1、String 的 split(String regex)方法参数注意点

使用这个方法时，当我们直接以“.”为参数时，是会出错的，如：

```
String str = "12.03";

String[] res = str.spilt(".");    //出错！！！
```

此时，我们得到的 res 是为空的（不是 null），即 str = [];

因为 String 的 split(String regex)根据给定的正则表达式的匹配来拆分此字符串，而"."是正则表达式中的关键字，没有经过转义 split 会把它当作一个正则表达式来处理的，需要写成 str.split("\.")进行转义处理。

2、关于 hashCode 方法

我们可以先通过 HashMap 中 hashCode 的作用来体验一下。

我们知道 HashMap 中是不允许插入重复元素的，如果是插入的同一个元素，会将前面的元素给覆盖掉，那势必在 HashMap 的 put 方法里对 key 值进行了判断，检测其是否是同一个对象。其 put 源码如下：

```
public V put(K key, V value) {

    if (table == EMPTY_TABLE) {

        //key 的hashCode 值放在了table 里面

        inflateTable(threshold);

    }

    if (key == null)

                return putForNullKey(value);

    int hash = hash(key);

    //计算我们传进来的key 的hashcode 值
```

```
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            //将传进来的key 的hashcode 值于HashMap 中的table 里面存放的hashCode 值比较
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }
    modCount++;
    addEntry(hash, key, value, i);
    return null;
}
```

可以看到这里的判断语句 if (e.hash == hash && ((k = e.key) == key || key.equals(k)))，里面通过&&逻辑运算符相连，先判断 e.hash == hash，即判断传进来的 key 的 hashCode 值与 table 中的已有的 hashCode 值比较，如果不存在该 key 值，也就不会再去执行&&后面的 equals 判断；当已经存在该 key 值时，再调用 equals 方法再次确定两个 key 值对象是否相同。从这里可以看出，hashCode 方法的存在是为了减少 equals 方法的调用次数，从而提高程序效率。

可以看到，判断两个对象是否相同，还是要取决于 equals 方法，而两个对象的 hashCode 值是否相等是两个对象是否相同的必要条件。所以有以下结论：

1. 如果两个对象的 hashCode 值不等，根据必要条件理论，那么这两个对象一定不是同一个对象，即他们的 equals 方法一定要返回 false；

2. 如果两个对象的 hashCode 值相等，这两个对象也不一定是同一个对象，即他们的 equals 方法返回值不确定；

反过来

1. 如果 equals 方法返回 true，即是同一个对象，它们的 hashCode 值一定相等；
2. 如果 equals 方法返回 false，hashCode 值也不一定不相等，即是不确定的；

（hashCode 返回的值一般是对象的存储地址或者与对象存储地址相关联的 hash 散列值）

然而，很多时候我们可能会重写 equals 方法，来判断这两个对象是否相等，此时，为了保证满足上面的结论，即满足 hashCode 值相等是 equals 返回 true 的必要条件，我们也需要重写 hashCode 方法，以保证判断两个对象的逻辑一致（所谓的逻辑一致，是指 equals 和 hashCode 方法都是用来判断对象是否相等）。如下例子：

```
public class Person {

    private String name;

    private int age;

    public Person(String name,int age){

        this.name = name;

        this.age = age;

    }

    public String getName() {

        return name;

    }

    public void setName(String name) {

        this.name = name;
```

```
    }

    public int getAge() {

        return age;

    }

    public void setAge(int age) {

        this.age = age;

    }

    @Override

        public Boolean equals(Object obj) {

        return this.name.equals(((Person)obj).name) && this.age== ((Person)obj).age
;

    }

}
```

在 Person 里面重写了 equals 方法，但是没有重写 hashCode 方法，如果就我们平时正常来使用的话也不会出什么问题，如：

```
Person p1 = new Person("lly",18);

Person p2 = new Person("lly",18);

System.out.println(p1.equals(p2));

//返回true
```

上面是按照了我们重写的 equals 方法，返回了我们想要的值。但是当我们使用 HashMap 来保存 Person 对象的时候就会出问题了，如下：

```
<span style="white-space:pre">  </span>Person p1 = new Person("lly", 18);

        System.out.println(p1.hashCode());
```

```
        HashMap<Person, Integer> hashMap = new HashMap<Person, Integer>();

        hashMap.put(p1, 1);

        System.out.println(hashMap.get(new Person("lly", 18)));     //此时返回了
null，没有按我们的意愿返回1
```

这是因为，我们没有重写 Person 的 hashCode 方法，使 hashCode 方法与我们 equals 方法的逻辑功能一致，此时的 Person 对象调用的 hashCode 方法还是父类的默认实现，即返回的是和对象内存地址相关的 int 值，这个时候，p1 对象和 new Person("lly",18);对象因为内存地址不一致，所以其 hashCode 返回值也是不同的。故 HashMap 会认为这是两个不同的 key，故返回 null。

所以，我们想要正确的结果，只需要重写 hashCode 方法，让 equals 方法和 hashCode 方法始终在逻辑上保持一致性。

在《Java 编程思想》一书中的 P495 页有如下的一段话：

“设计 hashCode()时最重要的因素就是：无论何时，对同一个对象调用 hashCode()都应该产生同样的值。如果在将一个对象用 put()添加进 HashMap 时产生一个 hashCdoe 值，而用 get()取出时却产生了另一个 hashCode 值，那么就无法获取该对象了。所以如果你的 hashCode 方法依赖于对象中易变的数据，用户就要当心了，因为此数据发生变化时，hashCode()方法就会生成一个不同的散列码”。

如下一个例子：

```
public class Person {

    private String name;

    private int age;

    public Person(String name,int age){

        this.name = name;

        this.age = age;

    }

    public String getName() {
```

```
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    @Override
        public int hashCode() {
        return name.hashCode()*37+age;
        //hashCode 的返回值依赖于对象中的易变数据
    }
    @Override
        public Boolean equals(Object obj) {
        return this.name.equals(((Person)obj).name) && this.age== ((Person)obj).age
;
    }
}
```

此时我们继续测试：

```
<span style="white-space:pre">  </span>Person p1 = new Person("lly", 18);

        System.out.println(p1.hashCode());

        HashMap<Person, Integer> hashMap = new HashMap<Person, Integer>();

        hashMap.put(p1, 1);

       p1.setAge(13);//改变依赖的一个值

        System.out.println(hashMap.get(p1));     //此时还是返回为null,这是因为我们p1
的hashCode 值已经改变了
```

所以，在设计 hashCode 方法和 equals 方法的时候，如果对象中的数据易变，则最好在 hashCode 方法中不要依赖于该字段。

3、Override 和 Overload 的区别

Override（重写）：

在子类中定义与父类具有完全相同的名称和参数的方法，通过子类创建的实例对象调用这个方法时，将调用子类中的定义方法，这相当于把父类中定义的那个完全相同的方法给覆盖了，是子类与父类之间多态性的一种体现。特点如下：

1. 子类方法的访问权限只能比父类的更大，不能更小（可以相同）；
2. 如果父类的方法是 private 类型，那么，子类则不存在覆盖的限制，相当于子类中增加了一个全新的方法；
3. 子类覆盖的方法所抛出的异常必须和父类被覆盖方法的所抛出的异常一致，或者是其子类；即子类的异常要少于父类被覆盖方法的异常；

Overload（重载）：

同一个类中可以有多个名称相同的方法，但方法的参数个数和参数类型或者参数顺序不同；

关于重载函数返回类型能否不一样，需分情况：

1. 如果几个 Overloaded 的方法的参数列表不一样（个数或类型），它们的返回者类型当然也可以不一样；

2. 两个方法的参数列表完全一样，则不能通过让其返回类型的不同来实现重载。
3. 不同的参数顺序也是可以实现重载的；如下：

```
public String getName(String str,int i){

    return null;

}

public String getName(int i,String str){

    return null;

}
```

我们可以用反证法来说明这个问题，因为我们有时候调用一个方法时也可以不定义返回结果变量，即不要关心其返回结果，例如，我们调用 map.remove(key)方法时，虽然 remove 方法有返回值，但是我们通常都不会定义接收返回结果的变量，这时候假设该类中有两个名称和参数列表完全相同的方法，仅仅是返回类型不同，java 就无法确定编程者倒底是想调用哪个方法了，因为它无法通过返回结果类型来判断。

所以，Overloaded 重载的方法是可以改变返回值的类型；只能通过不同的参数个数、不同的参数类型、不同的参数顺序来实现重载。

4、ArrayList、Vector、LinkedList 区别

ArrayList、Vector、LinkedList 都实现了 List 接口，其关系图如下：

三者都可以添加 null 元素对象，如下示例：

```
ArrayList<String> arrayList = new ArrayList<String>();

arrayList.add(null);

arrayList.add(null);

System.out.println(arrayList.size());

//输出为2
```

```
LinkedList<String> linkedList = new LinkedList<String>();
linkedList.add(null);

Vector<String> vectorList = new Vector<String>();
vectorList.add(null);
```

ArrayList 和 Vector 相同点：

ArrayList 和 Vector 两者在功能上基本完全相同，其底层都是通过 new 出的 Object[]数组实现。所以当我们能够预估到数组大小的时候，我们可以指定数组初始化的大小，这样可以减少后期动态扩充数组大小带来的消耗。如下：

```
ArrayList<String> list= new ArrayList<String>(20);
Vector<String> list2 = new Vector<String>(15);
```

由于这两者的数据结构为数组，所以在获取数据方面即 get()的时候比较高效，而在 add()插入或者 remove()的时候，由于需要移动元素，效率相对不高。（其实对于我们平常使用来说，由于一般使用 add(String element)都是让其加在数组末尾，所以并不需要移动元素，效率还是很好的，如果使用 add(int index, String element)指定了插入位置，此时就需要移动元素了。）

ArrayList 和 Vector 区别：

ArrayList 的所有方法都不是同步的，而 Vector 的大部分方法都加了 synchronized 同步，所以，就线程安全来说，ArrayList 不是线程安全的，而 Vector 是线程安全的，也因此 Vector 效率方面相较 ArrayList 就会更低，所以如果我们本身程序就是安全的，ArrayList 是更好的选择。

大多数的 Java 程序员使用 ArrayList 而不是 Vector,因为同步完全可以由程序员自己来控制。

LinkedList：

LinkedList 其底层是通过双向循环链表实现的，所以在大量增加或删除元素时（即 add 和 remove 操作），由于不需要移动元素有更好的性能，但是在获取数据（get 操作）方面要差。

所以，在三者的使用选择上，LinkedList 适合于有大量的增加/删除操作和较少随机读取操作，ArrayList 适合于大规模随机读取数据，而较少插入和删除元素情景下使用，Vector 在要求线程安全的情况下使用。

5、String、StringBuffer、StringBuilder 区别

String（since JDK1.0）：
字符串常量，不可更改，因为其内部定义的是一个 final 类型的数组来保存值的，如下：

```
private final char value[];
```

所以，当我们每次去“更改”String 变量的值的时候（包括重新赋值或者使用 String 内部的一些方法），其实是重新新建了一个 String 对象（new String）来保存新的值，然后让我们的变量指向新的对象。因此，当我们需要频繁改变字符串的时候，使用 String 会带来较大的开销。

定义 String 的方法有两种：

```
1. String str = "abc";

2. String str2 = new String("def");
```

第一种方式创建的 String 对象“abc”是存放在字符串常量池中，创建过程是，首先在字符串常量池中查找有没有"abc"对象，如果有则将 str 直接指向它，如果没有就在字符串常量池中创建出来“abc”，然后在将 str 指向它。当有另一个 String 变量被赋值为 abc 时，直接将字符串常量池中的地址给它。如下：

```
<span style="white-space:pre">  </span>String a = "abc";

        String b = "abc";

        System.out.println(a == b);     //打印 true
```

也就是说通过第一种方式创建的字符串在字符串常量池中，是可共享的。同时，也是不可更改的，体现在：

```
<span style="white-space:pre">  </span>String a = "abc";

        String b = "abc";

        b = b + "def";
```

此时，字符串常量池中存在了两个对象“abc”和“abcdef”。

第二种创建方式其实分为两步：

```
    String s = "def";

    String str2 = new String(s);
```

第一步就是上面的第一种情况；第二步在堆内存中 new 出一个 String 对象，将 str2 指向该堆内存地址，新 new 出的 String 对象内容，是在字符串常量池中找到的或创建出“def”对象，相当于此时存在两份“def”对象拷贝，一份存在字符串常量池中，一份被堆内存的 String 对象私有化管理着。所以使用 String str2 = new String("def");这种方式创建对象，实际上创建了两个对象。

StringBuffer（since JDK1.0）和 StringBuilder（since JDK1.5）：

StringBuffer 和 StringBuilder 在功能上基本完全相同，它们都继承自 AbstractStringBuilder，而 AbstractStringBuilder 是使用非 final 修饰的字符数组实现的，如：char[] value; ，所以，可以对 StringBuffer 和 StringBuilder 对象进行改变，每次改变还是再原来的对象上发生的，不会重新 new 出新的 StringBuffer 或 StringBuilder 对象来。所以，当我们需要频繁修改字符串内容的时候，使用 StringBuffer 和 StringBuilder 是很好地选择。

两者的核心操作都是 append 和 insert，append 是直接在字符串的末尾追加，而 insert(int index,String str)是在指定位置出插入字符串。 StringBuffer 和 StringBuilder 的最主要区别就是线程安全方面，由于在 StringBuffer 内大部分方法都添加了 synchronized 同步，所以 StringBuffer 是线程安全的，而 StringBuilder 不是线程安全的。因此，当我们处于多线程的环境下时，我们需要使用 StringBuffer，如果我们的程序是线程安全的使用 StringBuilder 在性能上就会更优一点。

三者的效率比较：
如上所述，

1. 当我们需要频繁的对字符串进行更改的时候，使用 StringBuffer 或 StringBuilder 是优先选择，对于 StringBuffer 和 StringBuilder 来说，只要程序是线程安全的，我们尽量使用 StringBuilder 来处理，要求线程安全的话只能使用 StringBuffer。平常情况下使用字符串（不常更改字符串内容），String 可以满足需求。
2. 有一种情况下使用 String 和 StringBuffer 或 StringBuilder 的效率是差不多的，如下：

```
<span style="white-space:pre">  </span>String a = "abc" + "def";    //速度很快

        StringBuilder sb = new StringBuilder();

        sb.append("abc").append("def");
```

对于第一条语句，Java 在编译的时候直接把 a 编译成 a = "abcdef"，但是当我们拼接的字符串是其他已定义的字符串对象时，就不会自动编译了，如下：

```
<span style="white-space:pre">  </span>String a = "abc";

        String b = "def";

        String c = a + b;
```

根据 String 源码中的解释，这种情况下是使用 concatenation 操作符（+），内部是新创建 StringBuffer 或 StringBuilder 对象，利用其 append 方法进行字符串追加，然后利用 toString 方法返回 String 串。所以此时的效率也是不高的。

6、Map、Set、List、Queue、Stack 的特点与用法

Java 集合类用来保存对象集合，对于基本类型，必须要使用其包装类型。Java 集合框架分为两种：

（1）Collection
以单个对象的形式保存，其关系图如下：

其中，Statck 类为 Vector 的子类。由于 Collection 类继承 Iterable 类，所以，所有 Collection 的实现类都可以通过 foreach 的方式进行遍历。

（2）Map
以<key,value>键值对的形式保存数据。Map 常用类的结构关系如下：

List：
List 集合里面存放的元素有序、可重复。List 集合的有序体现在它默认是按照我们的添加顺序设置索引值（即我们可以通过 get(索引值 index)的方式获取对象）；可重复，是由于我们给每个元素设置了索引值，可以通过索引值找到相应的对象。
关于 List 集合下的具体实现类 ArrayList、Vector、LinkedList 可以参考上面的第四点总结。对于 Statck，它是 Vector 的子类，模拟了栈后进先出的数据结构。

Queue：
接口，模拟了队列先进先出的数据结构。

Set：
Set 里面的元素无序、不可重复。由于无序性，我们不能通过 get 方式获取对象（因为 set 没有索引值）。如下：

```
<span style="white-space:pre">  </span>Set<String> set = new HashSet<String>();

        set.add("ddd");

        set.add("4444");

        set.add("555");

        set.add("777");

        for(String s : set){

            System.out.println(s);

        }
```

打印结果如下：

```
ddd

777

4444
```

```
555
```

而对于不可重复性，Set 的所有具体实现类其内部都是通过 Map 的实现类来保存对象的。如 HashSet 内部就是通过 HashMap 来保存数据的，如下源码：

```
// Dummy value to associate with an Object in the backing Map--一个没有实际意义的Object 对象

private static final Object PRESENT = new Object();

public HashSet() {

    map = new HashMap<>();

}

public Boolean add(E e) {

    return map.put(e, PRESENT)==null;

}
```

可以看到，new 出一个 HashSet 的时候，里面 new 出了一个 HashMap 对象，在使用 HashSet 进行 add 添加数据的时候，HashSet 将我们需要保存的数据作为 HashMap 的 key 值保存了起来，而 key 值是不允许重复的，相当于 HashSet 的元素也是不可重复的。

Map：
Map 里面的元素通过<key,value>键值对的形式保存，不允许重复（具体分析可参考第二点总结内容）。其实 Map<key,value>有点像 Java 中的 Model 对象类，如下使用：

```
    class Person{

        private String name;

        private int age;

        //set、get 方法省略

    }
```

```
List<Person> persons ;

//也可使用如下方式

List<Map<String,String>> persons ;

Map<String,String> map = new HashMap<String,String>();

map.put("name","lly");

map.put("age","18");

persons .add(map);
```