元素的位置，因此数组的访问速度很高。

2. 数组声明

a) int[] i;

b) int i[];

以上两种方式都是合法的，但推荐 a 方式。b 方式的风格来自于 c/c++, java 容纳了该风格。

3. 数组初始化

数组初始化有三种方式：

- int[] a = new int[3];//事先声明数组长度，但里面的元素值会设置，会自动初始化
- int[] a = new int[]{2,4,6};//初始化数组元素值
- int[] a = {2,4,6,3};//初始化数组元素值

4. 数组元素的增删改查

1) 数组是定长的，所以没有增与删，请看如下代码：

```
int[] a = {2,4,6};
for (int i = 0; i < a.length; i++) {
    int t = a[i];//取值
    System.out.println(t);
    a[i] = t+1;//修改值
    System.out.println(a[i]);
```

```
}
```

5. 多维数组

声明

1) int[][] a = {{1,2},{5,6},{9,10}};
2) int[][] b = new int[3][];

以上是两个二维数组的声明方式，其实有点不好理解，但是，如果你把{1,2}看成是一维数组中的一个元素，这样来理解，就方便些了。同时，在 2 中，同样的可以看成定义了一个长度为 3 的一维数组，在声明时，先声明长度为 3，至于里面的元素，是一个没有声明的一维数组，在往一维数组里添加一维数组时，自然会声明好需要添加的一维数组，即先声明最外层的一维数组，里层的一维数组，在添加时，会声明，如果不声明，编译无法通过。

```
int[][] b = new int[3][];
for (int i = 0; i < b.length; i++) {
    int[] c = new int[4];
    b[i] = c;
}
System.out.println(b[1][3]);
```

list 基本使用

1. 集合概念

Collection 接口，它是在集合层次结构的顶层。List 接口扩展了集合，并声明存储元素的序列集合的行为，也就是 List 接口里定义了一堆方法，让实现类去实现，我们接下来将主要演示 ArrayList 这个实现类的一些常用方法。List 的长度是不固定的，可以任意添加，删除，且 ArrayList 里面的元素值是可以重复的。

2. 代码示例

- List 基本操作

```
public void listAction(){
    List<String> list = new ArrayList<String>();//<>这里面只能接复合数据类型,List<int>就会报编译错误
    list.add("a");//增加一个元素
    int len = list.size();//获取list的大小，即元素个数
```

```
        System.out.println(len);//输出为1
        String s = list.get(0);//获取list中第一个值，是从0开始的，如果list长度为1而get(1)时，会报越界的异常
        System.out.println(s);//输出为a
        list.set(0, "b");//替换index为0的元素值
        System.out.println(list);//输出为[b]
        boolean c = list.contains("a");//判断list中是否包含有元素a
        System.out.println(c);//输出false
        List<String> l1 = new ArrayList<String>();
        l1.add("c");
       list.addAll(l1);//向list里面添加一个l1,可以理解为两个数组取并集，并将并集的值赋给list
        System.out.println(list);//输出[b,c]
        list.retainAll(l1);//取两个数组的交集，并将交集的值赋给list
        System.out.println(list);//输出[c]
        list.add("d");
        list.add("e");
        List<String> d = list.subList(0, 2);//从list中取一个子list，示例中是从0开始，到index为2，但不包含2
        System.out.println(d);//输出[c, d]
    }
```

- List 循环

```
    public void loopList(){
        List<Integer> list = new ArrayList<Integer>();
        list.add(3);
        list.add(1);
        list.add(2);
        for (int i = 0; i < list.size(); i++) {
            System.out.println(list.get(i));
        }
        for (Integer i : list) {
```

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

- List 删除

```
    public void listRemove(){
        List<Integer> list = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(1);
        /**
         * remove有两种形式，一种是按index删除，一种是按list中的值删除。
         * 按index删除时，入参是int类型，按值删除时，入参是复合数据类型
         */
        Integer e = 1;//事先定义好要删除的值对象
        /**
         * Removes the first occurrence of the specified element from this list
         * api说明中强调了，在remove(Object o)时，是删除第一次出现的这个值。
         */
        list.remove(e);
        System.out.println(list);//输出为[2, 1]
        int index = 0;//定义要删除的index,也可以不事先定义，直接remove(1)，这个1会被默认为是int型的
        list.remove(index);
        System.out.println(list);//输出[1]
        list.add(2);
        list.add(2);
        list.add(2);
        List<Integer> l1 = new ArrayList<Integer>();
        l1.add(2);
        /**
```

```
        * remove是单个remove,如果我想把list中的元素2全部一次性的remove掉，就要用removeAll了
        * removeAll会把list中包含了l1元素的元素全删除
        * 也可以理解为是取list与l1的差集，并将差集的值赋给list
        */
        list.removeAll(l1);
        System.out.println(list);//输出为[1]
    }

    public void loopRemoveList(){
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("a");
        list.add("a");
        for (int i = 0; i<list.size(); i++) {
            if(list.get(i).equals("a")){
                list.remove(list.get(i));
            }
        }
        System.out.println(list);//输出为[b, a]
    }
```

loopRemoveList 方面中，在循环删除时，最后发现其返回值中仍然有 a，并没有删除完全，这里因为在循环中用了 list.size()来做为循环退出判断条件，而 list 在 remove 掉元素后，其 list.size()的值就会发生变化，导致了数组没有循环完全就退出了。正确的循环删除应该引入 iterator,代码如下：

```
    public void iteratorRemoveList(){
        List<String> list = new ArrayList<String>();
        list.add("a");
        list.add("b");
        list.add("a");
        list.add("a");
        Iterator<String> it = list.iterator();
```

```
        while(it.hasNext()){
            String s = it.next();
            if(s.equals("a")){
                it.remove();
            }
        }
        System.out.println(list);//输出为[b]
    }
```

- 迭代器（Iterator）

迭代器是一种设计模式，它是一个对象，它可以遍历并选择序列中的对象，而开发人员不需要了解该序列的底层结构。迭代器通常被称为"轻量级"对象，因为创建它的代价小。

Java中的Iterator功能比较简单，并且只能单向移动：

(1) 使用方法iterator()要求容器返回一个Iterator。第一次调用Iterator的next()方法时，它返回序列的第一个元素。注意：iterator()方法是java.lang.Iterable接口,被Collection继承。

(2) 使用next()获得序列中的下一个元素。

(3) 使用hasNext()检查序列中是否还有元素。

(4) 使用remove()将迭代器新返回的元素删除。

Iterator 是 Java 迭代器最简单的实现，为 List 设计的 ListIterator 具有更多的功能，它可以从两个方向遍历 List，也可以从 List 中插入和删除元素。

- 数组转 List

```
    public void arrayToArrayList(){
        int[] i = new int[]{1,2,3};
        List<int[]> t = Arrays.asList(i);
    }
```

以上代码是数组转 list,但是转了后，我们发现数据结构是 List<int[]>，这不是我们预期的结果。这是因为 list 中只能接复合数据类型，而 int 是基本数据类型，所以，转换不了，正确的应该是：

```
    public void arrayToArrayList(){
        Integer[] i = new Integer[]{1,2,3};
        List<Integer> t = Arrays.asList(i);
        System.out.println(t);//输出为[1, 2, 3]
```

```
}
```

以上代码转换正确，且输出正确。但是我们读了Arrays.asList的API说明文档，发现其返回值是：Returns a fixed-size list backed by the specified array. 意思是返回一个固定长度的特殊的数组，也就是说Arrays.asList()的返回值还不是一个真正的list，只是一个特殊的数组，于是，要经过再一次的转换：

```
public void arrayToArrayList(){
    Integer[] i = new Integer[]{1,2,3};
    List<Integer> t = Arrays.asList(i);
    List<Integer> list = new ArrayList<>(t);//将数组转换为ArrayList的正确方式
    System.out.println(list);//输出为[1, 2, 3]
}
```

以上的代码，才是数组转ArrayList的正确方式，当然，数组转ArrayList还可以用循环的方式往ArrayList里面添加，都可以，选择一种自己合适的方式即可。

- 清除List

```
public void clearList(){
    List<String> list = null;
    boolean e = list.isEmpty();
    System.out.println(e);
}
```

以上的代码，list等于null，需要注意的是，当list等于null时，调用任何方法，都会报空指针异常。再看下面的：

```
public void clearList(){
    List<String> list = new ArrayList<String>();
    list.add("a");
    boolean e = list.isEmpty();//判断list里面是否存在元素
    System.out.println(e);//输出false
    list.clear();//清除list里的所有元素
    boolean m = list.isEmpty();
    System.out.println(m);//输出true
}
```

- List排序

简单排序：

```
public void sortList(){
    List<Integer> list = new ArrayList<Integer>();
    list.add(3);
    list.add(1);
    list.add(2);
    Collections.sort(list);//集合排序，升序排序
    System.out.println(list);//输出[1, 2, 3]
}
```

复杂排序：

```
List<List<Integer>> list = new ArrayList<List<Integer>>();
```

对于上面这种复杂数据类型的list，如果用Collections.sort(list);会报编译错误。假如我们想根据List里面的List的第一个值的大小来排序，代码如下：

```
public void sortList(){
    List<List<Integer>> list = new ArrayList<List<Integer>>();
    List<Integer> l1 = new ArrayList<Integer>();
    l1.add(3);
    List<Integer> l2 = new ArrayList<Integer>();
    l2.add(1);
    l2.add(3);
    List<Integer> l3 = new ArrayList<Integer>();
    l3.add(2);
    list.add(l1);
    list.add(l2);
    list.add(l3);
    Collections.sort(list, new Comparator<List<Integer>>(){
        @Override
        public int compare(List<Integer> o1, List<Integer> o2) {
            if(o1.get(0)>o2.get(0)){//按List<Integer>的第一个值进行降序排
序
                return -1;
            }else if(o1.get(0)<o2.get(0)){
                return 1;
            }
```

```
                return 0;
            }
        });
        System.out.println(list);//输出[[3], [2], [1, 3]]
    }
```

map 基本使用

1. Map 概念

Map 集合类用于存储元素对（称作“键”和“值”），其中每个键映射到一个值。Map 接口也是定义了一堆方法，用于各实现类去操作 Map 里的键值对。我们将演 HashMap 这个实现类，需特别注意的是 HashMap 是无序排列的。

2. 代码示例

- Map 基础操作

```
public void mapAction(){
    Map<String, String> map = new HashMap<String, String>();
    map.put("a", "this is a");//往map里添加一对键值对
    int len = map.size();//一共有多少对键值对
    System.out.println(len);//输出1
    String a = map.get("a");//获取key为a的值
    System.out.println(a);//输出this is a
    System.out.println(map.get("b"));//获取一个不存在的key时，返回值为null,并不会抛出异常
    map.put("a", "this is the second a");//添加一个已经存在的key,后面添加的value会覆盖前面已存在的value
    System.out.println(map.get("a"));//输出this is the second a
    boolean c = map.containsKey("a");//判断map中是否存在key a
    System.out.println(c);//输出true
    boolean v = map.containsValue("this is the second a");//判断map中是否存在给定的value
    System.out.println(v);//输出true
    Map<String, String> m1 = new HashMap<String, String>();
    m1.put("a", "this is the third a");
```

```
        m1.put("b", "this is b");
        map.putAll(m1);//取map与m1的并集，如果m1中有已经在map中存在的key,则m1中的value会覆盖map中的存在的key的value,并集的值会赋值给map
        System.out.println(map);//输出{b=this is b, a=this is the third a}
    }
```

- Map 循环

```
    public void loopMap(){
        Map<String, String> map = new HashMap<String, String>();
        map.put("a", "this is a");
        map.put("b", "this is b");
        /**
         * set可理解成为一种特殊的List,只是里面的元素对象是不允许重复的。keySet是将key都放到一个集合里面去
         * HashMap是无序排列的。keySet后，set对象也是无序的
         */
        Set<String> set = map.keySet();
        for (String s : set) {
            System.out.println(map.get(s));//循环输出map的value值
        }
        Collection<String> v = map.values();//将map的所有value值转换为一个集合类
        for (String s : v) {
            System.out.println(s);//循环输出map的value值
        }
    }
```

- Map 键值对删除

```
    public void mapRemove(){
        Map<String, String> map = new HashMap<String, String>();
        map.put("a", "this is a");
        map.put("b", "this is b");
        map.remove("a");//根据key来删除map里的键值对
        System.out.println(map);//输出{b=this is b}
```

```
}
```

循环删除：

```
public void loopRemoveMap(){
    Map<Integer, String> map = new HashMap<Integer, String>();
    map.put(1, "this is a");
    map.put(2, "this is b");
    /**
     *set虽然可以看作一个特殊的list,但是set没有像list一个通过get取值的方式
     *要想取值，可以将set转为List，或者转为iterator
     */
    Set<Integer> set = map.keySet();
    List<Integer> list = new ArrayList<Integer>(set);//set转为list
    for (int i = 0; i < list.size(); i++) {
        map.remove(list.get(i));
    }
    System.out.println(map);//输出{}
}
```

- Map 清空键值对

个人认为循环删除键值对的意义不大，使用的少，循环删除后，相当于清空了，还不如用以下的方式来清空 Map：

```
public void clearMap(){
    Map<Integer, String> map = new HashMap<Integer, String>();
    map.put(1, "this is a");
    map.put(2, "this is b");
    boolean e = map.isEmpty();//判断map里是否存在键值对
    System.out.println(e);//输出false
    map.clear();//清除map里所有的键值对
    e = map.isEmpty();
    System.out.println(e);//输出true
}
```

- Map 按键(key)排序

简单排序：

```
public void sortMapByKey(){
```

```
        Map<String, String> map = new HashMap<String, String>();
        map.put("a", "this is a");
        map.put("c", "this is c");
        map.put("b", "this is b");
        System.out.println(map);//{b=this is b, c=this is c, a=this is a}
        /**
         * TreeMap是一个按key进行升序排列的一个Map实现类
         * TreeMap会自动的把里面的键值对进行排序
         * 利用这一点，将HashMap转换为TreeMap，即可实现Map按key进行排序
         */
        Map<String, String> tm = new TreeMap<String, String>();
        tm.putAll(map);
        map = tm;
        System.out.println(map);//输出{a=this is a, b=this is b, c=this is c}
    }
```

复杂排序：

```
    public void sortMapByKey(){
        Map<String, String> map = new HashMap<String, String>();
        map.put("a", "this is a");
        map.put("c", "this is c");
        map.put("b", "this is b");
        System.out.println(map);//输出{b=this is b, c=this is c, a=this is a}
        /**
         * LinkedHashMap是会记录你put进去的顺序，输出时，会按你put进去的顺序进行输出
         * 利用这一点，将HashMap的key按要求排列好，然后再put进一个LinkedHashMap即能实现map的复杂排序
         */
        Map<String, String> lm = new LinkedHashMap<String, String>();
        List<String> list = new ArrayList<String>(map.keySet());
        Collections.sort(list, new Comparator<String>(){

            @Override
```

```
            public int compare(String o1, String o2) {
                return o2.compareTo(o1);//list降序排列
            }

        });
        System.out.println(list);//输出[c, b, a]
        for (String key : list) {
            lm.put(key, map.get(key));
        }
        System.out.println(lm);//输出{c=this is c, b=this is b, a=this is a}
    }
```

- Map 按值(value)排序

按值排序，使用的较少，测试人员基本用不上，且实现起来也比较复杂，所以不给出代码。

Java 第四天

冒泡排序

1. 冒泡排序介绍

冒泡排序是这样的：在一组数中（int[] a = {3,1,6,2,5}），从头开始（即从 0 开始），相邻的两个数进行比较(a[0]与 a[1]比较)，如果要升序排列，在比较时，如果后面一个小于前面一个（如果 a[0]>a[1]），则两个数进行交换(temp = a[0]; a[0] = a[1]; a[1]=temp)， 并且下标加 1（即用 a[1]与 a[2]进行比较了），当比较 4(长度为 5，最后一个就不用自己跟自己比较了，所以是比较 4 次)次之后，该组数据中的最大值 6 就会被冒到最右边了，就变成了{1,3,2,5,6}了，接下来对剩下的{1,3,2,5}再重复上面的步骤，一直到数据为{1,2,3,5,6}为止。

以上的这段话，再次整理下就是，循环一次，冒出一个最大的数，循环两次，冒出两个数，假如数组长度为 5，则以上的循环进行 4 次，就能冒出 4 个数了，那剩下的那个，自然就不用冒了。即 4 次整体的循环，这 4 次，也是一个循环，那就是循环里套循环了。如此一来，代码就出来了：

```
public void sort(){
    int[] a = {49,38,52,53,65,97,12,13,51};
    for (int i = 0; i < a.length-1; i++) {
        for (int j = 0; j < a.length-i-1; j++) {
            if(a[j]>a[j+1]){
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
            }
        }
    }
    for (int i : a) {
        System.out.println(i);//循环输出数组a
    }
}
```

快速排序

1. 快速排序介绍

快递排序，讲究一个快字。对于给定的一组数据，如何最快速的排好序？在排序过程中，我们不可避免的要对这组数据进行循环，比如 int[] a = {5,6,4,3,7}，这一组数据，我们肉眼一看，5 应该排在这个数组的第 3 位，但是计算机是没有眼睛的，计算机为了”看出”5 在第 3 位，它必须循环一遍，把 5 与所有的数据比较一遍之后，它就知道 5 在整个数组中应该排第几位了，为了突出快速排序的快，那么在循环时，从数组的前后一起循环这样是最快的了，在比较的过程中，将比 5 大的，全放在右边，比 5 小的，全放在左边，这样当循环结束时，5 的位置就被确定在了第 3 位了。所以，对于{5,6,4,3,7},我们先选定一个 5，循环时，先从后面开始，7 与 5 比较，7 大，那么继续循环至下一个，下一个是 3，3 比 5 小，那么要将 3 放在左边，数据变为了：{3,6,4,3,7}，此时，循环体换到从前面开始了，那么就是 3 与 5 比较，3 小，那么继续循环至下一个，下一个是 6，比 5 大，所以，要把 6 丢到右边去，这样数据变为了{3,6,4,6,7}，如此，循环体再切到了右边，6 比 5 大，接着从右边往左边走，4 比 5 小，那么 4 应该再丢到左边了{3,4,4,6,7},此时，整个一个循环结束了，把选定的 5 放到中间位置，就变成了{3,4,5,6,7},此时，以 5 的位置为分割点，就分成了两部分，剩下的两部分再用上面的方法递归下去，即可得到最后的排序结果了，代码如下：

```
public int getSortIndex(int[] a, int left, int right){
    int temp = a[left];
    while(left<right){
        while(left < right && a[right]>=temp){
            right--;
        }
        a[left] = a[right];
        while(left < right && a[left] <= temp){
            left++;
        }
        a[right] = a[left];
    }
```

```
        a[left] = temp;
        return left;
    }

    public void quickSort(int[] a, int left, int right){
        if(left < right){
            int index = this.getSortIndex(a, left, right);
            this.quickSort(a, left, index-1);
            this.quickSort(a, index+1, right);
        }
    }

    public void sort(int[] a){
        if(a!=null && a.length>0){
            this.quickSort(a, 0, a.length-1);
        }
    }
```

调用：

```
int[] a = {49,38,52,53,65,97,12,13,51};
t.sort(a);
```

Java 第五天

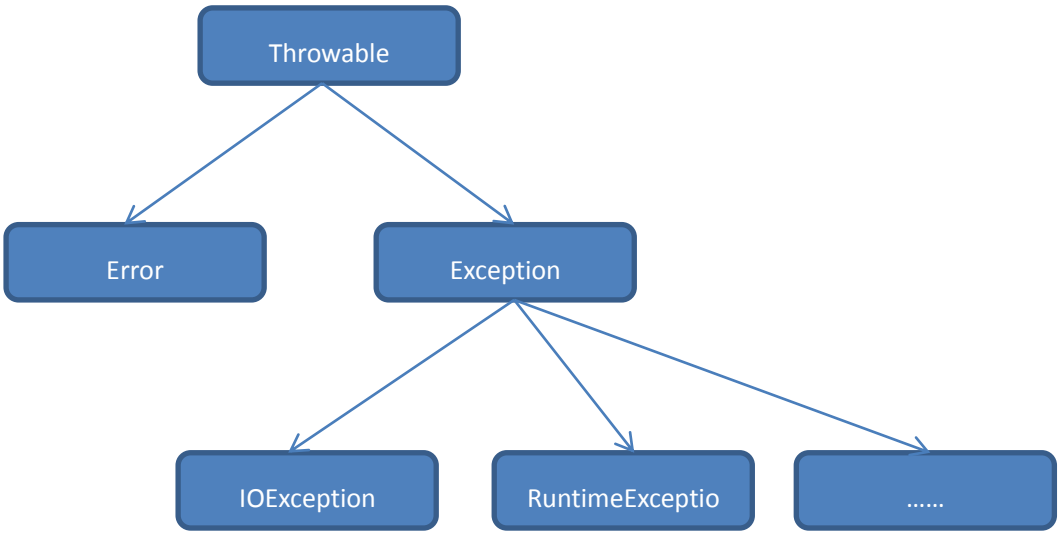
异常概述

1. 异常分类

   java 异常可以分为三类：

   1) 检查异常：对于一些可预见的问题产生的异常，是可控制的。比如要读取某个文件，但该文件根本不存在，则会发生异常，对于这类异常，往往需要事先处理好，如果不事先处理好，则不会被编译通过。
   2) 运行时异常：对于一些不可预见的问题产生的异常，是不可控制的。比如两个数相除，如果用户把除数输入成了 0，则在运行时，就会发生异常，但是这类异常在编译时是会被编译通过的。
   3) 错误：对于一些操作系统或硬件资源引起的异常，总是不可控制的。比如堆栈溢出产生的错误异常，还有我们做测试时当断言失败后，也会产生一个错误异常，这类错误异常在编译时也是会被编译通过的。

2. 异常层次结构



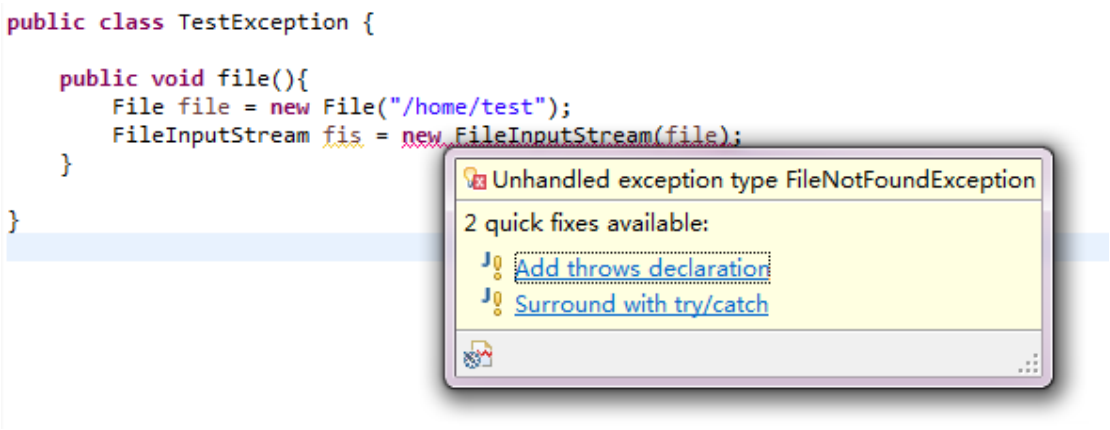
Throwable 类是所有异常类的超类，Error 类和 Exception 类都是 Throwable 类的子类，Exception 类中有两个重要的子类：IOException 类与 RuntimeException 类。在自定义异常时，也是在 Exception 类中派生出自己的异常类。

异常处理

1. 异常处理分类

由于异常是分为可控制与不可控制两种，所以，我们在处理时，也分为了两种：被动处理与主动处理两种。

被动处理：主要是对于可控制的异常，也就是检查异常，IDE 会提示你去处理，否则编译无法通过，是 IDE 提示我们，强制我们去处理的，所以称为被动处理。比如：



主动处理：主要是对于不可控制的异常，IDE 不会提示我们，需要根据程序员的经验来预判，如果认为这里可能出现异常，则主动的进行处理，称为主动处理。

2. 异常处理方式

1) 被动处理

异常的被动处理方法，有两种,即”Add throws declaration”与”Surround with try/catch”。

我们先来看”Surround with try/catch”，这个的意思是说加上 try/catch，我们来看代码：

```
public void file(){
    File file = new File("/home/test");
    try {
        FileInputStream fis = new FileInputStream(file);
        System.out.println("a");
    } catch (FileNotFoundException e) {
```

```
            System.out.println("b");
        }
        System.out.println("c");
    }
```

假如”/home/test”文件不存在，那么这段代码运行后会输出:”bc”。

当程序运行到 FileInputStream fis = new FileInputStream(file);这一句时，会产生异常，由于我们加了 try/catch，所以这个异常就被我们捕获了，那么 try{}块中产生异常的那一句代码的下面的代码就不会被执行了，就跳到了 catch{}块里面去了，就开始执行 catch{}块里面的代码了，所以，”a”没有被输出，”b”被输出了，catch{}块运行完后，就再运行方法体中 try/catch 下面的代码了，所以”c”被输出了。

我们再来看”Add throws declaration”，这个的意思是当产生异常时，把异常往外层的方法抛，比如下面的代码：

```
    public void file() throws FileNotFoundException{
        File file = new File("/home/test");
        FileInputStream fis = new FileInputStream(file);
        System.out.println("a");
    }
```

上面的代码用了 throws 关键字，是指当发生异常时，我不捕获这个异常，不处理这个异常，交给外层的方法来处理，比如外层方法(外层方法指要调用 file 方法的方法)：

```
public void testFile(){
    this.file();
}
```

这个时候，外层方法就要被动的处理这个异常了，要么 try/catch，要么再抛给外外层。我们来看外层方法处理后的输出：

```
    public void file() throws FileNotFoundException{
        File file = new File("/home/test");
        FileInputStream fis = new FileInputStream(file);
        System.out.println("a");
    }

    public void testFile(){
        try {
            this.file();
            System.out.println("b");
```

```
        } catch (FileNotFoundException e) {
            System.out.println("c");
        }
        System.out.println("d");
    }
```

上面的代码，调用 testFile 方法后会输出”cd”。用了 throws 把异常抛出后，产生异常的代码行以下的代码，都不会被执行，也就是说会跳出 file 方法了，跳出 file 方法后，代码又回到了 testFile 方法，同时 testFile 方法接收到了来自 file 方法的异常，即 this.file()这一行产生异常，testFile 经过 try/catch 处理后，输出了”cd”。

2) 主动处理

异常的主动处理，需要我们在可能出现异常的地方，将异常捕获，或者抛出给外层方法，比如如下的代码：

```
public void test(){
    int a = 0;
    int b = 1/a;
    System.out.println(b);
}
```

以上的代码会被编译通过，但是根据 a 的值的变化可能有异常，所以，我们需要主动的处理一下：

```
public void test(){
    try{
        int a = 0;
        int b = 1/a;
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

或者：

```
public void test() throws Exception{
    int a = 0;
    int b = 1/a;
}
```

3) 总结

既然异常处理方式有 try/catch 和 throws 两种，那我们一般用哪一种？其实一般有这样

一些原则（当然不是官方原则），在最外层的方法上用 try/catch 把整个方法体全捕获起来，其上层的方法，能抛出的，就都抛出，这样呢，所有的异常就都统一在最外层处理了。

异常类型

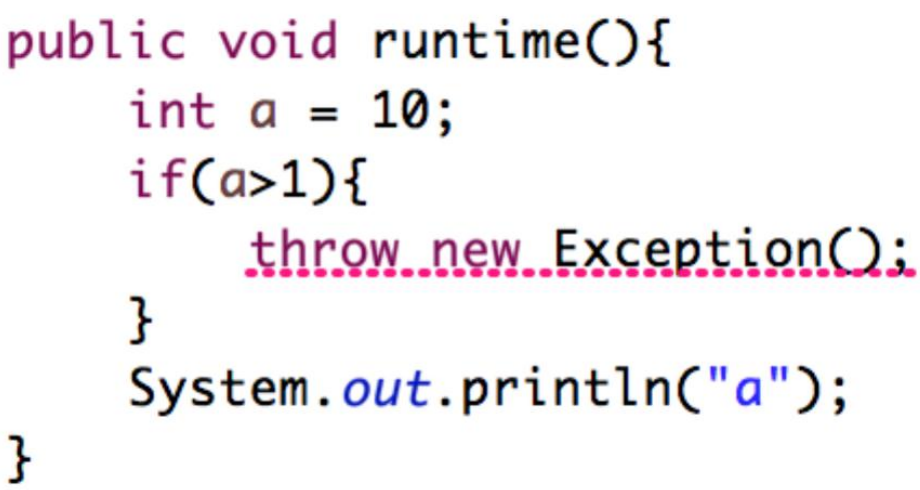
我们在 try/catch 时，要主动的告诉程序我想 catch 住一个什么类型的异常，如果是被动处理，IDE 会自动的帮我们找到是什么类型的异常，而如果是主动处理，那应该如何判断异常类型？我们再回过头来看异常的层次结构，Throwable 类有两个派生类:Error 类和 Exception 类，其它的异常类，都会继承于 Error 类或 Exception 类，所以，在不确定异常类型时，可以用 try{}catch(Error e){}或 try{}catch(Exception e){},或者干脆把两个都写上:
try{}catch(Error e){}catch(Exception e){}

产生异常

有异常发生时，运行线程会在异常发生的地方中断，然后异常或被抛出或被捕获。当有些场景，我们需要线程发生中断时，我们就可以利用这一特点，所以，需要我们去主动的产生一个异常，而产生异常的方式有多种，比如我们可以：

```
public void runtime(){
    int a = 10;
    if(a>1){
        int b = 1/0;
    }
    System.out.println("a");
}
```

但这样的代码未名也太 low 了点，所以，java 提供了一个关键字 throw(注意是 throw 不是 throws，面试时会被经常问到这两者间的区别，throws 是用在方法上抛出异常，throw 是用来产生异常)，让我们来主动的产生异常，如下代码：



利用 throw 来产生了一个 Exception 的异常，但是 Exception 的异常却需要我们被动的去捕获，这就达不到方便的目的了，于是，java 提供了一个 RuntimeException 的类，让其在产生异常的同时，不需要我们被动的去处理，这样就很方便了：

```
public void runtime(){
    int a = 10;
    if(a>1){
        throw new RuntimeException();
    }
    System.out.println("a");
}
```

自定义异常

所谓的自定义异常，是指通过继承异常类，派生出一个自己定义的异常类。自定义异常类有什么好处？好处当然有，可以自己控制异常时的输出，可以更加优化我们的代码结构等等，就跟我们写代码为什么要一层一层的封装一样的道理。

自定义异常有两种写法，第一种：

```
public class TestException extends RuntimeException{

    private static final long serialVersionUID =
-1484113975578015987L;
```

```
    public TestException() {
        super();
    }

    public TestException(String message) {
        super(message);
    }

}
```

调用时：

```
    public void test(){
        int a = 10;
        if(a>9){
            throw new TestException("this is exception");
        }
    }
```

第二种：

```
public class TestException {

    public static void throwException(String message){
        throw new RuntimeException(message);
    }

}
```

调用时：

```
    public void test(){
        int a = 10;
        if(a>9){
            TestException.throwException("this is exception");
        }
    }
```

两种方法都可以达到自定义异常的目的，个人觉得第一种更加的好一些，也更易于扩展。大家在使用时，根据自己的代码习惯自由选择即可。

Java 第六天

前言

IO 操作，无非就是读写文件，对于测试人员而言，也不需要弄懂 IO 背后的故事，重要的是我也不懂这个故事，那就让我们简单粗暴的来点代码吧。

读文件

1. 流读取

```
public void readByte(String fileName) {
    InputStream is = null;
    try {
        is = new FileInputStream(fileName);
        byte[] byteBuffer = new byte[1024];
        int read = 0;
        while((read = is.read(byteBuffer)) != -1){
            System.out.write(byteBuffer, 0, read);
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }finally{
        try {
            if(is != null){
                is.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
        }
    }
```

以上的代码，看着很复杂，感觉要记很多东西，我来给大家解读一下，希望对大家的理解有所帮助。

当自来水厂想要把水送往到用户家里的时候，有得个管道来支持水的流动，所以，当我们想要读取硬盘上的某个文件的时候，我们就得建立一个管道了，那这个管道是通向哪里？通向了内存，把文件内容读到内存中后，再由应用程序来决定把内容如何处理，所以

```
is = new FileInputStream(fileName);
```

这一句代码，可以理解为在文件与内存间建立了一个管道，有了管道，就可以传输了，于是

```
byte[] byteBuffer = new byte[1024];
```

这一句可以理解为申请了一段内存空间，然后通过管道，就把文件内容放到内存里去了，这个内存大小是 1024 个字节，所以，当管道输道的字节流达到了 1024 后，就不再输送字节流到内存了，就跟你把水龙头关了后，当然就不再出水了，所以当把内存里的数据读出来后，就又可以往里面输送字节流了，于是

```
while((read = is.read(byteBuffer)) != -1)
```

用这一句来循环读取内存里的数据，因为有可能读的文件大小是 2048 你读一次最多只能读 1024，所以需要循环的判断是否全部读完了。读出来后，你想怎么处理都可以：

```
System.out.write(byteBuffer, 0, read);
```

通过这一句代码，是把读出来的字节流给输出到控制台上。

2. buffer读取

现在很多小区楼顶上，都会有一个储存水的水箱，这是因为自来水厂的水直接输到用户家里存在诸多的弊端，于是自来水厂就把水不直接输送给用户家了，就输送到水箱。基于这一点，在原来流读取的方式上，进行了改进，把数据先放到一个缓冲区里，当缓冲区里的数据达到文本数据的一整行时，再把这一整行给读取出来，这样我们拿到的就是一行一行的数据了，这样对我们而言就更方便了。顺便这种方式也解决了中文乱码的问题。因为流读取可能中间截断，而一行一行的读取，就不存在截断的问题了。

```
    public String readFile(String filePath) {
        StringBuffer appInfolistInput = new StringBuffer();
        try {
            String encoding = "UTF8";
            File file = new File(filePath);
            if (file.isFile() && file.exists()) {
                InputStreamReader read = new InputStreamReader(new
```

```
FileInputStream(file), encoding);
                BufferedReader bufferedReader = new BufferedReader(read);
                String lineTxt = null;
                while ((lineTxt = bufferedReader.readLine()) != null) {
                    appInfolistInput.append(lineTxt.trim());
                }
                read.close();
                bufferedReader.close();
            } else {
                System.out.println("找不到指定的文件");
            }
        } catch (Exception e) {
            System.out.println("读取文件内容出错");
            e.printStackTrace();
        }
        return appInfolistInput.toString();
    }
```

以上代码中的 BufferedReader 就是一个缓冲区，通过 bufferedReader.readLine()来读取缓冲区里的一行一行数据。以上的代码我有些许偷懒，没有把 close 在 finally 里进行，大家复制代码后，自行修改一下即可。

写文件

1. 流写入

写入与读取，讲道理的话，是一样的，我们来看代码：

```
public void writeByte(String fileName){
    try {
        File file = new File(fileName);
        OutputStream os = new FileOutputStream(file);
        String s = "hello world";
        byte[] byteBuffer = s.getBytes();
        os.write(byteBuffer, 0, byteBuffer.length);
        os.flush();
```

```
            os.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

需要注意的是byte[] byteBuffer = s.getBytes();这一句代码是把我们从程序中的数据给放到了内存里去了，然后通过os.write(byteBuffer, 0, byteBuffer.length);把内存里的数据写到本地文件里去。

2. buffer 写入

与 buffer 读入的道理一样，代码如下：

```
public void writeBuffer(String fileName){
    try {
        File file = new File(fileName);
        BufferedWriter output = new BufferedWriter(new FileWriter(file));
        output.write("hello wrold");
        output.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }
```

上面的代码中，output.write("hello wrold");这一句是表示将应用程序中的数据给加载到缓冲区并写入本地文件。

说明

除了上面讲的流管道与缓冲管道外，还有很多派生出来的管道，万变不离其中，我们测试人员只要熟练掌握好这两种管道，然后有条件的，就去熟悉一下其它管道，做到灵活运用即可。

Java 第七天

多线程概念

提到线程，必不可避免的要讲到进程，当 java 程序开始运行时，就会起一个 java 进程，比如在 windows 的资源管理器中，可以看到一个 java.exe，在 Linux 中，用命令”ps –ef | grep java”就会看到多了一个这样的进程，同样的，当你打开 QQ 时，系统就会多了一个 QQ.exe 的进程，也就是说进程是指操作系统能运行的任务。当 java 的进程启动后，为了提高对某个事件的处理能力，就会多启动几个线程来同时处理这个事情，这就是多线程。通俗点讲，假如你有 1000 万，你想每天数一遍，如果用一台点钞机，则会点一天，假如你用 12 台点钞机同时清点，就可能只需要 1 个小时就点完了，增加一个点钞机就是启动一个线程，这就是多线程。线程与进程都有五个状态：创建、就绪、运行、阻塞、终止。这就是多线程的概念。

多线程应用场景

概念有了，好像懂了。但对我们测试人员而言，多线程一般用在什么地方？个人经验，对多线程的使用，大多在这两个地方：

1) 多个线程同时处理同样一件事情。比如假如我们有 1000 个文件要处理，我们可以起 10 个线程，每个线程处理 100 个文件，线程 1 处理编号为 1-100 的文件，线程 2 处理编号为 101-200 的文件，依此类推下去。
2) 一个线程死循环的处理一件事情。比如监控，假如我们想监控数据库表某个字段的值变化，我们就可以启一个线程，用 while(true)去不断的取这个字段的值，并记录下其变化过程。

多线程代码实例

1. Thread 类

```
public class TestThread extends Thread{
```

```
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        TestThread t1 = new TestThread();
        TestThread t2 = new TestThread();
        TestThread t3 = new TestThread();
        t1.setName("this is thread 1");
        t1.start();
        t2.setName("this is thread 2");
        t2.start();
        t3.setName("this is thread 3");
        t3.start();
    }

}
```

说明：

- 以上代码是启了三个线程，每个线程循环三次的输出线程名。
- setName 是设置线程名。
- 启动线程要用 start 方法，而不是直接调用 run 方法。请务必注意！
- Thread.*currentThread*().getName()是获取线程名

2. Runnable 接口

因为 java 只允许单继承，所以，假如 TestThread 类要继承其它的类时，就无法继承了，

但是 java 是允许完成多接口的，所以，jdk 也提供了一个线程接口 Runnable 接口。

```
public class TestThread implements Runnable{

    @Override
```

```
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(Thread.currentThread().getName());
        }
    }

    public static void main(String[] args) {
        TestThread t1 = new TestThread();
        Thread thread1 = new Thread(t1);
        TestThread t2 = new TestThread();
        Thread thread2 = new Thread(t2);
        TestThread t3 = new TestThread();
        Thread thread3 = new Thread(t3);
        thread1.setName("this is thread 1");
        thread1.start();
        thread2.setName("this is thread 2");
        thread2.start();
        thread3.setName("this is thread 3");
        thread3.start();
    }

}
```

说明：使用 Runnable 接口，稍微麻烦了那第一点，在启动线程时，需要把线程类给加到 Thread 类对象里面去，然后用 Thread 类对象去调用 start 方法来启动线程，也就是说不管是继承 Thread 类还是完成 Runnable 接口，最终都是通过 Thread 类对象调用 start 方法来启动线程。

3. Callable 接口

Callable 同样是一个接口，与 Runnable 接口的区别在于：

1) Callable 要 override 的方法是 call(),而 Runnable 要 override 的方法是 run()
2) Callable 执行完 call 方法后允许有返回值，而 Runnable 执行完 run 方法后不允许有返回值
3) call 方法可以 throws Exception，但是 run 方法不可以
4) Callable 有个 FutureTask 类来监控线程，同时 FutureTask 类对象可以取消

线程的执行，也可以获取线程执行完成后的返回值。

Callable 代码示例：

```
public class TestThread<V> implements Callable<V>{


    @Override
    public V call() throws Exception {
        String name = Thread.currentThread().getName();//获取线程名
        for (int i = 0; i < 3; i++) {
            System.out.println(name);//循环输出线程名
        }
        return (V) (name+" return result");//返回值
    }


    public static void main(String[] args) {
        TestThread<String> tt = new TestThread<String>();//Callable接口实现类的实例化
        FutureTask<String> task = new FutureTask<String>(tt);//FutureTask对象实例化，可以理解为启一个线程监控，并将Callable接口实现类对象放入监控
        Thread t = new Thread(task);//线程类实例化，并将线程监控与线程对象结合起来
        t.setName("this is thread 1");//设置线程名
        t.start();//启动线程
        String returnResult = null;
        try {
            returnResult = task.get();//获取线程执行完成后的返回值
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
        System.out.println(returnResult);//输出线程返回值
    }
```

}

4. 线程池

多线程固然好，但什么东西都架不住一个多，一多了后就面临着管理调配等问题了，如何管理好多线程？且线程数一多，对硬件资源的消耗，对系统的并发压力就都会是问题了，于是线程池就应运而生了。线程池很形象的就是把要运行的线程都放在一个池子里面，然后统一管理，服从调配。

Java 提供了 4 种线程池，大家自行去查阅下，作为我们测试人员，我们用的最多的就是 newFixedThreadPool 线程池，这个线程池的特点是：定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。比如线程池规定的容纳最多的线程数是 5 个，那么当有 8 个线程过来时，这 8 个线程会放入队列中去，然后在队列中选取 5 个线程进入线程池运行，余下 3 个线程仍在队列中等待，当运行的 5 个线程中一旦有某一个线程运行完了，则余下的 3 个线程中立即会有一个线程进入线程池运行，直到所有的线程被处理完成。

由于 Callable 接口有线程返回值，与 newFixedThreadPool 线程池的结合的例子多些，所以我也以 Callable 接口与 newFixedThreadPool 线程池为例子，给出代码：

```
public class TestThread<V> implements Callable<V>{

    @Override
    public V call() throws Exception {
        String name = Thread.currentThread().getName();//获取线程名
        System.out.println(name);//输出线程名
        return (V) (name+" return result");//返回值
    }

}

public class ThreadPool {

    private int threadCount = 10;//定义一共有多少个线程

    private int threadpoolCount = 3;//定义线程池的大小
```

```
    public void threadPoolControl() {
        ExecutorService service =
Executors.newFixedThreadPool(threadpoolCount);//产生线程池对象
        List<TestThread<String>> c = new ArrayList<TestThread<String>>();//
用来存放需要运行的所有的线程对象
        for (int i = 0; i < threadCount; i++) {
            c.add(new TestThread<String>());//将线程对象实例化并存放于List中
        }
        try {
            List<Future<String>> futures = service.invokeAll(c);//执行所有的线
程，并返回所有的线程监控对象Future，
            service.shutdown();//当所有线程处理完后，会关闭线程池，是非阻塞的方法。
            for (Future<String> future : futures) {
                System.out.println(future.get());//循环输出线程返回值
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ThreadPool tp = new ThreadPool();
        tp.threadPoolControl();
    }

}
```

线程安全

当多个线程同时去操作同一个对象的变量时，会使这个变量的值变得漂忽不

定，最后都不知道这个变量的值究竟是多少了。就跟有一框苹果，一群人一哄而上去抢，最后都不知道谁抢了多少，也不知道还剩下多少。比如：

```
public class TestDemo {

    public static int d = 10;

    public static int getDemo(){
        d = d -1;
        return d;
    }
}
public class TestThread extends Thread{
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" :
"+TestDemo.getDemo());
    }
    public static void main(String[] args) {
        for (int i = 0; i < 100; i++) {
            TestThread t = new TestThread();
            t.start();
        }
    }
}
```

运行上面的代码时，有可能会出现线程输出值是一样的，也就是说有可能多个线程同时去执行了 getDemo 这个方法，所以他们得到的返回值是一样的！

为了解决上面代码中的这个问题，排除掉一切无法预知的可能性，于是就有了线程安全的概念，即不管你多个线程如何操作这个对象的变量，都要明确的知道这个变量的值。如何做到随时掌握变量的值？我们再分析操作一个变量，可以分为两种：一是借过来用一下，并不改变变量的值，二是拿过来用，并且改变变量的值。总结：线程安全可以分为变量使用型安全与变量消费型安全。

1) 变量使用型

变量使用型，java 为我们提供了一个类 ThreadLocal，就是把要操作的对象变量，我复制一遍到自己的线程中，这样，我要用时，就直接从我自己的线程中去取，这样不就安全了吗？如此一来，就有人反问了：既然只是取用，那不管多少个线程，直接拿就行了，何必多次一举复制变量呢？复制变量很显然会占用内存，会消耗资源，这不得不偿失吗？其实不得不承认 ThreadLocal 不是用来解决对象变量访问问题的，也并不能真正的解决线程安全问题，而主要是提供了对对象变量的访问方式，通过这种方式，可能会使对象变量的访问速度更加高效更加便捷。举个例子，有三兄弟都想看一本书，但只有一本书，且三兄弟看书的进度不一样，书又不能被撕开，于是，就出现了一个很奇怪的场景：老大侧着头在看第 20 页，老二歪着在看第 10 页，老三只能跟着老二看第 10 页，自己脑补一下这个画面，是不是相当的别扭？他们老爸为了解决这个问题，就想出了一个办法：让他们三个人把这本书都给抄录一遍！这样以后自己想怎么看就怎么看了，这个书就相当于变量了，书有很多页，表示这个变量很复杂，三兄弟表示三个线程。在用 WebDriver 做 UI 自动化时，如果用多线程运行，driver 就会串了，就会有问题了，解决办法就是用 ThreadLocal 来为每一个线程产生的 driver 对象进行单独的维护，不至于在多线程中产生冲突，这也是我为什么把 ThreadLocal 拿出来讲的原因。我们来看 ThreadLocal 的使用代码：

```
public class TestDemo {

    private ThreadLocal<Map<String,String>> dLocal = new
ThreadLocal<Map<String,String>>();//定义一个ThreadLocal对象,dLocal对象是针对
所有线程的，所有线程产生的变量都会放入到该对象中

    public void setDemo(){
        dLocal.set(new HashMap<String,String>());//为每个线程产生一个新的Map，
并放入dLocal里
    }

    public void getDemo(){
        System.out.println(dLocal.get());
        dLocal.get().put("a", "a");//dLocal.get()是指从dLocal里获取该线程的Map
    }
```

```
}
public class TestThread extends Thread{

    private TestDemo td;

    public TestThread(TestDemo td) {
        this.td = td;
    }
    @Override
    public void run() {
        td.setDemo();//该句代码的意义是指线程启动后，就把产生一个变量到该线程中去，并且这个产生的变量Map只有这个线程可以使用
        td.getDemo();
    }
    public static void main(String[] args) {
        TestDemo td = new TestDemo();
        for (int i = 0; i < 10; i++) {
            TestThread t = new TestThread(td);
            t.start();
        }
    }
}
```

2) 变量消费型

变量消费型，这里就真正的涉及到了线程安全了。还是以那一框苹果为例，如何做到一帮人去抢时仍然能够掌控好苹果的数量？如何做到当两个人同时抢到一个苹果时这两人不打架？有人说让他们排好队，一个一个去拿苹果，那这样就不是多线程了，于是一个新的概念就出来了：当这帮人都跑到苹果框前时，随机的选出一个人来，让这个人去拿苹果，这样就做到了永远只有一个人在拿苹果，于是，苹果的数量就能被掌控住了，但新的问题是：如何保证当一个人在拿苹果时，其它人都乖乖听话的不动？聪明的人们就想到了，给苹果框架个盖子，并且加个锁，并且钥匙只有一把，当有一个人要去拿时，把这把锁的钥匙给他，他拿完后，把

锁锁上且钥匙归还，这样就能保证永远只有一个人在拿苹果了！基于这种场景，为了保证线程安全，java 也引入了一个关键字：synchronized，也可以把 synchronized 根据解成为锁！

在上面的场景中，我们可以试着去总结一下这个锁的特点：

- 锁是归对象所有的。当有两框苹果时，就相当于有两个对象了，显然不能用这框苹果的钥匙打开另一框苹果的锁。即多线程只有在操作同一个对象时，才需要考虑线程安全，要是一群人去抢苹果，也有一群筐的苹果在那，还有抢的乐趣吗？还需要抢吗？还有锁的必要吗？
- 对象的锁只有一个，且钥匙只有一把。如果有多把钥匙，抢苹果的人就又多了，又乱了，所以，钥匙只能有一把，多则乱。

synchronized 的使用，可以加在变量上，可以加在方法上，也可以加在代码块中，一般

就这两种用的比较多，分别以代码演示。

加在方法上：

```
public class TestDemo {

    public int d = 10;

    /**
     * synchronized加在方法上指这把锁是属于TestDemo对象的
     * @return
     */
    public synchronized int getDemo(){
        return d = d -1;
    }
}

public class TestThread extends Thread{

    private TestDemo td;

    public TestThread(TestDemo td) {
        this.td = td;
```

```
    }
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" :
"+td.getDemo());
    }
    public static void main(String[] args) {
        TestDemo td = new TestDemo();
        for (int i = 0; i < 10; i++) {
            TestThread t = new TestThread(td);
            t.start();
        }
    }
}
```

加在代码块上：

```
public class TestDemo {

    public int d = 10;

    /**
     * synchronized(this)指下面的代码块的锁是属于TestDemo对象实例的
     * @return
     */
    public int getDemo(){
        synchronized(this){
            d = d - 1;
            return d;
        }
    }
}

public class TestThread extends Thread{
```

```
    private TestDemo td;

    public TestThread(TestDemo td) {
        this.td = td;
    }
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" :
"+td.getDemo());
    }
    public static void main(String[] args) {
        TestDemo td = new TestDemo();
        for (int i = 0; i < 10; i++) {
            TestThread t = new TestThread(td);
            t.start();
        }
    }
}
```

加在代码块上的另一种方式：

```
public class TestDemo {

    public int d = 10;

    private byte[] lock = new byte[0];

    /**
     * synchronized(lock)指下面的代码块的锁是属于lock对象的
     * @return
     */
    public int getDemo(){
        synchronized(lock){
```

```
            d = d - 1;
            return d;
        }
    }
}

public class TestThread extends Thread{

    private TestDemo td;

    public TestThread(TestDemo td) {
        this.td = td;
    }
    @Override
    public void run() {
        System.out.println(Thread.currentThread().getName()+" :
"+td.getDemo());
    }
    public static void main(String[] args) {
        TestDemo td = new TestDemo();
        for (int i = 0; i < 10; i++) {
            TestThread t = new TestThread(td);
            t.start();
        }
    }
}
```

上面的三种方式，对于我们测试人员而言，哪种都可以使用，灵活运用即可。不管哪种方式，我们都要明确下面的几点：

- 对象的钥匙只有一把
- 一旦某个线程获取到了对象的钥匙，那么这个对象中的所有带 **synchronized** 的方法或者代码块其它线程都不能使用，只能阻塞等待在那，只有等当前线程执行完它锁的方法或代码块并归还钥匙给对象后，其它线程才能去竞争这把钥匙。

Java 第八天

泛型

首先，必须明确的一点是 java 是一种强类型的语言，任何对象在使用时，都必须明确指出其类型，否则就会编译报错。而我们在写一些基础方法时，可能不能确定传入参数的类型，我们知道，Object 是所有类的超类，所以，我们可以用 Object 作为参数类型，然后在代码中强转，于是就出现了下面的代码：

```
public class TestGenerics {

    public Object obj;

    public void setObj(Object obj) {
        this.obj = obj;
    }

    public void test(){
        System.out.println(Integer.valueOf(obj.toString())+1);
        System.out.println(obj.toString()+1);
    }

    public static void main(String[] args) {
        TestGenerics t = new TestGenerics();
        t.setObj(3);
        t.test();
    }

}
```

上面的代码也可以达到目的，但看上去别扭，且也不好维护。那可不可以在参数传递进去的时候，自动的判断其数据类型，并自动的帮我们转成传递进去的类型？这就是泛型诞生的原因。泛型，泛指所有的对象类型，所有类型的对象都可以接收，在编译时，jvm 会自动的把泛型转成我们传

递进去的参数类型，即泛型擦除，这样就不用我们自己进行强转了，需要强调的一点是：泛型是针对对象类型的，而java的基本数据类型是不属于对象类型的，所以泛型不支持基本数据类型。还是赶紧的来看看泛型的用法吧：

```
public class TestGenerics {

    public <T> T test(T t){//传入参数为泛型，泛型一般用T表示，<T>表示这是一个泛型方法,<T>后面的T表示返回类型是T
        if(t instanceof Integer){
            return (T)((Integer)((Integer)t+1));//(Integer)t+1返回的是int型的数据，泛型不支持，所以要转为Integer后再次强转为泛型
        }else{
            return (T) ((String)t+1);//参数是什么类型，那么返回值就是什么类型
        }
    }

    public static void main(String[] args) {
        TestGenerics t = new TestGenerics();
        Integer r = t.test(3);//重点是这里，返回值类型会自动转换
        System.out.println(r);//输出4
        String s = t.test("3");//重点是这里，返回值类型会自动转换
        System.out.println(s);//输出31
    }

}
```

以上介绍的是在方法上定义泛型，下面的代码将演示在类上定义泛型：

```
public class TestGenerics<T> {//<T>表示这是一个泛型类，在实例化时，可以指定泛型类型

    public T test(T t){//传入参数为泛型，泛型一般用T表示，<T>表示这是一个泛型方法,<T>后面的T表示返回类型是T
        if(t instanceof Integer){
            return (T)((Integer)((Integer)t+1));//(Integer)t+1返回的是int型的数据，泛型不支持，所以要转为Integer后再次强转为泛型
```

```
        }else{
            return (T) ((String)t+1);//参数是什么类型，那么返回值就是什么类型
        }
    }

    public static void main(String[] args) {
        TestGenerics<String> t = new TestGenerics<String>();//指定泛型类型是String类型
        String s = t.test("3");//重点是这里，返回值类型会自动转换
        System.out.println(s);//输出31
        TestGenerics<Integer> t1 = new TestGenerics<Integer>();//指定泛型类型是Integer类型
        Integer r = t1.test(3);//重点是这里，返回值类型会自动转换
        System.out.println(r);//输出4
    }

}
```

泛型的使用场景

泛型其实我们每天都在用，比如 List<String> list = new ArrayList<String>();这个就是泛型的应用，对于测试人员而言，结合 builder 模式，给出一个 MapBuilder 示例：

```
public class MapBuilder<K, V> {//泛型不只有T，还有K,V 还有通配符?

    public Builder<K, V> b;

    public MapBuilder(Builder<K, V> b){
        this.b = b;
    }

    public Map<K, V> map(){
        return b.map;
    }

    public V get(K key){
```

```
        return b.map.get(key);
    }

    public static class Builder<K,V>{//内部类

        public Map<K, V> map;

        public Builder(){
            map = new HashMap<K, V>();
        }

        public Builder<K, V> put(K key, V value){
            map.put(key, value);
            return this;//不断的put,所以要返回一个当前对象
        }

        public MapBuilder<K, V> build(){
            return new MapBuilder<K, V>(this);//返回一个MapBuilder对象
        }
    }

    public static void main(String[] args) {
        //构建出了一个map,而不需要像以前一样一行只能put一个值
        Map<String, String> map = new MapBuilder.Builder<String,
String>().put("a", "b").put("c", "d").build().map();
        System.out.println(map.get("a"));
        //在build()方面之前，可以随意的put,build之后，
        MapBuilder.Builder<String, String> b = new MapBuilder.Builder<String,
String>();
        b.put("c","c1").put("c1", "c2");
        b.put("c2", "c3");
        System.out.println(b.build().get("c2"));
    }
```

```
}
```

泛型的其它特性，大家可以自行的下去查阅一下，上面的讲解，对于测试人员而言，基本够用了。

反射

反是相对于正而言的，正讲究的是顺应，符合人们的生活习惯，反是指无法通过正常途径来获取到想要的东西，只能自己创造条件，这个创造条件的过程，是一个对个人阅历丰富的过程，但同样也是一个艰难的过程。在程序界也是这样的，在 java 中，一切都是对象，利用对象来获取其变量值，调用其方法，这是正。但如果 jvm 得到的不是对象，而是一个字符串，那怎么办？我们是不是得把这个字符串变为一个对象？同理，根据字符串，也可以获取对象的变量名，变量值，还可以根据字符串来调用对象的方法。是不是很神奇？这个就是反射。我们还是看看代码吧：

```
package com.test;

public class TestReflectObject {

    private int a = 0;

    public int b = 1;

    public int sum(int a, int b){
        return this.a+this.b+a+b;
    }

    private int sum(int a){
        return this.a+this.b+a;
    }

}
```

```
public class TestReflect {


    public void test(){
        String classPath = "com.test.TestReflectObject";//定义类的路径
        try {
            Class<?> clazz = Class.forName(classPath);//根据类的路径来产生一个Class对象
            Object obj = clazz.newInstance();//根据Class对象来产生一个实例对象
            Field f = clazz.getDeclaredField("b");//根据clazz对象及变量名来获取类的变量对象，变量称为Field。
            System.out.println(f.get(obj));//获取变量的值。要获取变量的值，必须是实例化的类对象才能获取到变量的值，所以，get方法的参数是实例化的类对象
            Field[] fields = clazz.getDeclaredFields();//获取已经声明的所有的变量，包含private的
            for (Field field : fields) {
                field.setAccessible(true);//由于private的变量不可访问，所以用setAccessible来使private的变量可访问及修改
                System.out.println(field.getName());//输出变量本身的名称，比如a b
                System.out.println(field.get(obj));//输出在类实例对象obj下的变量的值
            }
            Field fa = clazz.getDeclaredField("a");
            fa.setAccessible(true);
            fa.set(obj, 3);//将对象obj下的变量a赋值为3
            System.out.println(fa.get(obj));//输出3
            Method m = clazz.getDeclaredMethod("sum", new Class[]{int.class, int.class});//根据方法的名称及方法的参数的类型，来获取方法对象
            Object returnValue = m.invoke(obj, new Object[]{5,6});//用invoke方法，结合类实例对象，来执行方法，其中new Object[]{5,6}是执行方法时的入参，returnValue是方法执行完后的返回值
            System.out.println(returnValue);//输出返回值为15
            Method[] methods = clazz.getDeclaredMethods();//根据类对象clazz来
```

```
获取所有的声明的方法
            for (Method method : methods) {
                method.setAccessible(true);//如果方法为private的，则无法调用，通过setAccessible来设置方法的访问权限，
                Class<?>[] pts = method.getParameterTypes();//获取方法的所有的参数类型
                Object[] parames = new Object[pts.length];//构建要调用方法的参数
                for (int i = 0; i < pts.length; i++) {
                    parames[i] = 5+i;
                }
                Object rv = method.invoke(obj, parames);//用invoke方法，结合类实例对象，来执行方法，
                System.out.println(rv);//输出方法执行完后的返回值
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        } catch (NoSuchMethodException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (InvocationTargetException e) {
            e.printStackTrace();
        }
    }
```

```
    public static void main(String[] args) {
        TestReflect tr = new TestReflect();
        tr.test();
    }

}
```

以上就是反射的用法，对于测试人员而言，在关键字驱动中，在设计关键字时，就会用到这个反射，所以掌握好反射的基本用法是非常有必要的。

注解

注解，正在被使用的越来越多，因为其便捷，很多大型的框架中也大量的使用了注解，比如 TestNg，springmvc。所以，我们也有必要去了解下注解是如何定义及使用的。

```
/**
 * 定义一个注解TestAnnotation
 * @author zhangfei
 *java提供了四个元注解@Target @Retention @Documented @Inherited
 *@Inherited用的少
 */
@Target(ElementType.FIELD)//表示TestAnnotation是使用在成员变量上
@Retention(RetentionPolicy.RUNTIME)//表示TestAnnotation是在程序执行时生效
@Documented//表示TestAnnotation可以被javadoc记录
public @interface TestAnnotation {
    String description() default "";//定义TestAnnotation注解上的一个属性
}

public class TestAnnotationObject {

    @TestAnnotation(description="success.")
    public static int RET_CODE_SUCC= 200;
```

```
    @TestAnnotation(description="fail.")
    private static int RET_CODE_FAIL= 201;

}

public class TestAnnotationParse {

    public static String getDescription(Class<?> clazz, int key){
        Field[] fields = clazz.getDeclaredFields();
        Field f = null;
        try {
            for (Field field : fields) {
                field.setAccessible(true);
                /**
                 * 判断是否有TestAnnotation的注解，并且其变量的值是否与入参相等
                 */
                if(field.isAnnotationPresent(TestAnnotation.class) &&
Integer.valueOf(field.get(clazz).toString())==key){
                    f = field;
                    break;
                }
            }
        } catch (NumberFormatException e) {
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        }
        if(f!=null){
            return f.getAnnotation(TestAnnotation.class).description();//获
取变量上的TestAnnotation注解的description的值
        }
```

```
        return null;
    }

    public static void main(String[] args) {

    System.out.println(TestAnnotationParse.getDescription(TestAnnotationObj
ect.class, TestAnnotationObject.RET_CODE_SUCC));

    System.out.println(TestAnnotationParse.getDescription(TestAnnotationObj
ect.class, 201));
    }

}
```

由上可以看出，注解的使用，分为三部分：

1. 定义注解
2. 解析注解。原理是利用反射来获取注解及对应的值
3. 使用注解

至于四个元注解的具体说明，或者注解的特性，大家可以进一步查阅一下。

泛型/反射/注解综合示例

场景是这样的：类似于 IOC，也就是注入，通过注解去实例化一个对象，比如：

```
    @New
    private TestObjectOne one;
```

用 New 这个注解，就自动的把 TestObjectOne 实例化了。原理是这样的：

- 扫描某个文件夹下的类对象，并用反射的方式实例化该文件夹下的所有类对象
- 将实例化的类对象放入一个 Map 中，key 为类的 Class 对象
- 建一个 New 的注解
- 在使用类实例化时，去获取该类对象中的带有 New 注解的变量，并在 Map 中获取变量的实例化对象，通过反射给变量赋值

首先扫描某个文件夹下的类对象，并用反射的方式实例化该文件夹下的所有类对象：

```
public class LoadAllObject {

    private final String basePath = "com.test.demo";//定义要扫描的文件夹路径

    private String classPath = "";

    private List<String> allClass = new ArrayList<String>();

    public void loadAllObject(){
        File f = new File(this.getClass().getResource("/").getPath());
        classPath = f.getAbsolutePath();//获取类对象的路径

    this.listAllFiles(classPath+File.separator+basePath.replace(".","/"));//递归获取所有的类对象文件
        this.getInstance();
    }

    private void listAllFiles(String path){
        path = path.replace("\\", "/");
        File file = new File(path);
        if(file.isFile() && file.getName().endsWith(".class")){//找到所有的.class文件
            String filePath = file.getPath().replace("\\", "/");
            int startIndex = classPath.length()+1;
            int endIndex = filePath.lastIndexOf(".class");
            allClass.add(filePath.substring(startIndex, endIndex).replace("/", "."));//将class的路径放入到allClass对象中去，比如com.test.demo.TestObjectOne
        }else if(file.isDirectory()){
            File[] files = file.listFiles();
            for (File f : files) {
```

```
                this.listAllFiles(f.getPath());//递归扫描所有的.class文件
            }
        }
    }

    private void getInstance(){
        for (String clazz : allClass) {//遍历allClass,allClass里面放的都是文件夹下的.class文件路径
            try {
                Class<?> c = Class.forName(clazz);//根据路径产生Class对象
                ObjectManager.OBJ_MANAGER.put(c, c.newInstance());//实例化对象并将对象放进一个map中去，key为Class类型的对象
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } catch (SecurityException e) {
                e.printStackTrace();
            } catch (IllegalArgumentException e) {
                e.printStackTrace();
            }
        }
    }

}
```

将实例化的类对象放入一个 Map 中，key 为类的 Class 对象

```
public class ObjectManager {

    public static Map<Class<?>, Object> OBJ_MANAGER = new HashMap<Class<?>, Object>();
```

```
    /**
     * static静态块，会在要用到该类对象的时候被执行一次，且只会执行一次，一般用于配置文件
     */
    static{
        new LoadAllObject().loadAllObject();//实例化LoadAllObject类对象，并调用loadAllObject方法
    }

}
```

两个实例对象：

```
package com.test.demo;

public class TestObjectOne {

    public void test(){
        System.out.println("this is test object one!");
    }

}
```

```
package com.test.demo;

public class TestObjectTwo {

    public void test(){
        System.out.println("this is test object two!");
    }

}
```

建一个 New 的注解：

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface New {

}
```

注解解析：

```
public class NewObject {

    public static void autowired(Object obj){
        try{
            Field[] fields = obj.getClass().getDeclaredFields();
            for (Field field : fields) {
                if(field.isAnnotationPresent(New.class)){
                    field.setAccessible(true);
                    field.set(obj,
ObjectManager.OBJ_MANAGER.get(field.getType()));
                }
            }
        }catch(IllegalAccessException e){
            e.printStackTrace();
        }
    }

}
```

注解使用：

```
public class Demo {

    @New
    private TestObjectOne one;
```

```
    @New
    private TestObjectTwo two;

    public Demo() {
        NewObject.autowired(this);//开始注入加了注解的类对象
    }

    public void test(){
        one.test();//输出this is test object one!
        two.test();//输出this is test object two!
    }

    public static void main(String[] args) {
        Demo d = new Demo();
        d.test();
    }

}
```

abstract

abstract，也就是抽象。我的理解，抽象是把需要重复调用的方法或者需要多次实现的方法抽取出来放在一个单独的类或者象形的表述出来。所以，我认为抽象方法有两个作用：

- 抽取公共方法
- 定义抽象方法，让子类去实现

根据以上的作用，我们可以反推下抽象类的特点：

- 可以有变量存在
- 可以用实体方法存在
- 可以有抽象方法存在
- 因为抽象方法要其子类重写，要是不重写，抽象方法就无意义了，所以抽象类只能被继承，不能实例化
- 因为抽象方法要依附于子类才有意义，那么子类同样可以重写抽象类里

的实体方法

关于公共方法，可以这样用：

```
public abstract class TestObject {

    protected int a = 10;

    public void test(){
        System.out.println("this is test object");
    }

    public int testObj(){
        return 1;
    }

}

public class TestObjectOne extends TestObject{

    /**
     * 由于父类也有test方法，子类的test方法相当于是重写了父类的test方法
     * 子类的实例对象在调用test方法时，只会执行子类重写后的test方法，父类的test方法则不会被执行
     */
    public void test(){
        int b = a + this.testObj();//父类变量可以直接使用，也可以直接调用父类的实体方法
        System.out.println("this is test object one! "+b);
    }

    public void testOne(){
        System.out.println("this is test one!");
    }
```

```
}

public class TestObjectTwo extends TestObject{

    /**
     * 由于父类也有test方法，子类的test方法相当于是重写了父类的test方法
     * 子类的实例对象在调用test方法时，只会执行子类重写后的test方法，父类的test方法则不会被执行
     */
    public void test(){
        super.test();//如果仍想调用父类的test方法，用super对象去调用即可
        System.out.println("this is test object two!");
    }


    public static void main(String[] args) {
        TestObject one = new TestObjectOne();
        TestObject two = new TestObjectTwo();
        one.test();//输出this is test object one! 11
        two.test();//输出this is test object this is test object two!
        one.testOne();//报错，这是因为one的对象类型是TestObject,而TestObject中根本没有testOne方法，testOne方法是属于类型为TestObjectOne的对象的.
        TestObjectOne o = new TestObjectOne();
        o.testOne();//输出this is test one!
    }
}
```

公共方法的抽取在测试中的应用场景是很广泛的，大家掌握好其中的关系即可。其实上面的例子也是多态的一种表现形式。再也不要死记硬背多态的概念了，上面的代码理解清楚即可。

关于抽象方法，为什么要有抽象方法？比如一个父亲向两个儿子下达了一个命令，周末回家里来。那这两个儿子周末就必须回来，但他们回来的方式各不相同，老大条件相对好些，开着小汽车回来的，老二是转了两道地铁再坐公交回来

的。我们来解读一下这个场景：

- 父亲下达的这个命令，只是叫你回来，并没有说你以什么方式回来，买不买酒回来，只是一个抽象的命令，也就是抽象类中的抽象方法。
- 两个儿子接到命令后，必须得执行命令，完成命令，也就是重写抽象方法，由于他们回来的交通方式不一样，也就代表着他们重写抽象方法的方法体是不一样的。

以上就是抽象方法存在的意义，即定义一个抽象方法后，让继承它的子类去自由发挥。

```
public abstract class TestObject {

    protected int a = 10;

    public void test(){
        System.out.println("this is test object");
    }

    public int testObj(){
        return 1;
    }

    public abstract boolean come();//定义一个抽象方法

}

public class TestObjectOne extends TestObject{

    /**
     * 由于父类也有test方法，子类的test方法相当于是重写了父类的test方法
     * 子类的实例对象在调用test方法时，只会执行子类重写后的test方法，父类的test方法则不会被执行
     */
    public void test(){
```

```
        int b = a + this.testObj();//父类变量可以直接使用，也可以直接调用父类的实体方法
        System.out.println("this is test object one! "+b);
    }

    public void testOne(){
        System.out.println("this is test one!");
    }

    /**
     * 重写父类的抽象方法，且是必须重写，否则编译通不过
     */
    @Override
    public boolean come() {
        return this.goToByCar();//重写方法体
    }

    private boolean goToByCar(){
        System.out.println("come by car");
        return true;
    }

}

public class TestObjectTwo extends TestObject{

    /**
     * 由于父类也有test方法，子类的test方法相当于是重写了父类的test方法
     * 子类的实例对象在调用test方法时，只会执行子类重写后的test方法，父类的test方法则不会被执行
     */
    public void test(){
        super.test();//如果仍想调用父类的test方法，用super对象去调用即可
```

```
        System.out.println("this is test object two!");
    }

    /**
     * 重写父类的抽象方法，且是必须重写，否则编译通不过
     */
    @Override
    public boolean come() {
        return this.goToByTrain();//返回值即可以通知调用者
    }

    private boolean goToByTrain(){
        System.out.println("come by train");
        return true;
    }

    public static void main(String[] args) {
        TestObject one = new TestObjectOne();
        TestObject two = new TestObjectTwo();
        one.come();//输出come by car
        two.come();//输出come by train
    }

}
```

抽象类在测试中的应用场景

在 UI 自动化时，我们听到很多的一个词是 page-object 模式，简称 PO，也就是每一个页面的元素对象与业务逻辑都写在一个 page 类里，这样就会产生很多的 page 类，那这些 page 类里，会有些基础方法需要封装，那可以把这些基础方法都放在一个 BasePage 类里，所有的 page 类继承 BasePage 抽象类即可。

interface

interface，中文是接口的意思。所谓接口，就是为了方便其它人调用或者扩展而预留下来的一个类，实现这个接口的人根本不需要了解整个程序的逻辑结构，只要按照预留的接口的规范去实现，并且告知程序实现接口的类的路径即可。所以，接口的特点有：

- 接口中的方法全是抽象方法，无实体方法，接口中的变量会被认为是 static 的变量。
- 接口可以理解为一种特殊的抽象类
- 实现接口的类必须重写接口中的所有方法
- Java 只支持单继承，但可以多实现，所以如果需要多继承的地方，不妨把抽象类改为接口来实现
- 实现接口的类的路径必须告诉程序，否则，程序无法知道是否有实现接口的类存在

我们来看一个示例：

```
public interface TestObject {//接口的定义用interface

    public boolean come();//定义抽象方法,接口里的抽象方法只能是public的，如果不写修饰符，默认也是public的

}

public class TestObjectOne implements TestObject{

    @Override
    public boolean come() {
        System.out.println("this is test object one!");
        return false;
    }

}
```

```
public class TestObjectTwo implements TestObject{

    @Override
    public boolean come() {
        System.out.println("this is test object two!");
        return true;
    }

}

public class TestMain {

    private String classPath = "com.test.demo.TestObjectOne";//默认的使用
TestObjectOne类

    public void setClassPath(String classPath) {
        this.classPath = classPath;
    }

    public void test(){
        try {
            TestObject to = (TestObject)
Class.forName(classPath).newInstance();//通过路径实例化对象
            boolean b = to.come();//完成的接口方法的调用
            if(b){
                //do something
            }else{
                //do something
            }
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
```

```
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        TestMain t = new TestMain();
        t.test();
        t.setClassPath("com.test.demo.TestObjectTwo");//告知程序实现接口的类的路径
        t.test();//此时会执行com.test.demo.TestObjectTwo类里的test方法
    }

}
```

上面的例子中，如果我们把 classPath 也就是实现类的路径放在配置文件中，我们就可以通过修改配置文件来控制程序的走向了。Java 中的接口是非常好用且非常中实用的，设计模式中大量的用到了接口，我们在实际的应用中，如果应用程序打了一个 jar 包发给别人了，而如果别人想要扩展时，不可能解压 jar 包，然后修改源码，再打 jar 包后使用吧？如果这样就太麻烦了，所以，预留接口是一个非常好的做法。

总结

这一章的内容有点多，但都是 java 的特性，想要进阶的，了解这些特性并熟练使用是第一步。其实上面的例子中，对于 java 的三大特性：封装，继承，多态都有涉及，了解一门语言并不是死记硬背其概念，更多的是灵活的运用，从使用的角度去了解，熟悉并加以自己的理解，这样，你就快进阶了！

Java 第九天

接口测试概述

移动互联网时代，我们逐渐的发现离不开手机了，各种 app 应有尽有，用户用的爽，苦的是后面的一群工程师，各种接口需要维护，作为一个测试人员，我们需要保证接口的正确性，稳定性，安全性等。那问题来了：什么是接口？在我看来，接口是终端与服务端之间过行通信的一道门或者窗口，门的两边分别是终端与服务端，它们之间只需要进行数据交换就行了，服务端是不会关心终端拿到数据有什么用，终端也不会关心服务端把数据放在什么地方，它们彼此是独立的。那终端与服务端靠什么进行数据交换？当然是互联网。那在互联网中是如何进行数据交换并传输的？靠的是互联网中大家共同遵守的协议，也就是 http 协议。那交换的数据是什么样的格式？restful 风格的嘛，也就是 json 串，当然也有表单风格的。有的人可能会说我们不是走的 http 协议，或者说我们用的是 webservice，我只能表示大家现在普遍都是用的 http 协议的 restful 风格的接口，如果你们公司非要标新立异，那只能引用某大佬讲过的一句话：你为什么要选择那 1%？以下的代码示例我也会以此为基础给出实例。

http 协议的基本知识

- http 请求

  http 请求包含三部分:请求行，消息头(header)，请求正文(body)

  1. 请求行：请求方式与请求路径
     a) 请求方式：一般是 GET 和 POST 两种请求方式
     b) 请求路径：请求的 URL
  2. 消息头：User-Agent，Accept，Content-Type，Content-Length 等。User-Agent 是标识请求的浏览器或者终端设备，比如如果是安卓手机，返回 A 内容，如果是苹果手机返回 B 内容。Accept 是指请求回来的数据格式，标明哪些是能接受，哪些不能接受的。Content-Type 是指请求发送的数据格式，也就是请求正文的数据格式。Content-Length 也就是请求发送数据的长度，即请求正文的数据长度，Content-Length 一般不

会限制，爱传多长传多长嘛，是吧，限制干嘛。消息头还有一些其它的值以及可以自定义消息头里的值，都会随着请求一起提交到服务端去，服务端是可以获取到消息头里的所有内容的。有的人可能会问，那请求参数我也放在消息头里，是否可行？当然可行啊，但开发基本都不会这么干啊，一般消息头也就是存放一些约定好的不怎么会发生变化的内容，请求的参数变化的可能性很大，放在里面不太合适。

3. 请求正文：也就是请求参数，即要传给服务端的数据。

请求行，消息头，请求正文三者之间的关系

1. 请求方式是 GET 时，参数在请求路径上用?相连接，此时无请求正文。GET 请求的请求路径是有长度限制的。
2. 请求方式是 POST 时，参数是放在请求正文里
3. 消息头中的 Content-Type 决定请求正文的数据格式
4. 消息头中的 Content-Length 决定请求正文的数据长度

Content-Type 与请求正文的对应表：

Content-Type	参数格式示例
application/x-www-form-urlencoded	name=zhangsan&age=18
application/json	{"name":"zhangsan","age":"18"}
text/plain	zhangsan
text/html	<html>zhangsan</html>

说明：

以上对应表，只是列出了我们做接口测试常见的几种 Content-Type,错误的 Content-Type 很可能会引起请求失败。当你不确定是哪个 Content-Type 时，去抓个包或者看接口文档即可。当请求失败时，很大程度上是请求头没设置全或者参数不对引起的。

- http 响应

http 响应包含三部分：状态行，消息头(header)，响应正文(body)

1. 状态行：200，404，403，500 等这些都是状态行，其中 200 是 OK，表示请求成功
2. 消息头：与请求消息头的作用一样，是告诉请求端信息的。我们在做接口测试时对于响应消息头的关注不多
3. 响应正文：服务端返回给终端的数据。这个是我们做接口测试比较关注

的。现在一般是返回一个 json 串，方便解析以及各种终端平台通用。

状态行与响应正文的关系：

一般是状态行是 200 时，我们才去关注响应正文，这样才有意义，当状态行不是 200 时，直接判定请求失败。

HttpURLConnection 图文示例

GET 请求实例

```
public String get(String url){
        HttpURLConnection conn = null;
        BufferedReader rd = null ;
        StringBuilder sb = new StringBuilder();
        String line = null ;
```

```
        String response = null;
        try {
            conn = (HttpURLConnection) new URL(url).openConnection();
            conn.setRequestMethod("GET");
            conn.setDoInput(true);
            conn.setReadTimeout(20000);
            conn.setConnectTimeout(20000);
            conn.setUseCaches(false);
            conn.connect();
            rd  = new BufferedReader( new
InputStreamReader(conn.getInputStream(), "UTF-8"));
            while ((line = rd.readLine()) != null ) {
                sb.append(line);//获取响应正文
            }
            response = sb.toString();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(rd != null){
                    rd.close();
                }
                if(conn != null){
                    conn.disconnect();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return response;
    }
```

POST 表单请求

```
public String post(String url, Map<String, String> form){
        HttpURLConnection conn = null;
        PrintWriter pw = null ;
        BufferedReader rd = null ;
        StringBuilder out = new StringBuilder();
        StringBuilder sb = new StringBuilder();
        String line = null ;
        String response = null;
        //将map里的值转换成name=zhangsan&age=18这种数据格式
        for (String key : form.keySet()) {
            if(out.length()!=0){
                out.append("&");
            }
            out.append(key).append("=").append(form.get(key));
        }
        try {
            conn = (HttpURLConnection) new URL(url).openConnection();
            conn.setRequestMethod("POST");
            conn.setDoOutput(true);
            conn.setDoInput(true);
            conn.setReadTimeout(20000);
            conn.setConnectTimeout(20000);
            conn.setUseCaches(false);
            conn.connect();
            pw = new PrintWriter(conn.getOutputStream());
            pw.print(out.toString());
            pw.flush();
            rd  = new BufferedReader( new
InputStreamReader(conn.getInputStream(), "UTF-8"));
            while ((line = rd.readLine()) != null ) {
```

```
                sb.append(line);
            }
            response = sb.toString();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(pw != null){
                    pw.close();
                }
                if(rd != null){
                    rd.close();
                }
                if(conn != null){
                    conn.disconnect();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return response;
    }
```

POST 之 JSON 请求

```
public String post(String url, String json){
        HttpURLConnection conn = null;
        PrintWriter pw = null ;
        BufferedReader rd = null ;
        StringBuilder sb = new StringBuilder ();
```

```
        String line = null ;
        String response = null;
        try {
            conn = (HttpURLConnection) new URL(url).openConnection();
            conn.setRequestMethod("POST");
            conn.setDoOutput(true);
            conn.setDoInput(true);
            conn.setReadTimeout(20000);
            conn.setConnectTimeout(20000);
            conn.setUseCaches(false);
            conn.setRequestProperty("Content-Type", "application/json");
            conn.connect();
            pw = new PrintWriter(conn.getOutputStream());
            pw.print(json);
            pw.flush();
            rd  = new BufferedReader( new
InputStreamReader(conn.getInputStream(), "UTF-8"));
            while ((line = rd.readLine()) != null ) {
                sb.append(line);
            }
            response = sb.toString();
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(pw != null){
                    pw.close();
                }
                if(rd != null){
                    rd.close();
                }
```

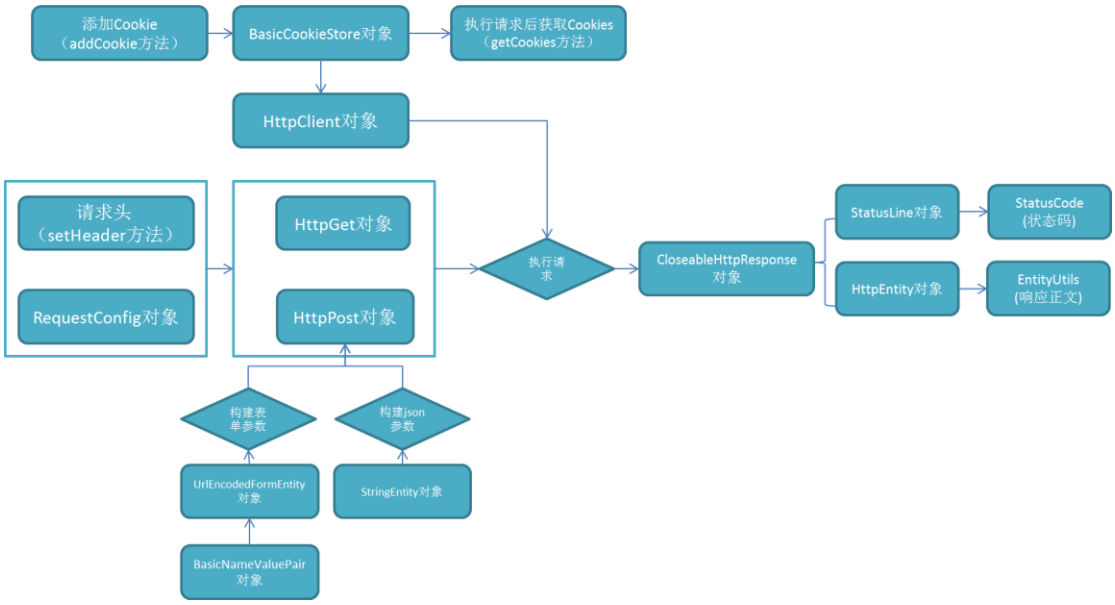
```
                if(conn != null){
                    conn.disconnect();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return response;
    }
```

Java 第十天

HttpClient 概述

上一章讲了接口的理论，也讲了用 HttpURLConnection 去请求一个接口并获取请求返回值。可能大家也感觉到了 HttpURLConnection 是比较原始的写法，面向过程的写法，缺少面向对象的概念，对于新手还不太好理解，于是 HttpClient 就诞生了，HttpClient 是在 HttpURLConnection 基础上进行了封装，对象结构清晰明了，且经历了多个版本的演变后，易用性得到了极大的改变，特别是在 4.X 版本上(以下的示例将会在 4.3.5 的版本上)。

图解 HttpClient



GET 请求

```
public String get(String url){
    CloseableHttpClient httpClient = null;
    HttpGet httpGet = null;
```

```
        try {
            httpClient = HttpClients.createDefault();//实例化HttpClient对象
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(20000).setConnectTimeout(20000).bui
ld(); //构建RequestConfig对象
            httpGet = new HttpGet(url);//构建HttpGet对象
            httpGet.setConfig(requestConfig);//设置RequestConfig
            CloseableHttpResponse response = httpClient.execute(httpGet);//执
行请求并获取response对象
            HttpEntity httpEntity = response.getEntity();//获取响应正文对象
            return EntityUtils.toString(httpEntity,"utf-8");//按编码输出响应正
文
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpGet!=null){
                    httpGet.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
```

POST 请求表单提交

```
    public String post(String url, Map<String, String> params){
        CloseableHttpClient httpClient = null;
        HttpPost httpPost = null;
        try {
            httpClient = HttpClients.createDefault();//实例化HttpClient对象
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(20000).setConnectTimeout(20000).bui
ld();//构建RequestConfig对象
            httpPost = new HttpPost(url);//构建HttpPost对象
            httpPost.setConfig(requestConfig);//设置RequestConfig
            /**
             * 开始构建表单参数，表单参数的数据格式是：name=zhangsan&age=18,
             * 其中，name与age为key,分别对应的value为zhangsan与18
             * 正因为表单参数的数据格式为key-value的键值对形式，所以入参为一个
Map<String, String> params
             */
            List<NameValuePair> ps = new ArrayList<NameValuePair>();
            for (String pKey : params.keySet()) {
                ps.add(new BasicNameValuePair(pKey, params.get(pKey)));
            }
            //构建表单参数完毕，产生一个List<NameValuePair>的对象,其中
BasicNameValuePair是产生一个key-value的键值对对象
            /**
             * new UrlEncodedFormEntity(ps)中的UrlEncodedFormEntity对象是将
List<NameValuePair>转换为表单参数的数据格式，
             * 并将Content-Type设置为application/x-www-form-urlencoded
             * 所以不需要我们自己设置Content-Type了
             */
            httpPost.setEntity(new UrlEncodedFormEntity(ps));//设置请求正文，
也就是设置表单参数
```

```
            CloseableHttpResponse response = httpClient.execute(httpPost);//
执行请求并获取response对象
            HttpEntity httpEntity = response.getEntity();//获取响应正文对象
            return EntityUtils.toString(httpEntity,"utf-8");//按编码输出响应正
文
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpPost!=null){
                    httpPost.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
```

POST 请求 JSON 提交

```
    public String post(String url, String body){
        CloseableHttpClient httpClient = null;
        HttpPost httpPost = null;
        try {
            httpClient = HttpClients.createDefault();//实例化HttpClient对象
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(2000).setConnectTimeout(2000).build
```

```
();//构建RequestConfig对象
            httpPost = new HttpPost(url);//构建HttpPost对象
            httpPost.setConfig(requestConfig);//设置RequestConfig
            /**
             * 在请求正文中，可以随意的提交一段字符串给服务器，现在为了数据交换的统一性，所以大家都是提交的一段json串，
             * 所以body的数据格式{"name":"zhangsan","age":18}，如果body是该数据格式，则Content-Type要设置为application/json
             * new StringEntity(body)是指利用body构建一个请求正文对象，且StringEntity是支持Content-Type为application/json的，
             * 所以，如果用了StringEntity对象且body的数据格式为json串，则无需我们再设置Content-Type
             */
            httpPost.setEntity(new StringEntity(body));//设置请求正文，也就是设置json串
            CloseableHttpResponse response = httpClient.execute(httpPost);
            HttpEntity httpEntity = response.getEntity();//获取响应正文对象
            return EntityUtils.toString(httpEntity,"utf-8");//按编码输出响应正文
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpPost!=null){
                    httpPost.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
```

```
            }
        }
        return null;
    }
```

POST 请求表单提交扩展

在介绍表单提交时，用到了 UrlEncodedFormEntity 对象，UrlEncodedFormEntity 对象会设置好数据格式，并设置好 Content-Type，我们知道请求正文，最终都是一个字符串，所以我们是否可以事先拼接起表单的字符串（如 name=zhangsan&age=18），然后设置好 Content-Type，用 StringEntity 对象来构建请求正文？答案是可以的，写法如下：

```
    /**
     * body的数据格式必须是name=zhangsan&age=18
     */
    public String post(String url, String body){
        CloseableHttpClient httpClient = null;
        HttpPost httpPost = null;
        try {
            httpClient = HttpClients.createDefault();//实例化HttpClient对象
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(2000).setConnectTimeout(2000).build
();//构建RequestConfig对象
            httpPost = new HttpPost(url);//构建HttpPost对象
            httpPost.setConfig(requestConfig);//设置RequestConfig
            httpPost.setHeader("Content-Type",
"application/x-www-form-urlencoded");//setHeader是设置请求头，该句代码是指设置
Content-Type,要设置其它的请求头，也是用setHeader方法。
            httpPost.setEntity(new StringEntity(body));//设置请求正文，也就是设
置json串
            CloseableHttpResponse response = httpClient.execute(httpPost);
            HttpEntity httpEntity = response.getEntity();//获取响应正文对象
            return EntityUtils.toString(httpEntity,"utf-8");//按编码输出响应正
文
```

```
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpPost!=null){
                    httpPost.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
```

另外，你也可以不用:

```
httpPost.setHeader("Content-Type", "application/x-www-form-urlencoded");
```

这一句来设置 Content-Type，StringEntity 对象本身也可以指定 Content-Type:

```
httpPost.setEntity(new StringEntity(body,"application/x-www-form-urlencoded"));
```

大家可以下去自行试验。

关于 SEESION 与 COOKIE

一句话概念：

session 是保存在服务端，cookie 是保存在客户端。

session 与 cookis 的联系：

在请求时，如果应用程序中用到了 session,服务器会先检测有没有 session id 传过来，没有就会创建一个 session，并将 session 保存在服务器内存中，同时将 session id 设置在消息头(Set-Cookie)中返回给客户端,客户端收到后，会将 Set-Cookie 中的 session id 给取出来，并保存在客户端本地，在下次请求时，会把本地的 session id 放在请求消息头中，或

者放在请求链接上，一起发给服务端。这就是为什么我们登录后，就可以操作我们用户名下的信息，就是通过 cookies 与服务器端的 session 进行信息验证的原因。

备注：

session 与 cookie 在我们的接口测试中，一般很少用到。

在 HttpClient 中，有一个专门的对象 BasicCookieStore 来添加与获取 cookie。比如我们要先 post 用户名与密码给服务器，然后保存请求后的 cookie,在下一次请求时，带上 cookie，我们来看代码的实现：

```
    /**
     * post请求，并返回服务端返回的cookies
     */
    public List<Cookie> post(String url, Map<String, String> params){
        CloseableHttpClient httpClient = null;
        HttpPost httpPost = null;
        try {
        CookieStore cookieStore = new BasicCookieStore();//建立cookieStore对
象
            httpClient =
HttpClients.custom().setDefaultCookieStore(cookieStore).build();//将
cookieStore对象设置到HttpClient对象中去。
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(20000).setConnectTimeout(20000).bui
ld();
            httpPost = new HttpPost(url);
            httpPost.setConfig(requestConfig);
            List<NameValuePair> ps = new ArrayList<NameValuePair>();
            for (String pKey : params.keySet()) {
                ps.add(new BasicNameValuePair(pKey, params.get(pKey)));
            }
            httpPost.setEntity(new UrlEncodedFormEntity(ps));
            CloseableHttpResponse response = httpClient.execute(httpPost);
            if(response.getStatusLine().getStatusCode() == HttpStatus.SC_OK){
             return cookieStore.getCookies();//当返回状态码是200时，获取服务端返
回的cookies并返回出去
```

```
            }
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpPost!=null){
                    httpPost.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return new ArrayList<Cookie>(0);
    }

    /**
     * 再次请求时，将已保存下来的cookies一起提交给服务端
     */
    public String get(String url, List<Cookie> cookies){
        CloseableHttpClient httpClient = null;
        HttpGet httpGet = null;
        try {
        CookieStore cookieStore = new BasicCookieStore();//建立cookieStore对
象
        for (Cookie cookie : cookies) {
            cookieStore.addCookie(cookie);//将已保存下来的cookies添加到
cookieStore对象中去
            }
```

```
        httpClient =
HttpClients.custom().setDefaultCookieStore(cookieStore).build();//将
cookieStore对象设置到HttpClient对象中去。
            RequestConfig requestConfig =
RequestConfig.custom().setSocketTimeout(20000).setConnectTimeout(20000).bui
ld();
            httpGet = new HttpGet(url);
            httpGet.setConfig(requestConfig);
            CloseableHttpResponse response = httpClient.execute(httpGet);
            if(response.getStatusLine().getStatusCode() == HttpStatus.SC_OK){
             return EntityUtils.toString(response.getEntity(),"utf-8");//当返
回状态码是200时，获取响应正文，并按指定编码返回
            }
        } catch (ClientProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                if(httpGet!=null){
                    httpGet.releaseConnection();
                }
                if(httpClient!=null){
                    httpClient.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        return null;
    }
```

Java 第十一天

Jsoup 概述

Jsoup 是可以做接口测试的，也就是可以发送 http 请求的，前面已经讲了 HttpURLConnection 和 HttpClient，都可以做接口测试，但我为什么要把 Jsoup 拿出来再讲一遍呢？这样不是知识重叠了吗？其实 Jsoup 有 HttpClient 没有的一个很重要的功能，就是解析 html，通过解析 html，可以获取到 html 中的数据，这也就是爬虫。我们还是一起来看看 Jsoup 的使用方式及如何爬虫吧。其所要使用的 jar 包是 jsoup-*.jar

GET 请求

```
    public void testJsop(){
        try {
            Document doc =
Jsoup.connect("http://www.cnblogs.com/zhangfei/p/4283930.html").get();
            System.out.println(doc);//返回的html数据
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

GET 请求参数处理

1. 直接放在 URL 上

```
    public void testJsop(){
        try {
            Document doc =
Jsoup.connect("http://www.cnblogs.com/zhangfei/p/?page=3").get();//page=3是
参数
```

```
            System.out.println(doc);//返回的html数据
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

2. 利用 jsoup 提供的 api

```
    public void testJsop(){
        try {
            Connection conn =
Jsoup.connect("http://www.cnblogs.com/zhangfei/p/");
            conn.data("page","3");//conn对象上添加参数
            Document doc = conn.get();//执行get请求
            System.out.println(doc);//返回的html数据
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

POST 请求

Jsoup 的 post 请求只支持表单提交：

```
    public void testJsop(){
        try {
            Connection conn =
Jsoup.connect("https://passport.jd.com/new/login.aspx");
            conn.data("loginname","test1");//conn对象上添加参数
            conn.data("loginpwd","test1");//conn对象上添加参数
            Document doc = conn.post();//执行post请求
            System.out.println(doc);//返回的html数据
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

请求设置

```
    public void testJsop(){
        try {
            Connection conn =
Jsoup.connect("https://passport.jd.com/new/login.aspx");
            conn.data("loginname","test1");
            conn.data("loginpwd","test1");
            conn.timeout(30000);//设置请求超时时间
            conn.ignoreContentType(true);//设置忽略Content-Type，忽略后，表示支
持任何形式的Content-Type.在请求失败时，可以尝试加上这句
            conn.header("User-Agent", "Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:32.0) Gecko/20100101 Firefox/32.0");//设置请求头
            Document doc = conn.post();
            System.out.println(doc);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

Cookies 管理

```
    public void testJsop(){
        try {
            Connection conn =
Jsoup.connect("https://passport.jd.com/new/login.aspx");
            conn.data("loginname","test1");
            conn.data("loginpwd","test1");
            conn.timeout(30000);
            conn.method(Method.POST);//设置请求方式
            /**
             * 当conn对象调用execute方法时，返回的是response对象,cookies是包含在
```

```
response对象中的，
             * 所以如果想获取cookies，必须调用execute方法，且在conn对象上必须指明请求方式,如:conn.method(Method.POST);
             */
            Response response = conn.execute();
            Map<String, String> cookies = response.cookies();//获取请求响应的cookies
            Document doc = Jsoup.connect("http://order.jd.com/center/list.action").cookies(cookies)//再次请求时将cookies带上
                .timeout(30000).get();
            System.out.println(doc);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

上面的示例中，可以看出，要取得 cookies,必须要有个 Response 的对象，所以，要用 execute 方法，如果直接用 conn.post()方法，返回的则是 Document 对象，但在用 execute 方法时，要事先调用一下 conn.method(Method.POST)方法设定好请求方式即可。

Jsoup 解析 html

```
    public void testJsop(){
        try {
            Document doc = Jsoup.connect("http://www.cnblogs.com/zhangfei/p/").get();
            String countText = doc.select("#myposts>div.pager:nth-of-type(1)>.Pager").text();
            System.out.println(countText);
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
}
```

说明：

- Jsoup 先得获取一个 Document 对象
- 用 select 方法来获取 html 中的元素对象 Elements
- select 方法里面的参数是一个 cssSelector
- text 方法是获取元素对象的文本值

cssSelector 说明

常见符号：

#表示 id

.表示 class

>表示子元素，层级

一个空格也表示子元素，但是是所有的后代子元素，相当于 xpath 中的相对路径

示例说明：

#input 选择 id 为 input 的节点

.Volvo 选择 class 为 Volvo 的节点

div#radio>input 选择 id 为 radio 的 div 下的所有的 input 节点

div#radio input 选择 id 为 radio 的 div 下的所有的子孙后代 input 节点

div#radio>input:nth-of-type(4) 选择 id 为 radio 的 div 下的第 4 个 input 节点

div#radio>:nth-child(1) 选择 id 为 radio 的 div 下的第 1 个子节点

div#radio>input:nth-of-type(4)+label 选择 id 为 radio 的 div 下的第 4 个 input 节点之后挨着的 label 节点

div#radio>input:nth-of-type(4)~label 选择 id 为 radio 的 div 下的第 4 个 input 节点之后的所有 label 节点

input.Volvo[name='identity'] 选择 class 为.Volvo 并且 name 为 identity 的 input 节点

input[name='identity'][type='radio']:nth-of-type(1) 选择 name 为 identity 且 type 为 radio 的第 1 个 input 节点

input[name^='ident'] 选择以 ident 开头的 name 属性的所有 input 节点

input[name$='entity'] 选择以'entity'结尾的 name 属性的所有 input 节点

input[name*='enti'] 选择包含'enti'的 name 属性的所有 input 节点

div#radio>*:not(input) 选择 id 为 radio 的 div 的子节点中不为 input 的所有子节点

input:not([type=radio]) 选择 input 节点中 type 不为 radio 的所有节点

爬虫示例

```
import java.io.IOException;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

public class TestDemo {
    public String baseUrl = "http://www.cnblogs.com/zhangfei/p/";

    public String pager = "?page=%s";

    public int getAllPageCount() {
        int count = 0;
        try {
            Document doc = Jsoup.connect(baseUrl).get();
            String countText = doc.select(
                    "#myposts>div.pager:nth-of-type(1)>.Pager").text();
            countText = countText.replaceFirst("\\D+(\\d+).*", "$1");
            count = Integer.valueOf(countText);
        } catch (IOException e) {
            e.printStackTrace();
        }
        return count;
    }

    public void crawler() {
```

```
        int count = this.getAllPageCount();
        for (int i = 1; i <= count; i++) {
            String url = baseUrl + String.format(pager, i);
            this.testJsop(url);
        }
    }

    public void testJsop(String url) {
        try {
            Document doc = Jsoup.connect(url).get();
            Elements element = doc.select("div.PostList a");
            for (Element e : element) {
                String text = e.text();
                String href = e.attr("href");
                System.out.println(text + " : " + href);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        TestDemo t = new TestDemo();
        t.crawler();
    }
}
```

上面的代码，如果通读了我的文档，我相信应该能看懂上面的代码，所以，我也没加注释了，大家自己去运行一下，不懂的 DEBUG 一下，再不懂的，可以问我。

Java 第十二天

json 概述

json 的概念及规范，没有必要在这里详述。之前我们讲了接口请求，也获取到了响应正文，如果响应正文是 json，且我们想去验证 json 里面的某个值是否正确，这时候，就需要解析 json 了。这里将会给大家演示用 fastjson 去解析 json，为什么是 fastjson？而不用 Google 的 Gson?fastjson 是阿里巴巴出品的，且 fastjson 传说中比 Gson 解析速度要快 30 倍左右，具体我没验证过，大家可以去验证一下。对我们测试人员而言，更重要的是 fastjson 的易用性非常好，很容易上手，不信请看下面的示例。需要 fastjson-*.jar。

fastjson 对象说明

- fastjson 中 JSONObject 类似于一个 Map
- fastjson 中 JSONArray 类似于一个 List

fastjson 解析示例

```
public class TestDemo {


    public void testJson(){
        String json = "{\"a\":[\"a1\",\"a2\",\"a1\"],"
                + "     \"cb\":{\"a\":1},"
                + "     \"d\":[\"a\",{\"a\":[1,20]},{\"a\":2},\"\"],"
                + "     \"e\":\"b\"}";
        JSONObject j = JSON.parseObject(json);//解析成一个JSONObject
        JSONArray a = j.getJSONArray("a");//将a的value解析成一个JSONArray
        System.out.println(a);//输出["a1","a2","a1"]
        JSONObject cb = j.getJSONObject("cb");//解析成一个JSONObject
        Integer cba = cb.getInteger("a");//获取cb下a的值
```

```
        System.out.println(cba);//输出1
    }
    public void testJsonObject(){
        String json = "{\"retCode\":200,\"retMsg\":\"success.\"}";
        RetInfo info = JSON.parseObject(json, RetInfo.class);//解析成一个
RetInfo对象
        System.out.println(info.getRetMsg());//输出success.
        String json1 =
"[{\"retCode\":200,\"retMsg\":\"success.\"},{\"retCode\":201,\"retMsg\":\"f
ail.\"}]";
        List<RetInfo> infos = JSON.parseArray(json1, RetInfo.class);//解析成
一个List<RetInfo>对象
        System.out.println(infos.get(1).getRetCode());//输出201
        /**
         * 如果不想写类似于RetInfo这种对象文件(也称为pojo)，那就可以用
TypeReference来解决
         * 这种方式在测试中用的普遍一些
         */
        List<Map<String, Object>> infos1 = JSON.parseObject(json1, new
TypeReference<List<Map<String, Object>>>(){});
        System.out.println(infos1.get(1).get("retMsg"));//输出fail.
        String json2 =
"{\"retCode\":200,\"retMsg\":\"success.\",\"data\":{\"name\":\"zhangsan\",\
"age\":18}}";
        JSONObject data = JSON.parseObject(json2).getJSONObject("data");//获
取data对象的JSONObject
        /**
         * 将data对象先转成一个json字符串后，再解析成一个map形式的数据结构
         * 这种方式在测试中也用的比较普遍
         */
        Map<String, Object> dataInfo = JSON.parseObject(data.toJSONString(),
new TypeReference<Map<String, Object>>(){});
        System.out.println(dataInfo.get("name"));//输出zhangsan
```

```
    }
    public static class RetInfo{
        private int retCode;
        private String retMsg;
        public int getRetCode() {
            return retCode;
        }
        public void setRetCode(int retCode) {
            this.retCode = retCode;
        }
        public String getRetMsg() {
            return retMsg;
        }
        public void setRetMsg(String retMsg) {
            this.retMsg = retMsg;
        }

    }

    public static void main(String[] args) {
        TestDemo t = new TestDemo();
        t.testJson();
        t.testJsonObject();
    }
}
```

关于 zson

zson 是我个人独立开发的一个专为测试人员打造的 json 解析 jar 包，根据给定的路径就能获取到相应的值，无需层层解析，且支持相对路径。相当的好用，不用不知道，一用爽到爆！具体大家可至 GITHUB 上去看：

https://github.com/zhangfei19841004/zson

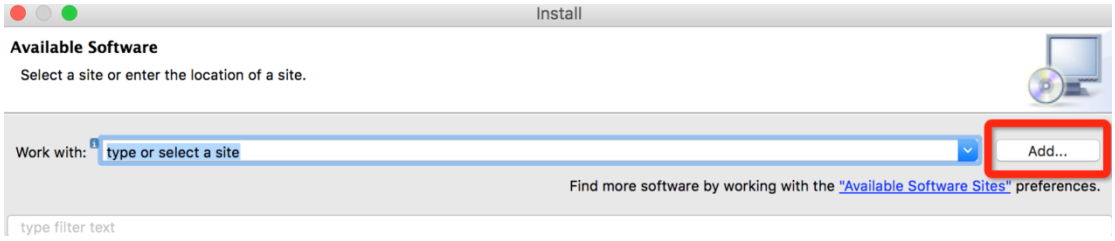
Java 第十三天

TestNg 概述

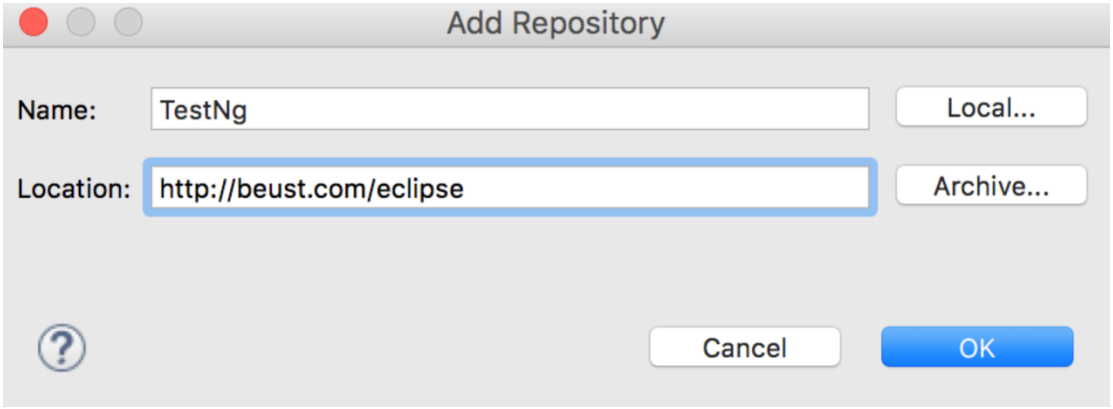
既然讲 java,作为一个测试人员，就不得不提到 TestNg,什么是 TestNg?TestNg 能帮我们做什么？大家可以注意到，在前面的代码示例中，要调用某个方法时，都是在 main 方法里面，new 一个实例对象，然后通过对象来调用方法，而我们在做接口测试时，如果写了一千个接口测试用例脚本，那么在运行时，就会要先 new 一千个对象，然后调用一千次！且每增加一个用例脚本或减少一个用例脚本，都会显得极其麻烦。那么有没有一种工具能帮助我们来管理用例脚本呢？对，就是 TestNg，TestNg 提供了一整套的解决方案，包括用例的组织，用例的管理，用例的运行，运行方式，报告的生成等，且提供出了非常丰富的接口扩展，下面我们来探索一下 TestNg 的一些运用方式吧！

TestNg 基本使用介绍

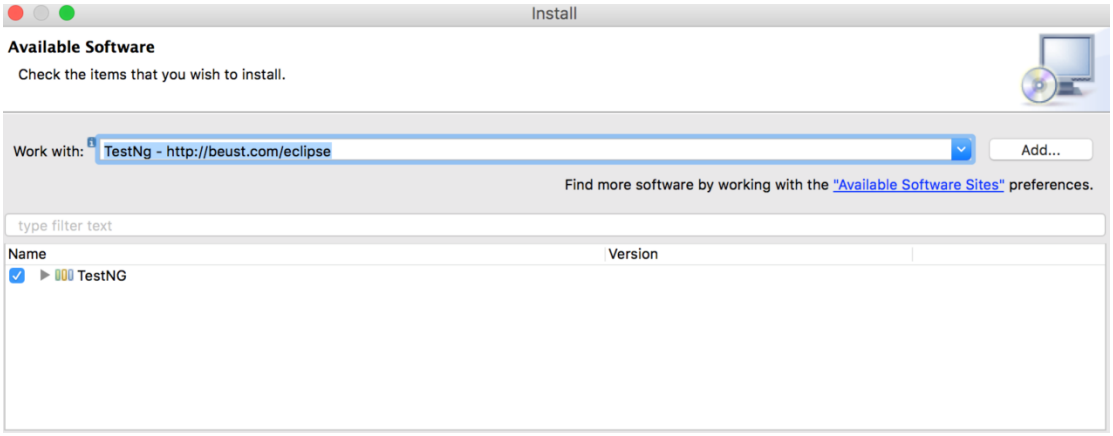
- 官方文档
  http://testng.org/doc/documentation-main.html
- eclipse 中安装 TestNg 插件
  当打开 eclipse 后，在菜单栏中 Help->Install New Software...



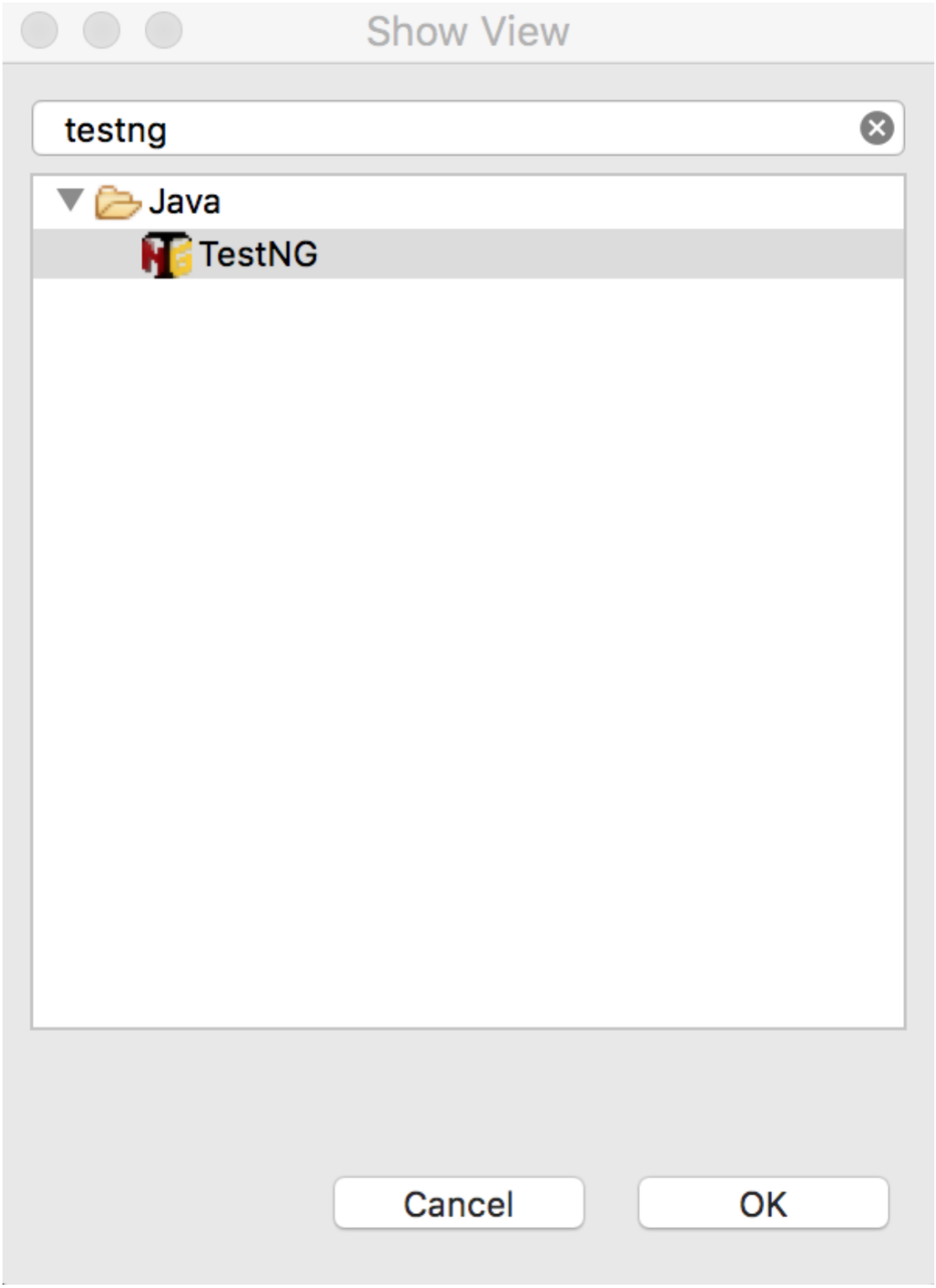
点击”Add...”



输入 Name 和 Location 后，点击”OK”



勾选 TestNG 后，点击”Next >”，在出现的界面中再次点击”Next >”，在下一个界面中，勾选”accept”后点击”Finish”，此时将会进入安装过程，过程会有点长，需要耐心等待，在安装过程中会出现一个提示，选择确定即可，等到出现需要重启的提示，然后重启 eclipse。重启后在菜单栏 Window->Show View->Other

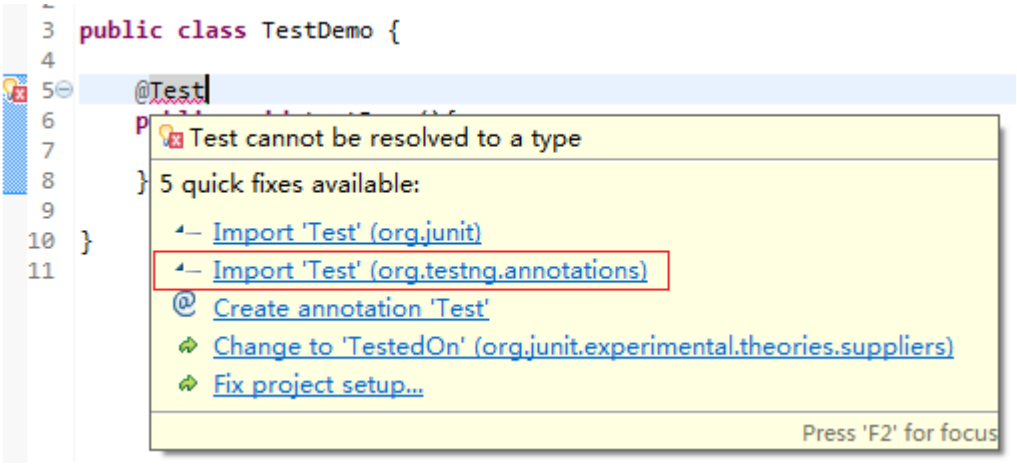


以上表示 TestNg 安装成功。

- 使用特点

  注解的形式来使用。前面专门讲过注解，大家可以再去温习下。

- @Test 标注在方法上面，表示该方法是一个测试方法。

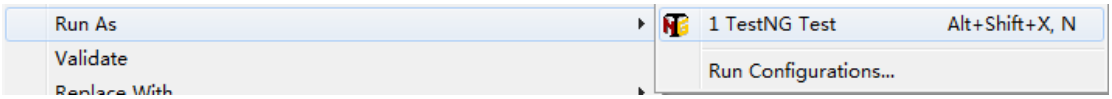


Import 时要注意一下，选择上图中红色标注出来的，第一个是 junit 的注解，第二个才是 TestNg 的注解，不要选择错了。代码如下：

```
import org.testng.annotations.Test;


public class TestDemo {


    @Test
    public void testDemo(){
        System.out.println("this is test method!");
    }


}
```

Eclipse 中运行：

在带有@Test 的 java 文件中，右键->Run As->TestNG Test



即可运行所有带有@Test 注解的方法。

- @BeforeMethod 标注在方法上面，表示该方法会在@Test 的测试方法运行之前被执行

```
public class TestDemo {


    @BeforeMethod
    public void testBeforeMethod(){
```

```
        System.out.println("this is before method!");
    }

    @Test
    public void testDemo(){
        System.out.println("this is test method!");
    }

    @Test
    public void testDemo1(){
        System.out.println("this is test method1!");
    }

}
```

运行输出：

```
this is before method!
this is test method!
this is before method!
this is test method1!
```

由此可以看出@BeforeMethod 的方法会在每一个@Test 方法之前被执行。

- @AfterMethod 标注在方法上面，表示该方法会在@Test 的测试方法运行之后被执行

```
public class TestDemo {

    @BeforeMethod
    public void testBeforeMethod(){
        System.out.println("this is before method!");
    }

    @Test
    public void testDemo(){
        System.out.println("this is test method!");
    }
```

```
    @Test
    public void testDemo1(){
        System.out.println("this is test method1!");
        System.out.println(1/0);
    }

    @AfterMethod
    public void testAfterMethod(){
        System.out.println("this is after method!");
    }

}
```

运行输出：

```
this is before method!
this is test method!
this is after method!
this is before method!
this is test method1!
this is after method!
```

由此可以看出@AfterMethod 的方法会在每一个@Test 方法之前被执行。在 testDemo1()方法中，System.out.println(1/0);这一句代码很明显会产生异常，但@AfterMethod 方法仍然被执行了，意味着@Test 方法不管是否有异常，@AfterMethod 方法始终会被执行。

- @BeforeClass 标注在方法上面，表示该方法会在所有的@Test 测试方法运行之前被执行，且只会被执行一次。

```
public class TestDemo {

    @BeforeClass
    public void testBeforeClass(){
        System.out.println("this is before class!");
    }
```

```
    @BeforeMethod
    public void testBeforeMethod(){
        System.out.println("this is before method!");
    }

    @Test
    public void testDemo(){
        System.out.println("this is test method!");
    }

    @Test
    public void testDemo1(){
        System.out.println("this is test method1!");
        System.out.println(1/0);
    }

    @AfterMethod
    public void testAfterMethod(){
        System.out.println("this is after method!");
    }

}
```

运行输出：

```
this is before class!
this is before method!
this is test method!
this is after method!
this is before method!
this is test method1!
this is after method!
```

由此可以看出@BeforeClass 的方法会在所有@Test 方法运行之前被执行。且只执行了一次。

- @AfterClass 标注在方法上面，表示该方法会在所有的@Test 测试方法运行之

后被执行，且只会被执行一次。

```
public class TestDemo {


    @BeforeClass
    public void testBeforeClass(){
        System.out.println("this is before class!");
    }


    @BeforeMethod
    public void testBeforeMethod(){
        System.out.println("this is before method!");
    }


    @Test
    public void testDemo(){
        System.out.println("this is test method!");
    }


    @Test
    public void testDemo1(){
        System.out.println("this is test method1!");
        System.out.println(1/0);
    }


    @AfterMethod
    public void testAfterMethod(){
        System.out.println("this is after method!");
    }


    @AfterClass
    public void testAfterClass(){
        System.out.println("this is after class!");
    }
```

```
}
```

运行输出：

```
this is before class!
this is before method!
this is test method!
this is after method!
this is before method!
this is test method1!
this is after method!
this is after class!
```

由此可以看出@AfterClass 的方法会在所有@Test 方法运行之后被执行。且只执行了一次。

- 备注

  @BeforeMethod，@AfterMethod，@BeforeClass，@AfterClass 是为了让我们制定一些执行策略的，比如在做自动化测试时，在每一个测试类被执行前，希望能备份一下数据库，且测试类执行完后，还原数据库，则可以用@BeforeClass，@AfterClass 就可以了。其实你也可以认为他们是一些公共操作或公共业务逻辑的提炼！

  以上介绍了 5 个注解，完全够大家的基本使用了，至于还有一些其它的注解，大家上官网去自行查阅并敲代码自行掌握。

TestNg 运行方式

上面介绍过 TestNg 在 eclipse 中如何运行，但也只是针对一个测试类而言的，如果我想只运行测试类中的某一个测试方法怎么做？或者一次运行多个测试类？再或者运行一个包下的所有测试类？既然有需求，就会有解决方案，TestNg 提供了一种 xml 配置文件的方式来解决了上面的问题。

- 运行多个测试类

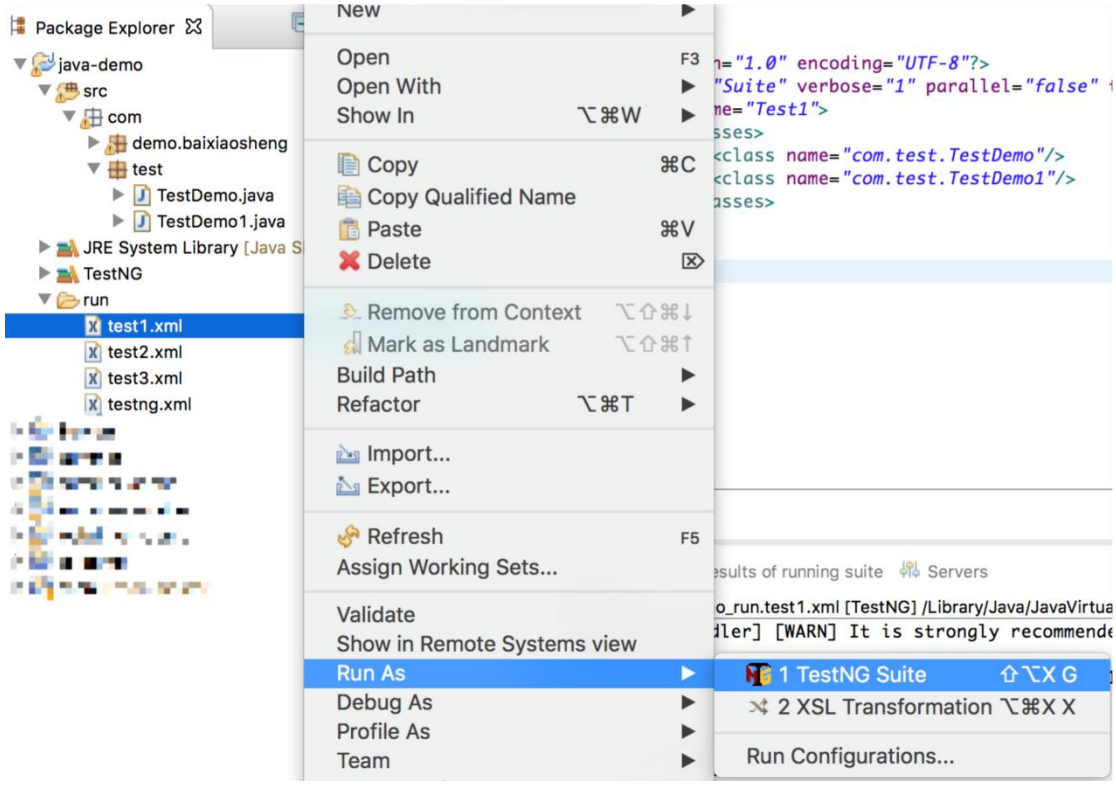
```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false"
thread-count="1">
    <test name="Test1">
        <classes>
            <class name="com.test.TestDemo"/>
```

```
            <class name="com.test.TestDemo1"/>
        </classes>
    </test>
</suite>
```

说明：

1. xml 文件的根结点为 suite，suite 中的属性所代表的意义：
   a) name: suite 的名称，多个 xml 的 suite 名称必须保持唯一
   b) verbose:表示测试报告中的显示内容的级别，0 到 10 个等级，0 为无，10 为最详细。
   c) parallel:表示是否多线程运行测试脚本
   d) thread-count:当 parallel 为 true 时，thread-count 的值才有意义，表示起几个线程去多线程执行测试脚本
2. suite 结点下面可以有多个 test 结点，每个 test 结点的名称必须唯一。
3. classes 结点表示测试类的集合。
4. class 结点表示运行的测试类，name 是要运行的测试类的全路径，可以有多个 class 结点，也就是一次可以执行多个测试类。

- 运行 TestNg 的 xml 配置文件



- 运行测试类中的部分测试方法

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<suite name="Suite" verbose="1" parallel="false"
thread-count="1">
    <test name="Test1" preserve-order="true">
        <classes>
            <class name="com.test.TestDemo">
                <methods>
                    <include name="testDemo" />
                </methods>
            </class>
        </classes>
    </test>
</suite>
```

说明：

1. methods 表式测试方法集。如果 methods 结点下不接任何结点，则表示运行所有的测试方法
2. 如果 methods 结点下接了 include，则表示只会运行 include 的测试方法，即 include 有几个就会运行几个测试方法
3. methods 结点下还可以接 exclude，则表示该测试类中，除了 exclude 的测试方法，其它的测试方法都会被执行
4. methods 结点下如果既接了 include，同时也接了 exclude，则以 include 为准，exclude 结点则会被忽略，即 exclude 结点不会产生任何作用。
5. test 结点中多了个属性 preserve-order，其值为 true 表示 methods 结点下有多个 include 时，按照 xml 中 include 的顺序来执行，如果 preserve-order 的值为 false 或者没这个属性，则会按照测试方法名称的升序顺序执行。

- 运行一个包下的所有测试类

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false"
thread-count="1">
    <test name="Test1">
        <packages>
            <package name="com.test" />
        </packages>
    </test>
```

```
</suite>
```

该 xml 表示运行一个包”com.test”下的所有的测试类中的测试方法。

Java 第十四天

数据驱动特点

数据驱动是指在脚本固定的情况下，根据数据的条数来决定脚本的运行次数，即有几组数据，脚本就会运行几遍。

数据驱动概述

数据驱动(Data Driven)，这里强调的是数据，驱动即执行，那么数据驱动就是根据数据来执行测试脚本。

- 场景：测试登录，分别用刘能和赵四的帐号去测试。

先写一个公共方法来描述登录的过程：(伪代码实现)

```
public boolean login(String username, String password){
    //do login
}
```

再到测试方法里面去调用这个 login 方法/函数：

```
public void test1(){
    login("liuneng","123456");
}

public void test2(){
    login("zhaosi","654321");
}
```

这样测试用例就写完了，执行 test1 与 test2 两个方法即可。但细心的你可能会发现 test1 与 test2 这两个测试方法里的方法体除了数据，其它完全一样，这就存在重构的空间了：

```
public boolean login(String[][] accounts){
    for(int i = 0; i<accounts.length; i++){
        //do login
    }
```

```
}

public void test(){
    String[][] accounts = [["liuneng","123456"],["zhaosi","654321"]];
    login(accounts);
}
```

经过重构后的代码，就有点数据驱动的意思了，根据 accounts 的 length 来决定 login 方法/函数体运行几次，这样维护起来就方便了，假如又有一个老王的帐号想用来测试，就不需要再加一个测试方法了，只需要：

```
String[][] accounts = [["liuneng","123456"],["zhaosi","654321"],["laowang","000000"]];
```

重构后的代码，是不是令你很激动？原来这就是数据驱动！别急，淡定，这还不是真的数据驱动，因为上面只有一个测试方法，最后执行完后，报告中记录的也是只有一个测试方法，而场景中：分别用刘能和赵四的帐号去测试，是希望在测试报告中有两个测试方法出现，显然上面的代码还不能满足我们的需求，于是进一步优化：

```
public boolean login(String username, String password){
    //do login
}

public void test(String username, String password){
    login(username,password);
}

public void executor(){
    String[][] accounts = [["liuneng","123456"],["zhaosi","654321"]];
    for(int i = 0; i<accounts.length; i++){
        test(accounts[i][0],accounts[i][1]);
    }
}
```

是的，离数据驱动原理真相越来越近了，上面多了个 executor 方法，这是啥？这就是车子的发动机引擎啊，就是测试脚本的执行引擎，让测试方法能够被执行起来，以及根据你所提供的测试数据的条数，决定测试方法的执行次数，并且报告中会显示是两个测试方法，这就是数据驱动。测试框架就是一个执行引擎，并

且测试框架都会支持数据驱动这一基本诉求，比如 TestNg 里的 dataProvider，junit 里的 Parameters 等。下面将为大家介绍 TestNg 里的 dataProvider 的用法。

TestNg 数据驱动

TestNg 的数据驱动也是以注解的形式来表达的：

```
public class TestData {

    @DataProvider(name="dataDemo")
    public Object[][] dataProvider(){
        return new Object[][]{{1,2},{3,4}};
    }

    @Test(dataProvider="dataDemo")
    public void testDemo(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }

}
```

说明：

1. TestNg 的数据驱动的提供数据的方法用@DataProvider 注解
2. @DataProvider 中的 name 属性表示该数据源的名称
3. @DataProvider 的方法要返回一个 Object[][]的二维数组，当然也可以是 Iterator<Object[]>的数据结构。
4. 在测试方法中要用到数据源，则要在@Test 中加上 dataProvider 属性，其值为数据源的名称。
5. Object[][]二维数组{{1,2},{3,4}}可以理解为有两组数据，分别为：{1,2}，{3,4}，所以测试方法会运行两次，每一次运行时，测试方法的第一个参数 a 对应这一组数据中的第一个，也就是 1 或者 3，第二个参数 b 则对应这一组数据中的第二个，也就是 2 或者 4，如果还有数据，则依次类推。当然数据源的每一组数据的个数要大于或等于测试方法的参数个数，否则会报错，且数据类

型要对应上，否则也会报错。

上面的示例中我们指定了@DataProvider 数据源的名称为 dataDemo，其实也可以不指定其名称，如果不指定其名称，则数据源的名称为该数据源方法的方法名，比如：

```
public class TestData {

    @DataProvider
    public Object[][] dataProvider(){
        return new Object[][]{{1,2},{3,4}};
    }

    @Test(dataProvider="dataProvider")
    public void testDemo(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }

}
```

上面的例子中，@DataProvider 数据源的方法与测试方法是在同一个测试类里，或者@DataProvider 数据源的方法放在测试类的父类中。但其实还有一种方式，@DataProvider 数据源的方法可以单独的放在一个类里，比如：

```
public class DataSource {

    @DataProvider
    public static Object[][] dataProvider(){
        return new Object[][]{{1,2},{3,4}};
    }

}

public class TestData {
```

```
    @Test(dataProvider="dataProvider",dataProviderClass=DataSou
rce.class)
    public void testDemo(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }

}
```

说明：

1. 在测试方法中可以指定一个数据源的类，用 dataProviderClass 来指定
2. 如果用了 dataProviderClass 指定数据源的类，则@DataProvider 数据源的方法必须是 static 的。

其实@DataProvider 数据源的方法，还可以提供一个参数 Method，这个 Method 是指要使用该数据源的测试方法的 Method 对象，比如：

```
public class TestData {

    @DataProvider
    public Object[][] dataProvider(Method method){
        //method对象指使用该数据源的测试方法的Method对象，这是java中的一
种反射
        System.out.println(method.getName());//输出testDemo
        return new Object[][]{{1,2},{3,4}};
    }

    @Test(dataProvider="dataProvider")
    public void testDemo(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }

}
```

很显然，有了这个 Method 对象后，就很方便我们扩展了，请看下面的例子：

先写一个类，将所有的数据都放在里面：

```
public class DataSource {
    /**
     * dataMap里有两个数据源，分别是testDemo与testDemo1
     * 根据测试方法的名称，来使用不同的数据源。
     * 比如testDemo方法就用数据源{{1,2},{3,4}}
     * testDemo1方法就用数据源{{5,6},{3,4}}
     * @return
     */
    public Map<String, Object[][]> dataSource(){
        Map<String, Object[][]> dataMap = new HashMap<String,
Object[][]>();
        Object[][] o1 = new Object[][]{{1,2},{3,4}};
        dataMap.put("testDemo", o1);
        Object[][] o2 = new Object[][]{{5,6},{7,8}};
        dataMap.put("testDemo1", o2);
        return dataMap;
    }

}
```

再结合到@DataProvider 数据源方法与测试方法中去：

```
public class TestData {

    @DataProvider
    public Object[][] dataProvider(Method method){
        DataSource data = new DataSource();
        Object[][] obj = data.dataSource().get(method.getName());
        return obj;
    }

    @Test(dataProvider="dataProvider")
    public void testDemo(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
```

```
    }

    @Test(dataProvider="dataProvider")
    public void testDemo1(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }

}
```

以上两个测试方法用到了同一个数据源@DataProvider，这样为我们以后写测试框架定下了基调。

enum 的使用

enum 是 java 中的一个关键字，中文翻译过来是枚举，先来看看用法：

```
public enum TestEnum {
    /**
     * 定义两组数据,分别是{200,"success."}与{400,"failed."}
     * 枚举的意思是根据定义好的数据，来生成对象，有几组数据，就生成几个对象
     */
    SUCCESS(200,"success."),
    FAIL(400,"failed.");

    private int retCode;

    private String retMsg;

    private TestEnum(int retCode, String retMsg) {
        this.retCode = retCode;
        this.retMsg = retMsg;
    }

    public int getRetCode() {
```

```
        return retCode;
    }

    public String getRetMsg() {
        return retMsg;
    }

    public static void main(String[] args) {
        System.out.println(TestEnum.SUCCESS.getRetCode());//输出
200
        System.out.println(TestEnum.FAIL.getRetMsg());//输出
failed.
    }

}
```

说明：枚举的意思是根据定义好的数据，来生成对象，有几组数据，就生成几个对象。上面的例子中生成了 SUCCESS 和 FAIL 两个对象。

联想到之前的数据驱动，是有几组数据测试方法就会执行几次，枚举是有几组数据就会生成几个对象，冥冥之中枚举与数据驱动好像有那么点联系，既然枚举已经定义好了，那么根据上面我们介绍到的知识点，把枚举与数据驱动联系起来。

```
public class DataSource {

    public Map<String, Object[][]> dataSource(){
        Map<String, Object[][]> dataMap = new HashMap<String,
Object[][]>();
        TestEnum[] te = TestEnum.values();//enum提供的方法，是把生成
的几组对象给放到一个数组里{TestEnum.SUCCESS,TestEnum.FAIL}
        Object[][] o1 = new Object[te.length][];//定义一个数据源，其
数据条数与TestEnum的数组条数一样
        for (int i = 0; i < o1.length; i++) {
            Object[] o1temp = new Object[]{te[i]};//将每一个TestEnum
再单独的放到一个数组里，o1temp为{TestEnum},
```

```
            o1[i] = o1temp;//{{TestEnum.SUCCESS},{TestEnum.FAIL}},
o1[0]就为{TestEnum.SUCCESS},枚举出来的TestEnum.SUCCESS与
TestEnum.FAIL都是一个TestEnum对象，只是对象里面的属性值不同
        }
        //循环添加后，数据结构为{{TestEnum.SUCCESS},{TestEnum.FAIL}}
        dataMap.put("testDemo", o1);
        Object[][] o2 = new Object[][]{{5,6},{7,8}};
        dataMap.put("testDemo1", o2);
        return dataMap;
    }

}

public class TestData {

    @DataProvider
    public Object[][] dataProvider(Method method){
        DataSource data = new DataSource();
        Object[][] obj = data.dataSource().get(method.getName());
        return obj;
    }

    @Test(dataProvider="dataProvider")
    public void testDemo(TestEnum te){
        System.out.println("retCode is: "+te.getRetCode() +"
retMsg is: "+te.getRetMsg());
    }

    @Test(dataProvider="dataProvider")
    public void testDemo1(int a, int b){
        int sum = a + b;
        System.out.println("this is sum: "+sum);
    }
```

```
}
```

以上对于对象放于二维数组中不懂的，请去翻阅前面的文档，有对二维数组的详细讲解。

关于@Parameters

这里所说的@Parameters 就 TestNg 的@Parameters，在前面我们介绍过用 TestNg 的 xml 配置文件来制定策略执行测试类，那 xml 配置文件与测试方法间如何进行参数传递？@Parameters 就是干这个事的，所以，@Parameters 只是参数的传递，并不是真正意义上的数据驱动，我个人对这个应用的不多，但还是有必要给大家介绍一下用法。

```
public class TestDemo {


    @Parameters({"a","b"})
    @Test
    public void testDemo(int a, int b){
        int sum = a+b;
        System.out.println("this is sum: "+sum); //输出3
    }


}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">
    <parameter name="a"  value="1"/>
    <parameter name="b"  value="2"/>
    <test name="Test1">
        <classes>
            <class name="com.test.demo.TestDemo" />
        </classes>
    </test>
</suite>
```

说明：

1. @Parameters 注解能用在测试方法或者@Before/@After/@Factory 上面。
2. @Parameters 中的字符串是在 xml 中的 parameter 结点的 name。
3. 在 xml 中的 parameter 结点的 value 就是该 name 对应的具体的参数值。
4. @Parameters 注解的{}中的参数的顺序，对应其标注的方法上的参数的顺序。

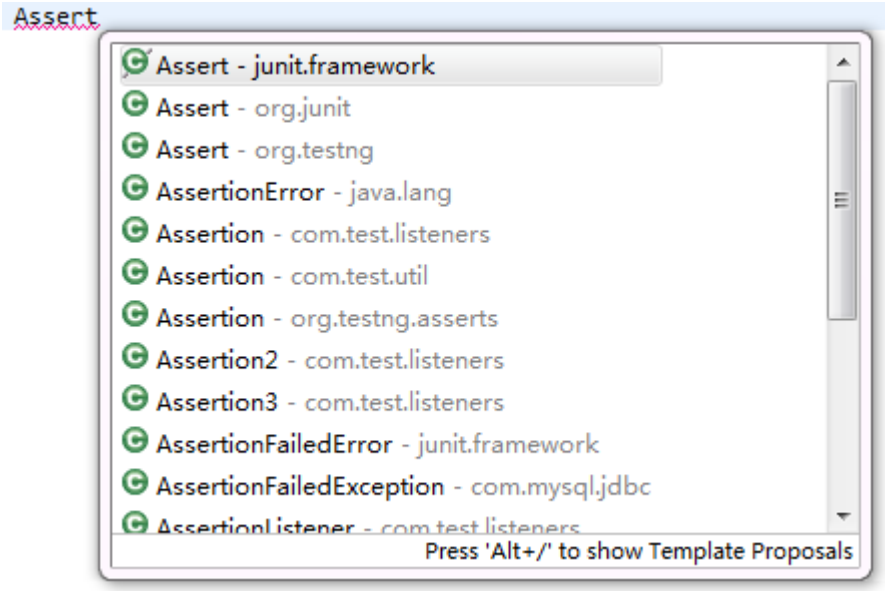
Java 第十五天

断言介绍

在做自动化测试中，断言是个很重要的概念，断言也就是检查点，重在判断我们通过测试脚本获取到的值与期望值是否相等，如果相等，则代表断言成功，程序会继续往下执行，如果不相等，则代表断言失败，程序就会在断言失败处中止。看一段断言的代码：

```
public class Test4 {
    @Test
    public void testAssert1(){
        System.out.println("开始断言");
        Assert.assertEquals("1", "1");
        System.out.println("结束断言");
    }
    @Test
    public void testAssert2(){
        System.out.println("开始断言");
        Assert.assertEquals(1, 2,"比较两个数是否相等：");
        System.out.println("结束断言");
    }
}
```

用到了 Assert 类，我们这里的 Assert 类里 testng 里的 Assert 类，在输入 Assert 后，Alt+/，在如下图中选择第三个 org.testng，这样就正确的 import 了。



在 testAssert1 方法中，assertEquals 里面有两个参数，前一个是代表实际值，后一个是代表期望值，在 testAssert2 方法中，assertEquals 方法里有三个参数，第一个是实际值，第二个是期望值，第三个测试用户自定义的一段字符串，这第三个参数可以写，也可以不写。如果断言失败后，这段字符串会输出。从上面的代码中，可以看出 assertEquals 方法里的实际值与期望值，即可以是 int 型的，也可以是 String 类型的，当然，其它类型的也可以，也就是说 java 的数据类型的值都可以当作参数传给 assertEquals 方法中去,包括 List,Map,数组等都可以用这个方法进行比较。我们来看一下运行后的输出结果：

```
开始断言
结束断言
开始断言
PASSED: testAssert1
FAILED: testAssert2
java.lang.AssertionError: 比较两个数是否相等： expected [2] but found [1]
    at org.testng.Assert.fail(Assert.java:94)
```

testAssert1运行通过，testAssert2在输出”开始断言”后，没有输出”结束断言”，这是因为断言失败后，该测试方法后面的语句就不执行了，就跳出测试方法并且标注该测试方法为失败了，且会输出失败的信息：

```
java.lang.AssertionError: 比较两个数是否相等： expected [2] but found [1]
```

这一句中，AssertionError 是失败的异常类，”比较两个数是否相等：”这一句是我们在 assertEquals 方法中自定义的第三个参数，expected [2] but found [1]

这是断言类给输出的，标注出了期望值与实际值。at org.testng.Assert.fail(Assert.java:94)这一种是指在程序的哪一行失败了。

Assert 类还有很多的静态断言方法去使用，不光只是基本数据类型的比较，还有

- Assert.assertFalse(condition);//判断传进去的参数是为是 false
- Assert.assertNotEquals(actual1, actual2);//判断实际值与期望值是否不相同
- Assert.assertNotNull(object);//判断传进去的参数值是否不为 null
- Assert.assertNotSame(actual, expected, message);//same 与 equals 是不一样的，equals 是比较实际值与期望值的值是否相等，same 中实际值与期望值如果是基本数据类型与 String 类型的值，则是比较值是否相同，如果是非基本数据类型与非 String 的，则是比较实际值与期望值的引用地址是否一样，举个例子：

```
String[] string1 = { "1", "2" };
String[] string3 = string1;
String[] string4 = { "1", "2" };
Assert.assertSame(string1,string3,"string1和string3不相同");
Assert.assertSame(string1, string4, "string1和string4不相等");
```

上面的代码中，string1与string3就是same的，而string1与string4就是notsame的

- Assert.assertNull(object, message);//判断传进来的参数值是否为 null
- Assert.assertSame(actual, expected);//与 assertNotSame 对应
- Assert.assertTrue(condition);//判断传进来的参数是否为 true

虽然 Assert 类提供了很多的断言的方法，但一般情况下我们只需要使用 assertEquals 这一个方法即可，所以在下面的封装中，也会只封装 assertEquals 这一个方法。

断言封装

到此，对断言应该有了一个认识了，但如果断言失败时不希望退出，直到把整个测试脚本运行完成后，再整体判断是否有断言失败的地方，这个就需要我们自己封装了，其实封装的原理很简单，断言失败也就是抛出了一个异常，之前介绍过异常捕捉，所以很容易就能联想到用 try/catch 去解决，于是代码如下：

```
public class Assertion {
```

```
    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){

        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){

        }
    }

}
```

增加了一个类，里面加了两个静态方法 verifyEquals,在方法体里面调用了 Assert.assertEquals，这样，当 Assert.assertEquals 断言失败后，异常就会 catch 住了，就不会跳出测试方法了，我们来看调用的方式：

```
    @Test
    public void testAssert3(){
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
        System.out.println("结束断言3");
    }
```

运行后，发现问题又来了，testAssert3 这个虽然断言后没有跳出测试方法，但是这个断言理论上是失败的，但被 TestNg 判断为了运行成功，这样就没达到目的，我们要在最后让 TestNg 判断 testAssert3 方法为失败，于是解决办法如下：

```
public class Assertion {
```

```
    public static boolean flag = true;

    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){
            flag = false;
        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){
            flag = false;
        }
    }

}
```

在里面加了一个标志符号，如果 catch 住异常后，就把标志符号赋为 false,这样在测试方法的最后判断一个标志符号是否为 true,如果为 true,测试整个断言过程中没有错误，如果为 false，则表示在脚本运行过程中曾经有断言失败过，则测试方法最后就被判定为失败：

```
    @Test
    public void testAssert3(){
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
        System.out.println("结束断言3");
        Assert.assertTrue(Assertion.flag);
    }
```

这时候执行，就会发现 testAssert3 方法在断言失败后也全部运行完了，且最后也被判定为了运行失败。但如果有多个测试方法，我们来看下面这种情况：

```
public class TestAssertion {

    @Test
    public void testAssert3(){
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
        System.out.println("结束断言3");
        Assert.assertTrue(Assertion.flag);
    }

    @Test
    public void testAssert4(){
        System.out.println("开始断言4");
        Assertion.verifyEquals(2, 2, "比较两个数是否相等：");
        System.out.println("结束断言4");
        Assert.assertTrue(Assertion.flag);
    }

}
```

上面的代码中，有两个测试方法，TestNg 默认按照测试方法名称的升序顺序进行执行，先执行 testAssert3 再执行 testAssert4，很明显 testAssert3 方法会失败，但 testAssert4 方法会测试通过，我们来看下执行后的测试结果：

```
===============================================
    Default test
    Tests run: 2, Failures: 2, Skips: 0
===============================================
```

两个方法都执行失败了。这是因为 `Assertion.flag` 这个静态变量在 testAssert3 方法中已经被赋值为 false 了，所以在 testAssert4 方法中 `Assert.assertTrue(Assertion.flag);` 这一句仍然被断言失败，于是改一下代码：

```
public class Assertion {

    public static boolean flag = true;
```

```
    public static void begin(){
        flag = true;
    }

    public static void end(){
        Assert.assertTrue(flag);
    }

    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){
            flag = false;
        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){
            flag = false;
        }
    }

}
public class TestAssertion {

    @Test
    public void testAssert3(){
        Assertion.begin();//断言开始，即将flag赋值为true
        System.out.println("开始断言3");
        Assertion.verifyEquals(1, 2, "比较两个数是否相等：");
```

```
        System.out.println("结束断言3");
        Assertion.end();//断言结束，即判断测试方法是否通过
    }

    @Test
    public void testAssert4(){
        Assertion.begin();//断言开始，即将flag赋值为true
        System.out.println("开始断言4");
        Assertion.verifyEquals(2, 2, "比较两个数是否相等：");
        System.out.println("结束断言4");
        Assertion.end();//断言结束，即判断测试方法是否通过
    }

}
```

以上代码就能符合我们的要求了，当然你可以把 Assertion.begin()放在@BeforeMethod 里面去，但是 Assertion.*end*()不能放在@AfterMethod 里，这是因为 Assertion.end()是在当前方法里抛出一个 Error 异常，如果放在@AfterMethod 里，则会被认为@AfterMethod 方法产生了异常,同时会判定@AfterMethod 方法运行失败，而测试方法则没有被判定，这样就又不符合我们的要求了。所以目前看来 Assertion.end()一定要加在测试方法的最后，以后了，会介绍如何利用 TestNg 的监听来将 Assertion.end()这一句代码给移出测试方法。

备注：上面的这个 Assertion 类中的变量 flag 是非线程安全的，如果要多线程执行，会产生问题，请参考以前的文档中 ThreadLocal 的使用，并自行加以改良吧。

Java 第十六天

Log4j 概述

在 java 中，Log4j 很重要。作为一个测试，会查看 log 是一个很重要的技能，同开发一样，在测试脚本中，打上 log 会让我们查找问题定位问题更加的快速，良好的 log 会完整的记录下整个运行过程。所以，我们也很有必要掌握 Log4j 的使用。在此之前，我们需要准备好 jar 包 log4j-*.jar

Log4j 使用

Log4j 的使用非常简单，配置文件加上 new 一个 logger 对象即可：

```
public class TestLog {


    private Logger logger = Logger.getLogger(this.getClass());//定义一个logger
对象
    public void testLog(){
        logger.error("this is error log!");//提交error的log
        logger.info("this is info log!");//提交info的log
        logger.debug("this is debug log!");//提交debug的log
        logger.warn("this is warn log!");//用的少，可以忽略
        logger.fatal("this is fatal log!");//用的少，可以忽略
    }
}
```

我们可以看到，logger 对象只是把 log 提交了，提交后，输出到哪里？是控制台？还是某个文件？这时候就需要依赖配置文件了。Log4j 的配置文件名称一定要为:log4j.properties(当然也可以为 log4j.xml)，且配置文件一定要存放在 resource 文件夹下，非 maven 工程的 resource 文件夹默认就是 src 文件夹，maven 工程的 resource 文件夹默认是 src/main/resources 文件夹，当然 resource 文件夹也可以自己定义。以下的配置示例中都是以 log4j.properties 作为配置示例，先看一段配置：

```
log4j.rootLogger=INFO, stdout, fileout
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.appender.fileout=org.apache.log4j.FileAppender
log4j.appender.fileout.File=d:/test.log
log4j.appender.file.DatePattern=yyyy-MM-dd'.log'
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n
```

说明：

- log4j.rootLogger 表示定义一个根 log，所有的 log 会默认提交到根 log。后面的值会以逗号分割，INFO 是指根 log 的级别，一定要放在第一个，且一般为(ERROR/INFO/DEBUG)。stdout 与 fileout 是指定义根 log 要输出的地方。
- log 级别的优先级为：ERROR>INFO>DEBUG，常用这三个，其它的级别自行查阅。当根 log 定义的级别为 INFO 时，优先级大于或等于 INFO 的都会被根 log 接收，比如:

  `logger.info("this is info log!");logger.error("this is error log!");`

  这两句 log 就会被根 log 接收。但是 `logger.debug("this is debug log!");`这一句就不会被根 log 接收。
- log4j.appender.stdout 与 log4j.appender.fileout 是定义接收的 log 的处理方式，stdout 是在控制台输出，fileout 是在文件中输出，其中 stdout 与 fileout 这两个名称都是自己定义的，且必须在 log4j.rootLogger 中加上才能生效。
- stdout 与 fileout 中的配置项，我觉得大家看一眼也能明白，对于 ConversionPattern 后面的值，大家去查阅一下，也就 OK 了，这里不细讲，我们主要还是讲 log4j 的使用技巧。

以上的根 log 的级别定义为了 INFO，但我们希望把 ERROR 的 LOG 存放在一个单独的文件里，如何办？很显然不能把根 log 的级别改为 ERROR，这样 INFO 的 log 就不能被接收了，或者再添加一个 error 的 appender？但这个 error 的 appender 同样会接收 INFO 的 log。于是就需要用到 Threshold 了：

```
log4j.rootLogger=INFO, stdout, fileout, error
```

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.appender.fileout=org.apache.log4j.FileAppender
log4j.appender.fileout.File=d:/test.log
log4j.appender.file.DatePattern=yyyy-MM-dd'.log'
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.appender.error = org.apache.log4j.DailyRollingFileAppender
log4j.appender.error.File =d:/test_error.log
log4j.appender.error.Threshold = ERROR
log4j.appender.error.DatePattern=yyyy-MM-dd'.log'
log4j.appender.error.layout = org.apache.log4j.PatternLayout
log4j.appender.error.layout.ConversionPattern==%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n
```

说明：

- Threshold 后面的值也是一个 log 级别
- Threshold，可以理解为过滤，即根 log 接收到的 log 再次进行过滤。上面示例中 Threshold 的值为 ERROR，表示根 log 接收的 log 中级别为 ERROR 的，同时会写进 d:/test_error.log 文件中。也就是说 Threshold 的级别要大于或等于根 log 的级别才有意义，否则就失去了意义了，因为比根 log 级别低的 log 根本不会被根 log 所接收。
- 上面示例中 logger.error("this is error log!");这一句，会在控制台输出，会输出到 d:/test.log 里，同时也会输出到 d:/test_error.log 里。

既然有根 log，相应的就会存在自定义的 log 了，比如：

```
log4j.rootLogger=INFO, stdout, fileout, error

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
```

```
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.appender.fileout=org.apache.log4j.FileAppender
log4j.appender.fileout.File=d:/test.log
log4j.appender.file.DatePattern=yyyy-MM-dd'.log'
log4j.appender.fileout.layout=org.apache.log4j.PatternLayout
log4j.appender.fileout.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.appender.error = org.apache.log4j.DailyRollingFileAppender
log4j.appender.error.File =d:/test_error.log
log4j.appender.error.Threshold = ERROR
log4j.appender.error.DatePattern=yyyy-MM-dd'.log'
log4j.appender.error.layout = org.apache.log4j.PatternLayout
log4j.appender.error.layout.ConversionPattern =%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n

log4j.logger.TEST_DEMO=INFO,demo
log4j.appender.demo=org.apache.log4j.FileAppender
log4j.appender.demo.File=d:/test_demo.log
log4j.appender.demo.layout=org.apache.log4j.PatternLayout
log4j.appender.demo.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n
```

说明：

- 自定义了一个 log：TEST_DEMO，其关联的 appender 为 demo。其定义 log 的方式与根 log 的定义方式一样。
- 上面示例中 `logger.info("this is info log!");`这一句，会在控制台输出，会输出到 `d:/test.log` 里，同时也会输出到 `d:/test_demo.log` 里。
- 自定义的 log 所接收到的 log 信息，如果符合根 log 接收的条件，默认也会被根 log 所接收。

上面的例子中，自定义的 log 默认会被根 log 所接收，假如希望自定义的 log 不被根 log 接收，只需要加一句配置即可：

```
log4j.logger.TEST_DEMO=INFO,demo
log4j.appender.demo=org.apache.log4j.FileAppender
log4j.appender.demo.File=d:/test_demo.log
log4j.appender.demo.layout=org.apache.log4j.PatternLayout
log4j.appender.demo.layout.ConversionPattern=%d{yyyy-MM-dd
HH:mm:ss} %c : %m%n
log4j.additivity.TEST_DEMO=false
```

也就是将自定义 log 的 additivity 设置为 false，则自定义 log 接收到的 log 信息不会再被根 log 所接收。

Log4j 对象生成

前面我们提了一下 log4j 对象的生成方式：

```
private Logger logger = Logger.getLogger(this.getClass());
```

需要注意的是 Logger.getLogger()里面的参数，比如 this.getClass()，就会用当前对象的 class 的名称:com.test.TestLog 去配置文件中找名称为 com.test.TestLog 的 log，如果没有这个自定义的 com.test.TestLog 的 log，则这个 logger 对象所提交的 log 会被根 log 所处理。所以如果想要用自定义的 log，这样就可以了：

```
private Logger logger = Logger.getLogger("TEST_DEMO");
```

总结：在生成 logger 对象时，会根据 getLogger 方法里的参数去配置文件里找对应的 log，如果有，就会按照配置文件中定义的 log 的配置去处理 log 信息，如果没有，则会被配置文件中的根 log 所处理。

Log4j 的封装

Log4j 的使用已经非常方便了，但我们为什么还要封装？当我们通过 TestNg 跑完测试脚本，纵观整个测试报告，发现没有任何 log 的输出信息，Log4j 可以输出到我们定义的任何地方，但却无法与 TestNg 结合。如果我们希望 log 也能被

输出到 TestNg 的报告中，那就必须进行封装了，这就是我们为什么在 Log4j 已经如此方便的情况下再进行封装的原因。

如果想要把 log 输出到 TestNg 的报告中，则就需要用到 TestNg 中的：Reporter.log();所以只需要把 Log4j 与 Reporter.log()结合起来，就能达到我们的目的了：

```
public class Log {

    private Logger logger;

    public Log(Class<?> clazz){
        logger = Logger.getLogger(clazz);
    }

    public Log(String s){
        logger = Logger.getLogger(s);
    }

    public Log(){
        logger = Logger.getLogger("");
    }

    public void info(Object message){
        logger.info(message);
        this.testngLogOutput(message);
    }

    public void error(Object message){
        logger.error(message);
        this.testngLogOutput(message);
    }

    public void warn(Object message){
        logger.warn(message);
        this.testngLogOutput(message);
```

```
    }

    public void debug(Object message){
        logger.debug(message);
        this.testngLogOutput(message);
    }

    private void testngLogOutput(Object message){
        Reporter.log(message.toString());
    }

}
```

如此封装之后，我们在用 log 时，只需要：

```
public class TestLog {

    private Log log = new Log(this.getClass());

    public void testLog(){
        log.error("this is error log!");
        log.info("this is info log!");
        log.debug("this is debug log!");
    }

}
```

上述代码调用完 testLog 方法后，请大家分别去配置文件中定义好的地方及报告里查看 log 的输出情况。

Java 第十七天

属性文件概述

Java 的属性文件是以.properties 结尾的文件，文件内容的每一行都是一个键值对，以第一个"="号分割，等号左边是 key，右边是 value。正是因为这种键值对的数据格式，所以，属性文件一般就在配置文件上。比如下面这个 config.properties 文件：

```
base.url=http://www.baidu.com
log.dir=/home/logs
download.dir=/home/files
```

属性文件解析

jdk 中自带解析属性文件的类 Properties 类，不需要第三方 jar 包就能解析属性文件。将上面的 config.properties 文件放在工程下面的 config 文件夹下面。解析如下：

```
public class PropertiesUtil {

    public void getProperties(){
        File file = new File("config/config.properties");//根据路径生成File对象
        Properties p = new Properties();//生成Properties对象
        InputStream in = null;
        try {
            in = new FileInputStream(file);//将File对象转成流对象
            p.load(in);//加载流至Properties对象
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        }finally {
            try {
                if(in!=null){
                    in.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        System.out.println(p.getProperty("base.url"));//输出
http://www.baidu.com
        System.out.println(p.getProperty("base.url1"));//输出null
        /**
         * 第二个参数是指getProperty方法返回的默认值，
         * 如果在config.properties文件中没有base.url1这个key,那么将输
出默认值
         * 也就是会输出http://cn.bing.com/
         */

    System.out.println(p.getProperty("base.url1","http://cn.bin
g.com/"));//输出http://cn.bing.com/
    }

    public static void main(String[] args) {
        PropertiesUtil p = new PropertiesUtil();
        p.getProperties();
    }

}
```

以上的代码，当然可以做成一个通用的方法：

```
public class PropertiesUtil {
```

```
    private static Properties p = null;

    private static String path = "config/config.properties";

    private static void loadProperties(){
        if(p==null){
            p = new Properties();
            File file = new File(path);
            InputStream in = null;
            try {
                in = new FileInputStream(file);
                p.load(in);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }finally {
                try {
                    if(in!=null){
                        in.close();
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    public static String getProperties(String key){
        PropertiesUtil.loadProperties();
        return p.getProperty(key);
    }
```

```
    public static String getProperties(String key, String
defaultValue){
        PropertiesUtil.loadProperties();
        return p.getProperty(key, defaultValue);
    }

    public static void main(String[] args) {

    System.out.println(PropertiesUtil.getProperties("base.url")
);
    }

}
```

static 块

上面提到了属性文件可以作为配置文件，配置文件的特点是在代码的任何地方都可以用，所以，配置文件的 Properties 变量应该是静态的，那么在使用配置文件里面的值时，是否需要每用一次就去把配置文件读一遍？显然是不需要的，像上面的代码一样，用个单例模式即可。现在隆重介绍一下 static 块，其特点有：

1. 块里要用到的变量，都必须是静态的
2. 块会在所在类被加载时就执行，且只会执行一次，就算后面该静态块所在的类被使用多次，块也不会再被执行。

正因为块有以上的特点，且完全符合配置文件的特点，所以 static 块是为配置文件而生的：

```
public class PropertiesUtil {

    private static Properties p = null;

    private static String path = "config/config.properties";

    static{
        p = new Properties();
```

```
        File file = new File(path);
        InputStream in = null;
        try {
            in = new FileInputStream(file);
            p.load(in);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }finally {
            try {
                if(in!=null){
                    in.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    public static String getProperties(String key){
        return p.getProperty(key);
    }

    public static String getProperties(String key, String
defaultValue){
        return p.getProperty(key, defaultValue);
    }

    public static void main(String[] args) {

    System.out.println(PropertiesUtil.getProperties("base.url")
);
```

```
    }

}
```

虽然用静态块与上面用单例模式的代码的效果是一样的，但 `static` 块明显在线程安全，代码的优雅性方面要高大上很多，推荐使用 `static` 块来加载配置文件。

Java 第十八天

操作 MySQL

- 所需 jar 包

  mysql-connector-java-*.jar

- 基本操作

```
public class ConnectMySQL {


    public static void execute(){
        try {
            Class.forName("com.mysql.jdbc.Driver").newInstance();//生成
Driver对象
            Connection conn =
DriverManager.getConnection("jdbc:mysql://127.0.0.1:3306/test_demo?useUnico
de=true&characterEncoding=UTF8","root","root");//产生连接对象Connection
            Statement stmt = conn.createStatement();//产生发送SQL指令的对象
Statement
            ResultSet rs = stmt.executeQuery("select * from demo");//执行查询
语句
            while(rs.next()){
                int id = rs.getInt(1);
                String name = rs.getString(2);
                String email = rs.getString(3);
                String contact = rs.getString(4);
                System.out.println("id: "+id+" name: "+name+" email: "+email+"
contact: "+contact);
            }
            stmt.executeUpdate("insert into demo(name,email,contact)
values('b','b','b')");//执行添加语句
            stmt.executeUpdate("update demo set name='a1' where id = 1");//
```

执行更新语句

```
            stmt.executeUpdate("delete from demo where id = 3");//执行删除语句
            stmt.close();//关闭
            conn.close();//关闭
        }  catch (SQLException e) {
            e.printStackTrace();
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        ConnectMySQL1.execute();
    }

}
```

- 封装操作

```
public class ConnectMySQL {

    public static String driver = "com.mysql.jdbc.Driver";

    private static String host;

    private static String user;

    private static String pwd;

    private static Connection conn = null;
```

```
    private static Statement stmt = null;

    public static void connect(String host, String user, String pwd) {
        ConnectMySQL.close();
        ConnectMySQL.host = host;
        ConnectMySQL.user = user;
        ConnectMySQL.pwd = pwd;
    }

    /**
     * @param sql
     * @return 字段名与数据组成的List,
如:[{"name"="zhangsan","phone"="1234"},{"name"="lisi","phone"="1234"}]
     */
    public static synchronized List<Map<String, String>> query(String sql) {
        return ConnectMySQL.result(sql);
    }

    /**
     * 用于进行insert update delete操作
     * @param sql
     */
    public static synchronized void update(String sql) {
        ConnectMySQL.resultUpdate(sql);
    }

    public static synchronized void close() {
        try {
            if (stmt != null) {
                stmt.close();
                stmt = null;
            }
            if (conn != null) {
```

```
                conn.close();
                conn = null;
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    private static void connectMySQL() {
        try {
            Class.forName(driver).newInstance();
            conn = (Connection) DriverManager.getConnection("jdbc:mysql://"
                    + host + "?useUnicode=true&characterEncoding=UTF8", user,
                    pwd);
        } catch (InstantiationException e) {
            e.printStackTrace();
        } catch (IllegalAccessException e) {
            e.printStackTrace();
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }

    }

    private static void statement() {
        if (conn == null) {
            ConnectMySQL.connectMySQL();
        }
        try {
            stmt = (Statement) conn.createStatement();
        } catch (SQLException e) {
```

```
            e.printStackTrace();
        }
    }

    private static ResultSet resultSet(String sql) {
        ResultSet rs = null;
        if (stmt == null) {
            ConnectMySQL.statement();
        }
        try {
            rs = stmt.executeQuery(sql);
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return rs;
    }

    private static void resultUpdate(String sql) {
        if (stmt == null) {
            ConnectMySQL.statement();
        }
        try {
             stmt.executeUpdate(sql);
         } catch (SQLException e) {
             e.printStackTrace();
         }
    }

    private static List<Map<String, String>> result(String sql) {
        ResultSet rs = ConnectMySQL.resultSet(sql);
        List<Map<String, String>> result = new ArrayList<Map<String,
String>>();
        try {
```

```
            ResultSetMetaData md = rs.getMetaData();
            int cc = md.getColumnCount();
            while (rs.next()) {
                Map<String, String> columnMap = new HashMap<String, String>();
                for (int i = 1; i <= cc; i++) {
                    columnMap.put(md.getColumnName(i), rs.getString(i));
                }
                result.add(columnMap);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return result;
    }

    public static void main(String[] args) throws SQLException {
        ConnectMySQL
                .connect("192.168.1.1/test", "test", "test");
        List<Map<String, String>> rs = ConnectMySQL
                .query("SELECT * from test");
        System.out.println(rs.get(0).get("test"));
        ConnectMySQL.close();
    }
}
```

操作 Excel

- 所需 jar 包

  poi-*.jar

  poi-ooxml.jar

  poi-ooxml-schemas.jar

  xmlbeans.jar

- 基本操作

```
public class ExcelReader {


    public static void excelAction(String filePath){
        File file = new File(filePath);
        try {
            FileInputStream in = new FileInputStream(file);
            Workbook workBook = WorkbookFactory.create(in);
            Sheet sheet = workBook.getSheet("Sheet1");
            int numOfRows = sheet.getLastRowNum()+1;
            Row row = sheet.getRow(0);
            int numOfColumn = row.getLastCellNum();
            System.out.println(numOfRows+" "+numOfColumn);
            Cell cell = row.getCell(0);
            String v = cell.getStringCellValue();
            System.out.println(v);
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (InvalidFormatException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }


    public static void main(String[] args) {
        ExcelReader1.excelAction("c:/tools/workspace/test.xlsx");
    }


}
```

- 封装操作

```
public class ExcelReader {
    private String filePath;
```

```
    private String sheetName;
    private Workbook workBook;
    private Sheet sheet;
    private List<String> columnHeaderList;
    private List<List<String>> listData;
    private List<Map<String,String>> mapData;
    private boolean flag;

    /**
     * 所需jar包：
poi-3.8.jar,poi-ooxml.jar,poi-ooxml-schemas.jar,xmlbeans.jar
     * 提供解析excel，兼容excel2003及2007+版本
     * @param filePath excel本地路径
     * @param sheetName excel的sheet名称
     */
    public ExcelReader(String filePath, String sheetName) {
        this.filePath = filePath;
        this.sheetName = sheetName;
        this.flag = false;
        this.load();
    }

    /**
     * 加载EXCEL文件内容，产生WorkBook对象，再产生Sheet对象
     */
    private void load() {
        FileInputStream inStream = null;
        try {
            inStream = new FileInputStream(new File(filePath));
            workBook = WorkbookFactory.create(inStream);
            sheet = workBook.getSheet(sheetName);
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }finally{
            try {
                if(inStream!=null){
                    inStream.close();
                }
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * 根据cell对象，来取得每个cell的值，所有的值的数据类型都转化为了String类型
     * @param cell Cell对象
     * @return
     */
    private String getCellValue(Cell cell) {
        String cellValue = "";
        DataFormatter formatter = new DataFormatter();
        if (cell != null) {
            switch (cell.getCellType()) {
                case Cell.CELL_TYPE_NUMERIC:
                    if (DateUtil.isCellDateFormatted(cell)) {
                        cellValue = formatter.formatCellValue(cell);
                    } else {
                        double value = cell.getNumericCellValue();
                        int intValue = (int) value;
                        cellValue = value - intValue == 0 ?
String.valueOf(intValue) : String.valueOf(value);
                    }
                    break;
                case Cell.CELL_TYPE_STRING:
                    cellValue = cell.getStringCellValue();
```

```
                    break;
                case Cell.CELL_TYPE_BOOLEAN:
                    cellValue = String.valueOf(cell.getBooleanCellValue());
                    break;
                case Cell.CELL_TYPE_FORMULA:
                    cellValue = String.valueOf(cell.getCellFormula());
                    break;
                case Cell.CELL_TYPE_BLANK:
                    cellValue = "";
                    break;
                case Cell.CELL_TYPE_ERROR:
                    cellValue = "";
                    break;
                default:
                    cellValue = cell.toString().trim();
                    break;
            }
        }
        return cellValue.trim();
    }

    /**
     * 取得sheet的data,listData是一行一个list,这个list里面放该行的所有列的值
     * mapData是一行一个list,这个list里面存放的是map,map的key是第一列的header值。
     */
    private void getSheetData() {
        listData = new ArrayList<List<String>>();
        mapData = new ArrayList<Map<String, String>>();
        columnHeaderList = new ArrayList<String>();
        int numOfRows = sheet.getLastRowNum() + 1;
        for (int i = 0; i < numOfRows; i++) {
            Row row = sheet.getRow(i);
            Map<String, String> map = new HashMap<String, String>();
```

```
            List<String> list = new ArrayList<String>();
            if (row != null) {
                for (int j = 0; j < row.getLastCellNum(); j++) {
                    Cell cell = row.getCell(j);
                    if (i == 0){
                        columnHeaderList.add(getCellValue(cell));
                    }
                    else{
                        map.put(columnHeaderList.get(j),
this.getCellValue(cell));
                    }
                    list.add(this.getCellValue(cell));
                }
            }
            if (i > 0){
                mapData.add(map);
            }
            listData.add(list);
        }
        flag = true;
    }

    /**
     * 根据行与列的index来得到相应的cell的值
     * @param row 从1开始
     * @param col 从1开始
     * @return
     */
    public String getCellData(int row, int col){
        if(row<=0 || col<=0){
            return null;
        }
        if(!flag){
```

```
            this.getSheetData();
        }
        if(listData.size()>=row && listData.get(row-1).size()>=col){
            return listData.get(row-1).get(col-1);
        }else{
            return null;
        }
    }

    /**
     * 根据行数及第一列的列名，取得相应的cell的值
     * @param row 从1开始
     * @param headerName 第一列的列名
     * @return
     */
    public String getCellData(int row, String headerName){
        if(row<=0){
            return null;
        }
        if(!flag){
            this.getSheetData();
        }
        if(mapData.size()>=row &&
mapData.get(row-1).containsKey(headerName)){
            return mapData.get(row-1).get(headerName);
        }else{
            return null;
        }
    }

    /**
     * 获取所有的数据，每一行作为一组数据放进List，每一行里的每一列作为一个数据放进
list
```

```
     * @return
     */
    public List<List<String>> getListData(){
        return listData;
    }

    /**
     * 获取所有的数据，第一行为Map的key,从第二行开始，每一列作为key的value,每一行作为一组数据放进list
     * @return
     */
    public List<Map<String,String>> getMapData(){
        return mapData;
    }


    public static void main(String[] args) {
        ExcelReader eh = new ExcelReader("c:\\log\\test.xlsx","test");
        System.out.println(eh.getCellData(1, 1));
        System.out.println(eh.getCellData(1, "name"));
    }
}
```

操作 XML

- 所需 jar 包

  dom4j-*.jar

  jaxen-*.jar
- 特点：利用 xpath 来解析 xml
- 封装操作

```
public class ParseXml {
    /**
```

```
     * 解析xml文件，我们需要知道xml文件的路径，然后根据其路径加载xml文件后，生成一个Document的对象，
     * 于是我们先定义两个变量String filePath,Document document
     * 然后再定义一个load方法，这个方法用来加载xml文件，从而产生document对象。
     */
    private String filePath;

    private Document document;

    /**
     *  构造器用来new ParseXml对象时，传一个filePath的参数进来,从而初始化filePath的值
     * 调用load方法，从而在ParseXml对象产生时，就会产生一个document的对象。
     */
    public ParseXml(String filePath) {
        this.filePath = filePath;
        this.load(this.filePath);
    }

    /**
     * 用来加载xml文件，并且产生一个document的对象
     */
    private void load(String filePath){
        File file = new File(filePath);
        if (file.exists()) {
            SAXReader saxReader = new SAXReader();
            try {
                document = saxReader.read(file);
            } catch (DocumentException e) {
                Log.logInfo("文件加载异常: " + filePath);
            }
        } else{
            Log.logInfo("文件不存在 : " + filePath);
```

```
        }
    }

    /**
     * 用xpath来得到一个元素节点对象
     * @param elementPath elementPath是一个xpath路径,比如"/config/driver"
     * @return 返回的是一个节点Element对象
     */
    public Element getElementObject(String elementPath) {
        return (Element) document.selectSingleNode(elementPath);
    }

    /**
     * 用xpath来获取一组元素节点对象
     * @param elementPath
     * @return
     */
    @SuppressWarnings("unchecked")
    public List<Element> getElementObjects(String elementPath) {
        return document.selectNodes(elementPath);
    }

    /**
     * 根据Element来获取该节点下的所有的子结点的信息
     * 且返回值中以子结点名称为key,子结点的text为value
     * @param element
     * @return
     */
    @SuppressWarnings("unchecked")
    public Map<String, String> getChildrenInfoByElement(Element element){
        Map<String, String> map = new HashMap<String, String>();
        List<Element> children = element.elements();
        for (Element e : children) {
```

```
            map.put(e.getName(), e.getText());
        }
        return map;
    }

    /**
     * 用xpath来判断一个结点对象是否存在
     */
    public boolean isExist(String elementPath){
        boolean flag = false;
        Element element = this.getElementObject(elementPath);
        if(element != null) flag = true;
        return flag;
    }

    /**
     * 用xpath来取得一个结点对象的值
     */
    public String getElementText(String elementPath) {
        Element element = this.getElementObject(elementPath);
        if(element != null){
            return element.getText().trim();
        }else{
            return null;
        }
    }

    public static void main(String[] args) {
        ParseXml px = new ParseXml("config/TestBaidu.xml");//给定config.xml
的路径
        List<Element> elements = px.getElementObjects("/*/testUI");
        Object[][] object = new Object[elements.size()][];
        for (int i =0; i<elements.size(); i++) {
```

```
            Object[] temp = new
Object[]{px.getChildrenInfoByElement(elements.get(i))};
            object[i] = temp;
        }
    }

}
```

Java 第十九天

外部文件数据源

在前面介绍 TestNg 数据驱动时，就介绍过如何把脚本与数据进行分离，是用了一个 DataSource 类来提供数据，如果我们要进行数据的修改，就要去修改这个 java 文件，我们知道，java 是需要编译的，修改 java 文件后又需要重新编译，很麻烦，但如果用非 java 文件来作为数据源，则修改数据后，就不需要重新编译了。且外部文件利用其文件的特性进行排版后，看起来也会比在 java 文件中更加的直观，清晰明了。

什么格式的外部文件

TestNg 的数据驱动的@DataProvider 返回的数据结构是 Object[][]，也就是说外部文件最好是对 List 数据结构比较友好，符合这个条件的有 XML 文件或 Excel 文件，这两种文件相比：

- XML 比 Excel 轻便
- Excel 比 XML 直观

大家觉得哪种好就用哪种，接下来的演示将会以 XML 为数据文件来演示。

XML 数据格式设计

在 DataSource 类中，有一个概念，就是用测试方法的名称来作为一个 key，以此实现多个测试方法同用一个数据源。在 XML 中，要能够很清晰明确的指出这些数据是提供给哪个测试方法的，所以，我们可以用测试方法的名称来作为一个节点，该节点的所有子节点都是这个测试方法的数据，则可以表示为：（假如有两个测试方法的名称分别为:testDemo,demoTest）

```
<data>
    <testDemo>
        <username>test1</username>
        <password>123456</password>
```

```
        <inputValue>test</inputValue>
    </testDemo>
    <testDemo>
        <username>test2</username>
        <password>123456</password>
        <inputValue>test</inputValue>
    </testDemo>
    <demoTest>
        <username>test1</username>
        <password>123456</password>
        <inputValue>test</inputValue>
    </demoTest>
</data>
```

上面 XML 的结构也是很清晰明了，testDemo 测试方法将会运行两遍，每一遍所对应的数据也很直观，demoTest 测试方法将会只运行一遍，这个 XML 的结构设计好后，那么 XML 文件的名称如何定？肯定有很多 XML 数据文件，那么哪个 XML 文件对应哪个测试类？于是我们可以用测试类的名称来命名 XML 文件名，通过这个潜规则来使脚本与数据联系起来，比如测试类叫：TestDemo,则对应的 XML 数据文件应该是 TestDemo.xml。定义好 XML 文件后，接下来我们就需要去解析这个 XML 文件，同时传给 TestNg 的@DataProvider 就可以了：

```
public class TestData {

    private String fileName;

    private ParseXml px;

    public TestData(String fileName){
        this.fileName = fileName;
        px = new ParseXml("test-data"+File.separator+this.fileName+".xml");//
将数据文件统一放在test-dta文件夹下
    }

    public List<Map<String, String>> getTestMethodData(String methodName){
```

```
        List<Element> elements = px.getElementObjects("/*/"+methodName);//根据方法名找出所有的与方法名相同的节点对象
        List<Map<String, String>> listData = new ArrayList<Map<String, String>>();
        for (Element element : elements) {
            Map<String, String> dataMap = px.getChildrenInfoByElement(element);//根据方法名节点对象找出其所有的子节点的信息
            listData.add(dataMap);
        }
        return listData;
    }

    public static void main(String[] args) {
        TestData td = new TestData("TestDemo");
        List<Map<String, String>> testDemo = td.getTestMethodData("testDemo");
        System.out.println(testDemo.get(0).get("username"));//输出test1
    }

}
```

备注：上面代码中的 ParseXml 类在上一章讲过了且提供出来了，大家自己去整理。

接下来要写一个@Dataprovider 的方法了，将这个方法放在 TestBase 类里，所有的测试方法都通过这个@Dataprovider 去获取测试数据，所以这个@Dataprovider 方法是一个基础方法，就把其放在一个基础类 TestBase 里。

```
public class TestBase {

    private TestData td;

    private void initailTestData(){
        if(td==null){
            td = new TestData(this.getClass().getSimpleName());//将测试类名传给TestData，从而找到对应的XML文件
```

```
        }
    }

    @DataProvider
    public Object[][] dataProvider(Method method){
        this.initailTestData();
        List<Map<String, String>> listData =
td.getTestMethodData(method.getName());
        Object[][] object = new Object[listData.size()][];
        for (int i = 0; i < object.length; i++) {
            object[i] = new Object[]{listData.get(i)};//最后返回的数据结构为
{{Map<String,String>},{Map<String,String>}}
        }
        return object;
    }

}
```

测试类只需要继承 TestBase 类，就能够所有的测试方法都通过@DataProvider 方法去寻找到对应的 XML 数据文件，从而进行数据驱动。

```
public class TestDemo extends TestBase{

    @Test(dataProvider="dataProvider")
    public void testDemo(Map<String, String> param){
        System.out.println(param.get("username"));
        System.out.println(param.get("password"));
        System.out.println(param.get("inputValue"));
    }

}
```

优化数据格式

以上的示例演示了用 XML 文件来作为数据源，再回首看看 XML 文件，我们

会发现<inputValue>这个节点的值在整个 XML 文件中都是一样的，与代码一样，既然相同，就有优化的空间了，于是，我们可以把相同的数据抽取出来，放在一个公共的<common>节点里：

```
<data>
    <common>
        <inputValue>test</inputValue>
    </common>
    <testDemo>
        <username>test1</username>
        <password>123456</password>
    </testDemo>
    <testDemo>
        <username>test2</username>
        <password>123456</password>
    </testDemo>
    <demoTest>
        <username>test1</username>
        <password>123456</password>
    </demoTest>
</data>
```

解析这个 XML 的原理就是先获取<common>下面的值，再获取测试方法的数据，然后将两这个 Map 给合并在一起，再返回出去就可以了，于是我们发现只需要修改 `TestData` 类里的 `getTestMethodData` 方法即可。

假如存在这样一种情况：

```
<data>
    <common>
        <inputValue>test</inputValue>
    </common>
    <testDemo>
        <username>test1</username>
        <password>123456</password>
        <inputValue>test1</inputValue>
    </testDemo>
```

```
    <testDemo>
        <username>test2</username>
        <password>123456</password>
    </testDemo>
    <demoTest>
        <username>test1</username>
        <password>123456</password>
    </demoTest>
</data>
```

第一个<tsetDemo>下面也有一个<inputValue>节点，很显然，第一个<tsetDemo>下面的<inputValue>节点是希望用自己所定义的值，而不是用这个<common>下面的值，于是我们在合并两个 Map 时，如果<testDemo>里存在与<common>里相同的节点名，则以<testDemo>里的节点为准。

```
public class TestData {

    private String fileName;

    private ParseXml px;

    private Map<String, String> commonData;

    public TestData(String fileName){
        this.fileName = fileName;
        px = new ParseXml("test-data"+File.separator+this.fileName+".xml");//
将数据文件统一放在test-dta文件夹下
    }

    public List<Map<String, String>> getTestMethodData(String methodName){
        List<Element> elements = px.getElementObjects("/*/"+methodName);//根
据方法名找出所有的与方法名相同的节点对象
        List<Map<String, String>> listData = new ArrayList<Map<String,
String>>();
        for (Element element : elements) {
```

```
            Map<String, String> dataMap =
px.getChildrenInfoByElement(element);//根据方法名节点对象找出其所有的子节点的信
息
            Map<String, String> data = this.getMergeMapData(dataMap,
this.getCommonData());//合并测试方法与common的数据
            listData.add(data);
        }
        return listData;
    }

    public Map<String, String> getCommonData(){
        if(commonData == null){
            try{
                Element element = px.getElementObject("/*/common");
                commonData = px.getChildrenInfoByElement(element);
            }catch(Exception e){
                commonData = null;
            }
        }
        return commonData;
    }

    /**
     * 合并两个Map,如果map1与map2存在相同的key,则以map1的值为准
     */
    private Map<String, String> getMergeMapData(Map<String, String> map1,
Map<String, String> map2){
        if(map2 == null){
            return map1;
        }
        Iterator<String> it = map2.keySet().iterator();
        while(it.hasNext()){
            String key = it.next();
```

```
            String value = map2.get(key);
            if(!map1.containsKey(key)){
                map1.put(key, value);
            }
        }
        return map1;
    }

}
```

既然有<common>的值，那么如果多个 XML 里的<common>节点下有相同的节点并且节点的值也相同，那同样可以优化，处理办法是单独的用一个 XML 文件来存放所有的测试类都要用到的一些公共数据，比如 url 啊，数据库连接信息啊等等。这些公共的数据由于所有的测试类都有可能用到，所以要在最开始的时候就解析出来，并存放到一个 Map 里，在测试方法去加载数据时，就将公共数据的 Map 与测试类的 common 的 Map 进行合并，然后再与测试方法的数据进行合并，同样的，如果存在相同的节点，则优先级是测试方法>common>公共数据。

将公共的 XML 命名为 global.xml，并存放于 test-data 里，并且在最开始的时候就解析出来：

```
public class Global {

    public static Map<String, String> global;

    static{
        ParseXml px = new ParseXml("test-data/global.xml");
        global = px.getChildrenInfoByElement(px.getElementObject("/*"));
    }

}
```

再来合并这三个 Map，同样的只需要修改 `TestData` 类里的 `getTestMethodData` 方法即可。

```
    public List<Map<String, String>> getTestMethodData(String methodName){
        List<Element> elements = px.getElementObjects("/*/"+methodName);//根
```

据方法名找出所有的与方法名相同的节点对象

```
        List<Map<String, String>> listData = new ArrayList<Map<String,
String>>();
        for (Element element : elements) {
            Map<String, String> dataMap =
px.getChildrenInfoByElement(element);
            Map<String, String> commonData =
this.getMergeMapData(this.getCommonData(), Global.global);
            Map<String, String> data = this.getMergeMapData(dataMap,
commonData);
            listData.add(data);
        }
        return listData;
    }
```

大家将以上的代码进行整理，然后加以理解，就会发现，原来数据驱动还可以这样玩，并用此方式来将 Excel 作为数据源，从而进一步提高自己的编程能力，如果这些都掌握了，还可以尝试一下多数据源的自由切换。挑战一下自己吧！

Java 第二十天

前言

本来我想写写监听的实现原理的，后来想想算了，如果通读前面的文档后，你会发现在讲 interface 的使用时，已经映射到了这个原理了，说白了，监听就是 interface 或抽象类的使用，参考前面的文档，稍加思考就能明白其原理。

TestNg 结果监听器

结果监听器，就是可以获取测试方法运行的结果及该测试方法的所有的数据，并且我们可以更改这些数据。我们先来看看基本的使用：

```
public class ListenerResult extends TestListenerAdapter{

    @Override
    public void onTestStart(ITestResult result) {
        System.out.println("this is test start.");
    }

    @Override
    public void onTestFailure(ITestResult tr) {
        System.out.println("this is test fail.");
    }

    @Override
    public void onTestSkipped(ITestResult tr) {
        System.out.println("this is test skip.");
    }

    @Override
    public void onTestSuccess(ITestResult tr) {
```

```
        System.out.println("this is test success.");
    }

}
```

在测试类中使用监听器：

```
@Listeners({ListenerResult.class})
public class TestListener {

    @Test
    public void test1(){
        System.out.println("this is test1");
    }

    @Test
    public void test2(){
        System.out.println("this is test2");
        System.out.println(1/0);
    }

}
```

输出结果：

```
this is test start.
this is test1
this is test success.
this is test start.
this is test2
this is test fail.
PASSED: test1
FAILED: test2
```

上面监听器的使用很简单，实现接口或继承抽象类，把实现类的`class`加入到`@Listeners`中去即可。同时我们也可以看出，实现的监听类中的`ITestResult`参数包含了监听到的测试方法的所有数据，根据此特性，再联想到我们之前的断言类的封装中，在每个测试方法的最后都要加上一句`Assertion.end();`以此来判断

测试方法是否都断言成功或失败，我们是否可以把两者联系起来？在监听器中去修改测试方法的运行状态？也就是说我们只需要判断flag的值，根据flag的值去修改ITestResult参数的值即可。

```
public class ListenerResult extends TestListenerAdapter{

    @Override
    public void onTestStart(ITestResult result) {
        Assertion.begin();
    }

    @Override
    public void onTestFailure(ITestResult tr) {
        if(!Assertion.flag){
            tr.setStatus(ITestResult.FAILURE);
        }
    }

    @Override
    public void onTestSkipped(ITestResult tr) {
        if(!Assertion.flag){
            tr.setStatus(ITestResult.FAILURE);
        }
    }

    @Override
    public void onTestSuccess(ITestResult tr) {
        if(!Assertion.flag){
            tr.setStatus(ITestResult.FAILURE);
        }
    }

}
```

上面的监听类用了后，再也不用在@BeforeMethod方法或测试方法的开头加上

一句Assertion.begin();也不用在测试方法的最后加Assertion.end();了，一切都交给监听实现类了。

将异常的输出进行改良，请参考以下的代码：

```
public class Assertion {

    public static boolean flag = true;

    public static List<Error> errors = new ArrayList<Error>();

    public static void begin(){
        flag = true;
    }

    public static void verifyEquals(Object actual, Object expected){
        try{
            Assert.assertEquals(actual, expected);
        }catch(Error e){
            errors.add(e);
            flag = false;
        }
    }

    public static void verifyEquals(Object actual, Object expected, String
message){
        try{
            Assert.assertEquals(actual, expected, message);
        }catch(Error e){
            errors.add(e);
            flag = false;
        }
    }

}
```

```
public class ListenerResult extends TestListenerAdapter{

    @Override
    public void onTestStart(ITestResult result) {
        Assertion.begin();
    }

    @Override
    public void onTestFailure(ITestResult tr) {
        this.handleAssertion(tr);
    }

    @Override
    public void onTestSkipped(ITestResult tr) {
        this.handleAssertion(tr);
    }

    @Override
    public void onTestSuccess(ITestResult tr) {
        this.handleAssertion(tr);
    }

    private int index;

    private boolean isNew = false;

    private void handleAssertion(ITestResult tr){
        Throwable throwable = tr.getThrowable();
        if(!Assertion.flag || throwable!=null){
            if(throwable==null){
                throwable = new Throwable();
                isNew = true;
            }
```

```
            StackTraceElement[] alltrace = new StackTraceElement[0];
            for (Error e : Assertion.errors) {
                alltrace = this.getAllStackTraceElement(tr, e, null,
alltrace);
            }
            if(!isNew){
                alltrace = this.getAllStackTraceElement(tr, null, throwable,
alltrace);
            }else{
            isNew = false;
            }
            throwable.setStackTrace(alltrace);
            tr.setThrowable(throwable);
            Assertion.flag = true;
            Assertion.errors.clear();
            tr.setStatus(ITestResult.FAILURE);
        }
    }

    private StackTraceElement[] getAllStackTraceElement(ITestResult tr, Error
e, Throwable throwable, StackTraceElement[] alltrace){
        StackTraceElement[] traces =
(e==null?throwable.getStackTrace():e.getStackTrace());
        StackTraceElement[] et = this.getKeyStackTrace(tr, traces);
        String msg = (e==null?throwable.getMessage():e.getMessage());
        StackTraceElement[] message = new StackTraceElement[]{new
StackTraceElement("message : "+msg+" in method : ",
tr.getMethod().getMethodName(),
tr.getTestClass().getRealClass().getSimpleName(), index)};
        index = 0;
        alltrace = this.merge(alltrace, message);
        alltrace = this.merge(alltrace, et);
        return alltrace;
```

```
    }

    private StackTraceElement[] getKeyStackTrace(ITestResult tr,
StackTraceElement[] stackTraceElements){
        List<StackTraceElement> ets = new ArrayList<StackTraceElement>();
        for (StackTraceElement stackTraceElement : stackTraceElements) {

if(stackTraceElement.getClassName().equals(tr.getTestClass().getName())){
                ets.add(stackTraceElement);
                index = stackTraceElement.getLineNumber();
            }
        }
        StackTraceElement[] et = new StackTraceElement[ets.size()];
        for (int i = 0; i < et.length; i++) {
            et[i] = ets.get(i);
        }
        return et;
    }

    private StackTraceElement[] merge(StackTraceElement[] traces1,
StackTraceElement[] traces2){
        StackTraceElement[] ste = new
StackTraceElement[traces1.length+traces2.length];
        for (int i = 0; i < traces1.length; i++) {
            ste[i] = traces1[i];
        }
        for (int i = 0; i < traces2.length; i++) {
            ste[traces1.length+i] = traces2[i];
        }
        return ste;
    }

}
```

备注：以上代码直接复制可用。但 Assertion 类中的变量 flag 及 errors 是非线程安全的，如果要多线程执行，会产生问题，请参考以前的文档中 **ThreadLocal** 的使用，并自行加以改良吧。

TestNg 报告监听器

TestNg 的监听器的原理都是一样的，报告监听器的用法与结果监听器的用法，也是一样的，不过，报告监听器是实现一个接口：

```
public class NewReport implements IReporter{


    @Override
    public void generateReport(List<XmlSuite> xmlSuites, List<ISuite> suites,
String outputDirectory) {
        List<ITestResult> list = new ArrayList<ITestResult>();
        for (ISuite suite : suites) {
            Map<String, ISuiteResult> suiteResults = suite.getResults();
            for (ISuiteResult suiteResult : suiteResults.values()) {
                ITestContext testContext = suiteResult.getTestContext();
                list.addAll(testContext.getPassedTests().getAllResults());
                list.addAll(testContext.getFailedTests().getAllResults());
                list.addAll(testContext.getSkippedTests().getAllResults());


    list.addAll(testContext.getFailedConfigurations().getAllResults());
            }
        }
        this.sort(list);
        this.outputResult(list, outputDirectory + "/test.txt");
    }


    private void sort(List<ITestResult> list){
        Collections.sort(list, new Comparator<ITestResult>(){


            @Override
```

```
            public int compare(ITestResult r1, ITestResult r2) {
                if(r1.getStartMillis()>r2.getStartMillis()){
                    return 1;
                }else{
                    return -1;
                }
            }

        });
    }

    private void outputResult(List<ITestResult> list, String path){
        try {
            BufferedWriter output = new BufferedWriter(new FileWriter(new
File(path), true));
            StringBuffer sb = new StringBuffer();
            for (ITestResult result : list) {
                if(sb.length()!=0){
                    sb.append("\r\n");
                }
                sb.append(result.getTestClass().getRealClass().getName())
                  .append(" ")
                  .append(result.getMethod().getMethodName())
                  .append(" ")
                  .append(this.formatDate(result.getStartMillis()))
                  .append(" ")
                  .append(result.getEndMillis()-result.getStartMillis())
                  .append("毫秒 ")
                  .append(this.getStatus(result.getStatus()));
            }
            output.write(sb.toString());
            output.flush();
            output.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private String formatDate(long date){
        SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss.SSS");
        return formatter.format(date);
    }

    private String getStatus(int status){
        String statusString = null;
        switch(status){
        case 1:
            statusString = "SUCCESS";
            break;
        case 2:
            statusString = "FAILURE";
            break;
        case 3:
            statusString = "SKIP";
            break;
        default:
            break;
        }
        return statusString;
    }

}
```

报告监听的接口实现后，需要把实现类 `NewReport` 给注册到 TestNg 中去，但这个能加在测试类的`@Listeners` 中吗？虽然可以，但加在测试类上代表这个 `NewReport` 只适用于这个测试类上，我们在单个测试类调试时可以这样用，但在

用 TestNg 的 XML 配置文件进行执行时，就必须在 XML 的配置文件中加上这个监听了，且最好不要加在测试类的@Listeners 中，以免引起冲突覆盖。

```
<?xml version="1.0" encoding="UTF-8"?>
<suite name="Suite" verbose="1" parallel="false" thread-count="1">

    <listeners>
        <listener class-name="com.test.listeners.NewReport"/>
    </listeners>
    <test name="Test1">
        <classes>
            <class name="com.test.demo.TestListener" />
        </classes>
    </test>
</suite>
```

后记

TestNg 还有一些其它的监听器，大家可以去自行的研究一下，并加以灵活运用，对自己的测试框架是很有帮助的。

Java 第二十一天

前言

这是最后一章了，如果你跟着文档一路走下来了，蓦然回首，会发现不知不觉中已然写了这么多的代码了，为自己鼓掌吧。鼓掌的同时，会发现我们之前的整个过程，都离不开 eclipse，包含代码的编写，脚本的执行。Java 是需要编译的，连编译 eclipse 都帮你自动的编译了。那在做 CI（持续集成）时，很显然是要摆脱 eclipse 的。那么我们就得找到一个工具来摆脱 eclipse，且这个工具能与 CI 工具进行结合，所以这个工具必须具备：编译功能 、执行功能、参数化功能。具备这三个功能的工具也有几个，在这里向大家介绍 ANT，我认为 ANT 对于测试人员而言的话，完全够用了。

ANT 安装

安装过程：

1. 到 http://ant.apache.org/bindownload.cgi 下载 ant 发布版本
2. 将下载后的 zip 文件解压缩到任意目录，比如 D:\ant
3. 在环境变量中增加 ANT_HOME=D:\ant(替换成你解压缩的目录)
4. 在环境变量 path 中增加;%ANT_HOME%\bin
5. 打开 cmd ，输入 ant ，如果提示一下信息证明成功了

```
Buildfile: build.xml does not exist!
Build failed
```

说明：

这里的 failed 并不是指你的 Ant 安装失败了，而是因为你只输入 ant 命令后，会在你当前目录下去寻找一个叫 build.xml 的文件，如果你当前目录下没有这个 build.xml 的文件，则会报 build.xml does not exist!，而 build.xml 里存放的是你需要去干的一些事情，比如编译，执行等。下面我们将介绍 build.xml。

build.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<project name="selenium" default="start_run_tests" basedir=".">
    <property name="src" value="src"/>
    <property name="dest" value="classes"/>
    <property name="lib.dir" value="${basedir}/lib"/>
    <property name="suite.dir" value="${basedir}/run"/>
    <property name="jar" value="selenium.jar"/>
    <path id="compile.path">
         <fileset dir="${lib.dir}/">
            <include name="*.jar"/>
        </fileset>
        <pathelement location="${src}"/>
        <pathelement location="${dest}"/>
    </path>
    <target name="init">
        <mkdir dir="${dest}"/>
    </target>
    <target name="compile" depends="init">
        <javac srcdir="${src}" destdir="${dest}" classpathref="compile.path">
         <compilerarg line="-encoding UTF-8 "/>
        </javac>
    </target>
     <!--run testng ant task-->
     <taskdef resource="testngtasks" classpathref="compile.path"/>

     <target name="start_run_tests" depends="compile" description="begin to
run tests">
         <parallel>
             <antcall target="run_tests">
             </antcall>
```

```
        </parallel>
    </target>

    <target name="run_tests" depends="compile">
        <testng classpathref="compile.path"  failureproperty="test.failed">
            <!--xml test suite file -->
            <xmlfileset dir="${suite.dir}">
                <include name="test1.xml"/>
            </xmlfileset>
        </testng>
        <antcall target="tearDown"/>
        <fail message="ERROR: test failed!!!!!" if="test.failed"/>
    </target>
   <target name=" tearDown">
       <delete dir="${dest}"/>
       <antcall target="transform"/>
   </target>
   <target name="transform">
       <xslt in="${basedir}/test-output/testng-results.xml"
style="${basedir}/test-output/testng-results.xsl"
out="${basedir}/test-output/report.html" classpathref="compile.path">
           <param name="testNgXslt.outputDir"
expression="${basedir}/test-output/"/>
           <param name="testNgXslt.showRuntimeTotals" expression="true"/>

       </xslt>
   </target>
</project>
```

将以上内容保存至 XML 文件并命名为 build.xml，且存放于工程根目录下面。简单说明一下，project 结点的 default 属性，是代表入口函数，表示会从"start_run_tests"这个 target 开始执行起，property 结点就是变量的定义，target 可以理解为函数方法，target 中的 depends 表示依赖，在执行 target 时，会优先执行其依赖的 target，所以，上述这个 build.xml 的执行顺序：

init->compile->start_run_tests->run_tests->tearDown->transform 这个顺序就是初始化，编译，执行 test1.xml，生成报告。最后的 transform 是用了一个 testng-xslt 的报告插件，这个插件的用法：

1. 下载 TestNG-xslt 把 saxon-8.7.jar 复制到测试项目的 lib 下
2. 从你下载的包中拷贝文件 testng-results.xsl 到 test-output 目录下。testng-results.xsl 文件的位置是 testng-xslt-1.1.1\src\main\resources，为什么要这个文件呢？因为我们的测试报告就是用这个 style 生成的。
3. 用 ant 运行这个 build.xml 就会在 test-output 目录下生成 report.html，打开它就能看到新生成的测试报告，这个报告可能会给你目耳一新的感觉。
4. 假如你不想用这个报告，也不想要用这个插件，将 <antcall target="transform"/>这一句删除，同时将<target name="transform">这个 target 删除即可。

运行 build.xml

运行的方式主要有三种：（主要是在 windows 下执行）

1. 打开 cmd，cd 至 build.xml 所在的文件夹，输入命令 ant 即可运行
2. 打开 cmd，cd 至 build.xml 所在的文件夹，输入命令 ant build.xml 即可运行
3. 打开 cmd，输入 ant c:\tool\workspace\TestDemo\build.xml，即是 ant 后面接上 build.xml 的全路径

build.xml 参数化

上面的 build.xml 中有一句<include name="test1.xml"/>，即是运行 test1.xml 这个 TestNg 的 XML 配置文件，这里面把 test1.xml 这个写死了，意味着这个 build.xml 只会运行 test1.xml 这一个配置文件，如果想运行 test2.xml，就得更改这个 build.xml，这相当于代码中的硬编码了，不是很合理，很显然需要进行参数化，ant 是这样进行参数化的：

1. 将这一句<include name="test1.xml"/>
   改为<include name="${testcase}.xml"/>
2. 在运行时，用 ant build.xml -Dtestcase=test1，这样表示将 test1 这

个值传给了 build.xml 中的${testcase}了，这样就实现了参数化了。

实现参数化的目的，第一是为了通过一个 build.xml 文件来执行所有的 TestNg 的配置文件，第二也是为了与 CI 工具进行结合，我们这里所说的 CI 工具，可能主要是指 jenkins，工具始终是工具，没啥太难的，大家下去翻翻资料，也差不多会了，然后将其与 ANT 结合起来执行 TestNg 的配置文件，来真正实现测试自动化。