Java提高篇

极客学院出版

前言

本文是作者成功自学 java 后总结出的，一整套知识点集合。

适用人群

适用正在自学 java 或准备自学的人群

> 致谢
>
> 内容撰写：http://cmsblogs.com/?cat=5

更新日期	更新内容
2015-07-23	Java 提高篇

目录

1

理解 Java 的三大特性之封装

从大二接触 Java 开始，到现在也差不多三个年头了。从最基础的 HTML、CSS 到最后的 SSH 自己都是一步一个脚印走出来的，其中开心过、失落过、寂寞过。虽然是半道出家但是经过自己的努力也算是完成了“学业”。期间参加过培训机构，但是极其不喜欢那种培训方式，于是毅然的放弃了选择自学(可怜我出了6000块钱啊)，虽然自学途中苦很多，但是乐更多，当中的付出和收获只有自己知道。黄天不负有心人，鄙人愚钝，在大四第一学期终于自学完成 Java (其中走了弯路，荒废半年)，并且凭借它得到了一份不错的工作，不胜感激！

闲话过多！进入正题，LZ 最近刚刚看完设计模式，感触良多。而且在工作过程中深感 Java 基础不够扎实，例如 IO 不熟、垃圾回收不知所云、多态七窍通五窍、反射不知、甚至连最基本的三大特性都搞得我迷糊了！所以我发狠心一定要好好弥补 Java 基础！从第一课开始一封装!!!!!!

三大特性之一封装

封装从字面上来理解就是包装的意思，专业点就是信息隐藏，是指利用抽象数据类型将数据和基于数据的操作封装在一起，使其构成一个不可分割的独立实体，数据被保护在抽象数据类型的内部，尽可能地隐藏内部的细节，只保留一些对外接口使之与外部发生联系。系统的其他对象只能通过包裹在数据外面的已经授权的操作来与这个封装的对象进行交流和交互。也就是说用户是无需知道对象内部的细节（当然也无从知道），但可以通过该对象对外的提供的接口来访问该对象。

对于封装而言，一个对象它所封装的是自己的属性和方法，所以它是不需要依赖其他对象就可以完成自己的操作。

使用封装有三大好处：

1、良好的封装能够减少耦合。

2、类内部的结构可以自由修改。

3、可以对成员进行更精确的控制。

4、隐藏信息，实现细节。

首先我们先来看两个类：Husband.java、Wife.java

```
public class Husband {
    /*
     * 对属性的封装
     * 一个人的姓名、性别、年龄、妻子都是这个人的私有属性
     */
    private String name ;
    private String sex ;
    private int age ;
    private Wife wife;

    /*
     * setter()、getter()是该对象对外开发的接口
     */
    public String getName() {
       return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setWife(Wife wife) {
        this.wife = wife;
    }
}

public class Wife {
    private String name;
    private int age;
    private String sex;
    private Husband husband;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }
```

```
    public void setAge(int age) {
        this.age = age;
    }

    public void setHusband(Husband husband) {
        this.husband = husband;
    }

    public Husband getHusband() {
        return husband;
    }

}
```

从上面两个实例我们可以看出 Husband 里面 wife 引用是没有 getter() 的，同时 wife 的 age 也是没有 getter() 方法的。至于理由我想各位都懂的，男人嘛深屋藏娇妻嘛，没有那个女人愿意别人知道她的年龄。

所以封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果不想被外界方法，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。比如我们将一个房子看做是一个对象，里面的漂亮的装饰，如沙发、电视剧、空调、茶桌等等都是该房子的私有属性，但是如果我们没有那些墙遮挡，是不是别人就会一览无余呢？没有一点儿隐私！就是存在那个遮挡的墙，我们既能够有自己的隐私而且我们可以随意的更改里面的摆设而不会影响到其他的。但是如果没有门窗，一个包裹的严严实实的黑盒子，又有什么存在的意义呢？所以通过门窗别人也能够看到里面的风景。所以说门窗就是房子对象留给外界访问的接口。

通过这个我们还不能真正体会封装的好处。现在我们从程序的角度来分析封装带来的好处。如果我们不使用封装，那么该对象就没有 setter() 和 getter()，那么 Husband 类应该这样写：

```
public class Husband {
    public String name ;
    public String sex ;
    public int age ;
    public Wife wife;
}
```

我们应该这样来使用它：

```
Husband husband = new Husband();
        husband.age = 30;
        husband.name = "张三";
        husband.sex = "男";   //貌似有点儿多余
```

但是那天如果我们需要修改 Husband，例如将 age 修改为 String 类型的呢？你只有一处使用了这个类还好，如果你有几十个甚至上百个这样地方，你是不是要改到崩溃。如果使用了封装，我们完全可以不需要做任何修改，只需要稍微改变下 Husband 类的 setAge() 方法即可。

```
public class Husband {

    /*
     * 对属性的封装
     * 一个人的姓名、性别、年龄、妻子都是这个人的私有属性
     */
    private String name ;
    private String sex ;
    private String age ;   /* 改成 String类型的*/
    private Wife wife;

    public String getAge() {
        return age;
    }

    public void setAge(int age) {
        //转换即可
        this.age = String.valueOf(age);
    }

    /** 省略其他属性的setter、getter **/

}
```

其他的地方依然那样引用 (husband.setAge(22)) 保持不变。

到了这里我们确实可以看出，**封装确实可以使我们容易地修改类的内部实现，而无需修改使用了该类的客户代码**。

我们在看这个好处：可以对成员变量进行更精确的控制。

还是那个 Husband，一般来说我们在引用这个对象的时候是不容易出错的，但是有时你迷糊了，写成了这样：

```
Husband husband = new Husband();
        husband.age = 300;
```

也许你是因为粗心写成了，你发现了还好，如果没有发现那就麻烦大了，逼近谁见过300岁的老妖怪啊！

但是使用封装我们就可以避免这个问题，我们对 age 的访问入口做一些控制 (setter) 如：

```
public class Husband {

    /*
     * 对属性的封装
     * 一个人的姓名、性别、年龄、妻子都是这个人的私有属性
     */
    private String name ;
    private String sex ;
    private int age ;    /* 改成 String类型的*/
    private Wife wife;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        if(age > 120){
            System.out.println("ERROR：error age input....");    //提示錯誤信息
        }else{
            this.age = age;
        }

    }

    /** 省略其他属性的setter、getter **/

}
```

上面都是对 setter 方法的控制，其实通过使用封装我们也能够对对象的出口做出很好的控制。例如性别我们在数据库中一般都是已1、0方式来存储的，但是在前台我们又不能展示1、0，这里我们只需要在 getter() 方法里面做一些转换即可。

```
public String getSexName() {
        if("0".equals(sex)){
            sexName = "女";
        }
        else if("1".equals(sex)){
            sexName = "男";
        }
        else{
            sexName = "人妖???";
        }
```

```
        return sexName;
    }
```

在使用的时候我们只需要使用 sexName 即可实现正确的性别显示。同理也可以用于针对不同的状态做出不同的操作。

```
public String getCzHTML(){
        if("1".equals(zt)){
            czHTML = "<a href='javascript:void(0)' onclick='qy("+id+")'>启用</a>";
        }
        else{
            czHTML = "<a href='javascript:void(0)' onclick='jy("+id+")'>禁用</a>";
        }
        return czHTML;
    }
```

鄙人才疏学浅，只能领悟这么多了，如果文中有错误之处，望指正，鄙人不胜感激！

吐槽

宅了三天今天就出去走走，在公交车上遇到一极品美女！我坐着，一美女上车，站在我旁边，开始还好，过了一会儿她就对我笑，笑笑还好，但是你也不能总对着人家笑吧！笑的我都不好意了(鄙人脸皮薄，容易害羞??)。难道是我没有洗脸？照照镜子蛮干净的啊！难道是我衣服又或者裤子没**，看来衣服裤子蛮好的！难道是我帅，对我有意思？(程序员屌丝一枚，貌似没可能)！实在受不了了，哥想惹不起我还躲不起么？哥下车！我一下车，那枚美女就飞速的占我座位！哥当时就憋出一个字：靠！！！！！！！！

2

理解 Java 的三大特性之继承

在《 Think in Java 》中有这样一句话：复用代码是 Java 众多引人注目的功能之一。但要想成为极具革命性的语言，仅仅能够复制代码并对加以改变是不够的，它还必须能够做更多的事情。在这句话中最引人注目的是“复用代码”,尽可能的复用代码使我们程序员一直在追求的，现在我来介绍一种复用代码的方式，也是 Java 三大特性之一—继承。

继承

在讲解之前我们先看一个例子，该例子是前篇博文（ Java提高篇一－理解java 的三大特性之封装）的。

从这里我们可以看出， Wife、Husband 两个类除了各自的 husband、wife 外其余部分全部相同，作为一个想最大限度实现复用代码的我们是不能够忍受这样的重复代码，如果再来一个小三、小四、小五……（扯远了大笑）我们是不是也要这样写呢？那么我们如何来实现这些类的可复用呢？利用继承！！

首先我们先离开软件编程的世界，从常识中我们知道丈夫、妻子、小三、小四…，他们都是人，而且都有一些共性，有名字、年龄、性别、头等等，而且他们都能够吃东西、走路、说话等等共同的行为，所以从这里我们可以发现他们都拥有人的属性和行为，同时也是从人那里继承来的这些属性和行为的。

从上面我们就可以基本了解了继承的概念了，继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码，能够大大的提高开发的效率。

对于 Wife、Husband 使用继承后，除了代码量的减少我们还能够非常明显的看到他们的关系。

继承所描述的是“is-a”的关系，如果有两个对象A和B，若可以描述为“A是B”，则可以表示 A 继承 B，其中 B 是被继承者称之为父类或者超类，A 是继承者称之为子类或者派生类。

实际上继承者是被继承者的特殊化，它除了拥有被继承者的特性外，还拥有自己独有得特性。例如猫有抓老鼠、爬树等其他动物没有的特性。同时在继承关系中，继承者完全可以替换被继承者，反之则不可以，例如我们可以说猫是动物，但不能说动物是猫就是这个道理，其实对于这个我们将其称之为“向上转型”，下面介绍。

诚然，继承定义了类如何相互关联，共享特性。对于若干个相同或者相识的类，我们可以抽象出他们共有的行为或者属相并将其定义成一个父类或者超类，然后用这些类继承该父类，他们不仅可以拥有父类的属性、方法还可以定义自己独有的属性或者方法。

同时在使用继承时需要记住三句话：

1、子类拥有父类非 private 的属性和方法。

2、子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

3、子类可以用自己的方式实现父类的方法。（ 以后介绍 ）。

综上所述，使用继承确实有许多的优点，除了将所有子类的共同属性放入父类，实现代码共享，避免重复外，还可以使得修改扩展继承而来的实现比较简单。

诚然，讲到继承一定少不了这三个东西：构造器、protected 关键字、向上转型。

构造器

通过前面我们知道子类可以继承父类的属性和方法，除了那些 private 的外还有一样是子类继承不了的—构造器。对于构造器而言，它只能够被调用，而不能被继承。调用父类的构造方法我们使用 super() 即可。

对于子类而已,其构造器的正确初始化是非常重要的,而且当且仅当只有一个方法可以保证这点：在构造器中调用父类构造器来完成初始化，而父类构造器具有执行父类初始化所需要的所有知识和能力。

```
public class Person {
    protected String name;
    protected int age;
    protected String sex;

    Person(String name){
        System.out.println("Person Constrctor-----" + name);
    }
}

public class Husband extends Person{
    private Wife wife;

    Husband(){
        super("chenssy");
        System.out.println("Husband Constructor...");
    }

    public static void main(String[] args) {
        Husband husband  = new Husband();
    }
}

Output:
Person Constrctor-----chenssy
Husband Constructor...
```

所以综上所述：对于继承而已，子类会默认调用父类的构造器，但是如果没有默认的父类构造器，子类必须要显示的指定父类的构造器，而且必须是在子类构造器中做的第一件事(第一行代码)。

protected 关键字

private 访问修饰符，对于封装而言，是最好的选择，但这个只是基于理想的世界，有时候我们需要这样的需求：我们需要将某些事物尽可能地对这个世界隐藏，但是仍然允许子类的成员来访问它们。这个时候就需要使用到 protected。

对于 protected 而言，它指明就类用户而言，他是 private，但是对于任何继承与此类的子类而言或者其他任何位于同一个包的类而言，他却是可以访问的。

```
public class Person {
    private String name;
    private int age;
    private String sex;

    protected String getName() {
      return name;
    }

    protected void setName(String name) {
       this.name = name;
    }

    public String toString(){
       return "this name is " + name;
    }

    /** 省略其他setter、getter方法 **/
}

public class Husband extends Person{
    private Wife wife;

    public  String toString(){
       setName("chenssy");   //调用父类的setName();
       return  super.toString();   //调用父类的toString()方法
    }

    public static void main(String[] args) {
       Husband husband = new Husband();

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

```
    }
}

Output:
this name is chenssy
```

从上面示例可以看书子类 Husband 可以明显地调用父类 Person 的 setName()。

诚然尽管可以使用 protected 访问修饰符来限制父类属性和方法的访问权限，但是最好的方式还是将属性保持为 private (我们应当一致保留更改底层实现)，通过 protected 方法来控制类的继承者的访问权限。

向上转型

在上面的继承中我们谈到继承是 is-a 的相互关系，猫继承与动物，所以我们可以说猫是动物，或者说猫是动物的一种。这样将猫看做动物就是向上转型。如下：

```
public class Person {
    public void display(){
        System.out.println("Play Person...");
    }

    static void display(Person person){
        person.display();
    }
}

public class Husband extends Person{
    public static void main(String[] args) {
        Husband husband = new Husband();
        Person.display(husband);     //向上转型
    }
}
```

在这我们通过 Person.display(husband)。这句话可以看出 husband 是 person 类型。

将子类转换成父类，在继承关系上面是向上移动的，所以一般称之为向上转型。由于向上转型是从一个叫专用类型向较通用类型转换，所以它总是安全的，唯一发生变化的可能就是属性和方法的丢失。这就是为什么编译器在“未曾明确表示转型”活“未曾指定特殊标记”的情况下，仍然允许向上转型的原因。

谨慎继承

上面讲了继承所带来的诸多好处，那我们是不是就可以大肆地使用继承呢？送你一句话：慎用继承。

首先我们需要明确，继承存在如下缺陷：

1、父类变，子类就必须变。

2、继承破坏了封装，对于父类而言，它的实现细节对与子类来说都是透明的。

3、继承是一种强耦合关系。

所以说当我们使用继承的时候，我们需要确信使用继承确实是有效可行的办法。那么到底要不要使用继承呢？《Think in Java》中提供了解决办法：问一问自己是否需要从子类向父类进行向上转型。如果必须向上转型，则继承是必要的，但是如果不需要，则应当好好考虑自己是否需要继承。

慎用继承！！！！！！！！！！！！！！！！！！！！！！！！！！！

3

理解 Java 的三大特性之多态

面向对象编程有三大特性：封装、继承、多态。

封装隐藏了类的内部实现机制，可以在不影响使用的情况下改变类的内部结构，同时也保护了数据。对外界而已它的内部细节是隐藏的，暴露给外界的只是它的访问方法。

继承是为了重用父类代码。两个类若存在 IS-A 的关系就可以使用继承。，同时继承也为实现多态做了铺垫。那么什么是多态呢？多态的实现机制又是什么？请看我一一为你揭开：

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。因为在程序运行时才确定具体的类，这样，不用修改源程序代码，就可以让引用变量绑定到各种不同的类实现上，从而导致该引用调用的具体方法随之改变，即不修改程序代码就可以改变程序运行时所绑定的具体代码，让程序可以选择多个运行状态，这就是多态性。

比如你是一个酒神，对酒情有独钟。某日回家发现桌上有几个杯子里面都装了白酒，从外面看我们是不可能知道这是些什么酒，只有喝了之后才能够猜出来是何种酒。你一喝，这是剑南春、再喝这是五粮液、再喝这是酒鬼酒……在这里我们可以描述成如下：

酒 a = 剑南春

酒 b = 五粮液

酒 c = 酒鬼酒

…

这里所表现的的就是多态。剑南春、五粮液、酒鬼酒都是酒的子类，我们只是通过酒这一个父类就能够引用不同的子类，这就是多态——我们只有在运行的时候才会知道引用变量所指向的具体实例对象。

诚然，要理解多态我们就必须要明白什么是“向上转型”。在继承中我们简单介绍了向上转型，这里就在啰嗦下：在上面的喝酒例子中，酒（Win）是父类，剑南春（JNC）、五粮液（WLY）、酒鬼酒（JGJ）是子类。我们定义如下代码：

JNC a = new JNC();

对于这个代码我们非常容易理解无非就是实例化了一个剑南春的对象嘛！但是这样呢？

Wine a = new JNC();

在这里我们这样理解，这里定义了一个 Wine 类型的 a，它指向 JNC 对象实例。由于 JNC 是继承与 Wine，所以JNC可以自动向上转型为 Wine，所以 a 是可以指向 JNC 实例对象的。这样做存在一

个非常大的好处，在继承中我们知道子类是父类的扩展，它可以提供比父类更加强大的功能，如果我们定义了一个指向子类的父类引用类型，那么它除了能够引用父类的共性外，还可以使用子类强大的功能。

但是向上转型存在一些缺憾，那就是它必定会导致一些方法和属性的丢失，而导致我们不能够获取它们。所以父类类型的引用可以调用父类中定义的所有属性和方法，对于只存在与子类中的方法和属性它就望尘莫及了。

```
public class Wine {
    public void fun1(){
        System.out.println("Wine 的Fun.....");
        fun2();
    }

    public void fun2(){
        System.out.println("Wine 的Fun2...");
    }
}

public class JNC extends Wine{
    /**
     * @desc 子类重载父类方法
     *       父类中不存在该方法，向上转型后，父类是不能引用该方法的
     * @param a
     * @return void
     */
    public void fun1(String a){
        System.out.println("JNC 的 Fun1...");
        fun2();
    }

    /**
     * 子类重写父类方法
     * 指向子类的父类引用调用fun2时，必定是调用该方法
     */
    public void fun2(){
       System.out.println("JNC 的Fun2...");
  }
}

public class Test {
    public static void main(String[] args) {
        Wine a = new JNC();
        a.fun1();
    }
```

```
}
-----------------------------------------------
Output:
Wine 的Fun.....
JNC 的Fun2...
```

从程序的运行结果中我们发现，a.fun1()首先是运行父类 Wine 中的 fun1().然后再运行子类 JNC 中的 fun2()。

分析：在这个程序中子类 JNC 重载了父类 Wine 的方法 fun1()，重写 fun2()，而且重载后的 fun1(String a) 与 fun1() 不是同一个方法，由于父类中没有该方法，向上转型后会丢失该方法，所以执行 JNC 的 Wine 类型引用是不能引用 fun1(String a) 方法。而子类 JNC 重写了 fun2()，那么指向 JNC 的 Wine 引用会调用 JNC 中 fun2() 方法。

所以对于多态我们可以总结如下：

指向子类的父类引用由于向上转型了，它只能访问父类中拥有的方法和属性，而对于子类中存在而父类中不存在的方法，该引用是不能使用的，尽管是重载该方法。若子类重写了父类中的某些方法，在调用该些方法的时候，必定是使用子类中定义的这些方法（动态连接、动态调用）。

对于面向对象而已，多态分为编译时多态和运行时多态。其中编辑时多态是静态的，主要是指方法的重载，它是根据参数列表的不同来区分不同的函数，通过编辑之后会变成两个不同的函数，在运行时谈不上多态。而运行时多态是动态的，它是通过动态绑定来实现的，也就是我们所说的多态性。

多态的实现

2.1实现条件

在刚刚开始就提到了继承在为多态的实现做了准备。子类 Child 继承父类 Father，我们可以编写一个指向子类的父类类型引用，该引用既可以处理父类Father对象，也可以处理子类 Child 对象，当相同的消息发送给子类或者父类对象时，该对象就会根据自己所属的引用而执行不同的行为，这就是多态。即多态性就是相同的消息使得不同的类做出不同的响应。

Java 实现多态有三个必要条件：继承、重写、向上转型。

继承：在多态中必须存在有继承关系的子类和父类。

重写：子类对父类中某些方法进行重新定义，在调用这些方法时就会调用子类的方法。

向上转型：在多态中需要将子类的引用赋给父类对象，只有这样该引用才能够具备技能调用父类的方法和子类的方法。

只有满足了上述三个条件，我们才能够在同一个继承结构中使用统一的逻辑实现代码处理不同的对象，从而达到执行不同的行为。

对于 Java 而言，它多态的实现机制遵循一个原则：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。

2.2实现形式

在 Java 中有两种形式可以实现多态。继承和接口。

2.2.1.基于继承实现的多态

基于继承的实现机制主要表现在父类和继承该父类的一个或多个子类对某些方法的重写，多个子类对同一方法的重写可以表现出不同的行为。

```
public class Wine {
    private String name;
    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Wine(){
    }

    public String drink(){
        return "喝的是 " + getName();
    }

    /**
     * 重写toString()
     */
    public String toString(){
        return null;
    }
}

public class JNC extends Wine{
    public JNC(){
        setName("JNC");
    }

    /**
     * 重写父类方法，实现多态
     */
    public String drink(){
        return "喝的是 " + getName();
    }

    /**
     * 重写toString()
     */
    public String toString(){
        return "Wine : " + getName();
    }
}

public class JGJ extends Wine{
    public JGJ(){
        setName("JGJ");
```

```
    }

    /**
     * 重写父类方法，实现多态
     */
    public String drink(){
        return "喝的是 " + getName();
    }

    /**
     * 重写toString()
     */
    public String toString(){
        return "Wine : " + getName();
    }
}

public class Test {
    public static void main(String[] args) {
        //定义父类数组
        Wine[] wines = new Wine[2];
        //定义两个子类
        JNC jnc = new JNC();
        JGJ jgj = new JGJ();

        //父类引用子类对象
        wines[0] = jnc;
        wines[1] = jgj;

        for(int i = 0 ; i < 2 ; i++){
            System.out.println(wines[i].toString() + "--" + wines[i].drink());
        }
        System.out.println("--------------------------------");

    }
}
OUTPUT:
Wine : JNC--喝的是 JNC
Wine : JGJ--喝的是 JGJ
--------------------------------
```

在上面的代码中 JNC、JGJ 继承 Wine，并且重写了 drink()、toString() 方法，程序运行结果是调用子类中方法，输出 JNC、JGJ 的名称，这就是多态的表现。不同的对象可以执行相同的行为，但是他们都需要通过自己的实现方式来执行，这就要得益于向上转型了。

我们都知道所有的类都继承自超类 Object，toString() 方法也是 Object 中方法，当我们这样写时：

```
Object o = new JGJ();

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

输出的结果是 Wine : JGJ。

Object、Wine、JGJ 三者继承链关系是：JGJ—>Wine—>Object。所以我们可以这样说：当子类重写父类的方法被调用时，只有对象继承链中的最末端的方法才会被调用。但是注意如果这样写：

```
Object o = new Wine();

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

输出的结果应该是 Null，因为 JGJ 并不存在于该对象继承链中。

所以基于继承实现的多态可以总结如下：对于引用子类的父类类型，在处理该引用时，它适用于继承该父类的所有子类，子类对象的不同，对方法的实现也就不同，执行相同动作产生的行为也就不同。

如果父类是抽象类，那么子类必须要实现父类中所有的抽象方法，这样该父类所有的子类一定存在统一的对外接口，但其内部的具体实现可以各异。这样我们就可以使用顶层类提供的统一接口来处理该层次的方法。

2.2.2.基于接口实现的多态

继承是通过重写父类的同一方法的几个不同子类来体现的，那么就可就是通过实现接口并覆盖接口中同一方法的几不同的类体现的。

在接口的多态中，指向接口的引用必须是指定这实现了该接口的一个类的实例程序，在运行时，根据对象引用的实际类型来执行对应的方法。

继承都是单继承，只能为一组相关的类提供一致的服务接口。但是接口可以是多继承多实现，它能够利用一组相关或者不相关的接口进行组合与扩充，能够对外提供一致的服务接口。所以它相对于继承来说有更好的灵活性。

经典实例

通过上面的讲述，可以说是对多态有了一定的了解。现在趁热打铁，看一个实例。该实例是有关多态的经典例子，摘自：http://blog.csdn.net/thinkGhoster/archive/2008/04/19/2307001.aspx 。

```
public class A {
    public String show(D obj) {
        return ("A and D");
    }

    public String show(A obj) {
        return ("A and A");
    }

}

public class B extends A{
    public String show(B obj){
        return ("B and B");
    }

    public String show(A obj){
        return ("B and A");
    }
}

public class C extends B{

}

public class D extends B{

}

public class Test {
    public static void main(String[] args) {
        A a1 = new A();
        A a2 = new B();
        B b = new B();
        C c = new C();
        D d = new D();
```

```
        System.out.println("1--" + a1.show(b));
        System.out.println("2--" + a1.show(c));
        System.out.println("3--" + a1.show(d));
        System.out.println("4--" + a2.show(b));
        System.out.println("5--" + a2.show(c));
        System.out.println("6--" + a2.show(d));
        System.out.println("7--" + b.show(b));
        System.out.println("8--" + b.show(c));
        System.out.println("9--" + b.show(d));
    }
}
```

运行结果：

```
1--A and A
2--A and A
3--A and D
4--B and A
5--B and A
6--A and D
7--B and B
8--B and B
9--A and D
```

在这里看结果1、2、3还好理解，从4开始就开始糊涂了，对于4来说为什么输出不是“B and B”呢?

首先我们先看一句话：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。这句话对多态进行了一个概括。其实在继承链中对象方法的调用存在一个优先级：this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)。

分析：

从上面的程序中我们可以看出 A、B、C、D 存在如下关系。

首先我们分析5，a2.show(c)，a2 是 A 类型的引用变量，所以 this 就代表了 A，a2.show(c),它在 A 类中找发现没有找到，于是到 A 的超类中找(super)，由于 A 没有超类（Object除外），所以跳到第三级，也就是 this.show((super)O)，C 的超类有 B、A，所以 (super)O 为 B、A，this 同样是 A，这里在 A 中找到了 show(A obj)，同时由于 a2 是 B 类的一个引用且 B 类重写了 show(A obj)，因此最终会调用子类B类的 show(A obj) 方法，结果也就是 B and A。

按照同样的方法我也可以确认其他的答案。

方法已经找到了但是我们这里还是存在一点疑问，我们还是来看这句话：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法。这我们用一个例子来说明这句话所代表的含义：a2.show(b);

这里 a2 是引用变量，为 A 类型，它引用的是 B 对象，因此按照上面那句话的意思是说有B来决定调用谁的方法,所以 a2.show(b) 应该要调用B中的 show(B obj)，产生的结果应该是“B and B”，但是为什么会与前面的运行结果产生差异呢？这里我们忽略了后面那句话“但是这儿被调用的方法必须是在超类中定义过的”，那么 show(B obj) 在 A 类中存在吗？根本就不存在！所以这句话在这里不适用？那么难道是这句话错误了？非也！其实这句话还隐含这这句话：它仍然要按照继承链中调用方法的优先级来确认。所以它才会在 A 类中找到 show(A obj)，同时由于 B 重写了该方法所以才会调用 B 类中的方法，否则就会调用 A 类中的方法。

所以多态机制遵循的原则概括为：当超类对象引用变量引用子类对象时，被引用对象的类型而不是引用变量的类型决定了调用谁的成员方法，但是这个被调用的方法必须是在超类中定义过的，也就是说被子类覆盖的方法，但是它仍然要根据继承链中方法调用的优先级来确认方法，该优先级为：this.show(O)、super.show(O)、this.show((super)O)、super.show((super)O)。

参考资料：http://blog.csdn.net/thinkGhoster/archive/2008/04/19/2307001.aspx。

百度文库：http://wenku.baidu.com/view/73f66f92daef5ef7ba0d3c03.html

在这里面向对象的三大特性已经介绍完成，下一步继续是 java 基础部分—巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

4

Java 的四舍五入

Java小事非小事！！！！！！！！！！！

四舍五入是我们小学的数学问题，这个问题对于我们程序猿来说就类似于1到10的加减乘除那么简单了。在讲解之间我们先看如下一个经典的案例：

```
public static void main(String[] args) {
        System.out.println("12.5的四舍五入值：" + Math.round(12.5));
        System.out.println("-12.5的四舍五入值：" + Math.round(-12.5));
    }
Output:
 12.5的四舍五入值：13
 -12.5的四舍五入值：-12
```

这是四舍五入的经典案例，也是我们参加校招时候经常会遇到的(貌似我参加笔试的时候遇到过好多次)。从这儿结果中我们发现这两个绝对值相同的数字，为何近似值会不同呢？其实这与 Math.round 采用的四舍五入规则来决定。

四舍五入其实在金融方面运用的非常多，尤其是银行的利息。我们都知道银行的盈利渠道主要是利息差，它从储户手里收集资金，然后放贷出去，期间产生的利息差就是银行所获得的利润。如果我们采用平常四舍五入的规则话，这里采用每 10 笔存款利息计算作为模型，如下：

四舍：0.000、0.001、0.002、0.003、0.004。这些舍的都是银行赚的钱。

五入：0.005、0.006、0.007、0.008、0.009。这些入的都是银行亏的钱，分别为：0.005、0.004、.003、0.002、0.001。

所以对于银行来说它的盈利应该是 0.000 + 0.001 + 0.002 + 0.003 + 0.004 - 0.005 - 0.004 - 0.003 - 0.002 - 0.001 = -0.005。从结果中可以看出每 10 笔的利息银行可能就会损失 0.005 元，千万别小看这个数字，这对于银行来说就是一笔非常大的损失。面对这个问题就产生了如下的银行家涉入法了。该算法是由美国银行家提出了，主要用于修正采用上面四舍五入规则而产生的误差。如下：

舍去位的数值小于 5 时，直接舍去。

舍去位的数值大于 5 时，进位后舍去。

当舍去位的数值等于 5 时，若 5 后面还有其他非 0 数值，则进位后舍去，若 5 后面是 0 时，则根据5前一位数的奇偶性来判断，奇数进位，偶数舍去。

对于上面的规则我们举例说明

11.556 = 11.56 ——六入

11.554 = 11.55 一 - 四舍

11.5551 = 11.56 一 - 五后有数进位

11.545 = 11.54 一 - 五后无数，若前位为偶数应舍去

11.555 = 11.56 一 - 五后无数，若前位为奇数应进位

下面实例是使用银行家舍入法：

```
public static void main(String[] args) {
        BigDecimal d = new BigDecimal(100000);     //存款
        BigDecimal r = new BigDecimal(0.001875*3);   //利息
        BigDecimal i = d.multiply(r).setScale(2,RoundingMode.HALF_EVEN);     //使用银行家算法

        System.out.println("季利息是："+i);
        }
Output:
季利息是：562.50
```

在上面简单地介绍了银行家舍入法，目前 Java 支持7中舍入法：

1、 ROUND_UP：远离零方向舍入。向绝对值最大的方向舍入，只要舍弃位非0即进位。

2、 ROUND_DOWN：趋向零方向舍入。向绝对值最小的方向输入，所有的位都要舍弃，不存在进位情况。

3、 ROUND_CEILING：向正无穷方向舍入。向正最大方向靠拢。若是正数，舍入行为类似于 ROUND_UP，若为负数，舍入行为类似于 ROUND_DOWN。 Math.round() 方法就是使用的此模式。

4、 ROUND_FLOOR：向负无穷方向舍入。向负无穷方向靠拢。若是正数，舍入行为类似于 ROUND_DOWN；若为负数，舍入行为类似于 ROUND_UP。

5、 HALF_UP：最近数字舍入(5进)。这是我们最经典的四舍五入。

6、 HALF_DOWN：最近数字舍入(5舍)。在这里5是要舍弃的。

7、 HAIL_EVEN：银行家舍入法。

提到四舍五入那么保留位就必不可少了，在 Java 运算中我们可以使用多种方式来实现保留位。

保留位

方式一：四舍五入

```
double f = 111231.5585;
BigDecimal b = new BigDecimal(f);
double f1 = b.setScale(2, RoundingMode.HALF_UP).doubleValue();
```

在这里使用 BigDecimal ，并且采用 setScale 方法来设置精确度，同时使用 RoundingMode.HALF_UP 表示使用最近数字舍入法则来近似计算。在这里我们可以看出 BigDecimal 和四舍五入是绝妙的搭配。

方式二：

```
java.text.DecimalFormat df =new java.text.DecimalFormat(”#.00”);
df.format(你要格式化的数字);
```

例：new java.text.DecimalFormat(”#.00”).format(3.1415926)

#.00 表示两位小数 #.0000四位小数 以此类推…

方式三：

```
double d = 3.1415926;

String result = String .format(”%.2f”);

%.2f %. 表示 小数点前任意位数 2 表示两位小数 格式后的结果为f 表示浮点型。
```

方式四：

此外如果使用 struts 标签做输出的话，有个 format 属性,设置为 format=”0.00”就是保留两位小数

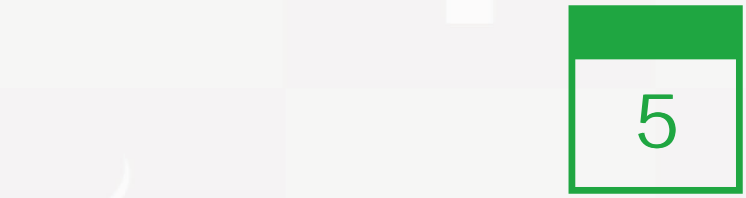
例如：

```
<bean:write name="entity" property="dkhAFSumPl"  format="0.00" />

或者

<fmt:formatNumber type="number" value="${10000.22/100}" maxFractionDigits="0"/>

maxFractionDigits表示保留的位数
```

不积跬步，无以至千里。

不积小流，无以成江海。

——荀子《劝学篇》



5

抽象类与接口

接口和内部类为我们提供了一种将接口与实现分离的更加结构化的方法。

抽象类与接口是 Java 语言中对抽象概念进行定义的两种机制，正是由于他们的存在才赋予 Java 强大的面向对象的能力。他们两者之间对抽象概念的支持有很大的相似，甚至可以互换，但是也有区别。

一、抽象类

我们都知道在面向对象的领域一切都是对象，同时所有的对象都是通过类来描述的，但是并不是所有的类都是来描述对象的。如果一个类没有足够的信息来描述一个具体的对象，而需要其他具体的类来支撑它，那么这样的类我们称它为抽象类。比如 new Animal()，我们都知道这个是产生一个动物 Animal 对象，但是这个 Animal 具体长成什么样子我们并不知道，它没有一个具体动物的概念，所以他就是一个抽象类，需要一个具体的动物，如狗、猫来对它进行特定的描述，我们才知道它长成啥样。

在面向对象领域由于抽象的概念在问题领域没有对应的具体概念，所以用以表征抽象概念的抽象类是不能实例化的。

同时，抽象类体现了数据抽象的思想，是实现多态的一种机制。它定义了一组抽象的方法，至于这组抽象方法的具体表现形式有派生类来实现。同时抽象类提供了继承的概念，它的出发点就是为了继承，否则它没有存在的任何意义。所以说定义的抽象类一定是用来继承的，同时在一个以抽象类为节点的继承关系等级链中，叶子节点一定是具体的实现类。（ 不知这样理解是否有错!!!高手指点…. ）

在使用抽象类时需要注意几点：

1、抽象类不能被实例化，实例化的工作应该交由它的子类来完成，它只需要有一个引用即可。

2、抽象方法必须由子类来进行重写。

3、只要包含一个抽象方法的抽象类，该方法必须要定义成抽象类，不管是否还包含有其他方法。

4、抽象类中可以包含具体的方法，当然也可以不包含抽象方法。

5、子类中的抽象方法不能与父类的抽象方法同名。

6、abstract 不能与 final 并列修饰同一个类。

7、abstract 不能与 private、static、final 或 native 并列修饰同一个方法。、

实例：

定义一个抽象动物类 Animal，提供抽象方法叫 cry()，猫、狗都是动物类的子类，由于 cry() 为抽象方法，所以 Cat、Dog 必须要实现 cry() 方法。如下：

```
public abstract class Animal {
    public abstract void cry();
}
```

```
public class Cat extends Animal{

    @Override
    public void cry() {
        System.out.println("猫叫：喵喵...");
    }
}

public class Dog extends Animal{

    @Override
    public void cry() {
        System.out.println("狗叫:汪汪...");
    }

}

public class Test {

    public static void main(String[] args) {
        Animal a1 = new Cat();
        Animal a2 = new Dog();

        a1.cry();
        a2.cry();
    }
}

----------------------------------------------------------------------
Output:
猫叫：喵喵...
狗叫:汪汪...
```

创建抽象类和抽象方法非常有用,因为他们可以使类的抽象性明确起来,并告诉用户和编译器打算怎样使用他们.抽象类还是有用的重构器,因为它们使我们可以很容易地将公共方法沿着继承层次结构向上移动。（From:Think in Java ）

二、接口

接口是一种比抽象类更加抽象的“类”。这里给“类”加引号是我找不到更好的词来表示，但是我们要明确一点就是，接口本身就不是类，从我们不能实例化一个接口就可以看出。如 new Runnable();肯定是错误的，我们只能 new 它的实现类。

接口是用来建立类与类之间的协议，它所提供的只是一种形式，而没有具体的实现。同时实现该接口的实现类必须要实现该接口的所有方法，通过使用 implements 关键字，他表示该类在遵循某个或某组特定的接口，同时也表示着"interface"只是它的外貌，但是现在需要声明它是如何工作的”。

接口是抽象类的延伸，java 了保证数据安全是不能多重继承的，也就是说继承只能存在一个父类，但是接口不同，一个类可以同时实现多个接口，不管这些接口之间有没有关系，所以接口弥补了抽象类不能多重继承的缺陷，但是推荐继承和接口共同使用，因为这样既可以保证数据安全性又可以实现多重继承。

在使用接口过程中需要注意如下几个问题：

1、1个 Interface 的方所有法访问权限自动被声明为 public。确切的说只能为 public，当然你可以显示的声明为 protected、private，但是编译会出错！

2、接口中可以定义“成员变量”，或者说是不可变的常量，因为接口中的“成员变量”会自动变为为 public static final。可以通过类命名直接访问：ImplementClass.name。

3、接口中不存在实现的方法。

4、实现接口的非抽象类必须要实现该接口的所有方法。抽象类可以不用实现。

5、不能使用 new 操作符实例化一个接口，但可以声明一个接口变量，该变量必须引用 (refer to) 一个实现该接口的类的对象。可以使用 instanceof 检查一个对象是否实现了某个特定的接口。例如：if(anObject instanceof Comparable){}。

6、在实现多接口的时候一定要避免方法名的重复。

三、抽象类与接口的区别

尽管抽象类和接口之间存在较大的相同点，甚至有时候还可以互换，但这样并不能弥补他们之间的差异之处。下面将从语法层次和设计层次两个方面对抽象类和接口进行阐述。

3.1语法层次

在语法层次，java 语言对于抽象类和接口分别给出了不同的定义。下面已 Demo 类来说明他们之间的不同之处。

使用抽象类来实现:

```
public abstract class Demo {
    abstract void method1();


    void method2(){
        //实现
    }
}
```

使用接口来实现

```
interface Demo {
    void method1();
    void method2();
}
```

抽象类方式中，抽象类可以拥有任意范围的成员数据，同时也可以拥有自己的非抽象方法，但是接口方式中，它仅能够有静态、不能修改的成员数据（但是我们一般是不会在接口中使用成员数据），同时它所有的方法都必须是抽象的。在某种程度上来说，接口是抽象类的特殊化。

对子类而言，它只能继承一个抽象类（这是 java 为了数据安全而考虑的），但是却可以实现多个接口。

3.2设计层次

上面只是从语法层次和编程角度来区分它们之间的关系，这些都是低层次的，要真正使用好抽象类和接口，我们就必须要从较高层次来区分了。只有从设计理念的角度才能看出它们的本质所在。一般来说他们存在如下三个不同点：

1、 抽象层次不同。抽象类是对类抽象，而接口是对行为的抽象。抽象类是对整个类整体进行抽象，包括属性、行为，但是接口却是对类局部（行为）进行抽象。

2、 跨域不同。抽象类所跨域的是具有相似特点的类，而接口却可以跨域不同的类。我们知道抽象类是从子类中发现公共部分，然后泛化成抽象类，子类继承该父类即可，但是接口不同。实现它的子类可以不存在任何关系，共同之处。例如猫、狗可以抽象成一个动物类抽象类，具备叫的方法。鸟、飞机可以实现飞 Fly 接口，具备飞的行为，这里我们总不能将鸟、飞机共用一个父类吧！所以说抽象类所体现的是一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is-a"关系，即父类和派生类在概念本质上应该是相同的。对于接口则不然，并不要求接口的实现者和接口定义在概念本质上是一致的， 仅仅是实现了接口定义的契约而已。

3、 设计层次不同。对于抽象类而言，它是自下而上来设计的，我们要先知道子类才能抽象出父类，而接口则不同，它根本就不需要知道子类的存在，只需要定义一个规则即可，至于什么子类、什么时候怎么实现它一概不知。比如我们只有一个猫类在这里，如果你这是就抽象成一个动物类，是不是设计有点儿过度？我们起码要有两个动物类，猫、狗在这里，我们在抽象他们的共同点形成动物抽象类吧！所以说抽象类往往都是通过重构而来的！但是接口就不同，比如说飞，我们根本就不知道会有什么东西来实现这个飞接口，怎么实现也不得而知，我们要做的就是事前定义好飞的行为接口。所以说抽象类是自底向上抽象而来的，接口是自顶向下设计出来的。

（上面纯属个人见解，如有出入、错误之处，望各位指点！！！！）

为了更好的阐述他们之间的区别，下面将使用一个例子来说明。该例子引自：http://blog.csdn.net/ttgjz/article/details/2960451

我们有一个 Door 的抽象概念，它具备两个行为 open() 和 close()，此时我们可以定义通过抽象类和接口来定义这个抽象概念：

抽象类：

```
abstract class Door{
    abstract void open();
    abstract void close();
}
```

接口

```
interface Door{
    void open();
    void close();
}
```

至于其他的具体类可以通过使用 extends 使用抽象类方式定义 Door 或者 Implements 使用接口方式定义 Door，这里发现两者并没有什么很大的差异。

但是现在如果我们需要门具有报警的功能，那么该如何实现呢?

解决方案一：给 Door 增加一个报警方法:clarm();

```
abstract class Door{
    abstract void open();
    abstract void close();
    abstract void alarm();
}
```

或者

```
interface Door{
    void open();
    void close();
    void alarm();
}
```

这种方法违反了面向对象设计中的一个核心原则 ISP (Interface Segregation Principle)一见批注，在 Door 的定义中把 Door 概念本身固有的行为方法和另外一个概念"报警器"的行为方 法混在了一起。这样引起的一个问题是那些仅仅依赖于 Door 这个概念的模块会因为”报警器”这个概念的改变而改变，反之依然。

解决方案二

既然 open()、close() 和alarm() 属于两个不同的概念，那么我们依据 ISP 原则将它们分开定义在两个代表两个不同概念的抽象类里面，定义的方式有三种：

1、两个都使用抽象类来定义。

2、两个都使用接口来定义。

3、一个使用抽象类定义，一个是用接口定义。

由于 Java 不支持多继承所以第一种是不可行的。后面两种都是可行的，但是选择何种就反映了你对问题域本质的理解。

如果选择第二种都是接口来定义，那么就反映了两个问题：1、我们可能没有理解清楚问题域，AlarmDoor 在概念本质上到底是门还报警器。2、如果我们对问题域的理解没有问题，比如我们在分析时确定了 AlarmDoor 在本质上概念是一致的，那么我们在设计时就没有正确的反映出我们的设计意图。因为你使用了两个接口来进行定义，他们概念的定义并不能够反映上述含义。

第三种，如果我们对问题域的理解是这样的：AlarmDoor 本质上 Door，但同时它也拥有报警的行为功能，这个时候我们使用第三种方案恰好可以阐述我们的设计意图。AlarmDoor 本质是们，所以对于这个概念我们使用抽象类来定义，同时 AlarmDoor 具备报警功能，说明它能够完成报警概念中定义的行为功能，所以 alarm 可以使用接口来进行定义。如下：

```
abstract class Door{
    abstract void open();
    abstract void close();
}

interface Alarm{
    void alarm();
}

class AlarmDoor extends Door implements Alarm{
    void open(){}
    void close(){}
    void alarm(){}
}
```

这种实现方式基本上能够明确的反映出我们对于问题领域的理解，正确的揭示我们的设计意图。其实抽象类表示的是"is-a"关系，接口表示的是"like-a"关系，大家在选择时可以作为一个依据，当然这是建立在对问题领域的理解上的，比如：如果我们认为 AlarmDoor 在概念本质上是报警器，同时又具有 Door 的功能，那么上述的定义方式就要反过来了。

批注： ISP（Interface Segregation Principle）`：面向对象的一个核心原则。它表明使用多个专门的接口比使用单一的总接口要好。

一个类对另外一个类的依赖性应当是建立在最小的接口上的。

一个接口代表一个角色，不应当将不同的角色都交给一个接口。没有关系的接口合并在一起，形成一个臃肿的大接口，这是对角色和接口的污染。

四、总结

1、抽象类在 Java 语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。

2、在抽象类中可以拥有自己的成员变量和非抽象类方法，但是接口中只能存在静态的不可变的成员数据（不过一般都不在接口中定义成员数据），而且它的所有方法都是抽象的。

3、抽象类和接口所反映的设计理念是不同的，抽象类所代表的是“is-a”的关系，而接口所代表的是“like-a”的关系。

抽象类和接口是 Java 语言中两种不同的抽象概念，他们的存在对多态提供了非常好的支持，虽然他们之间存在很大的相似性。但是对于他们的选择往往反应了您对问题域的理解。只有对问题域的本质有良好的理解，才能做出正确、合理的设计。

`巩固基础，提高技术，不惧困难，攀登高峰！！！！！！`

6

使用序列化实现对象的拷贝

我们知道在 Java 中存在这个接口 Cloneable，实现该接口的类都会具备被拷贝的能力，同时拷贝是在内存中进行，在性能方面比我们直接通过 new 生成对象来的快，特别是在大对象的生成上，使得性能的提升非常明显。然而我们知道拷贝分为深拷贝和浅拷贝之分，但是浅拷贝存在对象属性拷贝不彻底问题。关于深拷贝、浅拷贝的请参考这里：渐析 java 的浅拷贝和深拷贝

一、浅拷贝问题

我们先看如下代码：

```
public class Person implements Cloneable{
    /** 姓名 **/
    private String name;

    /** 电子邮件 **/
    private Email email;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Email getEmail() {
        return email;
    }

    public void setEmail(Email email) {
        this.email = email;
    }

    public Person(String name,Email email){
        this.name  = name;
        this.email = email;
    }

    public Person(String name){
        this.name = name;
    }

    protected Person clone() {
        Person person = null;
        try {
            person = (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
```

```
        }

        return person;
    }
}

public class Client {
    public static void main(String[] args) {
        //写封邮件
        Email email = new Email("请参加会议","请与今天12:30到二会议室参加会议...");

        Person person1 =  new Person("张三",email);

        Person person2 =  person1.clone();
        person2.setName("李四");
        Person person3 =  person1.clone();
        person3.setName("王五");

        System.out.println(person1.getName() + "的邮件内容是：" + person1.getEmail().getContent());
        System.out.println(person2.getName() + "的邮件内容是：" + person2.getEmail().getContent());
        System.out.println(person3.getName() + "的邮件内容是：" + person3.getEmail().getContent());
    }
}
--------------------
Output:
张三的邮件内容是：请与今天12:30到二会议室参加会议...
李四的邮件内容是：请与今天12:30到二会议室参加会议...
王五的邮件内容是：请与今天12:30到二会议室参加会议...
```

在该应用程序中，首先定义一封邮件，然后将该邮件发给张三、李四、王五三个人，由于他们是使用相同的邮件，并且仅有名字不同，所以使用张三该对象类拷贝李四、王五对象然后更改下名字即可。程序一直到这里都没有错，但是如果我们需要张三提前 30 分钟到，即把邮件的内容修改下：

```
public class Client {
    public static void main(String[] args) {
        //写封邮件
        Email email = new Email("请参加会议","请与今天12:30到二会议室参加会议...");

        Person person1 =  new Person("张三",email);

        Person person2 =  person1.clone();
        person2.setName("李四");
        Person person3 =  person1.clone();
        person3.setName("王五");
```

```
        person1.getEmail().setContent("请与今天12:00到二会议室参加会议...");

        System.out.println(person1.getName() + "的邮件内容是：" + person1.getEmail().getContent());
        System.out.println(person2.getName() + "的邮件内容是：" + person2.getEmail().getContent());
        System.out.println(person3.getName() + "的邮件内容是：" + person3.getEmail().getContent());
    }
}
```

在这里同样是使用张三该对象实现对李四、王五拷贝，最后将张三的邮件内容改变为：请与今天 12:00 到二会议室参加会议…。但是结果是：

```
张三的邮件内容是：请与今天12:00到二会议室参加会议...
李四的邮件内容是：请与今天12:00到二会议室参加会议...
王五的邮件内容是：请与今天12:00到二会议室参加会议...
```

这里我们就疑惑了为什么李四和王五的邮件内容也发送了改变呢？让他们提前30分钟到人家会有意见的！

其实出现问题的关键就在于 clone() 方法上，我们知道该 clone() 方法是使用 Object 类的 clone() 方法，但是该方法存在一个缺陷，它并不会将对象的所有属性全部拷贝过来，而是有选择性的拷贝，基本规则如下：

1、基本类型

如果变量是基本很类型，则拷贝其值，比如 int、float 等。

2、对象

如果变量是一个实例对象，则拷贝其地址引用，也就是说此时新对象与原来对象是公用该实例变量。

3、String 字符串

若变量为 String 字符串，则拷贝其地址引用。但是在修改时，它会从字符串池中重新生成一个新的字符串，原有紫都城对象保持不变。

基于上面上面的规则，我们很容易发现问题的所在，他们三者公用一个对象，张三修改了该邮件内容，则李四和王五也会修改，所以才会出现上面的情况。对于这种情况我们还是可以解决的，只需要在 clone() 方法里面新建一个对象，然后张三引用该对象即可：

```
protected Person clone() {
        Person person = null;
        try {
            person = (Person) super.clone();
            person.setEmail(new Email(person.getEmail().getObject(),person.getEmail().getContent()));
```

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

        return person;
    }
```

所以：浅拷贝只是 Java 提供的一种简单的拷贝机制，不便于直接使用。

对于上面的解决方案还是存在一个问题，若我们系统中存在大量的对象是通过拷贝生成的，如果我们每一个类都写一个 clone() 方法，并将还需要进行深拷贝，新建大量的对象，这个工程是非常大的，这里我们可以利用序列化来实现对象的拷贝。

二、利用序列化实现对象的拷贝

如何利用序列化来完成对象的拷贝呢？在内存中通过字节流的拷贝是比较容易实现的。把母对象写入到一个字节流中，再从字节流中将其读出来，这样就可以创建一个新的对象了，并且该新对象与母对象之间并不存在引用共享的问题，真正实现对象的深拷贝。

```
public class CloneUtils {
    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T   obj){
        T cloneObj = null;
        try {
            //写入字节流
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            ObjectOutputStream obs = new   ObjectOutputStream(out);
            obs.writeObject(obj);
            obs.close();

            //分配内存，写入原始对象，生成新对象
            ByteArrayInputStream ios = new  ByteArrayInputStream(out.toByteArray());
            ObjectInputStream ois = new ObjectInputStream(ios);
            //返回生成的新对象
            cloneObj = (T) ois.readObject();
            ois.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return cloneObj;
    }
}
```

使用该工具类的对象必须要实现 Serializable 接口，否则是没有办法实现克隆的。

```
public class Person implements Serializable{
    private static final long serialVersionUID = 2631590509760908280L;

    ..................
    //去除clone()方法

}

public class Email implements Serializable{
```

```
    private static final long serialVersionUID = 1267293988171991494L;

    ..................
}
```

所以使用该工具类的对象只要实现 Serializable 接口就可实现对象的克隆，无须继承 Cloneable 接口实现 clone() 方法。

```
public class Client {
    public static void main(String[] args) {
        //写封邮件
        Email email = new Email("请参加会议","请与今天12:30到二会议室参加会议...");

        Person person1 =  new Person("张三",email);

        Person person2 =  CloneUtils.clone(person1);
        person2.setName("李四");
        Person person3 =  CloneUtils.clone(person1);
        person3.setName("王五");
        person1.getEmail().setContent("请与今天12:00到二会议室参加会议...");

        System.out.println(person1.getName() + "的邮件内容是：" + person1.getEmail().getContent());
        System.out.println(person2.getName() + "的邮件内容是：" + person2.getEmail().getContent());
        System.out.println(person3.getName() + "的邮件内容是：" + person3.getEmail().getContent());
    }
}
--------------------
Output:
张三的邮件内容是：请与今天12:00到二会议室参加会议...
李四的邮件内容是：请与今天12:30到二会议室参加会议...
王五的邮件内容是：请与今天12:30到二会议室参加会议...
```

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

参考文献《 编写高质量代码 改善Java程序的151个建议 》——秦小波

7

关键字 static

一、 static 代表着什么

在 Java 中并不存在全局变量的概念，但是我们可以通过 static 来实现一个“伪全局”的概念，在 Java 中 static 表示“全局”或者“静态”的意思，用来修饰成员变量和成员方法，当然也可以修饰代码块。

Java 把内存分为栈内存和堆内存，其中栈内存用来存放一些基本类型的变量、数组和对象的引用，堆内存主要存放一些对象。在 JVM 加载一个类的时候，若该类存在 static 修饰的成员变量和成员方法，则会为这些成员变量和成员方法在固定的位置开辟一个固定大小的内存区域，有了这些“固定”的特性，那么 JVM 就可以非常方便地访问他们。同时如果静态的成员变量和成员方法不出作用域的话，它们的句柄都会保持不变。同时 static 所蕴含“静态”的概念表示着它是不可恢复的，即在那个地方，你修改了，他是不会变回原样的，你清理了，他就不会回来了。

同时被 static 修饰的成员变量和成员方法是独立于该类的，它不依赖于某个特定的实例变量，也就是说它被该类的所有实例共享。所有实例的引用都指向同一个地方，任何一个实例对其的修改都会导致其他实例的变化。

```
public class User {
    private static int userNumber  = 0 ;

    public User(){
        userNumber ++;
    }

    public static void main(String[] args) {
        User user1 = new User();
        User user2 = new User();

        System.out.println("user1 userNumber：" + User.userNumber);
        System.out.println("user2 userNumber：" + User.userNumber);
    }
}
------------
Output:
user1 userNumber：2
user2 userNumber：2
```

二、 怎么使用 static

static 可以用于修饰成员变量和成员方法，我们将其称之为静态变量和静态方法，直接通过类名来进行访问。

ClassName.propertyName

ClassName.methodName(……)

Static 修饰的代码块表示静态代码块，当 JVM 装载类的时候，就会执行这块代码，其用处非常大。（对于代码块的使用这几天介绍，敬请关注）

2.1、static 变量

static 修饰的变量我们称之为静态变量，没有用 static 修饰的变量称之为实例变量，他们两者的区别是：

静态变量是随着类加载时被完成初始化的，它在内存中仅有一个，且 JVM 也只会为它分配一次内存，同时类所有的实例都共享静态变量，可以直接通过类名来访问它。

但是实例变量则不同，它是伴随着实例的，每创建一个实例就会产生一个实例变量，它与该实例同生共死。

所以我们一般在这两种情况下使用静态变量：对象之间共享数据、访问方便。

2.2、static 方法

static 修饰的方法我们称之为静态方法，我们通过类名对其进行直接调用。由于他在类加载的时候就存在了，它不依赖于任何实例，所以 static 方法必须实现，也就是说他不能是抽象方法 abstract。

Static 方法是类中的一种特殊方法，我们只有在真正需要他们的时候才会将方法声明为 static。如 Math 类的所有方法都是静态 static 的。

2.3、static 代码块

被 static 修饰的代码块，我们称之为静态代码块，静态代码块会随着类的加载一块执行，而且他可以随意放，可以存在于该了的任何地方。

三、 Static 的局限

Static 确实是存在诸多的作用，但是它也存在一些缺陷。

1、它只能调用 static 变量。

2、它只能调用 static 方法。

3、不能以任何形式引用 this、super。

4、static 变量在定义时必须要进行初始化，且初始化时间要早于非静态变量。

总结：无论是变量，方法，还是代码块，只要用 static 修饰，就是在类被加载时就已经”准备好了”,也就是可以被使用或者已经被执行，都可以脱离对象而执行。反之，如果没有 static，则必须要依赖于对象实例。

8

详解内部类

可以将一个类的定义放在另一个类的定义内部，这就是内部类。

内部类是一个非常有用的特性但又比较难理解使用的特性(鄙人到现在都没有怎么使用过内部类，对内部类也只是略知一二)。

第一次见面

内部类我们从外面看是非常容易理解的，无非就是在一个类的内部在定义一个类。

```
public class OuterClass {
    private String name ;
    private int 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;
    }

    class InnerClass{
        public InnerClass(){
            name = "chenssy";
            age = 23;
        }
    }
}
```

在这里 InnerClass 就是内部类，对于初学者来说内部类实在是使用的不多，鄙人菜鸟一个同样没有怎么使用过(貌似仅仅只在做 swing 注册事件中使用过)，但是随着编程能力的提高，我们会领悟到它的魅力所在，它可以使用能够更加优雅的设计我们的程序结构。在使用内部类之间我们需要明白为什么要使用内部类，内部类能够为我们带来什么样的好处。

一、为什么要使用内部类

为什么要使用内部类？在《Think in java》中有这样一句话：使用内部类最吸引人的原因是：每个内部类都能独立地继承一个（接口的）实现，所以无论外围类是否已经继承了某个（接口的）实现，对于内部类都没有影响。

在我们程序设计中有时候会存在一些使用接口很难解决的问题，这个时候我们可以利用内部类提供的、可以继承多个具体的或者抽象的类的能力来解决这些程序设计问题。可以这样说，接口只是解决了部分问题，而内部类使得多重继承的解决方案变得更加完整。

```
public interface Father {

}

public interface Mother {

}

public class Son implements Father, Mother {

}

public class Daughter implements Father{

    class Mother_ implements Mother{

    }
}
```

其实对于这个实例我们确实是看不出来使用内部类存在何种优点，但是如果 Father、Mother 不是接口，而是抽象类或者具体类呢？这个时候我们就只能使用内部类才能实现多重继承了。

其实使用内部类最大的优点就在于它能够非常好的解决多重继承的问题，但是如果我们不需要解决多重继承问题，那么我们自然可以使用其他的编码方式，但是使用内部类还能够为我们带来如下特性（摘自《Think in java》）：

1、内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。

2、在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。

3、创建内部类对象的时刻并不依赖于外围类对象的创建。

4、内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。

5、内部类提供了更好的封装，除了该外围类，其他类都不能访问。

二、内部类基础

在这个部分主要介绍内部类如何使用外部类的属性和方法，以及使用.this与.new。

当我们在创建一个内部类的时候，它无形中就与外围类有了一种联系，依赖于这种联系，它可以无限制地访问外围类的元素。

```
public class OuterClass {
    private String name ;
    private int age;

    /**省略getter和setter方法**/

    public class InnerClass{
        public InnerClass(){
            name = "chenssy";
            age = 23;
        }

        public void display(){
            System.out.println("name：" + getName() +"   ;age：" + getAge());
        }
    }

    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();
        innerClass.display();
    }
}
--------------
Output：
name：chenssy   ;age：23
```

在这个应用程序中，我们可以看到内部了 InnerClass 可以对外围类 OuterClass 的属性进行无缝的访问，尽管它是 private 修饰的。这是因为当我们在创建某个外围类的内部类对象时，此时内部类对象必定会捕获一个指向那个外围类对象的引用，只要我们在访问外围类的成员时，就会用这个引用来选择外围类的成员。

其实在这个应用程序中我们还看到了如何来引用内部类：引用内部类我们需要指明这个对象的类型：OuterClasName.InnerClassName。同时如果我们需要创建某个内部类对象，必须要利用外部类的对象通过.new 来创建内部类： OuterClass.InnerClass innerClass = outerClass.new InnerClass();。

同时如果我们需要生成对外部类对象的引用，可以使用 OuterClassName.this，这样就能够产生一个正确引用外部类的引用了。当然这点实在编译期就知晓了，没有任何运行时的成本。

```
public class OuterClass {
    public void display(){
        System.out.println("OuterClass...");
    }

    public class InnerClass{
        public OuterClass getOuterClass(){
            return OuterClass.this;
        }
    }

    public static void main(String[] args) {
        OuterClass outerClass = new OuterClass();
        OuterClass.InnerClass innerClass = outerClass.new InnerClass();
        innerClass.getOuterClass().display();
    }
}
-------------
Output:
OuterClass...
```

到这里了我们需要明确一点，内部类是个编译时的概念，一旦编译成功后，它就与外围类属于两个完全不同的类（当然他们之间还是有联系的）。对于一个名为 OuterClass 的外围类和一个名为 InnerClass 的内部类，在编译成功后，会出现这样两个 class 文件：OuterClass.class 和OuterClass$InnerClass.class。

在 Java 中内部类主要分为成员内部类、局部内部类、匿名内部类、静态内部类。

三、成员内部类

成员内部类也是最普通的内部类，它是外围类的一个成员，所以他是可以无限制的访问外围类的所有 成员属性和方法，尽管是 private 的，但是外围类要访问内部类的成员属性和方法则需要通过内部类实例来访问。

在成员内部类中要注意两点，**第一**：成员内部类中不能存在任何 static 的变量和方法；**第二**：成员内部类是依附于外围类的，所以只有先创建了外围类才能够创建内部类。

```
public class OuterClass {
    private String str;

    public void outerDisplay(){
        System.out.println("outerClass...");
    }

    public class InnerClass{
        public void innerDisplay(){
            //使用外围内的属性
            str = "chenssy...";
            System.out.println(str);
            //使用外围内的方法
            outerDisplay();
        }
    }

    /*推荐使用getxxx()来获取成员内部类，尤其是该内部类的构造函数无参数时 */
    public InnerClass getInnerClass(){
        return new InnerClass();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.getInnerClass();
        inner.innerDisplay();
    }
}
--------------------
chenssy...
outerClass...
```

推荐使用 getxxx() 来获取成员内部类，尤其是该内部类的构造函数无参数时 。

四、局部内部类

有这样一种内部类，它是嵌套在方法和作用于内的，对于这个类的使用主要是应用与解决比较复杂的问题，想创建一个类来辅助我们的解决方案，到那时又不希望这个类是公共可用的，所以就产生了局部内部类，局部内部类和成员内部类一样被编译，只是它的作用域发生了改变，它只能在该方法和属性中被使用，出了该方法和属性就会失效。

对于局部内部类实在是想不出什么好例子，所以就引用《 Think in Java 》中的经典例子了。

定义在方法里：

```
public class Parcel5 {
    public Destionation destionation(String str){
        class PDestionation implements Destionation{
            private String label;
            private PDestionation(String whereTo){
                label = whereTo;
            }
            public String readLabel(){
                return label;
            }
        }
        return new PDestionation(str);
    }

    public static void main(String[] args) {
        Parcel5 parcel5 = new Parcel5();
        Destionation d = parcel5.destionation("chenssy");
    }
}
```

定义在作用域内：

```
public class Parcel6 {
    private void internalTracking(boolean b){
        if(b){
            class TrackingSlip{
                private String id;
                TrackingSlip(String s) {
                    id = s;
```

```
            }
            String getSlip(){
                return id;
            }
        }
        TrackingSlip ts = new TrackingSlip("chenssy");
        String string = ts.getSlip();
    }
}

public void track(){
    internalTracking(true);
}

public static void main(String[] args) {
    Parcel6 parcel6 = new Parcel6();
    parcel6.track();
}
}
```

第 8 章 五、匿名内部类

在做 Swing 编程中，我们经常使用这种方式来绑定事件

```
button2.addActionListener(
            new ActionListener(){
               public void actionPerformed(ActionEvent e) {
                  System.out.println("你按了按钮二");
               }
            });
```

我们咋一看可能觉得非常奇怪，因为这个内部类是没有名字的，在看如下这个例子：

```
public class OuterClass {
   public InnerClass getInnerClass(final int   num,String str2){
      return new InnerClass(){
         int number = num + 3;
         public int getNumber(){
            return number;
         }
      };       /* 注意：分号不能省 */
   }

   public static void main(String[] args) {
      OuterClass out = new OuterClass();
      InnerClass inner = out.getInnerClass(2, "chenssy");
      System.out.println(inner.getNumber());
   }
}

interface InnerClass {
   int getNumber();
}

----------------
Output:
5
```

这里我们就需要看清几个地方

1、匿名内部类是没有访问修饰符的。

2、new 匿名内部类，这个类首先是要存在的。如果我们将那个 InnerClass 接口注释掉，就会出现编译出错。

3、注意 getInnerClass() 方法的形参，第一个形参是用 final 修饰的，而第二个却没有。同时我们也发现第二个形参在匿名内部类中没有使用过，所以当所在方法的形参需要被匿名内部类使用，那么这个形参就必须为 final。

4、匿名内部类是没有构造方法的。因为它连名字都没有何来构造方法。

PS:由于篇幅有限，对匿名内部类就介绍到这里，有关更多关于匿名内部类的知识，我就会在下篇博客（Java 提高篇一－详解匿名内部类）做详细的介绍，包括为何形参要定义成 final，怎么对匿名内部类进行初始化等等，敬请期待……

六、静态内部类

在 Java 提高篇一－关键字 static 中提到 Static 可以修饰成员变量、方法、代码块，其他它还可以修饰内部类，使用 static 修饰的内部类我们称之为静态内部类，不过我们更喜欢称之为嵌套内部类。静态内部类与非静态内部类之间存在一个最大的区别，我们知道非静态内部类在编译完成之后会隐含地保存着一个引用，该引用是指向创建它的外围内，但是静态内部类却没有。没有这个引用就意味着：

1、它的创建是不需要依赖于外围类的。

2、它不能使用任何外围类的非static成员变量和方法。

```
public class OuterClass {
    private String sex;
    public static String name = "chenssy";

    /**
     *静态内部类
     */
    static class InnerClass1{
        /* 在静态内部类中可以存在静态成员 */
        public static String _name1 = "chenssy_static";

        public void display(){
            /*
             * 静态内部类只能访问外围类的静态成员变量和方法
             * 不能访问外围类的非静态成员变量和方法
             */
            System.out.println("OutClass name :" + name);
        }
    }

    /**
     * 非静态内部类
     */
    class InnerClass2{
        /* 非静态内部类中不能存在静态成员 */
        public String _name2 = "chenssy_inner";
        /* 非静态内部类中可以调用外围类的任何成员,不管是静态的还是非静态的 */
        public void display(){
            System.out.println("OuterClass name：" + name);
        }
```

```
    }

    /**
     * @desc 外围类方法
     * @author chenssy
     * @data 2013-10-25
     * @return void
     */
    public void display(){
        /* 外围类访问静态内部类：内部类. */
        System.out.println(InnerClass1._name1);
        /* 静态内部类 可以直接创建实例不需要依赖于外围类 */
        new InnerClass1().display();

        /* 非静态内部的创建需要依赖于外围类 */
        OuterClass.InnerClass2 inner2 = new OuterClass().new InnerClass2();
        /* 方位非静态内部类的成员需要使用非静态内部类的实例  */
        System.out.println(inner2._name2);
        inner2.display();
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        outer.display();
    }
}
----------------
Output:
chenssy_static
OutClass name :chenssy
chenssy_inner
OuterClass name：chenssy
```

上面这个例子充分展现了静态内部类和非静态内部类的区别。

到这里内部类的介绍就基本结束了！对于内部类其实本人认识也只是皮毛，逼近菜鸟一枚，认知有限！我会利用这几天时间好好研究内部类!

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

9

Java 提高篇(九)—— 实现多重继承

多重继承指的是一个类可以同时从多于一个的父类那里继承行为和特征，然而我们知道 Java 为了保证数据安全，它只允许单继承。有些时候我们会认为如果系统中需要使用多重继承往往都是糟糕的设计,这个时候我们往往需要思考的不是怎么使用多重继承,而是您的设计是否存在问题.但有时候我们确实是需要实现多重继承，而且现实生活中也真正地存在这样的情况，比如遗传：我们即继承了父亲的行为和特征也继承了母亲的行为和特征。可幸的是 Java 是非常和善和理解我们的,它提供了两种方式让我们曲折来实现多重继承：接口和内部类。

一、接口

在介绍接口和抽象类的时候了解到子类只能继承一个父类，也就是说只能存在单一继承，但是却可以实现多个接口，这就为我们实现多重继承做了铺垫。

对于接口而已，有时候它所表现的不仅仅只是一个更纯粹的抽象类，接口是没有任何具体实现的，也就是说，没有任何与接口相关的存储，因此也就无法阻止多个接口的组合了。

```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}


interface CanFly {
    void fly();
}

public class ActionCharacter {
    public void fight(){

    }
}

public class Hero extends ActionCharacter implements CanFight,CanFly,CanSwim{

    public void fly() {
    }

    public void swim() {
    }

    /**
     * 对于fight()方法，继承父类的，所以不需要显示声明
     */
}
```

二、内部类

上面使用接口实现多重继承是一种比较可行和普遍的方式，在介绍内部类的时候谈到内部类使的多继承的实现变得更加完美了，同时也明确了如果父类为抽象类或者具体类，那么我就仅能通过内部类来实现多重继承了。如何利用内部类实现多重继承，请看下面实例：儿子是如何利用多重继承来继承父亲和母亲的优良基因。

首先是父亲 Father 和母亲 Mother：

```
public class Father {
    public int strong(){
        return 9;
    }
}

public class Mother {
    public int kind(){
        return 8;
    }
}
```

重头戏在这里，儿子类 Son：

```
public class Son {

    /**
     * 内部类继承Father类
     */
    class Father_1 extends Father{
        public int strong(){
            return super.strong() + 1;
        }
    }

    class Mother_1 extends  Mother{
        public int kind(){
            return super.kind() - 2;
        }
    }

    public int getStrong(){
        return new Father_1().strong();
```

```
    }

    public int getKind(){
        return new Mother_1().kind();
    }
}
```

测试程序：

```
public class Test1 {

    public static void main(String[] args) {
        Son son = new Son();
        System.out.println("Son 的Strong：" + son.getStrong());
        System.out.println("Son 的kind：" + son.getKind());
    }

}
----------------------------------------
Output:
Son 的Strong：10
Son 的kind：6
```

儿子继承了父亲，变得比父亲更加强壮，同时也继承了母亲，只不过温柔指数下降了。这里定义了两个内部类，他们分别继承父亲 Father 类、母亲类 Mother 类，且都可以非常自然地获取各自父类的行为，这是内部类一个重要的特性：内部类可以继承一个与外部类无关的类，保证了内部类的独立性，正是基于这一点，多重继承才会成为可能。

有关于更多接口和内部类的详情，请参考这里:

内部类：Java提高篇——详解内部类 **接口：**Java提高篇一 - 抽象类与接口

10

Java 提高篇(十)—— 详解匿名内部类

在 Java 提高篇一 - 详解内部类中对匿名内部类做了一个简单的介绍，但是内部类还存在很多其他细节问题，所以就衍生出这篇博客。在这篇博客中你可以了解到匿名内部类的使用、匿名内部类要注意的事项、如何初始化匿名内部类、匿名内部类使用的形参为何要为 final。

一、使用匿名内部类内部类

匿名内部类由于没有名字，所以它的创建方式有点儿奇怪。创建格式如下：

```
new 父类构造器（参数列表）|实现接口（）
    {
     //匿名内部类的类体部分
    }
```

在这里我们看到使用匿名内部类我们必须要继承一个父类或者实现一个接口，当然也仅能只继承一个父类或者实现一个接口。同时它也是没有 class 关键字，这是因为匿名内部类是直接使用 new 来生成一个对象的引用。当然这个引用是隐式的。

```
public abstract class Bird {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public abstract int fly();
}

public class Test {

    public void test(Bird bird){
        System.out.println(bird.getName() + "能够飞 " + bird.fly() + "米");
    }

    public static void main(String[] args) {
        Test test = new Test();
        test.test(new Bird() {

            public int fly() {
                return 10000;
            }
```

```
            public String getName() {
                return "大雁";
            }
        });
    }
}
------------------
Output:
大雁能够飞 10000米
```

在 Test 类中，test() 方法接受一个 Bird 类型的参数，同时我们知道一个抽象类是没有办法直接 new 的，我们必须要先有实现类才能 new 出来它的实现类实例。所以在 main 方法中直接使用匿名内部类来创建一个 Bird 实例。

由于匿名内部类不能是抽象类，所以它必须要实现它的抽象父类或者接口里面所有的抽象方法。

对于这段匿名内部类代码其实是可以拆分为如下形式：

```
public class WildGoose extends Bird{
    public int fly() {
        return 10000;
    }

    public String getName() {
        return "大雁";
    }
}

WildGoose wildGoose = new WildGoose();
test.test(wildGoose);
```

在这里系统会创建一个继承自 Bird 类的匿名类的对象，该对象转型为对 Bird 类型的引用。

对于匿名内部类的使用它是存在一个缺陷的，就是它仅能被使用一次，创建匿名内部类时它会立即创建一个该类的实例，该类的定义会立即消失，所以匿名内部类是不能够被重复使用。对于上面的实例，如果我们需要对 test() 方法里面内部类进行多次使用，建议重新定义类，而不是使用匿名内部类。

二、注意事项

在使用匿名内部类的过程中，我们需要注意如下几点：

1、使用匿名内部类时，我们必须是继承一个类或者实现一个接口，但是两者不可兼得，同时也只能继承一个类或者实现一个接口。

2、匿名内部类中是不能定义构造函数的。

3、匿名内部类中不能存在任何的静态成员变量和静态方法。

4、匿名内部类为局部内部类，所以局部内部类的所有限制同样对匿名内部类生效。

5、匿名内部类不能是抽象的，它必须要实现继承的类或者实现的接口的所有抽象方法。

三、使用的形参为何要为 final

参考文件：http://android.blog.51cto.com/268543/384844

我们给匿名内部类传递参数的时候，若该形参在内部类中需要被使用，那么该形参必须要为 final。也就是说：**当所在的方法的形参需要被内部类里面使用时，该形参必须为 final。**

为什么必须要为 final 呢?

首先我们知道在内部类编译成功后，它会产生一个 class 文件，该 class 文件与外部类并不是同一 class 文件，仅仅只保留对外部类的引用。当外部类传入的参数需要被内部类调用时，从 java 程序的角度来看是直接被调用：

```
public class OuterClass {
    public void display(final String name,String age){
        class InnerClass{
            void display(){
                System.out.println(name);
            }
        }
    }
}
```

从上面代码中看好像 name 参数应该是被内部类直接调用？其实不然，在 java 编译之后实际的操作如下：

```
public class OuterClass$InnerClass {
    public InnerClass(String name,String age){
        this.InnerClass$name = name;
        this.InnerClass$age = age;
    }


    public void display(){
        System.out.println(this.InnerClass$name + "----" + this.InnerClass$age );
    }
}
```

所以从上面代码来看，内部类并不是直接调用方法传递的参数，而是利用自身的构造器对传入的参数进行备份，自己内部方法调用的实际上时自己的属性而不是外部方法传递进来的参数。

直到这里还没有解释为什么是 final？在内部类中的属性和外部方法的参数两者从外表上看是同一个东西，但实际上却不是，所以他们两者是可以任意变化的，也就是说在内部类中我对属性的改变并不会影响到外部的形参，而然这从程序员的角度来看这是不可行的，毕竟站在程序的角度来看这两个根本就是同一个，如果内部类该变了，而外部方法的形参却没有改变这是难以理解和不可接受的，所以为了保持参数的一致性，就规定使用 final 来避免形参的不改变。

简单理解就是，拷贝引用，为了避免引用值发生改变，例如被外部类的方法修改等，而导致内部类得到的值不一致，于是用 final 来让该引用不可改变。

故如果定义了一个匿名内部类，并且希望它使用一个其外部定义的参数，那么编译器会要求该参数引用是 final 的。

四、匿名内部类初始化

我们一般都是利用构造器来完成某个实例的初始化工作的，但是匿名内部类是没有构造器的！那怎么来初始化匿名内部类呢？使用构造代码块！利用构造代码块能够达到为匿名内部类创建一个构造器的效果。

```
public class OutClass {
    public InnerClass getInnerClass(final int     age,final String name){
        return new InnerClass() {
            int age_ ;
            String name_;
            //构造代码块完成初始化工作
            {
                if(0 < age && age < 200){
                    age_ = age;
                    name_ = name;
                }
            }
            public String getName() {
                return name_;
            }

            public int getAge() {
                return age_;
            }
        };
    }

    public static void main(String[] args) {
        OutClass out = new OutClass();

        InnerClass inner_1 = out.getInnerClass(201, "chenssy");
        System.out.println(inner_1.getName());

        InnerClass inner_2 = out.getInnerClass(23, "chenssy");
        System.out.println(inner_2.getName());
    }
}
```

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

11

强制类型转换

在 Java 中强制类型转换分为基本数据类型和引用数据类型两种，这里我们讨论的后者，也就是引用数据类型的强制类型转换。

在 Java 中由于继承和向上转型，子类可以非常自然地转换成父类，但是父类转换成子类则需要强制转换。因为子类拥有比父类更多的属性、更强的功能，所以父类转换为子类需要强制。那么，是不是只要是父类转换为子类就会成功呢？其实不然，他们之间的强制类型转换是有条件的。

当我们用一个类型的构造器构造出一个对象时，这个对象的类型就已经确定的，也就说它的本质是不会再发生变化了。在 Java 中我们可以通过继承、向上转型的关系使用父类类型来引用它，这个时候我们是使用功能较弱的类型引用功能较强的对象，这是可行的。但是将功能较弱的类型强制转功能较强的对象时，就不一定可以行了。

举个例子来说明。比如系统中存在 Father、Son 两个对象。首先我们先构造一个 Son 对象，然后用一个Father 类型变量引用它：

```
Father father = new Son();
```

在这里 Son 对象实例被向上转型为 father 了，但是请注意这个 Son 对象实例在内存中的本质还是 Son 类型的，只不过它的能力临时被消弱了而已，如果我们想变强怎么办？将其对象类型还原！

```
Son son = (Son)father;
```

这条语句是可行的，其实 father 引用仍然是 Father 类型的，只不过是将它的能力加强了，将其加强后转交给 son 引用了，Son 对象实例在 son 的变量的引用下，恢复真身，可以使用全部功能了。

前面提到父类强制转换成子类并不是总是成功，那么在什么情况下它会失效呢？

当引用类型的真实身份是父类本身的类型时，强制类型转换就会产生错误。例如：

```
Father father = new Father();
```

```
Son son = (Son) father;
```

这个系统会抛出 ClassCastException 异常信息。

所以编译器在编译时只会检查类型之间是否存在继承关系，有则通过；而在运行时就会检查它的真实类型，是则通过，否则抛出 ClassCastException异常。

所以在继承中，子类可以自动转型为父类，但是父类强制转换为子类时只有当引用类型真正的身份为子类时才会强制转换成功，否则失败。

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

12 代码块

在编程过程中我们可能会遇到如下这种形式的程序：

```
public class Test {
    {
        ////
    }
}
```

这种形式的程序段我们将其称之为代码块，所谓代码块就是用大括号({})将多行代码封装在一起，形成一个独立的数据体，用于实现特定的算法。一般来说代码块是不能单独运行的，它必须要有运行主体。在 Java 中代码块主要分为四种：

一、普通代码块

普通代码块是我们用得最多的也是最普遍的，它就是在方法名后面用{}括起来的代码段。普通代码块是不能够单独存在的，它必须要紧跟在方法名后面。同时也必须要使用方法名调用它。

```
public class Test {
    public void test(){
        System.out.println("普通代码块");
    }
}
```

二、静态代码块

想到静态我们就会想到 static，静态代码块就是用 static 修饰的用{}括起来的代码段，它的主要目的就是对静态属性进行初始化。

```
public class Test {
    static{
        System.out.println("静态代码块");
    }
}
```

三、同步代码块

使用 synchronized 关键字修饰，并使用“{}”括起来的代码片段，它表示同一时间只能有一个线程进入到该方法块中，是一种多线程保护机制。

四、 构造代码块

在类中直接定义没有任何修饰符、前缀、后缀的代码块即为构造代码块。我们明白一个类必须至少有一个构造函数，构造函数在生成对象时被调用。构造代码块和构造函数一样同样是在生成一个对象时被调用，那么构造代码在什么时候被调用？如何调用的呢？看如下代码：

```
public class Test {
    /**
     * 构造代码
     */
    {
        System.out.println("执行构造代码块...");
    }

    /**
     * 无参构造函数
     */
    public Test(){
        System.out.println("执行无参构造函数...");
    }

    /**
     * 有参构造函数
     * @param id  id
     */
    public Test(String id){
        System.out.println("执行有参构造函数...");
    }
}
```

上面定义了一个非常简单的类，该类包含无参构造函数、有参构造函数以及构造代码块，同时在上面也提过代码块是没有独立运行的能力，他必须要有一个可以承载的载体，那么编译器会如何来处理构造代码块呢？编译器会将代码块按照他们的顺序(假如有多个代码块)插入到所有的构造函数的最前端，这样就能保证不管调用哪个构造函数都会执行所有的构造代码块。上面代码等同于如下形式：

```
public class Test {
    /**
     * 无参构造函数
     */
    public Test(){
```

```
        System.out.println("执行构造代码块...");
        System.out.println("执行无参构造函数...");
    }

    /**
     * 有参构造函数
     * @param id  id
     */
    public Test(String id){
        System.out.println("执行构造代码块...");
        System.out.println("执行有参构造函数...");
    }

}
```

运行结果

```
public static void main(String[] args) {
        new Test();
        System.out.println("----------------");
        new Test("1");
    }
------------
Output:
执行构造代码块...
执行无参构造函数...
----------------
执行构造代码块...
执行有参构造函数...
```

从上面的运行结果可以看出在 new 一个对象的时候总是先执行构造代码，再执行构造函数，但是有一点需要注意构造代码不是在构造函数之前运行的，它是依托构造函数执行的。正是由于构造代码块有这几个特性，所以它常用于如下场景：

1、初始化实例变量

如果一个类中存在若干个构造函数，这些构造函数都需要对实例变量进行初始化，如果我们直接在构造函数中实例化，必定会产生很多重复代码，繁琐和可读性差。这里我们可以充分利用构造代码块来实现。这是利用编译器会将构造代码块添加到每个构造函数中的特性。

2、初始化实例环境

一个对象必须在适当的场景下才能存在，如果没有适当的场景，则就需要在创建对象时创建此场景。我们可以利用构造代码块来创建此场景，尤其是该场景的创建过程较为复杂。构造代码会在构造函数之前执行。

上面两个常用场景都充分利用构造代码块的特性，能够很好的解决在实例化对象时构造函数比较难解决的问题，利用构造代码不仅可以减少代码量，同时也是程序的可读性增强了。特别是当一个对象的创建过程比较复杂，需要实现一些复杂逻辑，这个时候如果在构造函数中实现逻辑，这是不推荐的，因为我们提倡构造函数要尽可能的简单易懂，所以我们可以使用构造代码封装这些逻辑实现部分。

五、 静态代码块、构造代码块、构造函数执行顺序

从词面上我们就可以看出他们的区别。静态代码块，静态，其作用级别为类，构造代码块、构造函数，构造，其作用级别为对象。

1、静态代码块，它是随着类的加载而被执行，只要类被加载了就会执行，而且只会加载一次，主要用于给类进行初始化。

2、构造代码块，每创建一个对象时就会执行一次，且优先于构造函数，主要用于初始化不同对象共性的初始化内容和初始化实例环境。

3、构造函数，每创建一个对象时就会执行一次。同时构造函数是给特定对象进行初始化，而构造代码是给所有对象进行初始化，作用区域不同。

通过上面的分析，他们三者的执行顺序应该为：静态代码块 > 构造代码块 > 构造函数。

```
public class Test {
    /**
     * 静态代码块
     */
    static{
        System.out.println("执行静态代码块...");
    }

    /**
     * 构造代码块
     */
    {
        System.out.println("执行构造代码块...");
    }

    /**
     * 无参构造函数
     */
    public Test(){
        System.out.println("执行无参构造函数...");
    }

    /**
     * 有参构造函数
     * @param id
```

```
     */
    public Test(String id){
        System.out.println("执行有参构造函数...");
    }

    public static void main(String[] args) {
        System.out.println("----------------------");
        new Test();
        System.out.println("----------------------");
        new Test("1");
    }
}
-----------
Output:
执行静态代码块...
----------------------
执行构造代码块...
执行无参构造函数...
----------------------
执行构造代码块...
执行有参构造函数...
```

13

equals() 方法总结

equals()

超类 Object 中有这个 equals() 方法，该方法主要用于比较两个对象是否相等。该方法的源码如下：

```
public boolean equals(Object obj) {
    return (this == obj);
    }
```

我们知道所有的对象都拥有标识(内存地址)和状态(数据)，同时“==”比较两个对象的的内存地址，所以说使用 Object 的 equals() 方法是比较两个对象的内存地址是否相等，即若 object1.equals(object2) 为 true，则表示 equals1 和 equals2 实际上是引用同一个对象。虽然有时候 Object 的 equals() 方法可以满足我们一些基本的要求，但是我们必须要清楚我们很大部分时间都是进行两个对象的比较，这个时候 Object 的 equals() 方法就不可以了，实际上 JDK 中，String、Math 等封装类都对 equals() 方法进行了重写。下面是 String 的 equals() 方法：

```
public boolean equals(Object anObject) {
    if (this == anObject) {
        return true;
    }
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = count;
        if (n == anotherString.count) {
        char v1[] = value;
        char v2[] = anotherString.value;
        int i = offset;
        int j = anotherString.offset;
        while (n-- != 0) {
            if (v1[i++] != v2[j++])
            return false;
        }
        return true;
        }
    }
    return false;
    }
```

对于这个代码段:if (v1[i++] != v2[j++])return false;我们可以非常清晰的看到 String 的 equals() 方法是进行内容比较，而不是引用比较。至于其他的封装类都差不多。

在 Java 规范中，它对 equals() 方法的使用必须要遵循如下几个规则：

equals 方法在非空对象引用上实现相等关系：

1、**自反性**：对于任何非空引用值 x，x.equals(x) 都应返回 true。

2、**对称性**：对于任何非空引用值 x 和 y，当且仅当 y.equals(x) 返回 true 时，x.equals(y) 才应返回 true。

3、**传递性**：对于任何非空引用值 x、y 和 z，如果 x.equals(y) 返回 true，并且 y.equals(z) 返回 true，那么 x.equals(z) 应返回 true。

4、**一致性**：对于任何非空引用值 x 和 y，多次调用 x.equals(y) 始终返回 true 或始终返回 false，前提是对象上 equals 比较中所用的信息没有被修改。

5、对于任何非空引用值 x，x.equals(null) 都应返回 false。

对于上面几个规则，我们在使用的过程中最好遵守，否则会出现意想不到的错误。

在 java 中进行比较，我们需要根据比较的类型来选择合适的比较方式：

1) 对象域，使用 equals 方法 。

2) 类型安全的枚举，使用 equals 或== 。

3) 可能为 null 的对象域 : 使用 == 和 equals 。

4) 数组域 : 使用 Arrays.equals 。

5)除 float 和 double 外的原始数据类型 : 使用 == 。

6) float 类型: 使用 Float.foatToIntBits 转换成 int 类型，然后使用==。

7) double 类型: 使用 Double.doubleToLongBit 转换成 long 类型，然后使用==。

至于6）、7）为什么需要进行转换，我们可以参考他们相应封装类的 equals() 方法，下面的是 Float 类的：

```
public boolean equals(Object obj) {
    return (obj instanceof Float)
          && (floatToIntBits(((Float)obj).value) ==   floatToIntBits(value));
    }
```

原因嘛，里面提到了两点：

```
However, there are two exceptions:
```

If f1 and f2 both represent
Float.NaN, then the equals method returns
true, even though Float.NaN==Float.NaN
has the value false.
If <code>f1 represents +0.0f while
f2 represents −0.0f, or vice
versa, the equal test has the value
false, even though 0.0f==−0.0f
has the value true.

在 equals() 中使用 getClass 进行类型判断

我们在覆写 equals() 方法时，一般都是推荐使用 getClass 来进行类型判断，不是使用 instanceof。我们都清楚 instanceof 的作用是判断其左边对象是否为其右边类的实例，返回 boolean 类型的数据。可以用来判断继承中的子类的实例是否为父类的实现。注意后面这句话：可以用来判断继承中的子类的实例是否为父类的实现，正是这句话在作怪。我们先看如下实例(摘自《高质量代码 改善 Java 程序的 151 个建议》)。

父类：Person

```
public class Person {
    protected String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Person(String name){
        this.name = name;
    }

    public boolean equals(Object object){
        if(object instanceof Person){
            Person p = (Person) object;
            if(p.getName() == null || name == null){
                return false;
            }
            else{
                return name.equalsIgnoreCase(p.getName ());
            }
        }
        return false;
    }
}
```

子类：Employee

```
public class Employee extends Person{
    private int id;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public Employee(String name,int id){
        super(name);
        this.id = id;
    }

    /**
     * 重写equals()方法
     */
    public boolean equals(Object object){
        if(object instanceof Employee){
            Employee e = (Employee) object;
            return super.equals(object) && e.getId() == id;
        }
        return false;
    }
}
```

上面父类 Person 和子类 Employee 都重写了 equals(),不过 Employee 比父类多了一个id属性。测试程序如下:

```
public class Test {
    public static void main(String[] args) {
        Employee e1 = new Employee("chenssy", 23);
        Employee e2 = new Employee("chenssy", 24);
        Person p1 = new Person("chenssy");

        System.out.println(p1.equals(e1));
        System.out.println(p1.equals(e2));
        System.out.println(e1.equals(e2));
    }
}
```

上面定义了两个员工和一个普通人，虽然他们同名，但是他们肯定不是同一人，所以按理来说输出结果应该全部都是 false，但是事与愿违，结果是：true、true、false。

对于那 e1!=e2 我们非常容易理解，因为他们不仅需要比较 name,还需要比较 ID。但是 p1 即等于 e1 也等于 e2，这是非常奇怪的，因为 e1、e2 明明是两个不同的类，但为什么会出现这个情况？首先 p1.equals(e1)，是调用 p1 的 equals 方法，该方法使用 instanceof 关键字来检查 e1 是否为 Person 类，这里我们再看看 instanceof：判断其左边对象是否为其右边类的实例，也可以用来判断继承中的子类的实例是否为父类的实现。他们两者存在继承关系，肯定会返回 true 了，而两者 name 又相同，所以结果肯定是 true。

所以出现上面的情况就是使用了关键字 instanceof，这是非常容易“专空子”的。故在覆写 equals 时推荐使用 getClass 进行类型判断。而不是使用 instanceof。

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

14

字符串

可以证明，字符串操作是计算机程序设计中最常见的行为。

一、String

首先我们要明确，String 并不是基本数据类型，而是一个对象，并且是不可变的对象。查看源码就会发现 String 类为 final 型的（当然也不可被继承），而且通过查看 JDK 文档会发现几乎每一个修改 String 对象的操作，实际上都是创建了一个全新的 String 对象。

字符串为对象，那么在初始化之前，它的值为 null，到这里就有必要提下 ”” 、null、new String() 三者的区别。null 表示 string 还没有 new ，也就是说对象的引用还没有创建，也没有分配内存空间给他，而 ”” 、new String() 则说明了已经 new 了，只不过内部为空，但是它创建了对象的引用，是需要分配内存空间的。打个比方：一个空玻璃杯，你不能说它里面什么都没有，因为里面有空气，当然也可以把它弄成真空，null与 “ 、new String() 的区别就象真空与空气一样。

在字符串中存在一个非常特殊的地方，那就是字符串池。每当我们创建一个字符串对象时，首先就会检查字符串池中是否存在面值相等的字符串，如果有，则不再创建，直接放回字符串池中对该对象的引用，若没有则创建然后放入到字符串池中并且返回新建对象的引用。这个机制是非常有用的，因为可以提高效率，减少了内存空间的占用。所以在使用字符串的过程中，推荐使用直接赋值（即String s=”aa”），除非有必要才会新建一个 String 对象（即String s = new String(”aa”)）。

对于字符串的使用无非就是这几个方面：

1、字符串比较

equals() ——判断内容是否相同。

compareTo() ——判断字符串的大小关系。

compareToIgnoreCase(String int) ——在比较时忽略字母大小写。

== ——判断内容与地址是否相同。

equalsIgnoreCase() ——忽略大小写的情况下判断内容是否相同。

reagionMatches() ——对字符串中的部分内容是否相同进行比较（详情请参考API）。

2、字符串查找

charAt(int index) ——返回指定索引 index 位置上的字符，索引范围从 0 开始。

indexOf(String str)——从字符串开始检索str，并返回第一次出现的位置，未出现返回 -1。

indexOf(String str，int fromIndex);——从字符串的第 fromIndex 个字符开始检索 str。

lastIndexOf(String str)——查找最后一次出现的位置。

lastIndexOf(String str，int fromIndex)——从字符串的第fromIndex个字符查找最后一次出现的位置。

starWith(String prefix，int toffset)——测试此字符串从指定索引开始的子字符串是否以指定前缀开始。

starWith(String prefix)——测试此字符串是否以指定的前缀开始。

endsWith(String suffix)——测试此字符串是否以指定的后缀结束。

3、字符串截取

public String subString(int beginIndex)——返回一个新的字符串，它是此字符串的一个子字符串。

public String subString(int beginIndex，int endIndex)——返回的字符串是从 beginIndex 开始到 endIndex-1 的串。

4、字符串替换

public String replace(char oldChar，char newChar)。

public String replace(CharSequence target，CharSequence replacement)——把原来的 etarget 子序列替换为 replacement 序列，返回新串。

public String replaceAll(String regex，String replacement)——用正则表达式实现对字符串的匹配。注意 replaceAll 第一个参数为正则表达式，鄙人曾经深受其害。

5、更多方法请参考 API

二、StringBuffer

StringBuffer 和 String 一样都是用来存储字符串的，只不过由于他们内部的实现方式不同，导致他们所使用的范围不同，对于 StringBuffer 而言，他在处理字符串时，若是对其进行修改操作，它并不会产生一个新的字符串对象，所以说在内存使用方面它是优于 String 的。

其实在使用方法，StringBuffer 的许多方法和 String 类都差不多，所表示的功能几乎一模一样，只不过在修改时 StringBuffer 都是修改自身，而 String 类则是产生一个新的对象，这是他们之间最大的区别。

同时 StringBuffer 是不能使用=进行初始化的，它必须要产生 StringBuffer 实例，也就是说你必须通过它的构造方法进行初始化。

在 StringBuffer 的使用方面，它更加侧重于对字符串的变化，例如追加、修改、删除，相对应的方法：

1、append()：追加指定内容到当前 StringBuffer 对象的末尾，类似于字符串的连接，这里 StringBuffer 对象的内容会发生改变。

2、insert：该类方法主要是在 StringBuffer 对象中插入内容。

3、delete：该类方法主要用于移除 StringBuffer 对象中的内容。

三、StringBuilder

StringBuilder 也是一个可变的字符串对象，他与 StringBuffer 不同之处就在于它是线程不安全的，基于这点，它的速度一般都比 StringBuffer 快。与 StringBuffer 一样，StringBuider 的主要操作也是 append 与 insert 方法。这两个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符添加或插入到字符串生成器中。

上面只是简单的介绍了 String、StringBuffer、StringBuilder，其实对于这三者我们应该更加侧重于他们只见到的区别，只有理清楚他们之间的区别才能够更好的使用他们。

四、正确使用 String、StringBuffer、StringBuilder

我们先看如下表格：

这里对于 String 是否为线程安全，鄙人也不是很清楚，原因：String 不可变，所有的操作都是不可能改变其值的，是否存在线程安全一说还真不好说？但是如果硬要说线程是否安全的话，因为内容不可变，永远都是安全的。

在使用方面由于 String 每次修改都需要产生一个新的对象，所以对于经常需要改变内容的字符串最好选择 StringBuffer 或者 StringBuilder.而对于 StringBuffer，每次操作都是对 StringBuffer 对象本身，它不会生成新的对象，所以 StringBuffer 特别适用于字符串内容经常改变的情况下。

但是并不是所有的 String 字符串操作都会比 StringBuffer 慢，在某些特殊的情况下，String 字符串的拼接会被 JVM 解析成 StringBuilder 对象拼接，在这种情况下 String 的速度比 StringBuffer 的速度快。如：

String name = ”I ” + ” am ” + ” chenssy ” ;

StringBuffer name = new StringBuffer(” I ”).append(” am ”).append(” chenssy ”);

对于这两种方式，你会发现第一种比第二种快太多了，在这里 StringBuffer 的优势荡然无存。其真实的原因就在于JVM做了一下优化处理，其实 String name = ”I ” + ” am ” + ” chenssy ” ;在 JVM 眼中就是 String name = ”I am chenssy ” ;这样的方式对于 JVM 而言，真的是不要什么时间。但是如果我们在这个其中增加一个 String 对象，那么 JVM 就会按照原来那种规范来构建 String 对象了。

对于这三者使用的场景做如下概括（参考：《编写搞质量代码：改善 Java 程序的 151 个建议》）：

1、String：在字符串不经常变化的场景中可以使用 String 类，如：常量的声明、少量的变量运算等。

2、StringBuffer：在频繁进行字符串的运算（拼接、替换、删除等），并且运行在多线程的环境中，则可以考虑使用 StringBuffer，例如 XML 解析、 HTTP 参数解析和封装等。

3、StringBuilder：在频繁进行字符串的运算（拼接、替换、删除等），并且运行在多线程的环境中，则可以考虑使用 StringBuffer，如 SQL 语句的拼装、JSON 封装等（貌似这两个我也是使用|StringBuffer）。

更多有关于他们之间区别，请参考：http://www.cnblogs.com/zuoxiaolong/p/lang1.html 。鄙人就不画蛇添足了。

五、字符串拼接方式

对于字符串而言我们经常是要对其进行拼装处理的，在 Java 中提高了三种拼装的方法：+、concat() 以及 append() 方法。这三者之间存在什么区别呢？先看如下示例：

```
public class StringTest {

    /**
     * @desc 使用+、concat()、append()方法循环10W次
     * @author chenssy
     * @data 2013-11-16
     * @param args
     * @return void
     */
    public static void main(String[] args) {
        //+
        long start_01 = System.currentTimeMillis();
        String a = "a";
        for(int i = 0 ; i < 100000 ; i++){
            a += "b";
        }
        long end_01 = System.currentTimeMillis();
        System.out.println("  +   所消耗的时间：" + (end_01 - start_01) + "毫米");

        //concat()
        long start_02 = System.currentTimeMillis();
        String c = "c";
        for(int i = 0 ; i < 100000 ; i++){
            c = c.concat("d");
        }
        long end_02 = System.currentTimeMillis();
        System.out.println("concat所消耗的时间：" + (end_02 - start_02) + "毫米");

        //append
        long start_03 = System.currentTimeMillis();
        StringBuffer e = new StringBuffer("e");
        for(int i = 0 ; i < 100000 ; i++){
            e.append("d");
        }
        long end_03 = System.currentTimeMillis();
        System.out.println("append所消耗的时间：" + (end_03 - start_03) + "毫米");
    }
```

```
}

------------
Output:
 +  所消耗的时间：19080毫米
concat所消耗的时间：9089毫米
append所消耗的时间：10毫米
```

从上面的运行结果可以看出，append()速度最快，concat()次之，+最慢。原因请看下面分解:

(一)+方式拼接字符串

在前面我们知道编译器对+进行了优化，它是使用 StringBuilder 的 append() 方法来进行处理的，我们知道 StringBuilder 的速度比 StringBuffer 的速度更加快，但是为何运行速度还是那样呢？主要是因为编译器使用 append() 方法追加后要同 toString() 转换成 String 字符串，也就说 str +=”b”等同于

str = new StringBuilder(str).append(“b”).toString();

它变慢的关键原因就在于 new StringBuilder() 和 toString()，这里可是创建了 10 W 个 StringBuilder 对象，而且每次还需要将其转换成 String，速度能不慢么?

（二）concat() 方法拼接字符串

```
public String concat(String str) {
    int otherLen = str.length();
    if (otherLen == 0) {
        return this;
    }
    char buf[] = new char[count + otherLen];
    getChars(0, count, buf, 0);
    str.getChars(0, otherLen, buf, count);
    return new String(0, count + otherLen, buf);
    }
```

这是 concat() 的源码，它看上去就是一个数字拷贝形式，我们知道数组的处理速度是非常快的，但是由于该方法最后是这样的：return new String(0, count + otherLen, buf);这同样也创建了 10 W 个字符串对象，这是它变慢的根本原因。

（三）append() 方法拼接字符串

```
public synchronized StringBuffer append(String str) {
    super.append(str);
```

```
        return this;
    }
```

StringBuffer 的 append() 方法是直接使用父类 AbstractStringBuilder 的 append() 方法，该方法的源码如下：

```
public AbstractStringBuilder append(String str) {
    if (str == null) str = "null";
        int len = str.length();
    if (len == 0) return this;
    int newCount = count + len;
    if (newCount > value.length)
        expandCapacity(newCount);
    str.getChars(0, len, value, count);
    count = newCount;
    return this;
   }
```

与 concat() 方法相似，它也是进行字符数组处理的，加长，然后拷贝，但是请注意它最后是返回并没有返回一个新串，而是返回本身，也就说这这个 10 W 次的循环过程中，它并没有产生新的字符串对象。

通过上面的分析，我们需要在合适的场所选择合适的字符串拼接方式，但是并不一定就要选择 append() 和 concat() 方法，原因在于+根据符合我们的编程习惯，只有到了使用 append() 和 concat() 方法确实是可以对我们系统的效率起到比较大的帮助，才会考虑，同时鄙人也真的没有怎么用过 concat() 方法。

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

15

关键字 final

在程序设计中，我们有时可能希望某些数据是不能够改变的，这个时候 final 就有用武之地了。final 是 Java 的关键字，它所表示的是“这部分是无法修改的”。不想被改变的原因有两个：效率、设计。使用到 final 的有三种情况：数据、方法、类。

一、final 数据

有时候数据的恒定不变是很有用的，它能够减轻系统运行时的负担。对于这些恒定不变的数据我可以叫做“常量”。“常量”主要应用与以下两个地方：

1、编译期常量，永远不可改变。

2、运行期初始化时，我们希望它不会被改变。

对于编译期常量，它在类加载的过程就已经完成了初始化，所以当类加载完成后是不可更改的，编译期可以将它代入到任何用到它的计算式中，也就是说可以在编译期执行计算式。当然对于编译期常量，只能使用基本类型，而且必须要在定义时进行初始化。

有些变量，我们希望它可以根据对象的不同而表现不同，但同时又不希望它被改变，这个时候我们就可以使用运行期常量。对于运行期常量，它既可是基本数据类型，也可是引用数据类型。基本数据类型不可变的是其内容，而引用数据类型不可变的是其引用，引用所指定的对象内容是可变的。

```
public class Person {
    private String name;

    Person(String name){
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

public class FinalTest {
    private final String final_01 = "chenssy";   //编 译期常量，必须要进行初始化，且不可更改
    private final String final_02;               //构造器常量，在实例化一个对象时被初始化

    private static Random random = new Random();
    private final int final_03 = random.nextInt(50);   //使用随机数来进行初始化

    //引用
```

```
    public final Person final_04 = new Person("chen_ssy");   //final指向引用数据类型

    FinalTest(String final_02){
        this.final_02 = final_02;
    }

    public String toString(){
        return "final_01 = " + final_01 +"   final_02  = " + final_02 + "   final_03 = " + final_03 +
                "   final_04 = " + final_04.getName();
    }

    public static void main(String[] args) {
        System.out.println("------------第一次创建对象------------");
        FinalTest final1 = new FinalTest("cm");
        System.out.println(final1);
        System.out.println("------------第二次创建对 象------------");
        FinalTest final2 = new FinalTest("zj");
        System.out.println(final2);
        System.out.println("------------修改引用对 象--------------");
        final2.final_04.setName("chenssy");
        System.out.println(final2);
    }
}

------------------
Output:
------------第一次创建对象------------
final_01 = chenssy   final_02 = cm   final_03 = 34    final_04 = chen_ssy
------------第二次创建对象------------
final_01 = chenssy   final_02 = zj   final_03 = 46   final_04 = chen_ssy
------------修改引用对象--------------
final_01 = chenssy   final_02 = zj   final_03 = 46   final_04 = chenssy
```

这里只阐述一点就是：不要以为某些数据是 final 就可以在编译期知道其值，通过 final_03 我们就知道了，在这里是使用随机数其进行初始化，他要在运行期才能知道其值。

二、final 方法

所有被 final 标注的方法都是不能被继承、更改的，所以对于 final 方法使用的第一个原因就是方法锁定，以防止任何子类来对它的修改。至于第二个原因就是效率问题，鄙人对这个效率问题理解的不是很清楚，在网上摘抄这段话：

> 在 Java 的早期实现中，如果将一个方法指明为 final，就是同意编译器将针对该方法的所有调用都转为内嵌调用。当编译器发现一个 final 方法调用命令时，它会根据自己的谨慎判断，跳过插入程序代码这种正常的调用方式而执行方法调用机制（将参数压入栈，跳至方法代码处执行，然后跳回并清理栈中的参数，处理返回值），并且以方法体中的实际代码的副本来代替方法调用。这将消除方法调用的开销。当然，如果一个方法很大，你的程序代码会膨胀，因而可能看不到内嵌所带来的性能上的提高，因为所带来的性能会花费于方法内的时间量而被缩减。

对这段话理解我不是很懂就照搬了，那位 Java 牛人可以解释解释下！！

父类的 final 方法是不能被子类所覆盖的，也就是说子类是不能够存在和父类一模一样的方法的。

```
public class Custom extends Person{
    public void method1(){
        System.out.println("Person's  method1....");
    }

//    Cannot override the final method from person：子类不能覆盖父类的final方法
//    public void method2(){
//        System.out.println("Person's method2...");
//    }
}
```

三、final 类

如果某个类用 final 修改，表明该类是最终类，它不希望也不允许其他来继承它。在程序设计中处于安全或者其他原因，我们不允许该类存在任何变化，也不希望它有子类，这个时候就可以使用 final 来修饰该类了。

对于 final 修饰的类来说，它的成员变量可以为 final，也可以为非 final。如果定义为 final，那么 final 数据的规则同样适合它。而它的方法则会自动的加上 final，因为 final 类是无法被继承，所以这个是默认的。

四、final 参数

在实际应用中，我们除了可以用 final 修饰成员变量、成员方法、类，还可以修饰参数、若某个参数被 final 修饰了，则代表了该参数是不可改变的。

如果在方法中我们修改了该参数，则编译器会提示你：The final local variable i cannot be assigned. It must be blank and not using a compound assignment。

```
public class Custom {
    public void test(final int i){
        //i++;    ---final参数不可改变
        System.out.println(i);
    }

    public void test(final Person p){
     //p = new Person();    --final参数不可变
     p.setName("chenssy");
    }
}
```

同 final 修饰参数在内部类中是非常有用的，**在匿名内部类中，为了保持参数的一致性，若所在的方法的形参需要被内部类里面使用时，该形参必须为 final。**详情参看：http://www.cnblogs.com/chenssy/p/3390871.html 。

六、final 与 static

final 和 static 在一起使用就会发生神奇的化学反应，他们同时使用时即可修饰成员变量，也可修饰成员方法。

对于成员变量，该变量一旦赋值就不能改变，我们称它为“全局常量”。可以通过类名直接访问。

对于成员方法，则是不可继承和改变。可以通过类名直接访问。

更多：

Java 提高篇——关键字static：http://www.cnblogs.com/chenssy/p/3386721.html

巩固基础，提高技术，不惧困难，攀登高峰！！！！！！

16

异常（一）

Java 的基本理念是“结构不佳的代码不能运行”！！！！！

大成若缺，其用不弊。

大盈若冲，其用不穷。

在这个世界不可能存在完美的东西，不管完美的思维有多么缜密，细心，我们都不可能考虑所有的因素，这就是所谓的智者千虑必有一失。同样的道理，计算机的世界也是不完美的，异常情况随时都会发生，我们所需要做的就是避免那些能够避免的异常，处理那些不能避免的异常。这里我将记录如何利用异常还程序一个“完美世界”。

一、为什么要使用异常

首先我们可以明确一点就是异常的处理机制可以确保我们程序的健壮性，提高系统可用率。虽然我们不是特别喜欢看到它，但是我们不能不承认它的地位，作用。有异常就说明程序存在问题，有助于我们及时改正。在我们的程序设计当做，任何时候任何地方因为任何原因都有可能会出现异常，在没有异常机制的时候我们是这样处理的：通过函数的返回值来判断是否发生了异常（这个返回值通常是已经约定好了的），调用该函数的程序负责检查并且分析返回值。虽然可以解决异常问题，但是这样做存在几个缺陷：

1、容易混淆。如果约定返回值为 -11111 时表示出现异常，那么当程序最后的计算结果真的为 -1111 呢?

2、代码可读性差。将异常处理代码和程序代码混淆在一起将会降低代码的可读性。

3、由调用函数来分析异常，这要求程序员对库函数有很深的了解。

在 OO 中提供的异常处理机制是提供代码健壮的强有力的方式。

> 使用异常机制它能够降低错误处理代码的复杂度，如果不使用异常，那么就必须检查特定的错误，并在程序中的许多地方去处理它，而如果使用异常，那就不必在方法调用处进行检查，因为异常机制将保证能够捕获这个错误，并且，只需在一个地方处理错误，即所谓的异常处理程序中。这种方式不仅节约代码，而且把“概述在正常执行过程中做什么事”的代码和“出了问题怎么办”的代码相分离。总之，与以前的错误处理方法相比，异常机制使代码的阅读、编写和调试工作更加井井有条。（摘自《Think in Java》）。

在初学时，总是听老师说把有可能出错的地方记得加异常处理，刚刚开始还不明白，有时候还觉得只是多此一举，现在随着自己的不断深入，代码编写多了，渐渐明白了异常是非常重要的。

二、基本定义

在《Think in Java》中是这样定义异常的：异常情形是指阻止当前方法或者作用域继续执行的问题。在这里一定要明确一点：异常代码某种程度的错误，尽管 Java 有异常处理机制，但是我们不能以“正常”的眼光来看待异常，异常处理机制的原因就是告诉你：这里可能会或者已经产生了错误，您的程序出现了不正常的情况，可能会导致程序失败！

那么什么时候才会出现异常呢？只有在你当前的环境下程序无法正常运行下去，也就是说程序已经无法来正确解决问题了，这时它所就会从当前环境中跳出，并抛出异常。抛出异常后，它首先会做几件事。首先，它会使用 new 创建一个异常对象，然后在产生异常的位置终止程序，并且从当前环境中弹出对异常对象的引用，这时。异常处理机制就会接管程序，并开始寻找一个恰当的地方来继续执行程序，这个恰当的地方就是异常处理程序，它的任务就是将程序从错误状态恢复，以使程序要么换一种方法执行，要么继续执行下去。

总的来说异常处理机制就是当程序发生异常时，它强制终止程序运行，记录异常信息并将这些信息反馈给我们，由我们来确定是否处理异常。

二、基本定义

在《Think in Java》中是这样定义异常的：异常情形是指阻止当前方法或者作用域继续执行的问题。在这里一定要明确一点：异常代码某种程度的错误，尽管 Java 有异常处理机制，但是我们不能以“正常”的眼光来看待异常，异常处理机制的原因就是告诉你：这里可能会或者已经产生了错误，您的程序出现了不正常的情况，可能会导致程序失败！

那么什么时候才会出现异常呢？只有在你当前的环境下程序无法正常运行下去，也就是说程序已经无法来正确解决问题了，这时它所就会从当前环境中跳出，并抛出异常。抛出异常后，它首先会做几件事。首先，它会使用 new 创建一个异常对象，然后在产生异常的位置终止程序，并且从当前环境中弹出对异常对象的引用，这时。异常处理机制就会接管程序，并开始寻找一个恰当的地方来继续执行程序，这个恰当的地方就是异常处理程序，它的任务就是将程序从错误状态恢复，以使程序要么换一种方法执行，要么继续执行下去。

总的来说异常处理机制就是当程序发生异常时，它强制终止程序运行，记录异常信息并将这些信息反馈给我们，由我们来确定是否处理异常。

三、异常体系

Java 为我们提供了非常完美的异常处理机制，使得我们可以更加专心于我们的程序，在使用异常之前我们需要了解它的体系结构：如下（该图摘自：http://blog.csdn.net/zhangerqing/article/details/8248186）。

从上面这幅图可以看出，Throwable 是 java 语言中所有错误和异常的超类（万物即可抛）。它有两个子类：Error、Exception。

其中 Error 为错误，是程序无法处理的，如 OutOfMemoryError、ThreadDeath 等，出现这种情况你唯一能做的就是听之任之，交由 JVM 来处理，不过 JVM 在大多数情况下会选择终止线程。

而 Exception 是程序可以处理的异常。它又分为两种 CheckedException（受捡异常），一种是 UncheckedException（不受检异常）。其中 CheckException 发生在编译阶段，必须要使用 try…catch（或者throws）否则编译不通过。而 UncheckedException 发生在运行期，具有不确定性，主要是由于程序的逻辑问题所引起的，难以排查，我们一般都需要纵观全局才能够发现这类的异常错误，所以在程序设计中我们需要认真考虑，好好写代码，尽量处理异常，即使产生了异常，也能尽量保证程序朝着有利方向发展。

所以：对于可恢复的条件使用被检查的异常（CheckedException），对于程序错误（言外之意不可恢复，大错已经酿成）使用运行时异常（RuntimeException）。

Java 的异常类实在是太多了，产生的原因也千变万化，所以下篇博文我将会整理，统计 Java 中经常出现的异常，望各位关注！！

四、异常使用

在网上看了这样一个搞笑的话：世界上最真情的相依，是你在 try 我在 catch。无论你发神马脾气，我都默默承受，静静处理。对于初学者来说异常就是 try…catch，（鄙人刚刚接触时也是这么认为的，碰到异常就是 try…catch）。个人感觉 try…catch 确实是用的最多也是最实用的。

在异常中 try 快包含着可能出现异常的代码块，catch 块捕获异常后对异常进行处理。先看如下实例：

```
public class ExceptionTest {
    public static void main(String[] args) {
        String file = "D:\\exceptionTest.txt";
        FileReader reader;
        try {
            reader = new FileReader(file);
            Scanner in = new Scanner(reader);
            String string = in.next();
            System.out.println(string + "不知道我有幸能够执行到不.....");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            System.out.println("对不起,你执行不到...");
        }
        finally{
            System.out.println("finally 在执行...");
        }
    }
}
```

这是段非常简单的程序，用于读取 D 盘目录下的 exceptionText.txt 文件，同时读取其中的内容、输出。首先 D 盘没有该文件，运行程序结果如下：

```
java.io.FileNotFoundException: D:\exceptionTest.txt（系统找不到指定的文件。）
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:106)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:41)
    at com.test9.ExceptionTest.main(ExceptionTest.java:19)
对不起,你执行不到...
finally 在执行...
```

从这个结果我们可以看出这些：

1、当程序遇到异常时会终止程序的运行（即后面的代码不在执行），控制权交由异常处理机制处理。

2、catch 捕捉异常后，执行里面的函数。

当我们在 D 盘目录下新建一个 exceptionTest.txt 文件后，运行程序结果如下：

```
1111不知道我有幸能够执行到不.....
finally 在执行...
```

11111 是该文件中的内容。从这个运行结果可以得出这个结果：**不论程序是否发生异常，finally 代码块总是会执行。所以 finally 一般用来关闭资源。**

在这里我们在看如下程序：

```
public class ExceptionTest {
    public static void main(String[] args) {
        int[] a = {1,2,3,4};
        System.out.println(a[4]);
        System.out.println("我执行了吗???");
    }
}
```

程序运行结果：

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at com.test9.ExceptionTest.main(ExceptionTest.java:14)
```

各位请注意这个异常信息和上面的异常信息错误，为了看得更加清楚，我将 他们列在一起：

```
java.io.FileNotFoundException: D:\exceptionTest.txt (系统找不到指定的文件。)
        Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
```

在这里我们发现两个异常之间存在如下区别：第二个异常信息多了Exception in thread “main”，这显示了出现异常信息的位置。在这里可以得到如下结论：**若程序中显示的声明了某个异常，则抛出异常时不会显示出处，若程序中没有显示的声明某个异常，当抛出异常时，系统会显示异常的出处。**

由于这篇博文会比较长，所以分两篇来介绍。下篇博文主要介绍 Java 异常的自定义异常、异常链、异常的使用误区、使用异常注意地方以及 try…catch、throw、throws。望各位看客关注！！！！

17

异常(二)

承接上篇博文：*Java提高篇—–异常(一)*

五、自定义异常

Java 确实给我们提供了非常多的异常，但是异常体系是不可能预见所有的希望加以报告的错误，所以 Java 允许我们自定义异常来表现程序中可能会遇到的特定问题，总之就是一句话：我们不必拘泥于 Java 中已有的异常类型。

Java 自定义异常的使用要经历如下四个步骤：

1、定义一个类继承 Throwable 或其子类。

2、添加构造方法(当然也可以不用添加，使用默认构造方法)。

3、在某个方法类抛出该异常。

4、捕捉该异常。

```
/** 自定义异常 继承Exception类 **/
public class MyException extends Exception{
    public MyException(){

    }

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

public class Test {
    public void display(int i) throws MyException{
        if(i == 0){
            throw new MyException("该值不能为0.....");
        }
        else{
            System.out.println( i / 2);
        }
    }

    public static void main(String[] args) {
        Test test = new Test();
        try {
            test.display(0);
            System.out.println ("----------------------");
```

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

运行结果：

六、异常链

在设计模式中有一个叫做责任链模式，该模式是将多个对象链接成一条链，客户端的请求沿着这条链传递直到被接收、处理。同样 Java 异常机制也提供了这样一条链：异常链。

我们知道每遇到一个异常信息，我们都需要进行 try…catch,一个还好，如果出现多个异常呢？分类处理肯定会比较麻烦，那就一个 Exception 解决所有的异常吧。这样确实是可以，但是这样处理势必会导致后面的维护难度增加。最好的办法就是将这些异常信息封装，然后捕获我们的封装类即可。

诚然在应用程序中，我们有时候不仅仅只需要封装异常，更需要传递。怎么传递？throws!！binge，正确！！但是如果仅仅只用 throws 抛出异常，那么你的封装类，怎么办？？

我们有两种方式处理异常，一是 throws 抛出交给上级处理，二是 try…catch 做具体处理。但是这个与上面有什么关联呢？try…catch 的 catch 块我们可以不需要做任何处理，仅仅只用 throw 这个关键字将我们封装异常信息主动抛出来。然后在通过关键字 throws 继续抛出该方法异常。它的上层也可以做这样的处理，以此类推就会产生一条由异常构成的异常链。

通过使用异常链，我们可以提高代码的可理解性、系统的可维护性和友好性。

同理，我们有时候在捕获一个异常后抛出另一个异常信息，并且希望将原始的异常信息也保持起来，这个时候也需要使用异常链。

在异常链的使用中，throw 抛出的是一个新的异常信息，这样势必会导致原有的异常信息丢失，如何保持？在 Throwable 及其子类中的构造器中都可以接受一个 cause 参数，该参数保存了原有的异常信息，通过 getCause()就可以获取该原始异常信息。

语法：

```
public void test() throws XxxException{
        try {
            //do something:可能抛出异常信息的代码块
        } catch (Exception e) {
            throw new XxxException(e);
        }
    }
```

示例：

```
public class Test {
```

```
    public void f() throws MyException{
        try {
            FileReader reader = new FileReader("G:\\myfile\\struts.txt");
             Scanner in = new Scanner(reader);
             System.out.println(in.next());
        } catch (FileNotFoundException e) {
            //e 保存异常信息
            throw new MyException("文件没有找到--01",e);
        }
    }

    public void g() throws MyException{
        try {
            f();
        } catch (MyException e) {
            //e 保存异常信息
            throw new MyException("文件没有找到--02",e);
        }
    }

    public static void main(String[] args) {
        Test t = new Test();
        try {
            t.g();
        } catch (MyException e) {
            e.printStackTrace();
        }
    }
}
```

运行结果:

```
com.test9.MyException: 文件没有找到--02
    at com.test9.Test.g(Test.java:31)
    at com.test9.Test.main(Test.java:38)
Caused by: com.test9.MyException: 文件没有找到--01
    at com.test9.Test.f(Test.java:22)
    at com.test9.Test.g(Test.java:28)
    ... 1 more
Caused by: java.io.FileNotFoundException: G:\myfile  \struts.txt (系统找不到指定的路径。)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:106)
    at java.io.FileInputStream.<init>(FileInputStream.java:66)
    at java.io.FileReader.<init>(FileReader.java:41)
```

```
    at com.test9.Test.f(Test.java:17)
    ... 2 more
```

如果在程序中,去掉 e，也就是：throw new MyException(“文件没有找到 –02”);

那么异常信息就保存不了，运行结果如下：

```
com.test9.MyException: 文件没有找到--02
    at com.test9.Test.g(Test.java:31)
    at com.test9.Test.main(Test.java:38)
```

PS:其实对于异常链鄙人使用的也不是很多，理解的不是很清楚，望各位指正！！！！

七、异常的使用误区

首先我们先看如下示例：该实例能够反映 Java 异常的不正确使用（其实这也是我刚刚学 Java 时写的代码）！！

```
OutputStreamWriter out = null;
      java.sql.Connection conn = null;
      try {          //  ----------1
         Statement stat = conn.createStatement();
         ResultSet rs = stat.executeQuery("select *from user");
         while (rs.next()){
            out.println("name:" + rs.getString("name") + "sex:"
                  + rs.getString("sex"));
         }
         conn.close();       //------2
         out.close();
      }
      catch (Exception ex){   //------3
         ex.printStackTrace();   //------4
      }
```

1、———-1

对于这个 try…catch 块，我想他的真正目的是捕获 SQL 的异常，但是这个 try 块是不是包含了太多的信息了。这是我们为了偷懒而养成的代码坏习惯。有些人喜欢将一大块的代码全部包含在一个 try 块里面，因为这样省事，反正有异常它就会抛出，而不愿意花时间来分析这个大代码块有那几块会产生异常，产生什么类型的异常，反正就是一篓子全部搞定。这就想我们出去旅游将所有的东西全部装进一个箱子里面，而不是分类来装，虽不知装进去容易，找出来难啊！！！所有对于一个异常块，我们应该仔细分清楚每块的抛出异常，因为一个大代码块有太多的地方会出现异常了。

结论一：尽可能的减小try块！！！

2、——-2

在这里你发现了什么？异常改变了运行流程！！不错就是异常改变了程序运行流程。如果该程序发生了异常那么 conn.close(); out.close();是不可能执行得到的，这样势必会导致资源不能释放掉。所以如果程序用到了文

件、Socket、JDBC 连接之类的资源，即使遇到了异常，我们也要确保能够正确释放占用的资源。这里 finally 就有用武之地了：不管是否出现了异常，finally 总是有机会运行的，所以 finally 用于释放资源是再适合不过了。

**结论二：保证所有资源都被正确释放。充分运用 finally 关键词。 **

3、———3

对于这个代码我想大部分人都是这样处理的，（LZ 也是）。使用这样代码的人都有这样一个心理，一个 catch 解决所有异常，这样是可以，但是不推荐！为什么！首先我们需要明白 catch 块所表示是它预期会出现何种异常，并且需要做何种处理，而使用 Exception 就表示他要处理所有的异常信息，但是这样做有什么意义呢?

这里我们再来看看上面的程序实例，很显然它可能需要抛出两个异常信息， SQLException 和 IOExceptio n。所以一个 catch 处理两个截然不同的 Exception 明显的不合适。如果用两个 catch，一个处理 SQLExcepti on、一个处理 IOException 就好多了。所以：

结论三：catch 语句应当尽量指定具体的异常类型，而不应该指定涵盖范围太广的 Exception 类。 不要一个 Ex ception 试图处理所有可能出现的异常。

4、———4

这个就问题多多了，我敢保证几乎所有的人都这么使用过。这里涉及到了两个问题，一是，捕获了异常不做处理，二是异常信息不够明确。

4.1、捕获异常不做处理，就是我们所谓的丢弃异常。我们都知道异常意味着程序出现了不可预期的问题，程序它希望我们能够做出处理来拯救它，但是你呢? 一句 ex.printStackTrace() 搞定，这是多么的不负责任对程序的异常情况不理不顾。虽然这样在调试可能会有一定的帮助，但是调试阶段结束后呢? 不是一句 ex.printStackTrac e() 就可以搞定所有的事情的!

那么怎么改进呢? 有四种选择:

1、处理异常。对所发生的的异常进行一番处理，如修正错误、提醒。再次申明 ex.printStackTrace() 算不上已经“处理好了异常”.

2、重新抛出异常。既然你认为你没有能力处理该异常，那么你就尽情向上抛吧！！！

3、封装异常。这是 LZ 认为最好的处理方法，对异常信息进行分类，然后进行封装处理。

4、不要捕获异常。

4.2、异常信息不明确。我想对于这样的：java.io.FileNotFoundException: ……信息除了我们 IT 人没有几个人看得懂和想看吧！所以在出现异常后，我们最好能够提供一些文字信息，例如当前正在执行的类、方法和其他状态信息，包括以一种更适合阅读的方式整理和组织 printStackTrace 提供的信息。起码我公司是需要将异常信息所在的类、方法、何种异常都需要记录在日志文件中的。

所以：

结论四：既然捕获了异常，就要对它进行适当的处理。不要捕获异常之后又把它丢弃，不予理睬。 不要做一个不负责的人。

结论五：在异常处理模块中提供适量的错误原因信息，组织错误信息使其易于理解和阅读。

对于异常还有以下几个注意地方：

六、不要在finally块中处理返回值。

七、不要在构造函数中抛出异常。

八、try…catch、throw、throws

在这里主要是区分 throw 和 throws。

throws 是方法抛出异常。在方法声明中，如果添加了 throws 子句，表示该方法即将抛出异常，异常的处理交由它的调用者，至于调用者任何处理则不是它的责任范围内的了。所以如果一个方法会有异常发生时，但是又不想处理或者没有能力处理，就使用 throws 吧！

而 throw 是语句抛出异常。它不可以单独使用，要么与 try…catch 配套使用，要么与 throws 配套使用。

```
//使用throws抛出异常
    public void f() throws MyException{
        try {
           FileReader reader = new FileReader("G:\\myfile\\struts.txt");
            Scanner in = new Scanner(reader);
            System.out.println(in.next());
        } catch (FileNotFoundException e) {
           throw new MyException("文件没有找到", e);    //throw
        }

    }
```

九、总结

其实对于异常使用的优缺点现在确实存在很多的讨论。例如：http://www.cnblogs.com/mailingfeng/archive/2012/11/14/2769974.html 。这篇博文对于是否需要使用异常进行了比较深刻的讨论。LZ实乃菜鸟一枚，不能理解异常深奥之处。但是有一点LZ可以肯定，那就是异常必定会影响系统的性能。

异常使用指南（摘自：Think in Java）

应该在下列情况下使用异常。

1、在恰当的级别处理问题（在知道该如何处理异常的情况下才捕获异常）。

2、解决问题并且重新调用产生异常的方法。

3、进行少许修补，然后绕过异常发生的地方继续执行。

4、用别的数据进行计算，以代替方法预计会返回的值。

5、把当前运行环境下能做的事情尽量做完。然后把相同（不同）的异常重新抛到更高层。

6、终止程序。

7、进行简化。

8、让类库和程序更加安全。（这既是在为调试做短期投资，也是在为程序的健壮做长期投资）

更多阅读：*Java提高篇—–异常(一)*。

18

数组之一：认识 JAVA 数组

噢，它明白了，河水既没有牛伯伯说的那么浅，也没有小松鼠说的那么深，只有自己亲自试过才知道！道听途说永远只能看到表明现象，只有亲自试过了，才知道它的深浅！！！！！

一、什么是数组

数组？什么是数组？在我印象中的数组是应该这样的：通过 new 关键字创建并组装他们，通过使用整形索引值访问它的元素，并且它的尺寸是不可变的！

但是这只是数组的最表面的东西！深一点？就是这样：数组是一个简单的复合数据类型，它是一系列有序数据的集合，它当中的每一个数据都具有相同的数据类型，我们通过数组名加上一个不会越界下标值来唯一确定数组中的元素。

还有更深的，那就是数组是一个特殊的对象！！（对于这个 LZ 理解的不是很好，对JVM也没有看，所以见解有限）。以下参考文献：http://developer.51cto.com/art/201001/176671.htm、http://www.blogjava.net/flysky19/articles/92763.html?opt=admin

不管在其他语言中数组是什么，在 Java 中它就是对象。一个比较特殊的对象。

```
public class Test {
    public static void main(String[] args) {
        int[] array = new int[10];
        System.out.println("array的父类是：" +  array.getClass().getSuperclass());
        System.out.println("array的类名是：" + array.getClass().getName());
    }
}
-------Output:
array的父类是：class java.lang.Object
array的类名是：[I
```

从上面示例可以看出,数组的是 Object 的直接子类,它属于“第一类对象”，但是它又与普通的 Java 对象存在很大的不同，从它的类名就可以看出：[I，这是什么东东？？在 JDK 中我就没有找到这个类，话说这个”[I”都不是一个合法标识符。怎么定义成类啊？所以我认为 SUM 那帮天才肯定对数组的底层肯定做了特殊的处理。

我们再看如下示例：

```
public class Test {
    public static void main(String[] args) {
        int[] array_00 = new int[10];
        System.out.println("一维数组：" + array_00.getClass().getName());
        int[][] array_01 = new int[10][10];
        System.out.println("二维数组：" + array_01.getClass().getName());
```

```
        int[][][] array_02 = new int[10][10][10];
        System.out.println("三维数组：" + array_02.getClass().getName());
    }
}
-----------------Output:
一维数组：[I
二维数组：[[I
三维数组：[[[I
```

通过这个实例我们知道：[代表了数组的维度，一个[表示一维，两个[表示二维。可以简单的说数组的类名由若干个’[‘和数组元素类型的内部名称组成。不清楚我们再看：

```
public class Test {
    public static void main(String[] args) {
        System.out.println("Object[]:" + Object  [].class);
        System.out.println("Object[][]:" + Object[][].class);
        System.err.println("Object[][][]:" + Object[]  [][].class);
        System.out.println("Object:" + Object.class);
    }
}
---------Output:
Object[]:class [Ljava.lang.Object;
Object[][]:class [[Ljava.lang.Object;
Object[][][]:class [[[Ljava.lang.Object;
Object:class java.lang.Object
```

从这个实例我们可以看出数组的“庐山真面目”。同时也可以看出数组和普通的 Java 类是不同的，普通的 Java 类是以全限定路径名 + 类名来作为自己的唯一标示的，而数组则是以若干个 [+L+ 数组元素类全限定路径+类来最为唯一标示的。这个不同也许在某种程度上说明了数组也普通 Java 类在实现上存在很大的区别，也许可以利用这个区别来使得 JVM 在处理数组和普通 Java 类时作出区分。

我们暂且不论这个[I 是什么东东，是由谁来声明的，怎么声明的（这些我现在也不知道！但是有一点可以确认：这个是在运行时确定的）。先看如下：

```
public class Test {
    public static void main(String[] args) {
        int[] array = new int[10];
        Class clazz = array.getClass();
        System.out.println(clazz.getDeclaredFields ().length);
        System.out.println(clazz.getDeclaredMethods ().length);
        System.out.println(clazz.getDeclaredConstructors().length);
        System.out.println(clazz.getDeclaredAnnotations().length);
        System.out.println(clazz.getDeclaredClasses().length);
```

```
    }
}
----------------Output:
0
0
0
0
0
```

从这个运行结果可以看出，我们亲爱的 [I 没有生命任何成员变量、成员方法、构造函数、Annotation 甚至连 length 成员变量这个都没有，它就是一个彻彻底底的空类。没有声明 length，那么我们 array.length 时，编译器怎么不会报错呢？确实，数组的 length 是一个非常特殊的成员变量。我们知道数组的是 Object 的直接之类，但是 Object 是没有 length 这个成员变量的，那么 length 应该是数组的成员变量，但是从上面的示例中，我们发现数组根本就没有任何成员变量，这两者不是相互矛盾么？

```
public class Main {
    public static void main(String[] args) {
        int a[] = new int[2];
        int i = a.length;
    }
}
```

打开 class 文件，得到 main 方法的字节码：

```
0 iconst_2                //将int型常量2压入操作数栈
    1 newarray 10 (int)         //将2弹出操作数栈，作为长度，创建一个元素类型为int, 维度为1的数组，并将数组的引用压入操作数栈
    3 astore_1                //将数组的引用从操作数栈中弹出，保存在索引为1的局部变量(即a)中
    4 aload_1                 //将索引为1的局部变量(即a)压入操作数栈
    5 arraylength               //从操作数栈弹出数组引用(即a)，并获取其长度(JVM负责实现如何获取)，并将长度压入操作数栈
    6 istore_2                //将数组长度从操作数栈弹出，保存在索引为2的局部变量(即i)中
    7 return                  //main方法返回
```

在这个字节码中我们还是没有看到 length 这个成员变量，但是看到了这个:arraylength ,这条指令是用来获取数组的长度的，所以说 JVM 对数组的长度做了特殊的处理，它是通过 arraylength 这条指令来实现的。

二、数组的使用方法

通过上面算是对数组是什么有了一个初步的认识，下面将简单介绍数组的使用方法。

数组的使用方法无非就是四个步骤：声明数组、分配空间、赋值、处理。

声明数组：就是告诉计算机数组的类型是什么。有两种形式：int[] array、int array[]。

分配空间：告诉计算机需要给该数组分配多少连续的空间，记住是连续的。array = new int[10];

赋值：赋值就是在已经分配的空间里面放入数据。array[0] = 1 、array[1] = 2…… 其实分配空间和赋值是一起进行的，也就是完成数组的初始化。有如下三种形式：

```
int a[] = new int[2];   //默认为0,如果是引用数据类型就为 null
        int b[] = new int[] {1,2,3,4,5};
        int c[] = {1,2,3,4,5};
```

处理：就是对数组元素进行操作。通过数组名+有效的下标来确认数据。

PS：由于能力有限，所以“什么是数组”主要是参考这篇博文：http://developer.51cto.com/art/201001/176671.htm 下篇将更多的介绍数组的一些特性，例如：效率问题、Array 的使用、浅拷贝以及与 list 之间的转换问题。

19

数组之二

前面一节主要介绍了数组的基本概念，对什么是数组稍微深入了一点点，在这篇博文中主要介绍数组的其他方面。

三、性能？请优先考虑数组

在 Java 中有很多方式来存储一系列数据，而且在操作上面比数组方便的多？但为什么我们还需要使用数组，而不是替代它呢？数组与其他种类的容器之间的区别有三个方面：效率、类型和保存基本类型的能力。在 Java 中，数组是一种效率最高的存储和随机访问对象引用序列的方式。

在项目设计中数组使用的越来越少了，而且它确实是没有 List、Set 这些集合使用方便，但是在某些方面数组还是存在一些优势的，例如：速度，而且集合类的底层也都是通过数组来实现的。

```
--------这是ArrayList的add()------
   public boolean add(E e) {
   ensureCapacity(size + 1);  // Increments modCount!!
   elementData[size++] = e;
   return true;
   }
```

下面利用数组和 list 来做一些操作比较。

一、求和

```
Long time1 = System.currentTimeMillis();
      for(int i = 0 ; i < 100000000 ;i++){
         sum += arrays[i%10];
      }
      Long time2 = System.currentTimeMillis();
      System.out.println("数组求和所花费时间：" + (time2 - time1) + "毫秒");
      Long time3 = System.currentTimeMillis();
      for (int i = 0; i < 100000000; i++) {
         sum  += list.get(i%10);
      }
      Long time4 = System.currentTimeMillis();
      System.out.println("List求和所花费时间：" + (time4 - time3) + "毫秒");
--------------Output:
数组求和所花费时间：696毫秒
List求和所花费时间：3498毫秒
```

从上面的时间消耗上面来说数组对于基本类型的求和计算的速度是集合的 5 倍左右。其实在 list 集合中，求和当中有一个致命的动作：list.get(i)。这个动作是进行拆箱动作，Integer 对象通过 intValue 方法自动转换成一个 int 基本类型，在这里就产生了不必要的性能消耗。

所以在性能要求较高的场景中请优先考虑数组。

四、变长数组?

数组是定长的，一旦初始化声明后是不可改变长度的。这对我们在实际开发中是非常不方便的，聪明的我们肯定是可以找到方法来实现的。就如 Java 不能实现多重继承一样，我们一样可以利用内部类和接口来实现(请参考：Java提高篇(九)—－实现多重继承)。

那么如何来实现变长数组呢? 我们可以利用 List 集合 add 方法里面的扩容思路来模拟实现。下面是 ArrayList 的扩容方法:

```
public void ensureCapacity(int minCapacity) {
        modCount++;
        int oldCapacity = elementData.length;
        /**
         * 若当前需要的长度超过数组长度时进行扩容处理
         */
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            int newCapacity = (oldCapacity * 3) / 2 + 1;    //扩容
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
            //拷贝数组，生成新的数组
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
```

这段代码对我们有用的地方就在于 if 语句后面。它的思路是将原始数组拷贝到新数组中，新数组是原始数组长度的 1.5 倍。所以模拟的数组扩容代码如下:

```
public class ArrayUtils {
    /**
     * @desc 对数组进行扩容
     * @author chenssy
     * @data 2013-12-8
     * @param <T>
     * @param datas 原始数组
     * @param newLen 扩容大小
     * @return T[]
     */
    public static <T> T[] expandCapacity(T[] datas,int newLen){
        newLen = newLen < 0 ? datas.length :datas.length + newLen;
```

```
        //生成一个新的数组
        return Arrays.copyOf(datas, newLen);
    }

    /**
     * @desc 对数组进行扩容处理，1.5倍
     * @author chenssy
     * @data 2013-12-8
     * @param <T>
     * @param datas  原始数组
     * @return T[]
     */
    public static <T> T[] expandCapacity(T[] datas){
        int newLen = (datas.length * 3) / 2;     //扩容原始数组的1.5倍
        //生成一个新的数组
        return Arrays.copyOf(datas, newLen);
    }

    /**
     * @desc 对数组进行扩容处理，
     * @author chenssy
     * @data 2013-12-8
     * @param <T>
     * @param datas 原始数组
     * @param mulitiple 扩容的倍数
     * @return T[]
     */
    public static <T> T[] expandCapacityMul(T[] datas,int mulitiple){
        mulitiple = mulitiple < 0 ? 1 : mulitiple;
        int newLen = datas.length * mulitiple;
        return Arrays.copyOf(datas,newLen );
    }
}
```

通过这种迂回的方式我们可以实现数组的扩容。因此在项目中如果确实需要变长的数据集，数组也是在考虑范围之内的，我们不能因为他是固定长度而排斥他！

五、数组复制问题

以前在做集合拷贝的时候由于集合没有拷贝的方法，所以一个一个的复制是非常麻烦的，所以我就干脆使用 List.toArray() 方法转换成数组然后再通过 Arrays.copyOf 拷贝，在转换成集合，个人觉得非常方便，殊不知我已经陷入了其中的陷进！我们知道若数组元素为对象，则数组里面数据是对象引用

```
public class Test {
    public static void main(String[] args) {
        Person person_01 = new Person("chenssy_01");

        Person[] persons1 = new Person[]{person_01};
        Person[] persons2 = Arrays.copyOf(persons1,persons1.length);

        System.out.println("数组persons1:");
        display(persons1);
        System.out.println("----------------------");
        System.out.println("数组persons2:");
        display(persons2);
        //改变其值
        persons2[0].setName("chessy_02");
        System.out.println("------------改变其值后------------");
        System.out.println("数组persons1:");
        display(persons1);
        System.out.println("----------------------");
        System.out.println("数组persons2:");
        display(persons2);
    }
    public static void display(Person[] persons){
        for(Person person : persons){
            System.out.println(person.toString());
        }
    }
}
-------------Output:
数组persons1:
姓名是：chenssy_01
----------------------
数组persons2:
姓名是：chenssy_01
------------改变其值后------------
数组persons1:
```

```
姓名是：chessy_02
----------------------
数组persons2:
姓名是：chessy_02
```

从结果中发现,persons1 中的值也发生了改变，这是典型的浅拷贝问题。所以通过 Arrays.copyOf() 方法产生的数组是一个浅拷贝。同时数组的 clone() 方法也是，集合的 clone() 方法也是，所以我们在使用拷贝方法的同时一定要注意浅拷贝这问题。

有关于深浅拷贝的博文，参考：

渐析Java的浅拷贝和深拷贝：http://www.cnblogs.com/chenssy/p/3308489.html 。

使用序列化实现对象的拷贝：http://www.cnblogs.com/chenssy/p/3382979.html 。

六、数组转换为 List 注意地方

我们经常需要使用到 Arrays 这个工具的 asList() 方法将其转换成列表。方便是方便，但是有时候会出现莫名其妙的问题。如下：

```
public static void main(String[] args) {
        int[] datas = new int[]{1,2,3,4,5};
        List list = Arrays.asList(datas);
        System.out.println(list.size());
    }
------------Output:
1
```

结果是 1,是的你没有看错, 结果就是 1。但是为什么会是 1 而不是 5 呢？先看 asList() 的源码

```
public static <T> List<T> asList(T... a) {
        return new ArrayList<T>(a);
    }
```

注意这个参数:T…a，这个参数是一个泛型的变长参数，我们知道基本数据类型是不可能泛型化的，也是就说 8 个基本数据类型是不可作为泛型参数的，但是为什么编译器没有报错呢？这是因为在 Java 中，数组会当做一个对象来处理，它是可以泛型的，所以我们的程序是把一个 int 型的数组作为了 T 的类型，所以在转换之后 List 中就只会存在一个类型为 int 数组的元素了。所以我们这样的程序 System.out.println(datas.equals(list.get(0)));输出结果肯定是 true。当然如果将int改为 Integer，则长度就会变成 5 了。

我们在看下面程序：

```
enum Week{Sum,Mon,Tue,Web,Thu,Fri,Sat}
    public static void main(String[] args) {
        Week[] weeks = {Week.Sum,Week.Mon,Week.Tue,Week.Web,Week.Thu,Week.Fri};
        List<Week> list = Arrays.asList(weeks);
        list.add(Week.Sat);
    }
```

这个程序非常简单，就是讲一个数组转换成 list，然后改变集合中值，但是运行呢？

```
Exception in thread "main"   java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(AbstractList.java:131)
```

```
    at java.util.AbstractList.add(AbstractList.java:91)
    at com.array.Test.main(Test.java:18)
```

编译没错，但是运行竟然出现了异常错误！UnsupportedOperationException，当不支持请求的操作时，就会抛出该异常。从某种程度上来说就是不支持add方法，我们知道这是不可能的！什么原因引起这个异常呢？先看 asList() 的源代码：

```
public static <T> List<T> asList(T... a) {
    return new ArrayList<T>(a);
}
```

这里是直接返回一个 ArrayList 对象返回，但是注意这个 ArrayList 并不是 java.util.ArrayList,而是 Arrays 工具类的一个内之类：

```
private static class ArrayList<E> extends AbstractList<E>
    implements RandomAccess, java.io.Serializable{
    private static final long serialVersionUID =  -2764017481108945198L;
    private final E[] a;
    ArrayList(E[] array) {
        if (array==null)
            throw new NullPointerException();
    a = array;
}
    /** 省略方法 **/
}
```

但是这个内部类并没有提供 add() 方法，那么查看父类：

```
public boolean add(E e) {
    add(size(), e);
    return true;
    }
    public void add(int index, E element) {
    throw new UnsupportedOperationException();
    }
```

这里父类仅仅只是提供了方法，方法的具体实现却没有，所以具体的实现需要子类自己来提供，但是非常遗憾

这个内部类 ArrayList 并没有提高 add 的实现方法。在 ArrayList 中，它主要提供了如下几个方法：

1、size：元素数量

2、toArray：转换为数组，实现了数组的浅拷贝。

3、get：获得指定元素。

4、contains：是否包含某元素。

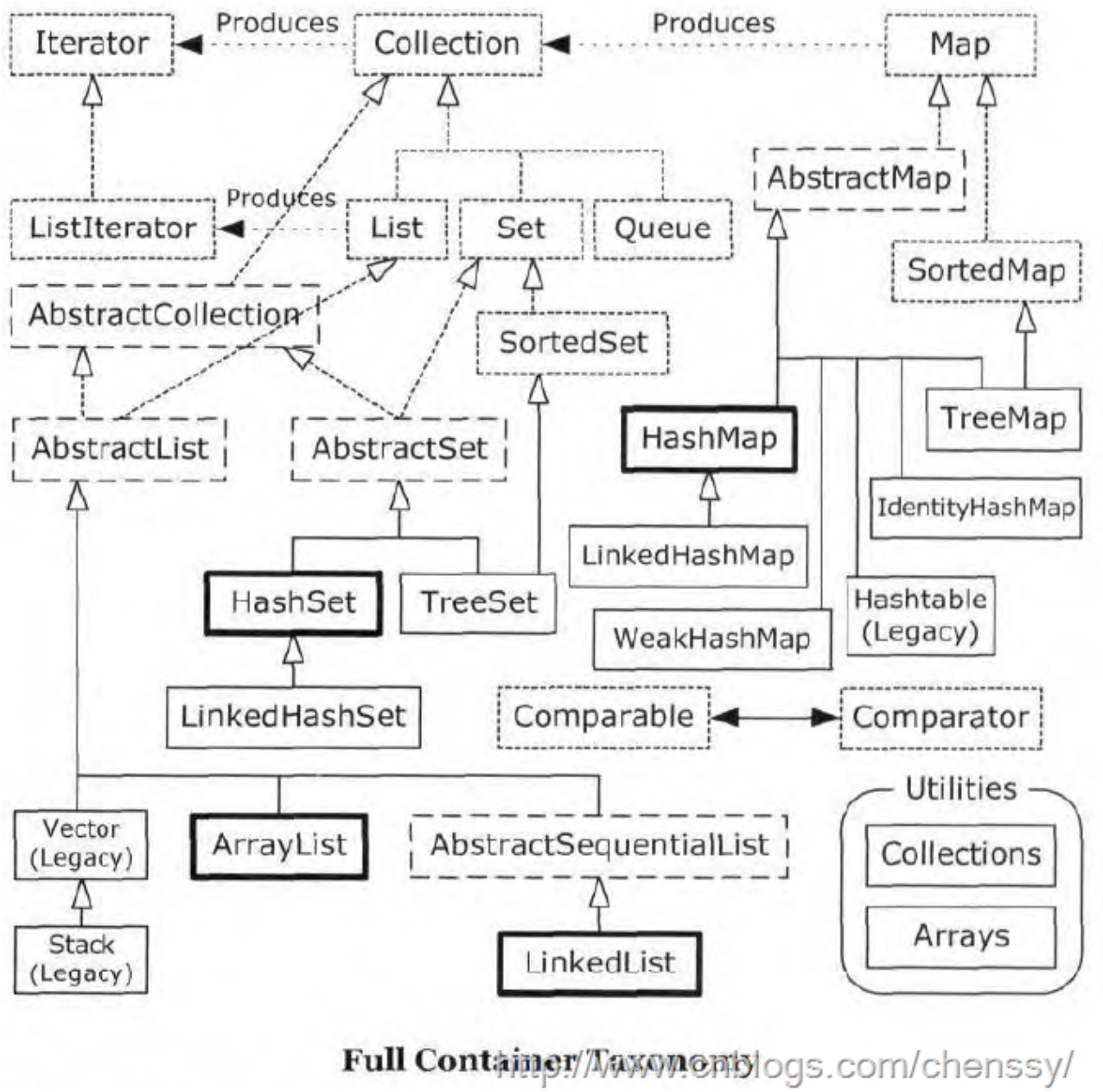
所以综上所述，asList 返回的是一个长度不可变的列表。数组是多长，转换成的列表是多长，我们是无法通过 add、remove 来增加或者减少其长度的。

参考文献：《编写高质量代码 - 改善 Java 程序的 151 个建议》

20

集合大家族

在编写 Java 程序中，我们最常用的除了八种基本数据类型，String 对象外还有一个集合类，在我们的的程序中到处充斥着集合类的身影！Java 中集合大家族的成员实在是太丰富了，有常用的 ArrayList、HashMap、HashSet，也有不常用的 Stack、Queue，有线程安全的 Vector、HashTable，也有线程不安全的 LinkedList、TreeMap 等等！



图片 20.1 fig.1

上面的图展示了整个集合大家族的成员以及他们之间的关系。下面就上面的各个接口、基类做一些简单的介绍(主要介绍各个集合的特点。区别)，更加详细的介绍会在不久的将来一一讲解。

一、Collection 接口

Collection 接口是最基本的集合接口，它不提供直接的实现，Java SDK提供的类都是继承自 Collection 的“子接口”如 List 和 Set。Collection 所代表的是一种规则，它所包含的元素都必须遵循一条或者多条规则。如有些允许重复而有些则不能重复、有些必须要按照顺序插入而有些则是散列，有些支持排序但是有些则不支持。

在 Java 中所有实现了 Collection 接口的类都必须提供两套标准的构造函数，一个是无参，用于创建一个空的 Collection，一个是带有 Collection 参数的有参构造函数，用于创建一个新的 Collection，这个新的 Collection 与传入进来的 Collection 具备相同的元素。

二、List 接口

List 接口为 Collection 直接接口。List 所代表的是有序的 Collection，即它用某种特定的插入顺序来维护元素顺序。用户可以对列表中每个元素的插入位置进行精确地控制，同时可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。实现 List 接口的集合主要有：ArrayList、LinkedList、Vector、Stack。

2.1、ArrayList

ArrayList 是一个动态数组，也是我们最常用的集合。它允许任何符合规则的元素插入甚至包括 null。每一个 ArrayList 都有一个初始容量（10），该容量代表了数组的大小。随着容器中的元素不断增加，容器的大小也会随着增加。在每次向容器中增加元素的同时都会进行容量检查，当快溢出时，就会进行扩容操作。所以如果我们明确所插入元素的多少，最好指定一个初始容量值，避免过多的进行扩容操作而浪费时间、效率。

size、isEmpty、get、set、iterator 和 listIterator 操作都以固定时间运行。add 操作以分摊的固定时间运行，也就是说，添加 n 个元素需要 O(n) 时间（由于要考虑到扩容，所以这不只是添加元素会带来分摊固定时间开销那样简单）。

ArrayList 擅长于随机访问。同时 ArrayList 是非同步的。

2.2、LinkedList

同样实现 List 接口的 LinkedList 与 ArrayList 不同，ArrayList是一个动态数组，而 LinkedList 是一个双向链表。所以它除了有 ArrayList 的基本操作方法外还额外提供了 get，remove，insert 方法在 LinkedList 的首部或尾部。

由于实现的方式不同，LinkedList 不能随机访问，它所有的操作都是要按照双重链表的需要执行。在列表中索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。这样做的好处就是可以通过较低的代价在 List 中进行插入和删除操作。

与 ArrayList 一样，LinkedList 也是非同步的。如果多个线程同时访问一个 List，则必须自己实现访问同步。一种解决方法是在创建 List 时构造一个同步的 List：

```
List list = Collections.synchronizedList(new LinkedList(…));
```

2.3、Vector

与 ArrayList 相似，但是 Vector 是同步的。所以说 Vector 是线程安全的动态数组。它的操作与 ArrayList 几乎一样。

2.4、Stack

Stack 继承自 Vector，实现一个后进先出的堆栈。Stack 提供 5 个额外的方法使得 Vector 得以被当作堆栈使用。基本的 push 和 pop 方法，还有 peek 方法得到栈顶的元素，empty 方法测试堆栈是否为空，search 方法检测一个元素在堆栈中的位置。Stack 刚创建后是空栈。

三、Set 接口

Set 是一种不包括重复元素的 Collection。它维持它自己的内部排序，所以随机访问没有任何意义。与 List 一样，它同样运行 null 的存在但是仅有一个。由于 Set 接口的特殊性，所有传入 Set 集合中的元素都必须不同，同时要注意任何可变对象，如果在对集合中元素进行操作时，导致e1.equals(e2)==true，则必定会产生某些问题。实现了 Set 接口的集合有：EnumSet、HashSet、TreeSet。

3.1、EnumSet

是枚举的专用 Set。所有的元素都是枚举类型。

3.2、HashSet

HashSet 堪称查询速度最快的集合，因为其内部是以 HashCode 来实现的。它内部元素的顺序是由哈希码来决定的，所以它不保证 set 的迭代顺序；特别是它不保证该顺序恒久不变。

3.3、TreeSet

基于 TreeMap，生成一个总是处于排序状态的 set，内部以 TreeMap 来实现。它是使用元素的自然顺序对元素进行排序，或者根据创建 Set 时提供的 `Comparator` 进行排序，具体取决于使用的构造方法。

四、Map 接口

Map 与 List、Set 接口不同，它是由一系列键值对组成的集合，提供了 key 到 Value 的映射。同时它也没有继承 Collection。在 Map 中它保证了 key 与 value 之间的一一对应关系。也就是说一个 key 对应一个 value，所以它不能存在相同的 key 值，当然 value 值可以相同。实现 map 的有：HashMap、TreeMap、HashTable、Properties、EnumMap。

4.1、HashMap

以哈希表数据结构实现，查找对象时通过哈希函数计算其位置，它是为快速查询而设计的，其内部定义了一个 hash 表数组（Entry[] table），元素会通过哈希转换函数将元素的哈希地址转换成数组中存放的索引，如果有冲突，则使用散列链表的形式将所有相同哈希地址的元素串起来，可能通过查看 HashMap.Entry 的源码它是一个单链表结构。

4.2、TreeMap

键以某种排序规则排序，内部以 red-black（红-黑）树数据结构实现，实现了 SortedMap 接口

4.3、HashTable

也是以哈希表数据结构实现的，解决冲突时与 HashMap 也一样也是采用了散列链表的形式，不过性能比 HashMap 要低

五、Queue

队列，它主要分为两大类，一类是阻塞式队列，队列满了以后再插入元素则会抛出异常，主要包括 ArrayBlockQueue、PriorityBlockingQueue、LinkedBlockingQueue。另一种队列则是双端队列，支持在头、尾两端插入和移除元素，主要包括：ArrayDeque、LinkedBlockingDeque、LinkedList。

六、异同点

出处：http://blog.csdn.net/softwave/article/details/4166598

6.1、Vector 和 ArrayList

1，vector 是线程同步的，所以它也是线程安全的，而 arraylist 是线程异步的，是不安全的。如果不考虑到线程的安全因素，一般用 arraylist 效率比较高。

2，如果集合中的元素的数目大于目前集合数组的长度时，vector 增长率为目前数组长度的 100%,而 arraylist 增长率为目前数组长度的 50%.如过在集合中使用数据量比较大的数据，用 vector 有一定的优势。

3，如果查找一个指定位置的数据，vector 和 arraylist 使用的时间是相同的，都是 0(1),这个时候使用 vector 和 arraylist 都可以。而如果移动一个指定位置的数据花费的时间为 0(n-i)n 为总长度，这个时候就应该考虑到使用 l inklist,因为它移动一个指定位置的数据所花费的时间为 0(1),而查询一个指定位置的数据时花费的时间为 0(i)。

ArrayList 和 Vector 是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要设计到数组元素移动等内存操作，所以索引数据快插入数据慢，Vector 由于使用了 synchronized 方法（线程安全）所以性能上比 ArrayList 要差，LinkedList 使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快!

6.2、Aarraylist 和 Linkedlist

1.ArrayList 是实现了基于动态数组的数据结构，LinkedList 基于链表的数据结构。

2.对于随机访问 get 和 set，ArrayList 觉得优于 LinkedList，因为 LinkedList 要移动指针。

3.对于新增和删除操作 add 和 remove，LinedList 比较占优势，因为 ArrayList 要移动数据。

这一点要看实际情况的。若只对单条数据插入或删除，ArrayList 的速度反而优于 LinkedList。但若是批量随机的插入删除数据，LinkedList 的速度大大优于 ArrayList. 因为 ArrayList 每插入一条数据，要移动插入点及之后的所有数据。

6.3、HashMap 与 TreeMap

1、HashMap 通过 hashcode 对其内容进行快速查找，而 TreeMap 中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用 TreeMap（HashMap 中元素的排列顺序是不固定的）。HashMap 中元素的排列顺序是不固定的）。

2、HashMap 通过 hashcode 对其内容进行快速查找，而 TreeMap 中所有的元素都保持着某种固定的顺序，如果你需要得到一个有序的结果你就应该使用 TreeMap（HashMap 中元素的排列顺序是不固定的）。集合框架”提供两种常规的 Map 实现：HashMap 和 TreeMap (TreeMap 实现 SortedMap 接口)。

3、在 Map 中插入、删除和定位元素，HashMap 是最好的选择。但如果您要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。使用 HashMap 要求添加的键类明确定义了 hashCode() 和 equals() 的实现。 这个 TreeMap 没有调优选项，因为该树总处于平衡状态。

6.4、hashtable 与 hashmap

1、历史原因:Hashtable 是基于陈旧的 Dictionary 类的，HashMap 是Java 1.2 引进的 Map 接口的一个实现。

2、同步性:Hashtable 是线程安全的，也就是说是同步的，而 HashMap 是线程序不安全的，不是同步的 。

3、值：只有 HashMap 可以让你将空值作为一个表的条目的 key 或value。

七、对集合的选择

7.1、对 List 的选择

1、对于随机查询与迭代遍历操作，数组比所有的容器都要快。所以在随机访问中一般使用 ArrayList

2、LinkedList 使用双向链表对元素的增加和删除提供了非常好的支持，而 ArrayList 执行增加和删除元素需要进行元素位移。

3、对于 Vector 而已，我们一般都是避免使用。

4、将 ArrayList 当做首选，毕竟对于集合元素而已我们都是进行遍历，只有当程序的性能因为 List 的频繁插入和删除而降低时，再考虑 LinkedList。

7.2、对 Set 的选择

1、HashSet 由于使用 HashCode 实现，所以在某种程度上来说它的性能永远比 TreeSet 要好，尤其是进行增加和查找操作。

3、虽然 TreeSet 没有 HashSet 性能好，但是由于它可以维持元素的排序，所以它还是存在用武之地的。

7.3、对 Map 的选择

1、HashMap 与 HashSet 同样，支持快速查询。虽然 HashTable 速度的速度也不慢，但是在 HashMap 面前还是稍微慢了些，所以 HashMap 在查询方面可以取代 HashTable。

2、由于 TreeMap 需要维持内部元素的顺序，所以它通常要比 HashMap 和 HashTable 慢。

个人网站：CMSBLOGS

ArrayList

一、ArrayList 概述

ArrayList 是实现 List 接口的动态数组，所谓动态就是它的大小是可变的。实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。

每个 ArrayList 实例都有一个容量，该容量是指用来存储列表元素的数组的大小。默认初始容量为 10。随着 ArrayList 中元素的增加，它的容量也会不断的自动增长。在每次添加新的元素时，ArrayList 都会检查是否需要进行扩容操作，扩容操作带来数据向新数组的重新拷贝，所以如果我们知道具体业务数据量，在构造 ArrayList 时可以给 ArrayList 指定一个初始容量，这样就会减少扩容时数据的拷贝问题。当然在添加大量元素前，应用程序也可以使用 ensureCapacity 操作来增加 ArrayList 实例的容量，这可以减少递增式再分配的数量。

注意，ArrayList 实现不是同步的。如果多个线程同时访问一个 ArrayList 实例，而其中至少一个线程从结构上修改了列表，那么它必须保持外部同步。所以为了保证同步，最好的办法是在创建时完成，以防止意外对列表进行不同步的访问：

```
List list = Collections.synchronizedList(new ArrayList(…));
```

二、ArrayList 源码分析

ArrayList 我们使用的实在是太多了，非常熟悉，所以在这里将不介绍它的使用方法。ArrayList 是实现 List 接口的，底层采用数组实现，所以它的操作基本上都是基于对数组的操作。

2.1、底层使用数组

```
private transient Object[] elementData;
```

transient? ？为 Java 关键字，为变量修饰符，如果用 transient 声明一个实例变量，当对象存储时，它的值不需要维持。Java 的 serialization 提供了一种持久化对象实例的机制。当持久化对象时，可能有一个特殊的对象数据成员，我们不想用 serialization 机制来保存它。为了在一个特定对象的一个域上关闭 serialization，可以在这个域前加上关键字 transient。当一个对象被序列化的时候，transient 型变量的值不包括在序列化的表示中，然而非 transient 型的变量是被包括进去的。

这里 Object[] elementData，就是我们的 ArrayList 容器，下面介绍的基本操作都是基于该 elementData 变量来进行操作的。

2.2、构造函数

ArrayList 提供了三个构造函数：

ArrayList()：默认构造函数，提供初始容量为 10 的空列表。

ArrayList(int initialCapacity)：构造一个具有指定初始容量的空列表。

ArrayList(Collection<? extends E> c)：构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。

```
/**
     * 构造一个初始容量为 10 的空列表
     */
    public ArrayList() {
        this(10);
    }
```

```
    /**
     * 构造一个具有指定初始容量的空列表。
     */
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "
                    + initialCapacity);
        this.elementData = new Object [initialCapacity];
    }

    /**
     *  构造一个包含指定 collection 的元素的列表，这些元素是按照该 collection 的迭代器返回它们的顺序排列的。
     */
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }
```

2.3、新增

ArrayList 提供了 add(E e)、add(int index, E element)、addAll(Collection<? extends E> c)、addAll(int index, Collection<? extends E> c)、set(int index, E element) 这个五个方法来实现 ArrayList 增加。

add(E e)：将指定的元素添加到此列表的尾部。

```
public boolean add(E e) {
    ensureCapacity(size + 1);  // Increments  modCount!!
    elementData[size++] = e;
    return true;
    }
```

这里 ensureCapacity() 方法是对 ArrayList 集合进行扩容操作，elementData(size++) = e，将列表末尾元素指向e。

add(int index, E element)：将指定的元素插入此列表中的指定位置。

```
public void add(int index, E element) {
        //判断索引位置是否正确
```

```
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException(
            "Index: "+index+", Size: "+size);
        //扩容检测
        ensureCapacity(size+1);
        /*
         * 对源数组进行复制处理（位移），从index + 1到size-index。
         * 主要目的就是空出index位置供数据插入，
         * 即向右移动当前位于该位置的元素以及所有后续元素。
         */
        System.arraycopy(elementData, index,  elementData, index + 1,
                size - index);
        //在指定位置赋值
        elementData[index] = element;
        size++;
        }
```

在这个方法中最根本的方法就是 System.arraycopy() 方法，该方法的根本目的就是将 index 位置空出来以供新数据插入，这里需要进行数组数据的右移，这是非常麻烦和耗时的，所以如果指定的数据集合需要进行大量插入（中间插入）操作，推荐使用 LinkedList。

addAll(Collection<? extends E> c)：按照指定 collection 的迭代器所返回的元素顺序，将该 collection 中的所有元素添加到此列表的尾部。

```
public boolean addAll(Collection<? extends E> c) {
        // 将集合C转换成数组
        Object[] a = c.toArray();
        int numNew = a.length;
        // 扩容处理，大小为size + numNew
        ensureCapacity(size + numNew); // Increments modCount
        System.arraycopy(a, 0, elementData, size, numNew);
        size += numNew;
       return numNew != 0;
    }
```

这个方法无非就是使用 System.arraycopy() 方法将 C 集合(先准换为数组)里面的数据复制到 elementData 数组中。这里就稍微介绍下 System.arraycopy()，因为下面还将大量用到该方法。该方法的原型为：public static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)。它的根本目的就是进行数组元素的复制。即从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。将源数组 src从srcPos 位置开始复制到 dest 数组中，复制长度为 length，数据从 dest 的 destPos 位置开始粘贴。

addAll(int index, Collection<? extends E> c)：从指定的位置开始，将指定 collection 中的所有元素插入到此列表中。

```
public boolean addAll(int index, Collection<? extends E> c) {
        //判断位置是否正确
        if (index > size || index < 0)
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: "
                    + size);
        //转换成数组
        Object[] a = c.toArray();
        int numNew = a.length;
        //ArrayList容器扩容处理
        ensureCapacity(size + numNew); // Increments modCount
        //ArrayList容器数组向右移动的位置
        int numMoved = size - index;
        //如果移动位置大于0，则将ArrayList容器的数据向右移动numMoved个位置，确保增加的数据能够增加
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew,
                    numMoved);
        //添加数组
        System.arraycopy(a, 0, elementData, index,  numNew);
        //容器容量变大
        size += numNew;
        return numNew != 0;
    }
```

set(int index, E element)：用指定的元素替代此列表中指定位置上的元素。

```
public E set(int index, E element) {
        //检测插入的位置是否越界
        RangeCheck(index);

        E oldValue = (E) elementData[index];
        //替代
        elementData[index] = element;
        return oldValue;
    }
```

2.4、删除

ArrayList 提供了 remove(int index)、remove(Object o)、removeRange(int fromIndex, int toIndex)、removeAll() 四个方法进行元素的删除。

remove(int index)：移除此列表中指定位置上的元素。

```
public E remove(int index) {
        //位置验证
        RangeCheck(index);

       modCount++;
       //需要删除的元素
        E oldValue = (E) elementData[index];
        //向左移的位数
        int numMoved = size - index - 1;
        //若需要移动，则想左移动numMoved位
        if (numMoved > 0)
            System.arraycopy(elementData, index + 1, elementData, index,
                    numMoved);
        //置空最后一个元素
        elementData[--size] = null; // Let gc do its work

        return oldValue;
    }
```

remove(Object o)：移除此列表中首次出现的指定元素（如果存在）。

```
public boolean remove(Object o) {
        //因为ArrayList中允许存在null，所以需要进行null判断
        if (o == null) {
            for (int index = 0; index < size; index++)
                if (elementData[index] == null) {
                    //移除这个位置的元素
                    fastRemove(index);
                    return true;
                }
        } else {
            for (int index = 0; index < size; index++)
                if (o.equals(elementData[index])) {
                    fastRemove(index);
                    return true;
                }
        }
        return false;
    }
```

其中 fastRemove() 方法用于移除指定位置的元素。如下

```
private void fastRemove(int index) {
```

```
        modCount++;
        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // Let gc do its work
    }
```

removeRange(int fromIndex, int toIndex)：移除列表中索引在 fromIndex（包括）和 toIndex（不包括）之间的所有元素。

```
protected void removeRange(int fromIndex, int toIndex)   {
        modCount++;
        int numMoved = size - toIndex;
        System
                .arraycopy(elementData, toIndex, elementData, fromIndex,
                        numMoved);

        // Let gc do its work
        int newSize = size - (toIndex - fromIndex);
        while (size != newSize)
            elementData[--size] = null;
    }
```

removeAll()：是继承自 AbstractCollection 的方法，ArrayList 本身并没有提供实现。

```
public boolean removeAll(Collection<?> c) {
        boolean modified = false;
        Iterator<?> e = iterator();
        while (e.hasNext()) {
            if (c.contains(e.next())) {
                e.remove();
                modified = true;
            }
        }
        return modified;
    }
```

2.5、查找

ArrayList 提供了 get(int index) 用读取 ArrayList 中的元素。由于 ArrayList 是动态数组，所以我们完全可以根据下标来获取 ArrayList 中的元素，而且速度还比较快，故 ArrayList 长于随机访问。

```
public E get(int index) {
        RangeCheck(index);

        return (E) elementData[index];
    }
```

2.6、扩容

在上面的新增方法的源码中我们发现每个方法中都存在这个方法： ensureCapacity()，该方法就是 ArrayList 的扩容方法。在前面就提过 ArrayList 每次新增元素时都会需要进行容量检测判断，若新增元素后元素的个数会超过 ArrayList 的容量，就会进行扩容操作来满足新增元素的需求。所以当我们清楚知道业务数据量或者需要插入大量元素前，我可以使用 ensureCapacity 来手动增加 ArrayList 实例的容量，以减少递增式再分配的数量。

```
public void ensureCapacity(int minCapacity) {
        //修改计时器
        modCount++;
        //ArrayList容量大小
        int oldCapacity = elementData.length;
        /*
         * 若当前需要的长度大于当前数组的长度时，进行扩容操 作
         */
        if (minCapacity > oldCapacity) {
            Object oldData[] = elementData;
            //计算新的容量大小，为当前容量的1.5倍
            int newCapacity = (oldCapacity * 3) / 2 + 1;
            if (newCapacity < minCapacity)
            newCapacity = minCapacity;
            //数组拷贝，生成新的数组
            elementData = Arrays.copyOf(elementData, newCapacity);
        }
    }
```

在这里有一个疑问，为什么每次扩容处理会是 1.5 倍，而不是 2.5、3、4 倍呢? 通过 google 查找，发现 1.5 倍的扩容是最好的倍数。因为一次性扩容太大(例如 2.5 倍)可能会浪费更多的内存(1.5 倍最多浪费 33%，而 2.5 被最多会浪费 60%，3.5 倍则会浪费 71%……)。但是一次性扩容太小，需要多次对数组重新分配内存，对性能消耗比较严重。所以 1.5 倍刚刚好，既能满足性能需求，也不会造成很大的内存消耗。

处理这个 ensureCapacity() 这个扩容数组外，ArrayList 还给我们提供了将底层数组的容量调整为当前列表保存的实际元素的大小的功能。它可以通过 trimToSize() 方法来实现。该方法可以最小化 ArrayList 实例的存储量。

```
public void trimToSize() {
        modCount++;
        int oldCapacity = elementData.length;
        if (size < oldCapacity) {
            elementData = Arrays.copyOf(elementData, size);
        }
    }
```

LinkedList

一、概述

LinkedList 与 ArrayList 一样实现 List 接口，只是 ArrayList 是 List 接口的大小可变数组的实现，LinkedList 是 List 接口链表的实现。基于链表实现的方式使得 LinkedList 在插入和删除时更优于 ArrayList，而随机访问则比 ArrayList 逊色些。

LinkedList 实现所有可选的列表操作，并允许所有的元素包括 null。

除了实现 List 接口外，LinkedList 类还为在列表的开头及结尾 get、remove 和 insert 元素提供了统一的命名方法。这些操作允许将链接列表用作堆栈、队列或双端队列。

此类实现 Deque 接口，为 add、poll 提供先进先出队列操作，以及其他堆栈和双端队列操作。

所有操作都是按照双重链接列表的需要执行的。在列表中编索引的操作将从开头或结尾遍历列表（从靠近指定索引的一端）。

同时，与 ArrayList 一样此实现不是同步的。

（以上摘自JDK 6.0 API）。

二、源码分析

2.1、定义

首先我们先看 LinkedList 的定义：

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
```

从这段代码中我们可以清晰地看出 LinkedList 继承 AbstractSequentialList，实现 List、Deque、Cloneable、Serializable。其中 AbstractSequentialList 提供了 List 接口的骨干实现，从而最大限度地减少了实现受“连续访问”数据存储（如链接列表）支持的此接口所需的工作,从而以减少实现 List 接口的复杂度。Deque 一个线性 collection，支持在两端插入和移除元素，定义了双端队列的操作。

2.2、属性

在 LinkedList 中提供了两个基本属性 size、header。

```
private transient Entry<E> header = new Entry<E>(null, null, null);
private transient int size = 0;
```

其中 size 表示的 LinkedList 的大小，header 表示链表的表头，Entry 为节点对象。

```
private static class Entry<E> {
        E element;       //元素节点
        Entry<E> next;   //下一个元素
        Entry<E> previous;  //上一个元素

        Entry(E element, Entry<E> next, Entry<E> previous) {
            this.element = element;
            this.next = next;
            this.previous = previous;
        }
    }
```

上面为 Entry 对象的源代码，Entry 为 LinkedList 的内部类，它定义了存储的元素。该元素的前一个元素、后一个元素，这是典型的双向链表定义方式。

2.3、构造方法

LinkedList 提供了两个构造方法：LinkedList() 和 LinkedList(Collection<? extends E> c)。

```
/**
 * 构造一个空列表。
 */
    public LinkedList() {
        header.next = header.previous = header;
    }

/**
 * 构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列。
 */
    public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }
```

LinkedList() 构造一个空列表。里面没有任何元素，仅仅只是将 header 节点的前一个元素、后一个元素都指向自身。

LinkedList(Collection<? extends E> c)： 构造一个包含指定 collection 中的元素的列表，这些元素按其 collection 的迭代器返回的顺序排列。该构造函数首先会调用 LinkedList()，构造一个空列表，然后调用了 addAll() 方法将 Collection 中的所有元素添加到列表中。以下是 addAll() 的源代码：

```
/**
 * 添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序。
 */
    public boolean addAll(Collection<? extends E> c) {
        return addAll(size, c);
    }

/**
 * 将指定 collection 中的所有元素从指定位置开始插入此列表。其中index表示在其中插入指定collection中第一个元素的索引
 */
    public boolean addAll(int index, Collection<? extends E> c) {
        //若插入的位置小于0或者大于链表长度，则抛出IndexOutOfBoundsException异常
```

```
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);
        Object[] a = c.toArray();
        int numNew = a.length;  //插入元素的个数
        //若插入的元素为空，则返回false
        if (numNew == 0)
            return false;
        //modCount:在AbstractList中定义的，表示从结构上修改列表的次数
        modCount++;
        //获取插入位置的节点，若插入的位置在size处，则是头节点，否则获取index位置处的节点
        Entry<E> successor = (index == size ? header : entry(index));
        //插入位置的前一个节点，在插入过程中需要修改该节点的next引用：指向插入的节点元素
        Entry<E> predecessor = successor.previous;
        //执行插入动作
        for (int i = 0; i < numNew; i++) {
            //构造一个节点e，这里已经执行了插入节点动作同时修改了相邻节点的指向引用
            //
            Entry<E> e = new Entry<E>((E) a[i], successor, predecessor);
            //将插入位置前一个节点的下一个元素引用指向当前元素
            predecessor.next = e;
            //修改插入位置的前一个节点，这样做的目的是将插入位置右移一位，保证后续的元素是插在该元素的后面，确保这些元素的
            predecessor = e;
        }
        successor.previous = predecessor;
        //修改容量大小
        size += numNew;
        return true;
    }
```

在 addAll() 方法中，涉及到了两个方法，一个是 entry(int index)，该方法为 LinkedList 的私有方法，主要是用来查找 index 位置的节点元素。

```
/**
 * 返回指定位置(若存在)的节点元素
 */
    private Entry<E> entry(int index) {
        if (index < 0 || index >= size)
            throw new IndexOutOfBoundsException("Index: " + index + ", Size: "
                    + size);
        //头部节点
        Entry<E> e = header;
        //判断遍历的方向
        if (index < (size >> 1)) {
            for (int i = 0; i <= index; i++)
```

```
            e = e.next;
        } else {
            for (int i = size; i > index; i--)
                e = e.previous;
        }
        return e;
    }
```

从该方法有两个遍历方向中我们也可以看出 LinkedList 是双向链表，这也是在构造方法中为什么需要将 header 的前、后节点均指向自己。

如果对数据结构有点了解，对上面所涉及的内容应该问题，我们只需要清楚一点：LinkedList 是双向链表，其余都迎刃而解。

由于篇幅有限，下面将就 LinkedList 中几个常用的方法进行源码分析。

2.4、增加方法

add(E e): 将指定元素添加到此列表的结尾。

```
public boolean add(E e) {
    addBefore(e, header);
        return true;
    }
```

该方法调用 addBefore 方法，然后直接返回 true，对于 addBefore() 而已，它为 LinkedList 的私有方法。

```
private Entry<E> addBefore(E e, Entry<E> entry) {
        //利用Entry构造函数构建一个新节点 newEntry，
        Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
        //修改newEntry的前后节点的引用，确保其链表的引用关系是正确的
        newEntry.previous.next = newEntry;
        newEntry.next.previous = newEntry;
        //容量+1
        size++;
        //修改次数+1
        modCount++;
        return newEntry;
    }
```

在 addBefore 方法中无非就是做了这件事：构建一个新节点 newEntry，然后修改其前后的引用。

LinkedList 还提供了其他的增加方法：

add(int index, E element)：在此列表中指定的位置插入指定的元素。

addAll(Collection<? extends E> c)：添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序。

addAll(int index, Collection<? extends E> c)：将指定 collection 中的所有元素从指定位置开始插入此列表。

AddFirst(E e): 将指定元素插入此列表的开头。

addLast(E e): 将指定元素添加到此列表的结尾。

2.5、移除方法

remove(Object o)：从此列表中移除首次出现的指定元素（如果存在）。该方法的源代码如下：

```
public boolean remove(Object o) {
    if (o==null) {
        for (Entry<E> e = header.next; e != header; e = e.next) {
            if (e.element==null) {
                remove(e);
                return true;
            }
        }
    } else {
        for (Entry<E> e = header.next; e != header; e = e.next) {
            if (o.equals(e.element)) {
                remove(e);
                return true;
            }
        }
    }
    return false;
}
```

该方法首先会判断移除的元素是否为 null，然后迭代这个链表找到该元素节点，最后调用 remove(Entry e)，remove(Entry e) 为私有方法，是 LinkedList 中所有移除方法的基础方法，如下：

```
private E remove(Entry<E> e) {
    if (e == header)
```

```
            throw new NoSuchElementException();

        //保留被移除的元素：要返回
        E result = e.element;

        //将该节点的前一节点的next指向该节点后节点
        e.previous.next = e.next;
        //将该节点的后一节点的previous指向该节点的前节点
        //这两步就可以将该节点从链表从除去：在该链表中是无法遍历到该节点的
        e.next.previous = e.previous;
        //将该节点归空
        e.next = e.previous = null;
        e.element = null;
        size--;
        modCount++;
        return result;
    }
```

其他的移除方法：

clear()： 从此列表中移除所有元素。

remove()：获取并移除此列表的头（第一个元素）。

remove(int index)：移除此列表中指定位置处的元素。

remove(Objec o)：从此列表中移除首次出现的指定元素（如果存在）。

removeFirst()：移除并返回此列表的第一个元素。

removeFirstOccurrence(Object o)：从此列表中移除第一次出现的指定元素（从头部到尾部遍历列表时）。

removeLast()：移除并返回此列表的最后一个元素。

removeLastOccurrence(Object o)：从此列表中移除最后一次出现的指定元素（从头部到尾部遍历列表时）。

2.6、查找方法

对于查找方法的源码就没有什么好介绍了，无非就是迭代，比对，然后就是返回当前值。

get(int index)：返回此列表中指定位置处的元素。

getFirst()：返回此列表的第一个元素。

getLast()：返回此列表的最后一个元素。

indexOf(Object o)：返回此列表中首次出现的指定元素的索引，如果此列表中不包含该元素，则返回 −1。

lastIndexOf(Object o)：返回此列表中最后出现的指定元素的索引，如果此列表中不包含该元素，则返回 −1。

23

HashMap

HashMap 也是我们使用非常多的 Collection，它是基于哈希表的 Map 接口的实现，以 key-value 的形式存在。在 HashMap 中，key-value 总是会当做一个整体来处理，系统会根据 hash 算法来来计算 key-value 的存储位置，我们总是可以通过 key 快速地存、取 value。下面就来分析 HashMap 的存取。

一、定义

HashMap 实现了 Map 接口，继承 AbstractMap。其中 Map 接口定义了键映射到值的规则，而 AbstractMap 类提供 Map 接口的骨干实现，以最大限度地减少实现此接口所需的工作，其实 AbstractMap 类已经实现了Map，这里标注 Map LZ 觉得应该是更加清晰吧！

```
public class HashMap<K,V>
    extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
```

二、构造函数

HashMap 提供了三个构造函数：

HashMap()：构造一个具有默认初始容量 (16) 和默认加载因子 (0.75) 的空 HashMap。

HashMap(int initialCapacity)：构造一个带指定初始容量和默认加载因子 (0.75) 的空 HashMap。

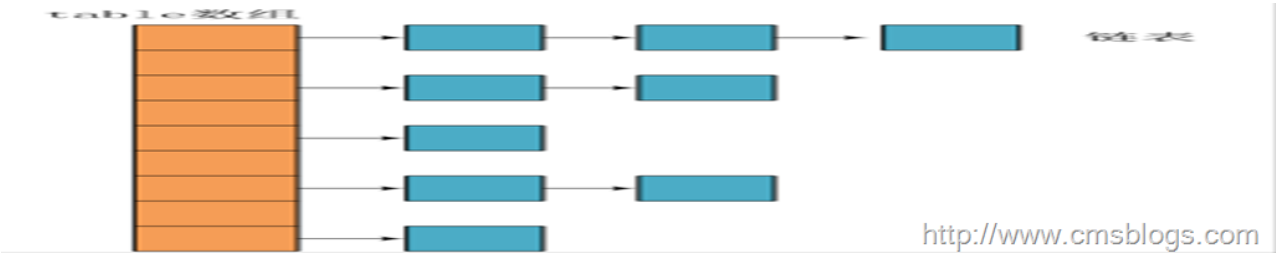
HashMap(int initialCapacity, float loadFactor)：构造一个带指定初始容量和加载因子的空 HashMap。

在这里提到了两个参数：初始容量，加载因子。这两个参数是影响 HashMap 性能的重要参数，其中容量表示哈希表中桶的数量，初始容量是创建哈希表时的容量，加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，它衡量的是一个散列表的空间的使用程度，负载因子越大表示散列表的装填程度越高，反之愈小。对于使用链表法的散列表来说，查找一个元素的平均时间是 O(1+a)，因此如果负载因子越大，对空间的利用更充分，然而后果是查找效率的降低；如果负载因子太小，那么散列表的数据将过于稀疏，对空间造成严重浪费。系统默认负载因子为 0.75，一般情况下我们是无需修改的。

HashMap 是一种支持快速存取的数据结构，要了解它的性能必须要了解它的数据结构。

三、数据结构

我们知道在 Java 中最常用的两种结构是数组和模拟指针(引用)，几乎所有的数据结构都可以利用这两种来组合实现，HashMap 也是如此。实际上 HashMap 是一个“链表散列”，如下是它数据结构：



图片 23.1 fig.1

从上图我们可以看出 HashMap 底层实现还是数组，只是数组的每一项都是一条链。其中参数 initialCapacity 就代表了该数组的长度。下面为 HashMap 构造函数的源码：

```
public HashMap(int initialCapacity, float loadFactor) {
        //初始容量不能<0
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: "
                    + initialCapacity);
        //初始容量不能 > 最大容量值，HashMap的最大容量值为2^30
        if (initialCapacity > MAXIMUM_CAPACITY)
            initialCapacity = MAXIMUM_CAPACITY;
        //负载因子不能 < 0
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: "
                    + loadFactor);

        // 计算出大于 initialCapacity 的最小的 2 的 n 次方值。
        int capacity = 1;
        while (capacity < initialCapacity)
            capacity <<= 1;

        this.loadFactor = loadFactor;
        //设置HashMap的容量极限，当HashMap的容量达到该极限时就会进行扩容操作
        threshold = (int) (capacity * loadFactor);
        //初始化table数组
        table = new Entry[capacity];
        init();
    }
```

从源码中可以看出，每次新建一个 HashMap 时，都会初始化一个 table 数组。table 数组的元素为 Entry 节点。

```
static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        Entry<K,V> next;
        final int hash;

        /**
         * Creates new entry.
         */
        Entry(int h, K k, V v, Entry<K,V> n) {
            value = v;
            next = n;
            key = k;
            hash = h;
        }
        .......
    }
```

其中 Entry 为 HashMap 的内部类，它包含了键 key、值 value、下一个节点 next，以及 hash 值，这是非常重要的，正是由于 Entry 才构成了 table 数组的项为链表。

上面简单分析了 HashMap 的数据结构，下面将探讨 HashMap 是如何实现快速存取的。

四、存储实现：put(key,vlaue)

首先我们先看源码

```
public V put(K key, V value) {
    //当key为null，调用putForNullKey方法，保存null与table第一个位置中，这是HashMap允许为null的原因
    if (key == null)
        return putForNullKey(value);
    //计算key的hash值
    int hash = hash(key.hashCode());              ------(1)
    //计算key hash 值在 table 数组中的位置
    int i = indexFor(hash, table.length);         ------(2)
    //从i出开始迭代 e,找到 key 保存的位置
    for (Entry<K, V> e = table[i]; e != null; e = e.next) {
        Object k;
        //判断该条链上是否有hash值相同的(key相同)
        //若存在相同，则直接覆盖value，返回旧value
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;   //旧值 = 新值
            e.value = value;
            e.recordAccess(this);
            return oldValue;     //返回旧值
        }
    }
    //修改次数增加1
    modCount++;
    //将key、value添加至i位置处
    addEntry(hash, key, value, i);
    return null;
}
```

通过源码我们可以清晰看到 HashMap 保存数据的过程为：首先判断 key 是否为 null，若为 null，则直接调用 putForNullKey 方法。若不为空则先计算 key 的 hash 值，然后根据 hash 值搜索在 table 数组中的索引位置，如果 table 数组在该位置处有元素，则通过比较是否存在相同的 key，若存在则覆盖原来 key 的 value，否则将该元素保存在链头（最先保存的元素放在链尾）。若 table 在该处没有元素，则直接保存。这个过程看似比较简单，其实深有内幕。有如下几点：

1、 先看迭代处。此处迭代原因就是为了防止存在相同的 key 值，若发现两个 hash 值（key）相同时，HashMap 的处理方式是用新 value 替换旧 value，这里并没有处理 key，这就解释了 HashMap 中没有两个相同的 key。

2、 在看（1）、（2）处。这里是 HashMap 的精华所在。首先是 hash 方法，该方法为一个纯粹的数学计算，就是计算 h 的 hash 值。

```
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
  }
```

我们知道对于 HashMap 的 table 而言，数据分布需要均匀（最好每项都只有一个元素，这样就可以直接找到），不能太紧也不能太松，太紧会导致查询速度慢，太松则浪费空间。计算 hash 值后，怎么才能保证 table 元素分布均与呢？我们会想到取模，但是由于取模的消耗较大，HashMap 是这样处理的：调用 indexFor 方法。

```
static int indexFor(int h, int length) {
    return h & (length-1);
  }
```

HashMap 的底层数组长度总是 2 的 n 次方，在构造函数中存在：capacity <<= 1;这样做总是能够保证 HashMap 的底层数组长度为 2 的 n 次方。当 length 为 2 的 n 次方时，h&(length - 1) 就相当于对 length 取模，而且速度比直接取模快得多，这是 HashMap 在速度上的一个优化。至于为什么是 2 的 n 次方下面解释。

我们回到 indexFor 方法，该方法仅有一条语句：h&(length - 1)，这句话除了上面的取模运算外还有一个非常重要的责任：均匀分布 table 数据和充分利用空间。

这里我们假设 length 为 16(2^n) 和 15，h 为 5、6、7。

length = 16			
h	length-1	h&length-1	
5	15	0101 & 1111 = 00101	5
6	15	0110 & 1111 = 00110	6
7	15	0111 & 1111 = 00111	7
length = 15			
5	14	0101 & 1110 = 00101	5
6	14	0110 & 1110 = 00110	6
7	14	0111 & 1110 = 00110	6

图片 23.2 fig.2

当 n=15 时，6 和 7 的结果一样，这样表示他们在 table 存储的位置是相同的，也就是产生了碰撞，6、7 就会在一个位置形成链表，这样就会导致查询速度降低。诚然这里只分析三个数字不是很多，那么我们就看 0-15。

h	length-1	h&length-1	
0	14	0000 & 1110 = 0000	0
1	14	0001 & 1110 = 0000	0
2	14	0010 & 1110 = 0010	2
3	14	0011 & 1110 = 0010	2
4	14	0100 & 1110 = 0100	4
5	14	0101 & 1110 = 0100	4
6	14	0110 & 1110 = 0110	6
7	14	0111 & 1110 = 0110	6
8	14	1000 & 1110 = 1000	8
9	14	1001 & 1110 = 1000	8
10	14	1010 & 1110 = 1010	10
11	14	1011 & 1110 = 1010	10
12	14	1100 & 1110 = 1100	12
13	14	1101 & 1110 = 1100	12
14	14	1110 & 1110 = 1110	14
15	14	1111 & 1110 = 1110	14

图片 23.3 fig.3

从上面的图表中我们看到总共发生了 8 此碰撞，同时发现浪费的空间非常大，有 1、3、5、7、9、11、13、15 处没有记录，也就是没有存放数据。这是因为他们在与 14 进行 & 运算时，得到的结果最后一位永远都是 0，即 0001、0011、0101、0111、1001、1011、1101、1111 位置处是不可能存储数据的，空间减少，进一步增加碰撞几率，这样就会导致查询速度慢。而当 length = 16 时，length － 1 = 15 即 1111，那么进行低位 & 运算时，值总是与原来 hash 值相同，而进行高位运算时，其值等于其低位值。所以说当 length = 2^n 时，不同的 hash 值发生碰撞的概率比较小，这样就会使得数据在 table 数组中分布较均匀，查询速度也较快。

这里我们再来复习 put 的流程：当我们想一个 HashMap 中添加一对 key-value 时，系统首先会计算 key 的 hash 值，然后根据 hash 值确认在 table 中存储的位置。若该位置没有元素，则直接插入。否则迭代该处元素链表并依此比较其 key 的 hash 值。如果两个 hash 值相等且 key 值相等 (e.hash == hash && ((k = e.key) == key || key.equals(k))),则用新的 Entry 的 value 覆盖原来节点的 value。如果两个 hash 值相等但 key 值不等，则将该节点插入该链表的链头。具体的实现过程见 addEntry 方法，如下：

```
void addEntry(int hash, K key, V value, int bucketIndex) {
      //获取bucketIndex处的Entry
      Entry<K, V> e = table[bucketIndex];
      //将新创建的 Entry 放入 bucketIndex 索引处，并让新的 Entry 指向原来的 Entry
      table[bucketIndex] = new Entry<K, V>(hash, key, value, e);
      //若HashMap中元素的个数超过极限了，则容量扩大两倍
      if (size++ >= threshold)
         resize(2 * table.length);
   }
```

这个方法中有两点需要注意：

一、链的产生

这是一个非常优雅的设计。系统总是将新的 Entry 对象添加到 bucketIndex 处。如果 bucketIndex 处已经有了对象，那么新添加的 Entry 对象将指向原有的 Entry 对象，形成一条 Entry 链，但是若 bucketIndex 处没有 Entry 对象，也就是 e==null,那么新添加的 Entry 对象指向 null，也就不会产生 Entry 链了。

二、扩容问题。

随着 HashMap 中元素的数量越来越多，发生碰撞的概率就越来越大，所产生的链表长度就会越来越长，这样势必会影响 HashMap 的速度，为了保证 HashMap 的效率，系统必须要在某个临界点进行扩容处理。该临界点在当 HashMap 中元素的数量等于 table 数组长度 * 加载因子。但是扩容是一个非常耗时的过程，因为它需要重新计算这些数据在新 table 数组中的位置并进行复制处理。所以如果我们已经预知 HashMap 中元素的个数，那么预设元素的个数能够有效的提高 HashMap 的性能。

五、读取实现：get(key)

相对于 HashMap 的存而言，取就显得比较简单了。通过 key 的 hash 值找到在 table 数组中的索引处的 Entry，然后返回该 key 对应的 value 即可。

```
public V get(Object key) {
        // 若为null，调用getForNullKey方法返回相对应的value
        if (key == null)
            return getForNullKey();
        // 根据该 key 的 hashCode 值计算它的 hash 码
        int hash = hash(key.hashCode());
        // 取出 table 数组中指定索引处的值
        for (Entry<K, V> e = table[indexFor(hash, table.length)]; e != null; e = e.next) {
            Object k;
            //若搜索的key与查找的key相同，则返回相对应的value
            if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
                return e.value;
        }
        return null;
    }
```

在这里能够根据 key 快速的取到 value 除了和 HashMap 的数据结构密不可分外，还和 Entry 有莫大的关系，在前面就提到过，HashMap 在存储过程中并没有将 key，value 分开来存储，而是当做一个整体 key-value 来处理的，这个整体就是 Entry 对象。同时 value 也只相当于 key 的附属而已。在存储的过程中，系统根据 key 的 hashcode 来决定 Entry 在 table 数组中的存储位置，在取的过程中同样根据 key 的 hashcode 取出相对应的 Entry 对象。

24

HashSet

在前篇博文中（Java 提高篇（二三）—- HashMap）详细讲解了 HashMap 的实现过程，对于 HashSet 而言，它是基于 HashMap 来实现的，底层采用 HashMap 来保存元素。所以如果对 HashMap 比较熟悉，那么 HashSet是 so easy!!

一、定义

```
public class HashSet<E>
    extends AbstractSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

HashSet 继承 AbstractSet 类，实现 Set、Cloneable、Serializable 接口。其中 AbstractSet 提供 Set 接口的骨干实现，从而最大限度地减少了实现此接口所需的工作。Set 接口是一种不包括重复元素的 Collection，它维持它自己的内部排序，所以随机访问没有任何意义。

基本属性

```
//基于HashMap实现，底层使用HashMap保存所有元素
        private transient HashMap<E,Object> map;

        //定义一个Object对象作为HashMap的value
        private static final Object PRESENT = new Object();
```

构造函数

```
/**
         * 默认构造函数
         * 初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。
         */
        public HashSet() {
            map = new HashMap<>();
        }

        /**
         * 构造一个包含指定 collection 中的元素的新 set。
         */
        public HashSet(Collection<? extends E> c) {
            map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
            addAll(c);
        }

        /**
         * 构造一个新的空 set，其底层 HashMap 实例具有指定的初始容量和指定的加载因子
         */
        public HashSet(int initialCapacity, float  loadFactor) {
```

```
        map = new HashMap<>(initialCapacity, loadFactor);
    }

    /**
     * 构造一个新的空 set，其底层 HashMap 实例具有指定的初始容量和默认的加载因子（0.75）。
     */
     public HashSet(int initialCapacity) {
       map = new HashMap<>(initialCapacity);
    }

    /**
     * 在API中我没有看到这个构造函数，今天看源码才发现（原来访问权限为包权限，不对外公开的）
     * 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。
     * dummy 为标识 该构造函数主要作用是对LinkedHashSet起到一个支持作用
     */
    HashSet(int initialCapacity, float loadFactor, boolean dummy) {
       map = new LinkedHashMap<>(initialCapacity, loadFactor);
    }
```

从构造函数中可以看出 HashSet 所有的构造都是构造出一个新的 HashMap，其中最后一个构造函数，为包访问权限是不对外公开，仅仅只在使用 LinkedHashSet 时才会发生作用。

二、方法

既然 HashSet 是基于 HashMap，那么对于 HashSet 而言，其方法的实现过程是非常简单的。

```
public Iterator<E> iterator() {
        return map.keySet().iterator();
    }
```

iterator() 方法返回对此 set 中元素进行迭代的迭代器。返回元素的顺序并不是特定的。底层调用 HashMap 的 keySet 返回所有的 key，这点反应了 HashSet 中的所有元素都是保存在 HashMap 的 key 中，value 则是使用的 PRESENT 对象，该对象为 static final。

```
public int size() {
        return map.size();
    }
```

size() 返回此 set 中的元素的数量（ set 的容量 ）。底层调用 HashMap的 size 方法，返回 HashMap 容器的大小。

```
public boolean isEmpty() {
        return map.isEmpty();
    }
```

isEmpty()，判断 HashSet() 集合是否为空，为空返回 true，否则返回false。

```
public boolean contains(Object o) {
        return map.containsKey(o);
    }
```

contains()，判断某个元素是否存在于 HashSet() 中，存在返回 true，否则返回 false。更加确切的讲应该是要满足这种关系才能返回true：(o==null ? e==null : o.equals(e))。底层调用 containsKey 判断 HashMap 的 key 值是否为空。

```
public boolean add(E e) {
        return map.put(e, PRESENT)==null;
    }
```

add() 如果此 set 中尚未包含指定元素，则添加指定元素。如果此 Set 没有包含满足 (e==null ? e2==null : e.equals(e2)) 的 e2 时，则将 e2 添加到 Set 中，否则不添加且返回 false。由于底层使用 HashMap 的 put 方法将 key = e，value=PRESENT 构建成 key-value 键值对，当此 e 存在于 HashMap 的 key 中，则 value 将会覆盖原有 value，但是 key 保持不变，所以如果将一个已经存在的 e 元素添加中 HashSet 中，新添加的元素是不会保存到 HashMap 中，所以这就满足了 HashSet 中元素不会重复的特性。

```
public boolean remove(Object o) {
        return map.remove(o)==PRESENT;
    }
```

remove 如果指定元素存在于此 set 中，则将其移除。底层使用 HashMap 的 remove 方法删除指定的 Entry。

```
public void clear() {
        map.clear();
    }
```

clear 从此 set 中移除所有元素。底层调用 HashMap 的 clear 方法清除所有的 Entry。

```
public Object clone() {
        try {
            HashSet<E> newSet = (HashSet<E>) super.clone();
            newSet.map = (HashMap<E, Object>) map.clone();
            return newSet;
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }
    }
```

clone 返回此 HashSet 实例的浅表副本：并没有复制这些元素本身。

后记：

由于 HashSet 底层使用了 HashMap 实现，使其的实现过程变得非常简单，如果你对 HashMap 比较了解，那么 HashSet 简直是小菜一碟。有两个方法对 HashMap 和 HashSet 而言是非常重要的，下篇将详细讲解 hashcode 和 equals。

25

HashTable

在 Java 中与有两个类都提供了一个多种用途的 hashTable 机制，他们都可以将可以 key 和 value 结合起来构成键值对通过 put(key,value)方法保存起来，然后通过 get(key) 方法获取相对应的 value 值。一个是前面提到的 HashMap，还有一个就是马上要讲解的 HashTable。对于 HashTable 而言，它在很大程度上和 HashMap 的实现差不多，如果我们对 HashMap 比较了解的话，对 HashTable 的认知会提高很大的帮助。他们两者之间只存在几点的不同，这个后面会阐述。

一、定义

HashTable 在 Java 中的定义如下：

```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable
```

从中可以看出 HashTable 继承 Dictionary 类，实现 Map 接口。其中 Dictionary 类是任何可将键映射到相应值的类（如 `Hashtable` ）的抽象父类。每个键和每个值都是一个对象。在任何一个 Dictionary 对象中，每个键至多与一个值相关联。Map 是”key-value键值对”接口。

HashTable 采用”拉链法”实现哈希表，它定义了几个重要的参数：table、count、threshold、loadFactor、modCount。

table：为一个 Entry[] 数组类型，Entry 代表了“拉链”的节点，每一个 Entry代表了一个键值对，哈希表的”key-value键值对”都是存储在 Entry 数组中的。

count：HashTable 的大小，注意这个大小并不是 HashTable 的容器大小，而是他所包含 Entry 键值对的数量。

threshold：Hashtable 的阈值，用于判断是否需要调整 Hashtable 的容量。threshold 的值=”容量*加载因子”。

loadFactor：加载因子。

modCount：用来实现“fail-fast”机制的（也就是快速失败）。所谓快速失败就是在并发集合中，其进行迭代操作时，若有其他线程对其进行结构性的修改，这时迭代器会立马感知到，并且立即抛出ConcurrentModificationException 异常，而不是等到迭代完成之后才告诉你（你已经出错了）。

二、构造方法

在 HashTabel 中存在 5 个构造函数。通过这 5 个构造函数我们构建出一个我想要的 HashTable。

```
public Hashtable() {
        this(11, 0.75f);
    }
```

默认构造函数，容量为 11，加载因子为 0.75。

```
public Hashtable(int initialCapacity) {
        this(initialCapacity, 0.75f);
    }
```

用指定初始容量和默认的加载因子 (0.75) 构造一个新的空哈希表。

```
public Hashtable(int initialCapacity, float loadFactor) {
        //验证初始容量
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+
                                               initialCapacity);
        //验证加载因子
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal Load: "+loadFactor);

        if (initialCapacity==0)
            initialCapacity = 1;

        this.loadFactor = loadFactor;

        //初始化table，获得大小为initialCapacity的table数组
        table = new Entry[initialCapacity];
        //计算阀值
        threshold = (int)Math.min(initialCapacity * loadFactor, MAX_ARRAY_SIZE + 1);
        //初始化HashSeed值
        initHashSeedAsNeeded(initialCapacity);
    }
```

用指定初始容量和指定加载因子构造一个新的空哈希表。其中 initHashSeedAsNeeded 方法用于初始化 hashSeed 参数，其中 hashSeed 用于计算 key 的 hash 值，它与 key 的 hashCode 进行按位异或运算。这个 hashSeed 是一个与实例相关的随机值，主要用于解决 hash 冲突。

```
private int hash(Object k) {
        return hashSeed ^ k.hashCode();
    }
```

构造一个与给定的 Map 具有相同映射关系的新哈希表。

```
public Hashtable(Map<? extends K, ? extends V> t) {
        //设置table容器大小，其值==t.size * 2 + 1
        this(Math.max(2*t.size(), 11), 0.75f);
        putAll(t);
    }
```

三、主要方法

HashTable 的 API 对外提供了许多方法，这些方法能够很好帮助我们操作 HashTable，但是这里我只介绍两个最根本的方法：put、get。

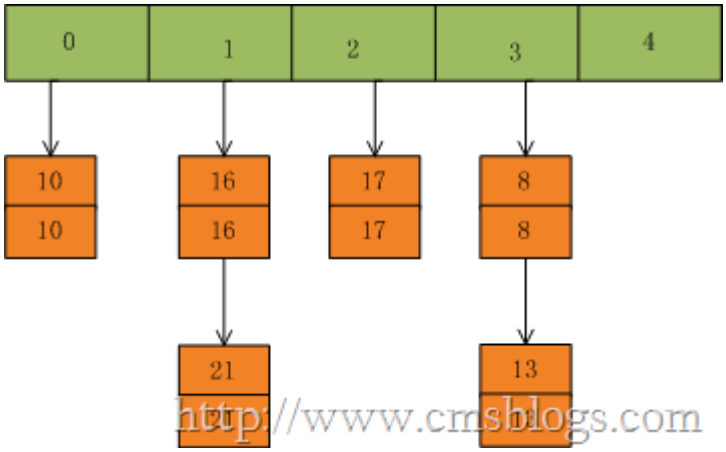
首先我们先看 put 方法：将指定 key 映射到此哈希表中的指定 value。注意这里键 key 和值 value 都不可为空。

```
public synchronized V put(K key, V value) {
        // 确保value不为null
        if (value == null) {
            throw new NullPointerException();
        }

        /*
         * 确保key在table[]是不重复的
         * 处理过程：
         * 1、计算key的hash值，确认在table[]中的索引位置
         * 2、迭代index索引位置，如果该位置处的链表中存在一个一样的key，则替换其value，返回旧值
         */
        Entry tab[] = table;
        int hash = hash(key);    //计算key的hash值
        int index = (hash & 0x7FFFFFFF) % tab.length;    //确认该key的索引位置
        //迭代，寻找该key，替换
        for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                V old = e.value;
                e.value = value;
                return old;
            }
        }

        modCount++;
        if (count >= threshold) {  //如果容器中的元素数量已经达到阀值，则进行扩容操作
            rehash();
            tab = table;
            hash = hash(key);
            index = (hash & 0x7FFFFFFF) % tab.length;
        }

        // 在索引位置处插入一个新的节点
```

```
        Entry<K,V> e = tab[index];
        tab[index] = new Entry<>(hash, key, value, e);
        //容器中元素+1
        count++;
        return null;
    }
```

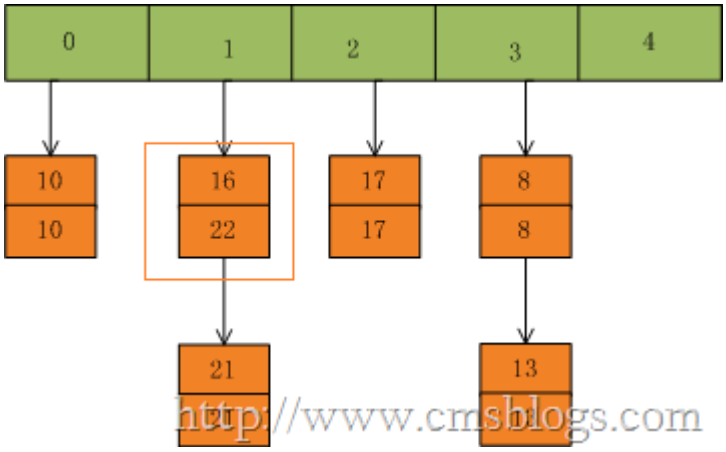
put 方法的整个处理流程是：计算 key 的 hash 值，根据 hash 值获得 key 在 table 数组中的索引位置，然后迭代该 key 处的 Entry 链表（我们暂且理解为链表），若该链表中存在一个这个的 key 对象，那么就直接替换其 value 值即可，否则在将改 key-value 节点插入该 index 索引位置处。如下：

首先我们假设一个容量为 5 的 table，存在 8、10、13、16、17、21。他们在 table 中位置如下：



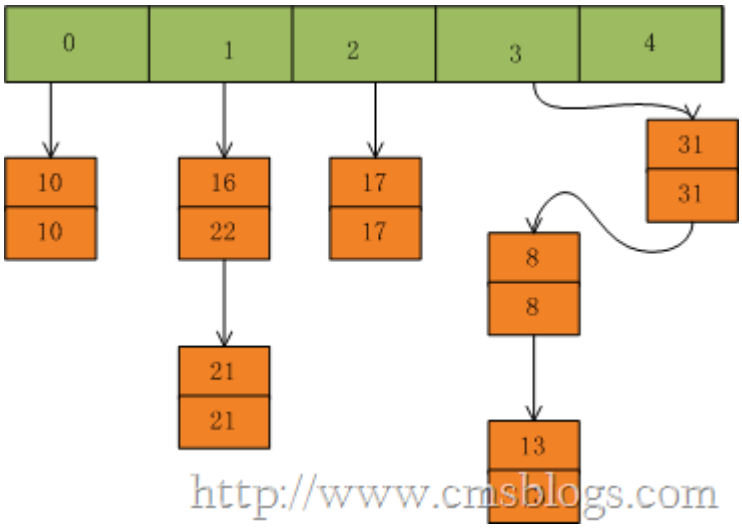
图片 25.1 fig.1

然后我们插入一个数：put(16,22)，key=16 在 table 的索引位置为 1，同时在 1 索引位置有两个数，程序对该“链表”进行迭代，发现存在一个 key=16,这时要做的工作就是用 newValue=22 替换 oldValue16，并将 oldValue=16 返回。



图片 25.2 fig.2

在 put(33,33)，key=33 所在的索引位置为 3，并且在该链表中也没有存在某个 key=33 的节点，所以就将该节点插入该链表的第一个位置。



图片 25.3 fig.3

在 HashTabled 的 put 方法中有两个地方需要注意：

1、HashTable 的扩容操作，在 put 方法中，如果需要向 table[] 中添加 Entry 元素，会首先进行容量校验，如果容量已经达到了阀值， HashTable 就会进行扩容处理 rehash()，如下：

```
protected void rehash() {
        int oldCapacity = table.length;
        //元素
        Entry<K,V>[] oldMap = table;

        //新容量=旧容量 * 2 + 1
        int newCapacity = (oldCapacity << 1) + 1;
        if (newCapacity - MAX_ARRAY_SIZE > 0) {
            if (oldCapacity == MAX_ARRAY_SIZE)
                return;
            newCapacity = MAX_ARRAY_SIZE;
        }

        //新建一个size = newCapacity 的HashTable
        Entry<K,V>[] newMap = new Entry[];

        modCount++;
        //重新计算阀值
        threshold = (int)Math.min(newCapacity *  loadFactor, MAX_ARRAY_SIZE + 1);
        //重新计算hashSeed
        boolean rehash = initHashSeedAsNeeded(newCapacity);
```

```
        table = newMap;
        //将原来的元素拷贝到新的HashTable中
        for (int i = oldCapacity ; i-- > 0 ;) {
            for (Entry<K,V> old = oldMap[i] ; old != null ; ) {
                Entry<K,V> e = old;
                old = old.next;

                if (rehash) {
                    e.hash = hash(e.key);
                }
                int index = (e.hash & 0x7FFFFFFF) % newCapacity;
                e.next = newMap[index];
                newMap[index] = e;
            }
        }
    }
```

在这个 rehash() 方法中我们可以看到容量扩大两倍 +1，同时需要将原来 HashTable 中的元素一一复制到新的 HashTable 中，这个过程是比较消耗时间的，同时还需要重新计算 hashSeed 的，毕竟容量已经变了。这里对阀值啰嗦一下：比如初始值 11、加载因子默认 0.75，那么这个时候阀值 threshold=8，当容器中的元素达到 8 时，HashTable 进行一次扩容操作，容量 = 8 * 2 + 1 =17，而阀值 threshold=17*0.75 = 13，当容器元素再一次达到阀值时，HashTable 还会进行扩容操作，一次类推。

2、其实这里是我的一个疑问，在计算索引位置 index 时， HashTable 进行了一个与运算过程（ hash & 0x7FFFFFFF ），为什么需要做一步操作，这么做有什么好处？如果哪位知道，望指导，LZ 不胜感激！！ 下面是计算 key 的 hash 值，这里 hashSeed 发挥了作用。

```
    private int hash(Object k) {
        return hashSeed ^ k.hashCode();
    }
```

相对于 put 方法，get 方法就会比较简单，处理过程就是计算 key 的 hash 值，判断在 table 数组中的索引位置，然后迭代链表，匹配直到找到相对应 key 的 value,若没有找到返回 null。

```
    public synchronized V get(Object key) {
        Entry tab[] = table;
        int hash = hash(key);
        int index = (hash & 0x7FFFFFFF) % tab.length;
        for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                return e.value;
```

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

四、HashTable 与 HashMap 的区别

HashTable 和 HashMap 存在很多的相同点，但是他们还是有几个比较重要的不同点。

第一：我们从他们的定义就可以看出他们的不同，HashTable 基于 Dictionary 类，而 HashMap 是基于 AbstractMap。Dictionary 是什么？它是任何可将键映射到相应值的类的抽象父类，而 AbstractMap 是基于 Map 接口的骨干实现，它以最大限度地减少实现此接口所需的工作。

第二：HashMap 可以允许存在一个为 null 的 key 和任意个为 null 的 value，但是 HashTable 中的 key 和 value 都不允许为 null。如下：

当 HashMap 遇到为 null 的 key 时，它会调用 putForNullKey 方法来进行处理。对于 value 没有进行任何处理，只要是对象都可以。

```
if (key == null)
        return putForNullKey(value);
```

而当 HashTable 遇到 null 时，他会直接抛出 NullPointerException异常信息。

```
if (value == null) {
        throw new NullPointerException();
    }
```

第三：Hashtable 的方法是同步的，而 HashMap 的方法不是。所以有人一般都建议如果是涉及到多线程同步时采用 HashTable，没有涉及就采用 HashMap，但是在 Collections 类中存在一个静态方法： synchronizedMap()，该方法创建了一个线程安全的 Map 对象，并把它作为一个封装的对象来返回，所以通过 Collections 类的 synchronizedMap 方法是可以我们你同步访问潜在的 HashMap。这样君该如何选择呢？？？

hashCode

在前面三篇博文中 LZ 讲解了（HashMap、HashSet、HashTable），在其中 LZ 不断地讲解他们的 put 和 get 方法，在这两个方法中计算 key 的 hashCode 应该是最重要也是最精华的部分，所以下面 LZ 揭开 hashCode 的“神秘”面纱。

hashCode 的作用

要想了解一个方法的内在原理，我们首先需要明白它是干什么的，也就是这个方法的作用。在讲解数组时（java提高篇（十八）——数组），我们提到数组是java中效率最高的数据结构，但是“最高”是有前提的。第一我们需要知道所查询数据的所在位置。第二：如果我们进行迭代查找时，数据量一定要小，对于大数据量而言一般推荐集合。

在 Java 集合中有两类，一类是 List，一类是 Set 他们之间的区别就在于 List 集合中的元素师有序的，且可以重复，而 Set 集合中元素是无序不可重复的。对于 List 好处理，但是对于 Set 而言我们要如何来保证元素不重复呢？通过迭代来 equals() 是否相等。数据量小还可以接受，当我们的数据量大的时候效率可想而知（当然我们可以利用算法进行优化）。比如我们向 HashSet 插入 1000 数据，难道我们真的要迭代 1000 次，调用 1000 次 equals() 方法吗？hashCode 提供了解决方案。怎么实现？我们先看 hashCode 的源码(Object)。

```
public native int hashCode();
```

它是一个本地方法，它的实现与本地机器有关，这里我们暂且认为他返回的是对象存储的物理位置（实际上不是，这里写是便于理解）。当我们向一个集合中添加某个元素，集合会首先调用 hashCode 方法，这样就可以直接定位它所存储的位置，若该处没有其他元素，则直接保存。若该处已经有元素存在，就调用 equals 方法来匹配这两个元素是否相同，相同则不存，不同则散列到其他位置（具体情况请参考（Java 提高篇（）—–HashMap））。这样处理，当我们存入大量元素时就可以大大减少调用 equals() 方法的次数，极大地提高了效率。

所以 **hashCode 在上面扮演的角色为寻域**（寻找某个对象在集合中区域位置）。hashCode 可以将集合分成若干个区域，每个对象都可以计算出他们的 hash 码，可以将 hash 码分组，每个分组对应着某个存储区域，根据一个对象的 hash 码就可以确定该对象所存储区域，这样就大大减少查询匹配元素的数量，提高了查询效率。

hashCode 对于一个对象的重要性

hashCode 重要么？不重要，对于 List 集合、数组而言，他就是一个累赘，但是对于 HashMap、HashSet、HashTable 而言，它变得异常重要。所以在使用 HashMap、HashSet、HashTable 时一定要注意 hashCode。对于一个对象而言，其 hashCode 过程就是一个简单的 Hash 算法的实现，其实现过程对你实现对象的存取过程起到非常重要的作用。

在前面 LZ 提到了 HashMap 和 HashTable 两种数据结构，虽然他们存在若干个区别，但是他们的实现原理是相同的，这里我以 HashTable 为例阐述 hashCode 对于一个对象的重要性。

一个对象势必会存在若干个属性，如何选择属性来进行散列考验着一个人的设计能力。如果我们将所有属性进行散列，这必定会是一个糟糕的设计，因为对象的 hashCode 方法无时无刻不是在被调用，如果太多的属性参与散列，那么需要的操作数时间将会大大增加，这将严重影响程序的性能。但是如果较少属相参与散列，散列的多样性会削弱，会产生大量的散列“冲突”，除了不能够很好的利用空间外，在某种程度也会影响对象的查询效率。其实这两者是一个矛盾体，散列的多样性会带来性能的降低。

那么如何对对象的 hashCode 进行设计，LZ 也没有经验。从网上查到了这样一种解决方案：设置一个缓存标识来缓存当前的散列码，只有当参与散列的对象改变时才会重新计算，否则调用缓存的 hashCode，这样就可以从很大程度上提高性能。

在 HashTable 计算某个对象在 table[] 数组中的索引位置，其代码如下：

```
int index = (hash & 0x7FFFFFFF) % tab.length;
```

为什么要 &0x7FFFFFFF？因为某些对象的 hashCode 可能会为负值，与 0x7FFFFFFF 进行与运算可以确保 index 为一个正数。通过这步我可以直接定位某个对象的位置，所以从理论上来说我们是完全可以利用 hashCode 直接定位对象的散列表中的位置，但是为什么会存在一个 key-value 的键值对，利用 key 的 hashCode 来存入数据而不是直接存放 value 呢？这就关系 HashTable 性能问题的最重要的问题:Hash 冲突！

我们知道冲突的产生是由于不同的对象产生了相同的散列码，假如我们设计对象的散列码可以确保 99.999999999% 的不重复，但是有一种绝对且几乎不可能遇到的冲突你是绝对避免不了的。我们知道 hashcode 返回的是 int，它的值只可能在 int 范围内。如果我们存放的数据超过了 int 的范围呢？这样就必定会产生两个相同的 index，这时在 index 位置处会存储两个对象，我们就可以利用 key 本身来进行判断。所以具有相索引的对象，在该 index 位置处存在多个对象，我们必须依靠 key 的 hashCode和key 本身来进行区分。

hashCode 与 equals

在 Java 中 hashCode 的实现总是伴随着 equals，他们是紧密配合的，你要是自己设计了其中一个，就要设计另外一个。当然在多数情况下，这两个方法是不用我们考虑的，直接使用默认方法就可以帮助我们解决很多问题。但是在有些情况，我们必须要自己动手来实现它，才能确保程序更好的运作。

对于 equals，我们必须遵循如下规则：

对称性：如果 x.equals(y) 返回是“true”，那么 y.equals(x) 也应该返回是“true”。

反射性：x.equals(x) 必须返回是“true”。

类推性：如果 x.equals(y) 返回是“true”，而且 y.equals(z) 返回是“true”，那么 z.equals(x) 也应该返回是“true”。

一致性：如果 x.equals(y) 返回是“true”，只要x和y内容一直不变，不管你重复 x.equals(y) 多少次，返回都是“true”。

任何情况下，x.equals(null)，永远返回是“false”；x.equals(和x不同类型的对象)永远返回是“false”。

对于 hashCode，我们应该遵循如下规则：

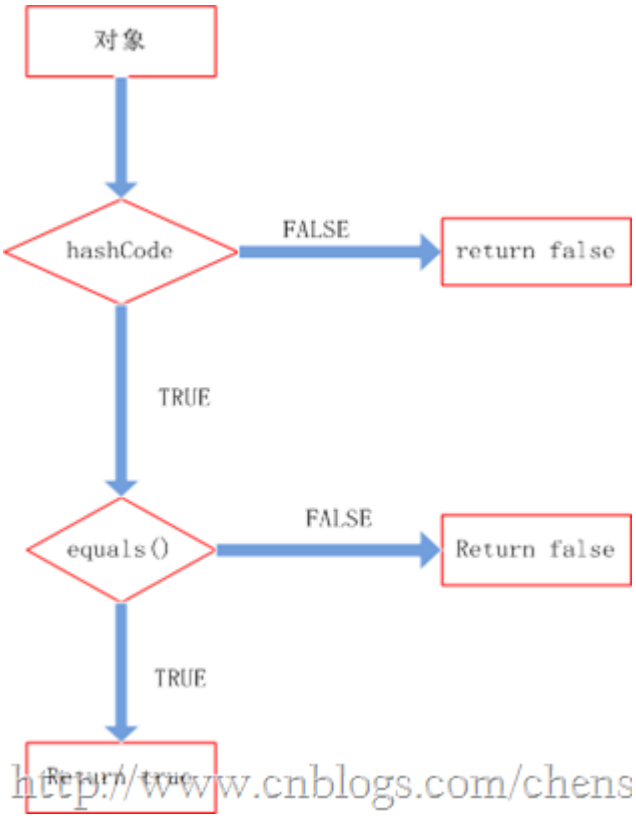
1. 在一个应用程序执行期间，如果一个对象的 equals 方法做比较所用到的信息没有被修改的话，则对该对象调用 hashCode 方法多次，它必须始终如一地返回同一个整数。
2. 如果两个对象根据 equals(Object o) 方法是相等的，则调用这两个对象中任一对象的 hashCode 方法必须产生相同的整数结果。
3. 如果两个对象根据 equals(Object o) 方法是不相等的，则调用这两个对象中任一个对象的 hashCode 方法，不要求产生不同的整数结果。但如果能不同，则可能提高散列表的性能。

至于两者之间的关联关系，我们只需要记住如下即可：

如果 x.equals(y) 返回“true”，那么 x 和 y 的 hashCode() 必须相等。

如果 x.equals(y) 返回“false”，那么 x 和 y 的 hashCode() 有可能相等，也有可能不等。

理清了上面的关系我们就知道他们两者是如何配合起来工作的。先看下图：



图片 26.1 fig.1

整个处理流程是：

1、判断两个对象的 hashcode 是否相等，若不等，则认为两个对象不等，完毕，若相等，则比较 equals。

2、若两个对象的 equals 不等，则可以认为两个对象不等，否则认为他们相等。

实例：

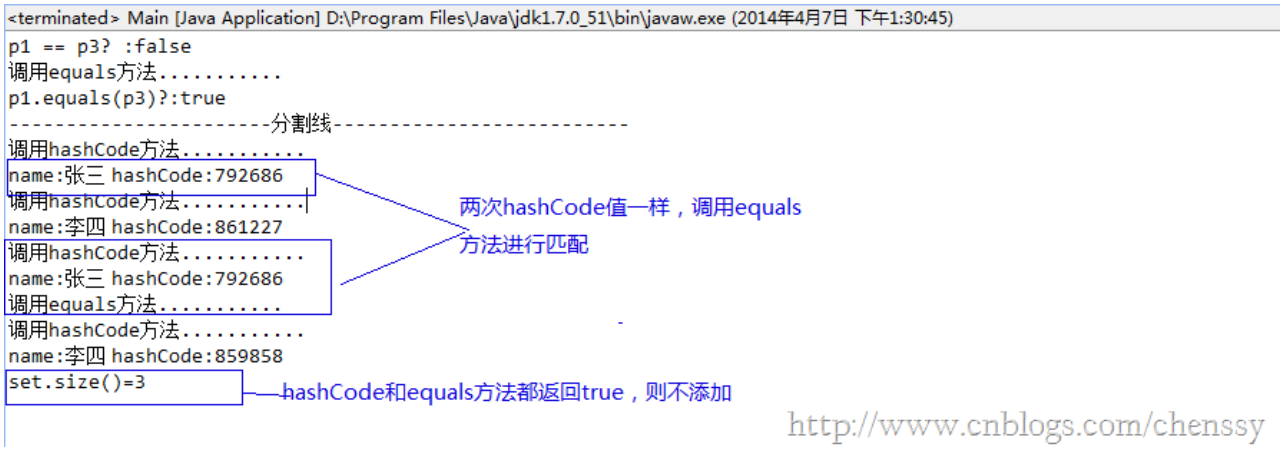
```
public class Person {
    private int age;
    private int sex;    //0：男，1：女
    private String name;

    private final int PRIME = 37;

    Person(int age ,int sex ,String name){
        this.age = age;
        this.sex = sex;
        this.name = name;
    }

    /** 省略getter、setter方法 **/
```

```
    @Override
    public int hashCode() {
        System.out.println("调用hashCode方法.....");

        int hashResult = 1;
        hashResult = (hashResult + Integer.valueOf(age).hashCode() + Integer.valueOf(sex).hashCode()) * PRIME;
        hashResult = PRIME * hashResult + ((name == null) ? 0 : name.hashCode());
        System.out.println("name:"+name +" hashCode:" + hashResult);

        return hashResult;
    }

    /**
     * 重写hashCode()
     */
    public boolean equals(Object obj) {
        System.out.println("调用equals方法.....");

        if(obj == null){
            return false;
        }
        if(obj.getClass() != this.getClass()){
            return false;
        }
        if(this == obj){
            return true;
        }

        Person person = (Person) obj;

        if(getAge() != person.getAge() || getSex()!=   person.getSex()){
            return false;
        }

        if(getName() != null){
            if(!getName().equals(person.getName())){
                return false;
            }
        }
        else if(person != null){
            return false;
        }
        return true;
    }
}
```

该 Bean 为一个标准的 Java Bean，重新实现了 hashCode 方法和 equals 方法。

```
public class Main extends JPanel {

    public static void main(String[] args) {
        Set<Person> set = new HashSet<Person>();

        Person p1 = new Person(11, 1, "张三");
        Person p2 = new Person(12, 1, "李四");
        Person p3 = new Person(11, 1, "张三");
        Person p4 = new Person(11, 1, "李四");

        //只验证p1、p3
        System.out.println("p1 == p3? :" + (p1 == p3));
        System.out.println("p1.equals(p3)?:"+p1.equals(p3));
        System.out.println("----------------------分割线----------------------------");
        set.add(p1);
        set.add(p2);
        set.add(p3);
        set.add(p4);
        System.out.println("set.size()="+set.size());
    }
}
```

运行结果如下：



图片 26.2 fig.2

从上图可以看出，程序调用四次 hashCode 方法，一次 equals 方法，其 set 的长度只有 3。add 方法运行流程完全符合他们两者之间的处理流程。

更多请关注：

>>>>>>>>>Java提高篇（十三）——equals()

>>>>>>>>>Java提高篇（二三）——HashMap

>>>>>>>>>Java提高篇（二四）——HashSet

>>>>>>>>>Java提高篇（二五）——HashTable

TreeMap

TreeMap 的实现是红黑树算法的实现，所以要了解 TreeMap 就必须对红黑树有一定的了解,其实这篇博文的名字叫做：根据红黑树的算法来分析 TreeMap 的实现，但是为了与 Java 提高篇系列博文保持一致还是叫做 TreeMap 比较好。通过这篇博文你可以获得如下知识点：

1、红黑树的基本概念。

2、红黑树增加节点、删除节点的实现过程。

3、红黑树左旋转、右旋转的复杂过程。

4、Java 中 TreeMap 是如何通过 put、deleteEntry 两个来实现红黑树增加、删除节点的。

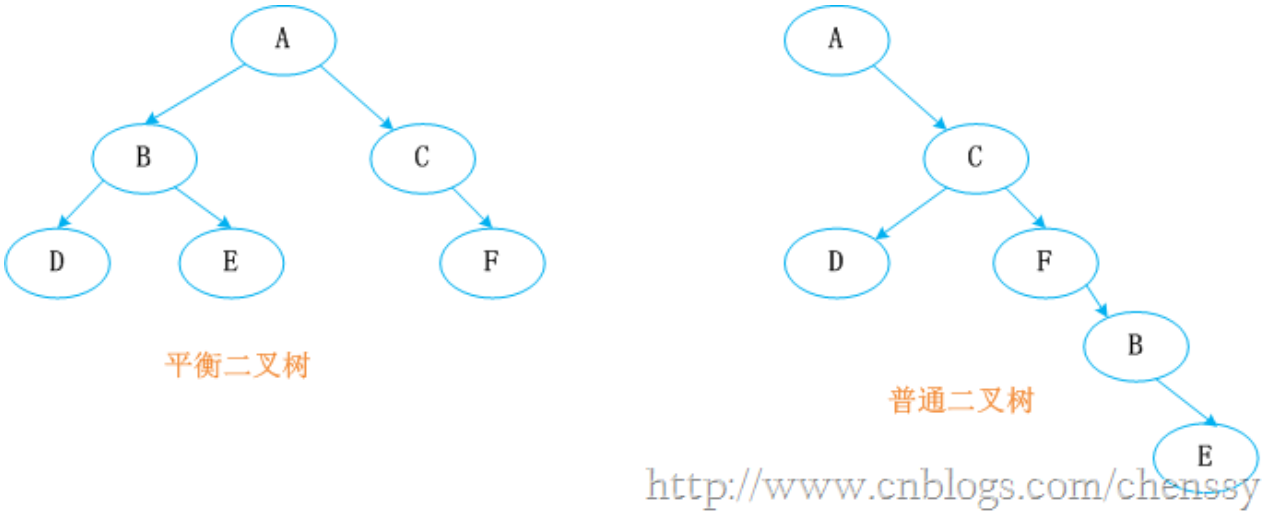
我想通过这篇博文你对 TreeMap 一定有了更深的认识。好了，下面先简单普及红黑树知识。

一、红黑树简介

红黑树又称红-黑二叉树，它首先是一颗二叉树，它具体二叉树所有的特性。同时红黑树更是一颗自平衡的排序二叉树。

我们知道一颗基本的二叉树他们都需要满足一个基本性质 - 即树中的任何节点的值大于它的左子节点，且小于它的右子节点。按照这个基本性质使得树的检索效率大大提高。我们知道在生成二叉树的过程是非常容易失衡的，最坏的情况就是一边倒（只有右/左子树），这样势必会导致二叉树的检索效率大大降低（O(n)），所以为了维持二叉树的平衡，大牛们提出了各种实现的算法，如：AVL，SBT，伸展树，TREAP ，红黑树等等。

平衡二叉树必须具备如下特性：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。也就是说该二叉树的任何一个等等子节点，其左右子树的高度都相近。



图片 27.1 fig.1

红黑树顾名思义就是节点是红色或者黑色的平衡二叉树，它通过颜色的约束来维持着二叉树的平衡。对于一棵有效的红黑树二叉树而言我们必须增加如下规则：

1、每个节点都只能是红色或者黑色

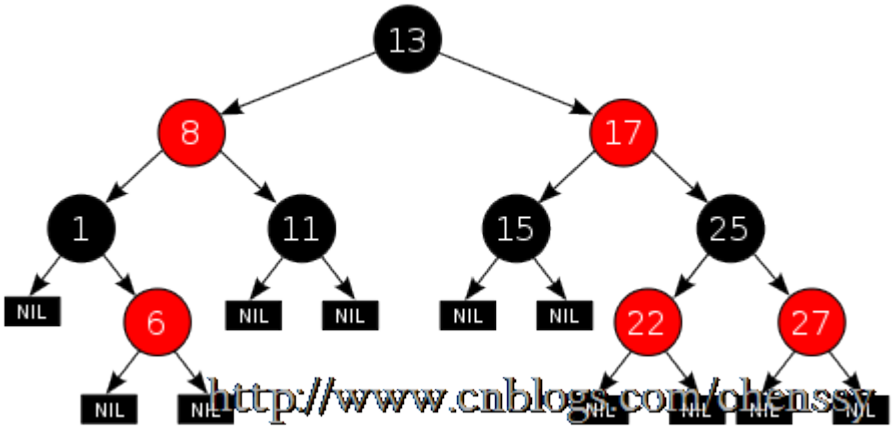
2、根节点是黑色

3、每个叶节点（NIL 节点，空节点）是黑色的。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

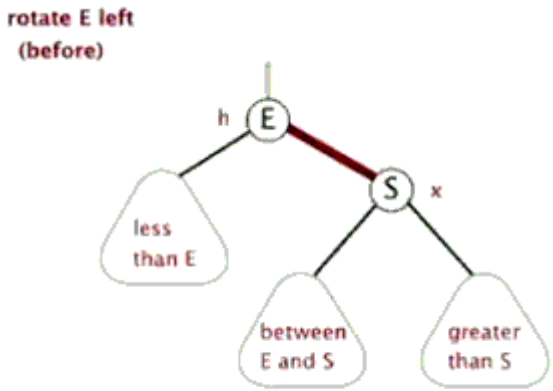
5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这棵树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。所以红黑树它是复杂而高效的，其检索效率 O(log n)。下图为一颗典型的红黑二叉树。



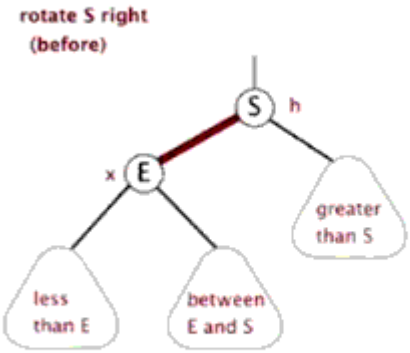
图片 27.2 fig.2

对于红黑二叉树而言它主要包括三大基本操作：左旋、右旋、着色。



图片 27.3 fig.3

左旋



图片 27.4 fig.4

右旋

（图片来自：http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html）

本节参考文献：http://baike.baidu.com/view/133754.htm?fr=aladdin一－百度百科

注：由于本文主要是讲解 Java 中 TreeMap，所以并没有对红黑树进行非常深入的了解和研究，如果诸位想对其进行更加深入的研究Lz提供几篇较好的博文：

1、红黑树系列集锦

2、红黑树数据结构剖析

3、红黑树

二、TreeMap 数据结构

>>>>>>回归主角：TreeMap<<<<<<

TreeMap 的定义如下：

```
public class TreeMap<K,V>
    extends AbstractMap<K,V>
    implements NavigableMap<K,V>, Cloneable, java.io.Serializable
```

TreeMap 继承 AbstractMap，实现 NavigableMap、Cloneable、Serializable 三个接口。其中 AbstractMap 表明 TreeMap 为一个 Map 即支持 key-value 的集合，NavigableMap（更多）则意味着它支持一系列的导航方法，具备针对给定搜索目标返回最接近匹配项的导航方法 。

TreeMap 中同时也包含了如下几个重要的属性：

```
//比较器，因为TreeMap是有序的，通过comparator接口我们可以对TreeMap的内部排序进行精密的控制
        private final Comparator<? super K> comparator;
        //TreeMap红-黑节点，为TreeMap的内部类
        private transient Entry<K,V> root = null;
        //容器大小
        private transient int size = 0;
        //TreeMap修改次数
        private transient int modCount = 0;
        //红黑树的节点颜色--红色
        private static final boolean RED = false;
        //红黑树的节点颜色--黑色
        private static final boolean BLACK = true;
```

对于叶子节点 Entry 是 TreeMap 的内部类，它有几个重要的属性：

```
//键
        K key;
        //值
        V value;
        //左孩子
        Entry<K,V> left = null;
        //右孩子
        Entry<K,V> right = null;
        //父亲
```

```
        Entry<K,V> parent;
        //颜色
        boolean color = BLACK;
```

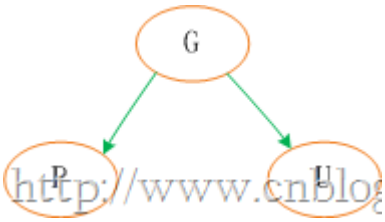
注：前面只是开胃菜，下面是本篇博文的重中之重，在下面两节我将重点讲解 treeMap 的 put()、delete() 方法。通过这两个方法我们会了解红黑树增加、删除节点的核心算法。

三、TreeMap put() 方法

在了解 TreeMap 的 put() 方法之前，我们先了解红黑树增加节点的算法。

红黑树增加节点

红黑树在新增节点过程中比较复杂，复杂归复杂它同样必须要依据上面提到的五点规范，同时由于规则 1、2、3 基本都会满足，下面我们主要讨论规则 4、5。假设我们这里有一棵最简单的树，我们规定新增的节点为 N、它的父节点为 P、P 的兄弟节点为 U、P 的父节点为 G。



图片 27.5 fig.5

对于新节点的插入有如下三个关键地方：

1、插入新节点总是红色节点 。 2、如果插入节点的父节点是黑色, 能维持性质 。 3、如果插入节点的父节点是红色, 破坏了性质. 故插入算法就是通过重新着色或旋转, 来维持性质 。

为了保证下面的阐述更加清晰和根据便于参考，我这里将红黑树的五点规定再贴一遍：

1、每个节点都只能是红色或者黑色

2、根节点是黑色

3、每个叶节点（NIL 节点，空节点）是黑色的。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

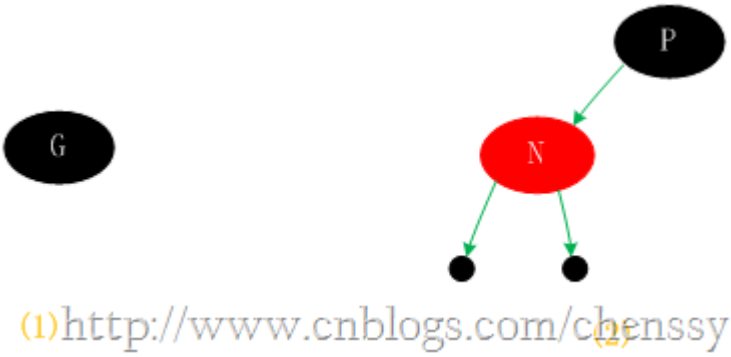
5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

一、为跟节点

若新插入的节点N没有父节点，则直接当做根据节点插入即可，同时将颜色设置为黑色。（如图一（1））

二、父节点为黑色

这种情况新节点 N 同样是直接插入，同时颜色为红色，由于根据规则四它会存在两个黑色的叶子节点，值为 null。同时由于新增节点 N 为红色，所以通过它的子节点的路径依然会保存着相同的黑色节点数，同样满足规则 5。（如图一（2））

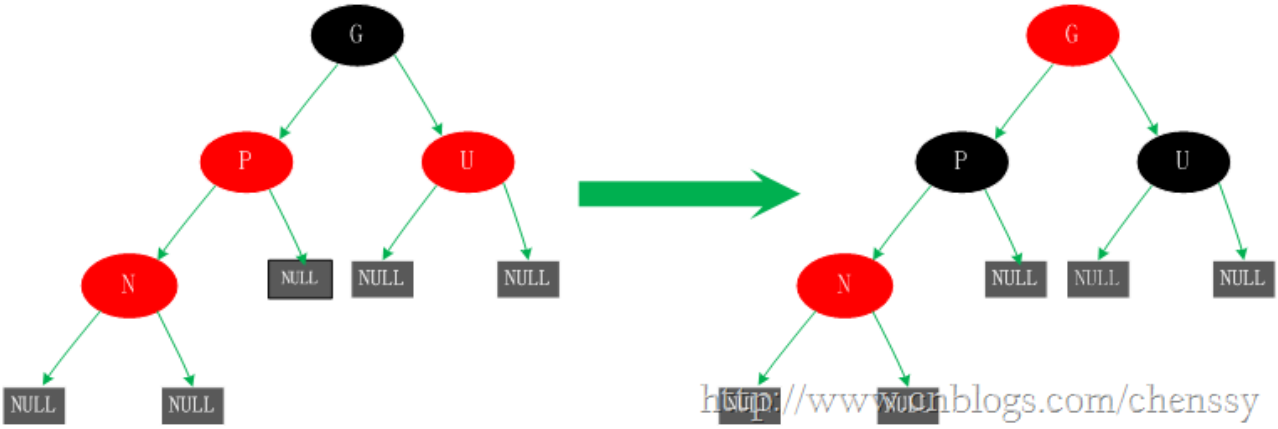


图片 27.6 fig.6

（图一）

三、若父节点 P 和 P 的兄弟节点 U 都为红色

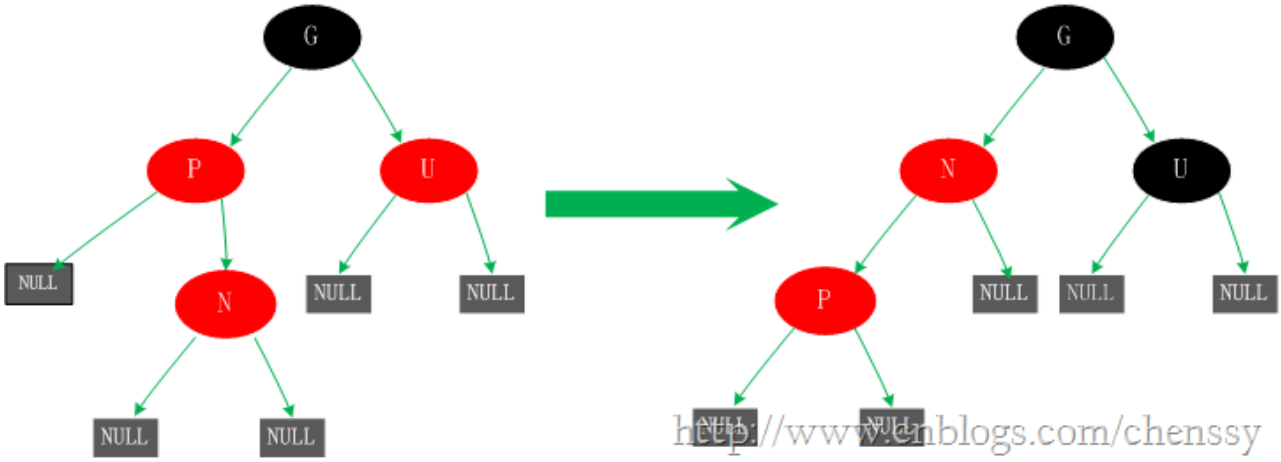
对于这种情况若直接插入肯定会出现不平衡现象。怎么处理？P、U 节点变黑、G 节点变红。这时由于经过节点 P、U 的路径都必须经过 G 所以在这些路径上面的黑节点数目还是相同的。但是经过上面的处理，可能 G 节点的父节点也是红色，这个时候我们需要将 G 节点当做新增节点递归处理。



图片 27.7 fig.7

四、若父节点 P 为红色，叔父节点U为黑色或者缺少，且新增节点 N 为 P 节点的右孩子

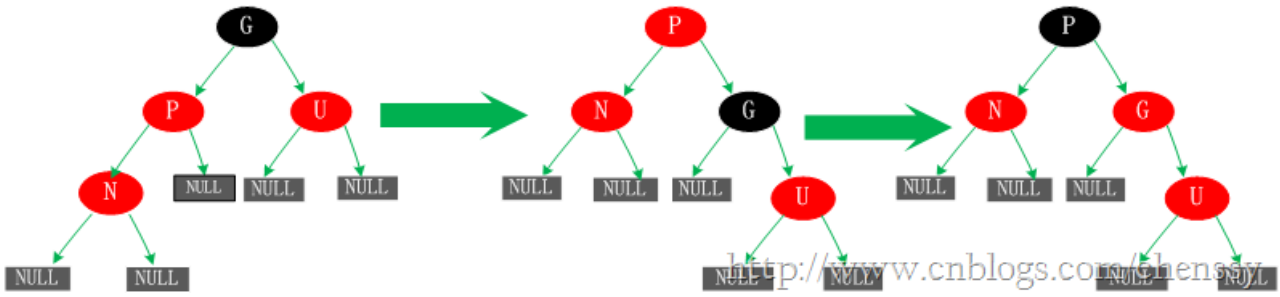
对于这种情况我们对新增节点 N、P 进行一次左旋转。这里所产生的结果其实并没有完成，还不是平衡的（违反了规则四），这是我们需要进行情况5的操作。



图片 27.8 fig.8

五、父节点 P 为红色，叔父节点 U 为黑色或者缺少，新增节点 N 为父节点 P 左孩子

这种情况有可能是由于情况四而产生的，也有可能不是。对于这种情况先已 P 节点为中心进行右旋转，在旋转后产生的树中，节点 P 是节点 N、G 的父节点。但是这棵树并不规范，它违反了规则 4，所以我们将 P、G 节点的颜色进行交换，使之其满足规范。开始时所有的路径都需要经过 G 其他们的黑色节点数一样，但是现在所有的路径改为经过 P，且 P 为整棵树的唯一黑色节点，所以调整后的树同样满足规范 5。



图片 27.9 fig.9

上面展示了红黑树新增节点的五种情况，这五种情况涵盖了所有的新增可能，不管这棵红黑树多么复杂，都可以根据这五种情况来进行生成。下面就来分析 Java 中的 TreeMap 是如何来实现红黑树的。

TreeMap put() 方法实现分析

在 TreeMap 的 put() 的实现方法中主要分为两个步骤，第一：构建排序二叉树，第二：平衡二叉树。

对于排序二叉树的创建，其添加节点的过程如下：

1、以根节点为初始节点进行检索。

2、与当前节点进行比对，若新增节点值较大，则以当前节点的右子节点作为新的当前节点。否则以当前节点的左子节点作为新的当前节点。

3、循环递归 2 步骤知道检索出合适的叶子节点为止。

4、将新增节点与 3 步骤中找到的节点进行比对，如果新增节点较大，则添加为右子节点；否则添加为左子节点。

按照这个步骤我们就可以将一个新增节点添加到排序二叉树中合适的位置。如下：

```
public V put(K key, V value) {
        //用t表示二叉树的当前节点
        Entry<K,V> t = root;
        //t为null表示一个空树，即TreeMap中没有任何元素，直接插入
        if (t == null) {
            //比较key值，个人觉得这句代码没有任何意义，空树还需要比较、排序?
            compare(key, key); // type (and possibly null) check
            //将新的key-value键值对创建为一个Entry节点，并将该节点赋予给root
            root = new Entry<>(key, value, null);
            //容器的size = 1，表示TreeMap集合中存在一 个元素
            size = 1;
            //修改次数 + 1
            modCount++;
            return null;
        }
        int cmp;    //cmp表示key排序的返回结果
        Entry<K,V> parent;  //父节点
        // split comparator and comparable paths
        Comparator<? super K> cpr = comparator;   //指定的排序算法
        //如果cpr不为空，则采用既定的排序算法进行创建TreeMap集合
        if (cpr != null) {
            do {
                parent = t;     //parent指向上次循环后的t
                //比较新增节点的key和当前节点key的大小
```

```
            cmp = cpr.compare(key, t.key);
            //cmp返回值小于0，表示新增节点的key小于当前节点的key，则以当前节点的左子节点作为新的当前节点
            if (cmp < 0)
                t = t.left;
            //cmp返回值大于0，表示新增节点的key大于当前节点的key，则以当前节点的右子节点作为新的当前节点
            else if (cmp > 0)
                t = t.right;
            //cmp返回值等于0，表示两个key值相等，则新值覆盖旧值，并返回新值
            else
                return t.setValue(value);
        } while (t != null);
    }
    //如果cpr为空，则采用默认的排序算法进行创建 TreeMap集合
    else {
        if (key == null)    //key值为空抛出异常
            throw new NullPointerException();
        /* 下面处理过程和上面一样 */
        Comparable<? super K> k =  (Comparable<? super K>) key;
        do {
            parent = t;
            cmp = k.compareTo(t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                t = t.right;
            else
                return t.setValue(value);
        } while (t != null);
    }
    //将新增节点当做parent的子节点
    Entry<K,V> e = new Entry<>(key, value,  parent);
    //如果新增节点的key小于parent的key，则当做左子节点
    if (cmp < 0)
        parent.left = e;
    //如果新增节点的key大于parent的key，则当做右子节点
    else
        parent.right = e;
    /*
     *  上面已经完成了排序二叉树的的构建，将新增节点 插入该树中的合适位置
     *  下面fixAfterInsertion()方法就是对这棵树进行调整、平衡，具体过程参考上面的五种情况
     */
    fixAfterInsertion(e);
    //TreeMap元素数量 + 1
    size++;
    //TreeMap容器修改次数 + 1
```

```
            modCount++;
            return null;
        }
```

上面代码中 do{} 代码块是实现排序二叉树的核心算法，通过该算法我们可以确认新增节点在该树的正确位置。找到正确位置后将插入即可，这样做了其实还没有完成，因为我知道 TreeMap 的底层实现是红黑树，红黑树是一棵平衡排序二叉树，普通的排序二叉树可能会出现失衡的情况，所以下一步就是要进行调整。fixAfterInsertion(e); 调整的过程务必会涉及到红黑树的左旋、右旋、着色三个基本操作。代码如下：

```
/**
    * 新增节点后的修复操作
    * x 表示新增节点
    */
    private void fixAfterInsertion(Entry<K,V> x) {
        x.color = RED;   //新增节点的颜色为红色

        //循环 直到 x不是根节点，且x的父节点不为红色
        while (x != null && x != root && x.parent.color == RED) {
            //如果X的父节点（P）是其父节点的父节点（G）的左节点
            if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
                //获取X的叔节点(U)
                Entry<K,V> y = rightOf(parentOf (parentOf(x)));
                //如果X的叔节点（U）为红色（情况三）
                if (colorOf(y) == RED) {
                    //将X的父节点（P）设置为黑色
                    setColor(parentOf(x), BLACK);
                    //将X的叔节点（U）设置为黑色
                    setColor(y, BLACK);
                    //将X的父节点的父节点（G）设置红色
                    setColor(parentOf(parentOf(x)), RED);
                    x = parentOf(parentOf(x));
                }
                //如果X的叔节点（U为黑色）；这里会存在  两种情况（情况四、情况五）
                else {
                    //如果X节点为其父节点（P）的右子 树，则进行左旋转（情况四）
                    if (x == rightOf(parentOf(x))) {
                        //将X的父节点作为X
                        x = parentOf(x);
                        //右旋转
                        rotateLeft(x);
                    }
                    //（情况五）
                    //将X的父节点（P）设置为黑色
                    setColor(parentOf(x), BLACK);
```

```
                //将X的父节点的父节点（G）设置红色
                setColor(parentOf(parentOf(x)), RED);
                //以X的父节点的父节点（G）为中心右旋转
                rotateRight(parentOf(parentOf(x)));
            }
        }
        //如果X的父节点（P）是其父节点的父节点（G）的右节点
        else {
            //获取X的叔节点（U）
            Entry<K,V> y = leftOf(parentOf(parentOf(x)));
            //如果X的叔节点（U）为红色（情况三）
            if (colorOf(y) == RED) {
                //将X的父节点（P）设置为黑色
                setColor(parentOf(x), BLACK);
                //将X的叔节点（U）设置为黑色
                setColor(y, BLACK);
                //将X的父节点的父节点（G）设置红色
                setColor(parentOf(parentOf(x)), RED);
                x = parentOf(parentOf(x));
            }
          //如果X的叔节点（U为黑色）；这里会存在两种情况（情况四、情况五）
            else {
                //如果X节点为其父节点（P）的右子树，则进行左旋转（情况四）
                if (x == leftOf(parentOf(x))) {
                    //将X的父节点作为X
                    x = parentOf(x);
                    //右旋转
                    rotateRight(x);
                }
                //（情况五）
                //将X的父节点（P）设置为黑色
                setColor(parentOf(x), BLACK);
                //将X的父节点的父节点（G）设置红色
                setColor(parentOf(parentOf(x)), RED);
                //以X的父节点的父节点（G）为中心右旋转
                rotateLeft(parentOf(parentOf(x)));
            }
        }
    }
    //将根节点G强制设置为黑色
    root.color = BLACK;
}
```

对这段代码的研究我们发现,其处理过程完全符合红黑树新增节点的处理过程。所以在看这段代码的过程一定要对红黑树的新增节点过程有了解。在这个代码中还包含几个重要的操作。左旋 (rotateLeft())、右旋 (rotateRight()) 、着色 (setColor()) 。

左旋：rotateLeft()

所谓左旋转，就是将新增节点 (N) 当做其父节点 (P) ，将其父节点P当做新增节点 (N) 的左子节点。即：G.left —> N ,N.left —> P。

```
private void rotateLeft(Entry<K,V> p) {
        if (p != null) {
            //获取P的右子节点，其实这里就相当于新增节点N ( 情况四而言 )
            Entry<K,V> r = p.right;
            //将R的左子树设置为P的右子树
            p.right = r.left;
            //若R的左子树不为空，则将P设置为R左子树的父亲
            if (r.left != null)
                r.left.parent = p;
            //将P的父亲设置R的父亲
            r.parent = p.parent;
            //如果P的父亲为空，则将R设置为跟节点
            if (p.parent == null)
                root = r;
            //如果P为其父节点 ( G ) 的左子树，则将R设置为P父 节点(G)左子树
            else if (p.parent.left == p)
                p.parent.left = r;
            //否则R设置为P的父节点 ( G ) 的右子树
            else
                p.parent.right = r;
            //将P设置为R的左子树
            r.left = p;
            //将R设置为P的父节点
            p.parent = r;
        }
    }
```

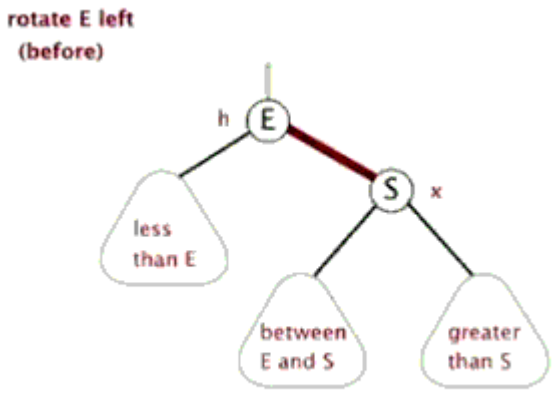
右旋：rotateRight()

所谓右旋转即，P.right —> G、G.parent —> P。

```
private void rotateRight(Entry<K,V> p) {
        if (p != null) {
            //将L设置为P的左子树
```

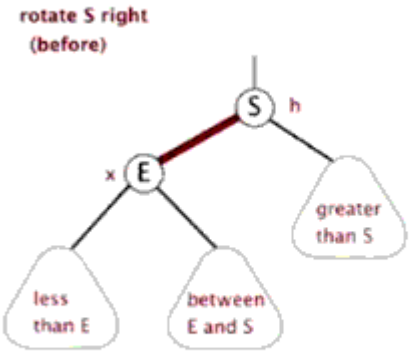
```
            Entry<K,V> l = p.left;
            //将L的右子树设置为P的左子树
            p.left = l.right;
            //若L的右子树不为空，则将P设置L的右子树的父节点
            if (l.right != null)
                l.right.parent = p;
            //将P的父节点设置为L的父节点
            l.parent = p.parent;
            //如果P的父节点为空，则将L设置根节点
            if (p.parent == null)
                root = l;
            //若P为其父节点的右子树，则将L设置为P的父节点的右子树
            else if (p.parent.right == p)
                p.parent.right = l;
            //否则将L设置为P的父节点的左子树
            else
                p.parent.left = l;
            //将P设置为L的右子树
            l.right = p;
            //将L设置为P的父节点
            p.parent = l;
        }
    }
```

左旋、右旋的示意图如下：



图片 27.10 fig.3

（左旋）



图片 27.11 fig.3

（右旋）

（图片来自：http://www.cnblogs.com/yangecnu/p/Introduce-Red-Black-Tree.html）

着色：setColor()

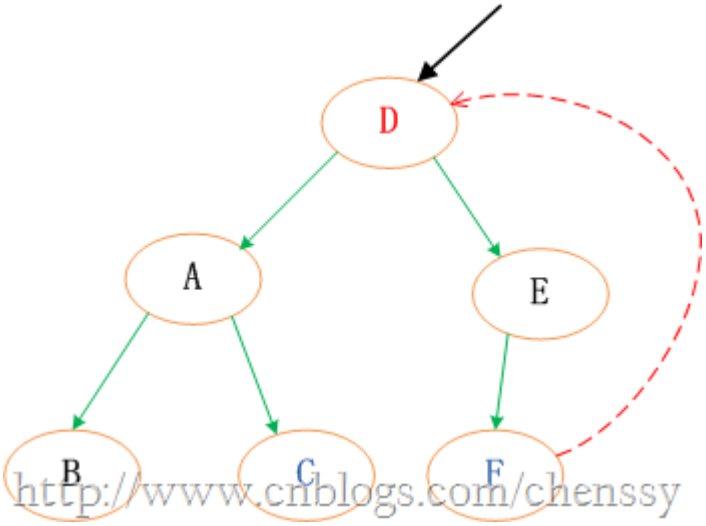
着色就是改变该节点的颜色，在红黑树中，它是依靠节点的颜色来维持平衡的。

```
private static <K,V> void setColor(Entry<K,V> p, boolean c) {
        if (p != null)
            p.color = c;
    }
```

四、TreeMap delete() 方法

红黑树删除节点

针对于红黑树的增加节点而言，删除显得更加复杂，使原本就复杂的红黑树变得更加复杂。同时删除节点和增加节点一样，同样是找到删除的节点，删除之后调整红黑树。但是这里的删除节点并不是直接删除，而是通过走了“弯路”通过一种捷径来删除的：**找到被删除的节点 D 的子节点 C，用 C 来替代 D，不是直接删除 D，因为 D 被 C 替代了，直接删除 C 即可。**所以这里就将删除父节点 D 的事情转变为了删除子节点 C 的事情，这样处理就将复杂的删除事件简单化了。**子节点 C 的规则是：右分支最左边，或者左分支最右边的。**



图片 27.12 fig.10

红-黑二叉树删除节点，最大的麻烦是要保持 各分支黑色节点数目相等。 因为是删除，所以不用担心存在颜色冲突问题——插入才会引起颜色冲突。

红黑树删除节点同样会分成几种情况，这里是按照待删除节点有几个儿子的情况来进行分类：

1、没有儿子，即为叶结点。直接把父结点的对应儿子指针设为 NULL，删除儿子结点就 OK 了。

2、只有一个儿子。那么把父结点的相应儿子指针指向儿子的独生子，删除儿子结点也 OK 了。

3、有两个儿子。这种情况比较复杂，但还是比较简单。上面提到过用子节点 C 替代代替待删除节点 D，然后删除子节点 C 即可。

下面就论各种删除情况来进行图例讲解，但是在讲解之前请允许我再次啰嗦一句，**请时刻牢记红黑树的 5 点规定：**

1、每个节点都只能是红色或者黑色

2、根节点是黑色

3、每个叶节点（NIL 节点，空节点）是黑色的。

4、如果一个结点是红的，则它两个子节点都是黑的。也就是说在一条路径上不能出现相邻的两个红色结点。

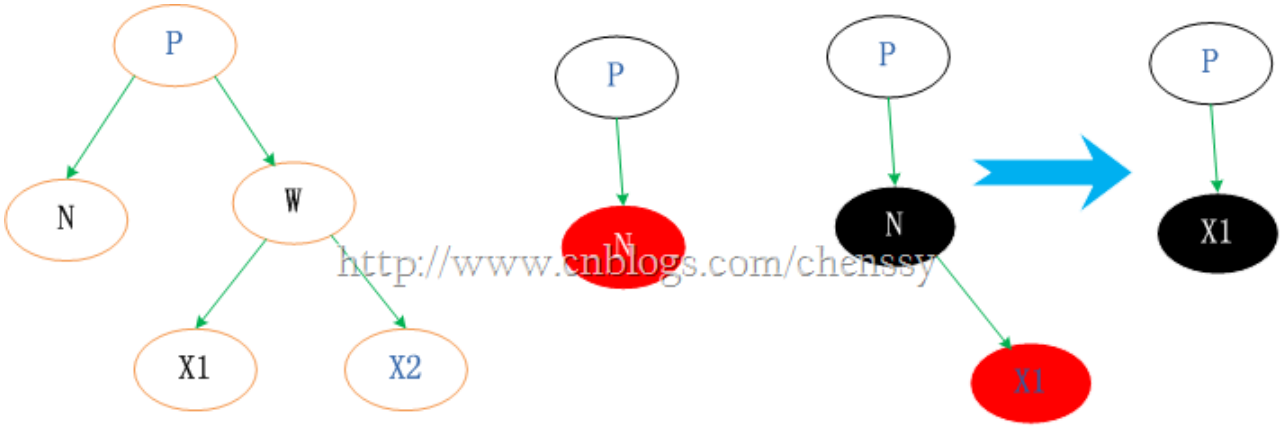
5、从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

（注：已经讲三遍了，再不记住我就怀疑你是否适合搞 IT 了 O(∩_∩)O~）

诚然，既然删除节点比较复杂，那么在这里我们就约定一下规则：

1、下面要讲解的删除节点一定是实际要删除节点的后继节点（N），如前面提到的C。

2、下面提到的删除节点的树都是如下结构，该结构所选取的节点是待删除节点的右树的最左边子节点。这里我们规定真实删除节点为 N、父节点为 P、兄弟节点为 W 兄弟节点的两个子节点为 X1、X2。如下图（2.1）。



图（2.1）　图（2.2）　图（2.3）

图片 27.13 fig.11

现在我们就上面提到的三种情况进行分析、处理。

情况一、无子节点（红色节点）

这种情况对该节点直接删除即可，不会影响树的结构。因为该节点为叶子节点它不可能存在子节点——如子节点为黑，则违反黑节点数原则（规定 5），为红，则违反“颜色”原则（规定 4）。如上图（2.2）。

情况二、有一个子节点

这种情况处理也是非常简单的，用子节点替代待删除节点，然后删除子节点即可。如上图（2.3）

情况三、有两个子节点

这种情况可能会稍微有点儿复杂。它需要找到一个替代待删除节点（N）来替代它，然后删除 N 即可。它主要分为四种情况。

1、N 的兄弟节点 W 为红色

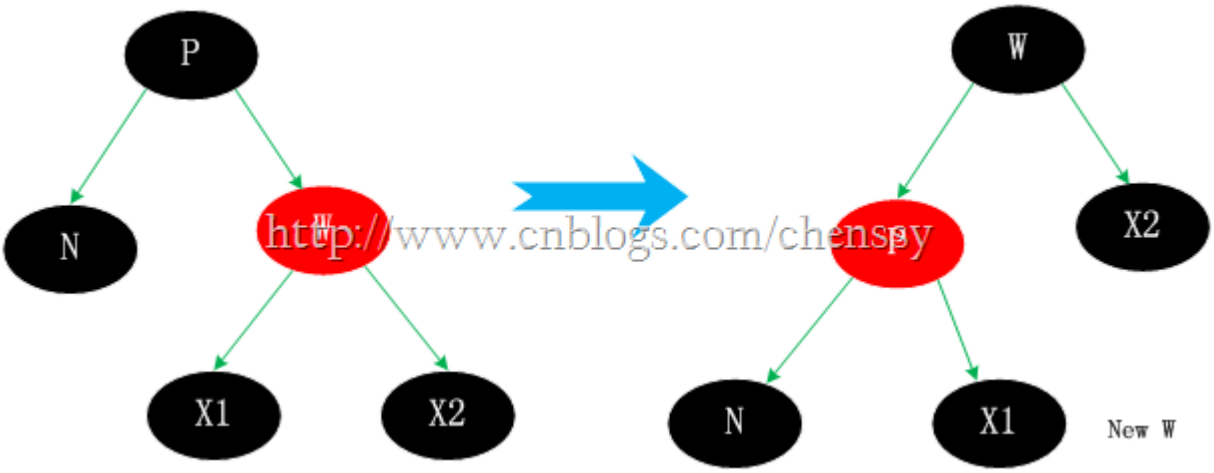
2、N 的兄弟 w 是黑色的，且w的俩个孩子都是黑色的。

3、N 的兄弟 w 是黑色的，w 的左孩子是红色，w 的右孩子是黑色。

4、N 的兄弟 w 是黑色的，且 w 的右孩子时红色的。

情况 3.1、N 的兄弟节点 W 为红色

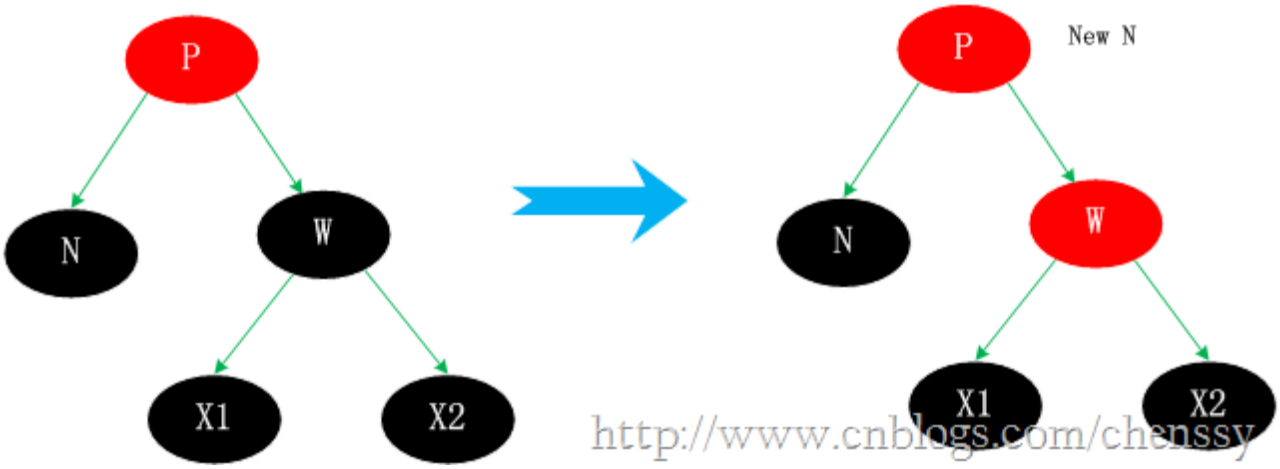
W 为红色，那么其子节点 X1、X2 必定全部为黑色，父节点 P 也为黑色。处理策略是：改变 W、P 的颜色，然后进行一次左旋转。这样处理就可以使得红黑性质得以继续保持。N 的新兄弟 new w 是旋转之前 w 的某个孩子，为黑色。这样处理后将情况 3.1、转变为 3.2、3.3、3.4 中的一种。如下：



图片 27.14 fig.12

情况 3.2、N 的兄弟 w 是黑色的，且 w 的俩个孩子都是黑色的

这种情况其父节点可红可黑，由于 W 为黑色，这样导致 N 子树相对于其兄弟W子树少一个黑色节点，这时我们可以将 W 置为红色。这样，N 子树与 W 子树黑色节点一致，保持了平衡。如下

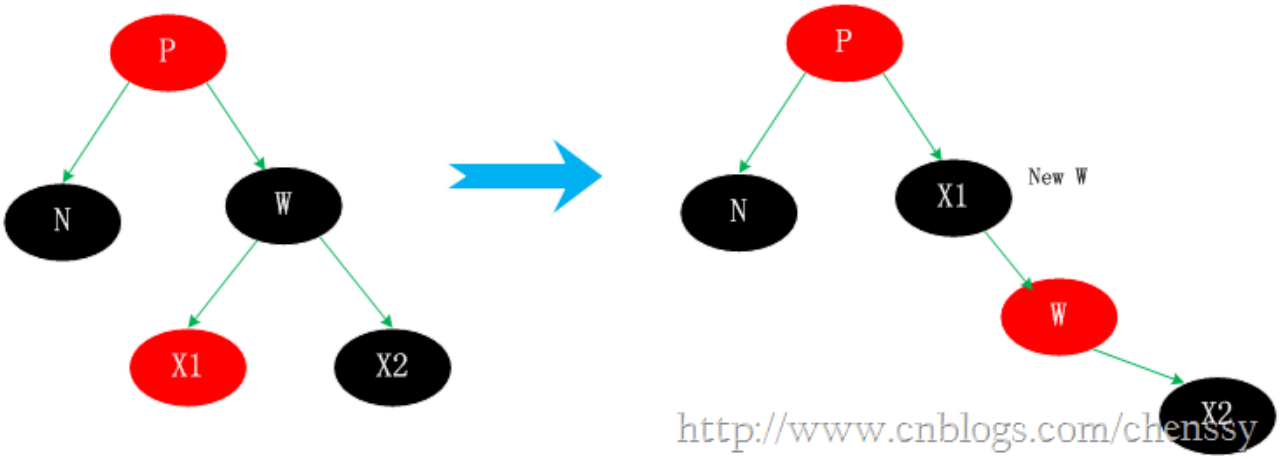


图片 27.15 fig.13

将 W 由黑转变为红，这样就会导致新节点 new N 相对于它的兄弟节点会少一个黑色节点。但是如果 new x 为红色，我们直接将 new x 转变为黑色，保持整棵树的平衡。否则情况 3.2 会转变为情况 3.1、3.3、3.4 中的一种。

情况3.3、N的兄弟w是黑色的，w的左孩子是红色，w的右孩子是黑色

针对这种情况是将节点 W 和其左子节点进行颜色交换，然后对 W 进行右旋转处理。

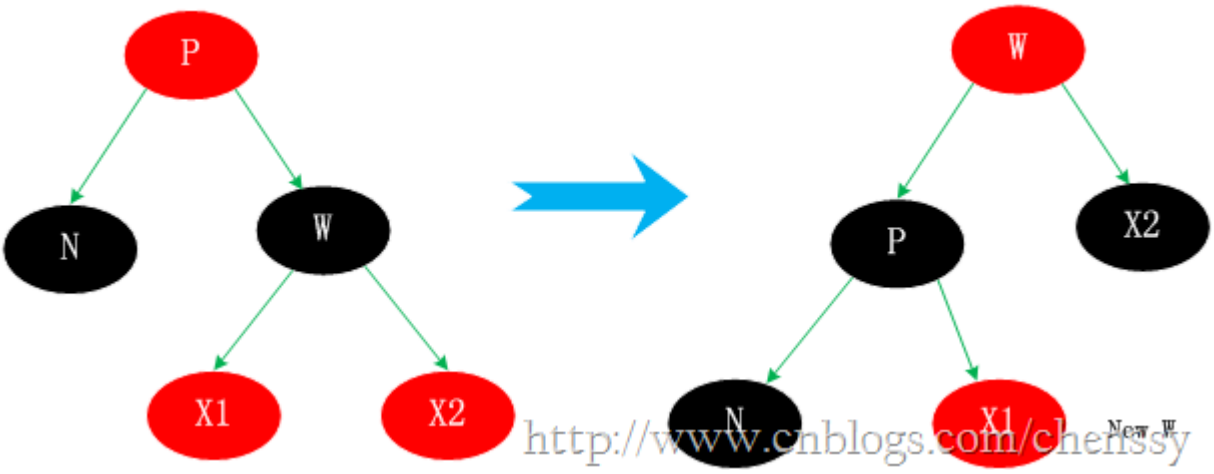


图片 27.16 fig.14

此时 N 的新兄弟 X1(new w) 是一个有红色右孩子的黑结点，于是将情况 3 转化为情况 4.

情况 3.4、N 的兄弟 w 是黑色的，且 w 的右孩子时红色的

交换 W 和父节点 P 的颜色，同时对 P 进行左旋转操作。这样就把左边缺失的黑色节点给补回来了。同时将 W 的右子节点 X2 置黑。这样左右都达到了平衡。



图片 27.17 fig.15

总结

个人认为这四种情况比较难理解，首先他们都不是单一的某种情况，他们之间是可以进行互转的。相对于其他的几种情况，情况 3.2 比较好理解，仅仅只是一个颜色的转变，通过减少右子树的一个黑色节点使之保持平衡，同时将不平衡点上移至 N 与 W 的父节点，然后进行下一轮迭代。情况 3.1，是将W旋转将其转成情况 2、3、4 情况进行处理。而情况 3.3 通过转变后可以化成情况 3.4 来进行处理，从这里可以看出情况 3.4 应该最终结。情况 3.4、右子节点为红色节点，那么将缺失的黑色节点交由给右子节点，通过旋转达到平衡。

通过上面的分析，我们已经初步了解了红黑树的删除节点情况，相对于增加节点而言它确实是选的较为复杂。下面我将看到在 Java TreeMap 中是如何实现红黑树删除的。

TreeMap deleteEntry() 方法实现分析

通过上面的分析我们确认删除节点的步骤是：找到一个替代子节点 C 来替代 P，然后直接删除 C，最后调整这棵红黑树。下面代码是寻找替代节点、删除替代节点。

```
private void deleteEntry(Entry<K,V> p) {
        modCount++;     //修改次数 +1
        size--;         //元素个数 -1

        /*
         * 被删除节点的左子树和右子树都不为空，那么就用 p节点的中序后继节点代替 p 节点
         * successor(P)方法为寻找P的替代节点。规则是右分支 最左边，或者 左分支最右边的节点
         * ---------------------（1）
         */
        if (p.left != null && p.right != null) {
            Entry<K,V> s = successor(p);
```

```
        p.key = s.key;
        p.value = s.value;
        p = s;
    }

    //replacement为替代节点，如果P的左子树存在那么就用左子树替代，否则用右子树替代
    Entry<K,V> replacement = (p.left != null ? p.left : p.right);

    /*
     * 删除节点，分为上面提到的三种情况
     * ----------------------（2）
     */
    //如果替代节点不为空
    if (replacement != null) {
        replacement.parent = p.parent;
        /*
         *replacement来替代P节点
         */
        //若P没有父节点，则跟节点直接变成replacement
        if (p.parent == null)
            root = replacement;
        //如果P为左节点，则用replacement来替代为左节点
        else if (p == p.parent.left)
            p.parent.left  = replacement;
        //如果P为右节点，则用replacement来替代为右节点
        else
            p.parent.right = replacement;

        //同时将P节点从这棵树中剔除掉
        p.left = p.right = p.parent = null;

        /*
         * 若P为红色直接删除，红黑树保持平衡
         * 但是若P为黑色，则需要调整红黑树使其保持平衡
         */
        if (p.color == BLACK)
            fixAfterDeletion(replacement);
    } else if (p.parent == null) {    //p没有父节点，表示为P根节点，直接删除即可
        root = null;
    } else {     //P节点不存在子节点，直接删除即可
        if (p.color == BLACK)       //如果P节点的颜色为黑色，对红黑树进行调整
            fixAfterDeletion(p);

        //删除P节点
        if (p.parent != null) {
```

```
            if (p == p.parent.left)
                p.parent.left = null;
            else if (p == p.parent.right)
                p.parent.right = null;
            p.parent = null;
        }
    }
}
```

（1）除是寻找替代节点 replacement，其实现方法为 successor()。如下：

```
static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
    if (t == null)
        return null;
    /*
     * 寻找右子树的最左子树
     */
    else if (t.right != null) {
        Entry<K,V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    }
    /*
     * 选择左子树的最右子树
     */
    else {
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

（2）处是删除该节点过程。它主要分为上面提到的三种情况，它与上面的 if…else if… else一一对应 。如下：

1、有两个儿子。这种情况比较复杂，但还是比较简单。上面提到过用子节点 C 替代代替待删除节点 D，然后删除子节点 C 即可。

2、没有儿子，即为叶结点。直接把父结点的对应儿子指针设为 NULL，删除儿子结点就 OK 了。

3、只有一个儿子。那么把父结点的相应儿子指针指向儿子的独生子，删除儿子结点也 OK 了。

删除完节点后，就要根据情况来对红黑树进行复杂的调整：fixAfterDeletion()。

```
private void fixAfterDeletion(Entry<K,V> x) {
    // 删除节点需要一直迭代，知道 直到 x 不是根节点，且  x 的颜色是黑色
    while (x != root && colorOf(x) == BLACK) {
        if (x == leftOf(parentOf(x))) {     //若X 节点为左节点
            //获取其兄弟节点
            Entry<K,V> sib = rightOf(parentOf(x));

            /*
             * 如果兄弟节点为红色----（情况3.1）
             * 策略：改变W、P的颜色，然后进行一次左旋转
             */
            if (colorOf(sib) == RED) {
                setColor(sib, BLACK);
                setColor(parentOf(x), RED);
                rotateLeft(parentOf(x));
                sib = rightOf(parentOf(x));
            }

            /*
             * 若兄弟节点的两个子节点都为黑色----（情况3.2）
             * 策略：将兄弟节点编程红色
             */
            if (colorOf(leftOf(sib))  == BLACK &&
                colorOf(rightOf(sib)) == BLACK) {
                setColor(sib, RED);
                x = parentOf(x);
            }
            else {
                /*
                 * 如果兄弟节点只有右子树为黑色----  （情况3.3）
                 * 策略：将兄弟节点与其左子树进行颜色互换然后进行右转
                 * 这时情况会转变为3.4
                 */
                if (colorOf(rightOf(sib)) == BLACK) {
                    setColor(leftOf(sib), BLACK);
                    setColor(sib, RED);
                    rotateRight(sib);
                    sib = rightOf(parentOf(x));
                }
                /*
                 *----情况3.4
                 *策略：交换兄弟节点和父节点的颜色，
```

```
                *同时将兄弟节点右子树设置为黑色，最后左旋转
                */
                setColor(sib, colorOf(parentOf  (x)));
                setColor(parentOf(x), BLACK);
                setColor(rightOf(sib), BLACK);
                rotateLeft(parentOf(x));
                x = root;
            }
        }

        /**
         * X节点为右节点与其为做节点处理过程差不多，这里 就不在累述了
         */
        else {
            Entry<K,V> sib = leftOf(parentOf(x));

            if (colorOf(sib) == RED) {
                setColor(sib, BLACK);
                setColor(parentOf(x), RED);
                rotateRight(parentOf(x));
                sib = leftOf(parentOf(x));
            }

            if (colorOf(rightOf(sib)) == BLACK &&
                colorOf(leftOf(sib)) == BLACK) {
                setColor(sib, RED);
                x = parentOf(x);
            } else {
                if (colorOf(leftOf(sib)) == BLACK)  {
                    setColor(rightOf(sib), BLACK);
                    setColor(sib, RED);
                    rotateLeft(sib);
                    sib = leftOf(parentOf(x));
                }
                setColor(sib, colorOf(parentOf (x)));
                setColor(parentOf(x), BLACK);
                setColor(leftOf(sib), BLACK);
                rotateRight(parentOf(x));
                x = root;
            }
        }
    }

    setColor(x, BLACK);
}
```

这是红黑树在删除节点后，对树的平衡性进行调整的过程，其实现过程与上面四种复杂的情况一一对应，所以在这个源码的时候一定要对着上面提到的四种情况看。

五、写在最后

这篇博文确实是有点儿长，在这里非常感谢各位看客能够静下心来读完，我想你通过读完这篇博文一定收获不小。同时这篇博文很大篇幅都在阐述红黑树的实现过程，对 Java 的 TreeMap 聊的比较少，但是我认为如果理解了红黑树的实现过程，对 TreeMap 那是手到擒来，小菜一碟。

同时这篇博文我写了四天，看了、参考了大量的博文。同时不免会有些地方存在借鉴之处，在这里对其表示感谢。LZ 大二开始学习数据结构，自认为学的不错，现在发现数据结构我还有太多的地方需要学习了，同时也再一次体味了算法的魅力！！！！

参考资料：

1、红黑树数据结构剖析：http://www.cnblogs.com/fanzhidongyzby/p/3187912.html

2、红黑二叉树详解及理论分析：http://blog.csdn.net/kartorz/article/details/8865997

3、教你透彻了解红黑树：http://blog.csdn.net/v_july_v/article/details/6105630

4、经典算法研究系列：五、红黑树算法的实现与剖析：http://blog.csdn.net/v_JULY_v/article/details/6109153

5、示例，红黑树插入和删除过程：http://saturnman.blog.163.com/blog/static/557611201097221570/

6、红黑二叉树详解及理论分析：http://blog.csdn.net/kartorz/article/details/8865997

TreeSet

与 HashSet 是基于 HashMap 实现一样，TreeSet 同样是基于 TreeMap 实现的。在《Java 提高篇（二七）—–TreeMap》中 LZ 详细讲解了 TreeMap 实现机制，如果客官详情看了这篇博文或者多 TreeMap 有比较详细的了解，那么 TreeSet 的实现对您是喝口水那么简单。

一、TreeSet 定义

我们知道 TreeMap 是一个有序的二叉树，那么同理 TreeSet 同样也是一个有序的，它的作用是提供有序的 Set 集合。通过源码我们知道 TreeSet 基础 AbstractSet，实现 NavigableSet、Cloneable、Serializable 接口。其中 AbstractSet 提供 Set 接口的骨干实现，从而最大限度地减少了实现此接口所需的工作。NavigableSet 是扩展的 SortedSet，具有了为给定搜索目标报告最接近匹配项的导航方法，这就意味着它支持一系列的导航方法。比如查找与指定目标最匹配项。Cloneable 支持克隆，Serializable 支持序列化。

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
```

同时在 TreeSet 中定义了如下几个变量。

```
private transient NavigableMap<E,Object> m;

//PRESENT会被当做Map的value与key构建成键值对
 private static final Object PRESENT = new Object();
```

其构造方法：

```
//默认构造方法，根据其元素的自然顺序进行排序
    public TreeSet() {
        this(new TreeMap<E,Object>());
    }

    //构造一个包含指定 collection 元素的新 TreeSet，它按照其元素的自然顺序进行排序。
    public TreeSet(Comparator<? super E> comparator) {
            this(new TreeMap<>(comparator));
    }

    //构造一个新的空 TreeSet，它根据指定比较器进行排序。
    public TreeSet(Collection<? extends E> c) {
        this();
        addAll(c);
    }

    //构造一个与指定有序 set 具有相同映射关系和相同排序的新 TreeSet。
    public TreeSet(SortedSet<E> s) {
        this(s.comparator());
```

```
    addAll(s);
}

TreeSet(NavigableMap<E,Object> m) {
    this.m = m;
}
```

二、TreeSet 主要方法

1、add：将指定的元素添加到此 set（如果该元素尚未存在于 set 中）。

```
public boolean add(E e) {
        return m.put(e, PRESENT)==null;
    }
```

2、addAll：将指定 collection 中的所有元素添加到此 set 中。

```
public  boolean addAll(Collection<? extends E> c) {
        // Use linear-time version if applicable
        if (m.size()==0 && c.size() > 0 &&
            c instanceof SortedSet &&
            m instanceof TreeMap) {
            SortedSet<? extends E> set = (SortedSet<? extends E>) c;
            TreeMap<E,Object> map = (TreeMap<E, Object>) m;
            Comparator<? super E> cc = (Comparator<? super E>) set.comparator();
            Comparator<? super E> mc = map.comparator();
            if (cc==mc || (cc != null && cc.equals(mc))) {
                map.addAllForTreeSet(set, PRESENT);
                return true;
            }
        }
        return super.addAll(c);
    }
```

3、ceiling：返回此 set 中大于等于给定元素的最小元素；如果不存在这样的元素，则返回 null。

```
public E ceiling(E e) {
        return m.ceilingKey(e);
    }
```

4、clear：移除此 set 中的所有元素。

```
public void clear() {
        m.clear();
    }
```

5、clone：返回 TreeSet 实例的浅表副本。属于浅拷贝。

```
public Object clone() {
        TreeSet<E> clone = null;
        try {
            clone = (TreeSet<E>) super.clone();
        } catch (CloneNotSupportedException e) {
            throw new InternalError();
        }

        clone.m = new TreeMap<>(m);
        return clone;
    }
```

6、comparator：返回对此 set 中的元素进行排序的比较器；如果此 set 使用其元素的自然顺序，则返回 null。

```
public Comparator<? super E> comparator() {
        return m.comparator();
    }
```

7、contains：如果此 set 包含指定的元素，则返回 true。

```
public boolean contains(Object o) {
        return m.containsKey(o);
    }
```

8、descendingIterator：返回在此 set 元素上按降序进行迭代的迭代器。

```
public Iterator<E> descendingIterator() {
         return m.descendingKeySet().iterator();
    }
```

9、descendingSet：返回此 set 中所包含元素的逆序视图。

```
public NavigableSet<E> descendingSet() {
        return new TreeSet<>(m.descendingMap());
    }
```

10、first：返回此 set 中当前第一个（最低）元素。

```
public E first() {
        return m.firstKey();
    }
```

11、floor：返回此 set 中小于等于给定元素的最大元素；如果不存在这样的元素，则返回 null。

```
public E floor(E e) {
        return m.floorKey(e);
    }
```

12、headSet：返回此 set 的部分视图，其元素严格小于 toElement。

```
public SortedSet<E> headSet(E toElement) {
        return headSet(toElement, false);
    }
```

13、higher：返回此 set 中严格大于给定元素的最小元素；如果不存在这样的元素，则返回 null。

```
public E higher(E e) {
        return m.higherKey(e);
    }
```

14、isEmpty：如果此 set 不包含任何元素，则返回 true。

```
public boolean isEmpty() {
        return m.isEmpty();
    }
```

15、iterator：返回在此 set 中的元素上按升序进行迭代的迭代器。

```
public Iterator<E> iterator() {
        return m.navigableKeySet().iterator();
    }
```

16、last：返回此 set 中当前最后一个（最高）元素。

```
public E last() {
        return m.lastKey();
    }
```

17、lower：返回此 set 中严格小于给定元素的最大元素；如果不存在这样的元素，则返回 null。

```
public E lower(E e) {
```

```
        return m.lowerKey(e);
    }
```

18、pollFirst：获取并移除第一个（最低）元素；如果此 set 为空，则返回 null。

```
public E pollFirst() {
        Map.Entry<E,?> e = m.pollFirstEntry();
        return (e == null) ? null : e.getKey();
    }
```

19、pollLast：获取并移除最后一个（最高）元素；如果此 set 为空，则返回 null。

```
public E pollLast() {
        Map.Entry<E,?> e = m.pollLastEntry();
        return (e == null) ? null : e.getKey();
    }
```

20、remove：将指定的元素从 set 中移除（如果该元素存在于此 set 中）。

```
public boolean remove(Object o) {
        return m.remove(o)==PRESENT;
    }
```

21、size：返回 set 中的元素数（set 的容量）。

```
public int size() {
        return m.size();
    }
```

22、subSet：返回此 set 的部分视图

```
/**
     * 返回此 set 的部分视图，其元素范围从 fromElement 到 toElement。
     */
     public NavigableSet<E> subSet(E fromElement, boolean fromInclusive,
            E toElement,   boolean toInclusive) {
            return new TreeSet<>(m.subMap(fromElement, fromInclusive,
                toElement,   toInclusive));
     }

     /**
      * 返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。
```

```
     */
    public SortedSet<E> subSet(E fromElement, E toElement) {
        return subSet(fromElement, true, toElement, false);
    }
```

23、tailSet：返回此 set 的部分视图

```
/**
     * 返回此 set 的部分视图，其元素大于（或等于，如果 inclusive 为 true）fromElement。
     */
    public NavigableSet<E> tailSet(E fromElement, boolean inclusive) {
        return new TreeSet<>(m.tailMap(fromElement, inclusive));
    }

    /**
     * 返回此 set 的部分视图，其元素大于等于 fromElement。
     */
    public SortedSet<E> tailSet(E fromElement) {
        return tailSet(fromElement, true);
    }
```

三、最后

由于 TreeSet 是基于 TreeMap 实现的，所以如果我们对 treeMap 有了一定的了解，对 TreeSet 那是小菜一碟，我们从 TreeSet 中的源码可以看出，其实现过程非常简单，几乎所有的方法实现全部都是基于 TreeMap 的。

29

详解 Java 定时任务

在我们编程过程中如果需要执行一些简单的定时任务，无须做复杂的控制，我们可以考虑使用 JDK 中的 Timer 定时任务来实现。下面 LZ 就其原理、实例以及 Timer 缺陷三个方面来解析 Java Timer 定时器。

一、简介

在 Java 中一个完整定时任务需要由 Timer、TimerTask 两个类来配合完成。 API 中是这样定义他们的，Timer：一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。由TimerTask：Timer 安排为一次执行或重复执行的任务。我们可以这样理解 **Timer 是一种定时器工具，用来在一个后台线程计划执行指定任务，而 TimerTask 一个抽象类，它的子类代表一个可以被 Timer 计划的任务。**

Timer 类

在工具类 Timer 中，提供了四个构造方法，每个构造方法都启动了计时器线程，同时 Timer 类可以保证多个线程可以共享单个 Timer 对象而无需进行外部同步，所以 Timer 类是线程安全的。但是由于每一个 Timer 对象对应的是单个后台线程，用于顺序执行所有的计时器任务，一般情况下我们的线程任务执行所消耗的时间应该非常短，但是由于特殊情况导致某个定时器任务执行的时间太长，那么他就会“独占”计时器的任务执行线程，其后的所有线程都必须等待它执行完，这就会延迟后续任务的执行，使这些任务堆积在一起，具体情况我们后面分析。

当程序初始化完成 Timer 后，定时任务就会按照我们设定的时间去执行， Timer 提供了 schedule 方法，该方法有多中重载方式来适应不同的情况，如下：

schedule(TimerTask task, Date time)：安排在指定的时间执行指定的任务。

schedule(TimerTask task, Date firstTime, long period)： 安排指定的任务在指定的时间开始进行重复的固定延迟执行。

schedule(TimerTask task, long delay)： 安排在指定延迟后执行指定的任务。

schedule(TimerTask task, long delay, long period)： 安排指定的任务从指定的延迟后开始进行重复的固定延迟执行。

同时也重载了 scheduleAtFixedRate 方法，scheduleAtFixedRate 方法与 schedule 相同，只不过他们的侧重点不同，区别后面分析。

scheduleAtFixedRate(TimerTask task, Date firstTime, long period)：安排指定的任务在指定的时间开始进行重复的固定速率执行。

scheduleAtFixedRate(TimerTask task, long delay, long period)：安排指定的任务在指定的延迟后开始进行重复的固定速率执行。

TimerTask

TimerTask 类是一个抽象类，由 Timer 安排为一次执行或重复执行的任务。它有一个抽象方法 run() 方法，该方法用于执行相应计时器任务要执行的操作。因此每一个具体的任务类都必须继承 TimerTask，然后重写 run() 方法。

另外它还有两个非抽象的方法：

boolean cancel()：取消此计时器任务。

long scheduledExecutionTime()：返回此任务最近实际执行的安排执行时间。

二、实例

2.1、指定延迟时间执行定时任务

```
public class TimerTest01 {
    Timer timer;
    public TimerTest01(int time){
        timer = new Timer();
        timer.schedule(new TimerTaskTest01(), time * 1000);
    }

    public static void main(String[] args) {
        System.out.println("timer begin....");
        new TimerTest01(3);
    }
}

public class TimerTaskTest01 extends TimerTask{

    public void run() {
        System.out.println("Time's up!!!!");
    }
}
```

运行结果：

```
首先打印：timer begin....

3秒后打印：Time's up!!!!
```

2.2、在指定时间执行定时任务

```
public class TimerTest02 {
    Timer timer;

    public TimerTest02(){
        Date time = getTime();
        System.out.println("指定时间time=" + time);
```

```
        timer = new Timer();
        timer.schedule(new TimerTaskTest02(), time);
    }

    public Date getTime(){
        Calendar calendar = Calendar.getInstance();
        calendar.set(Calendar.HOUR_OF_DAY, 11);
        calendar.set(Calendar.MINUTE, 39);
        calendar.set(Calendar.SECOND, 00);
        Date time = calendar.getTime();

        return time;
    }

    public static void main(String[] args) {
        new TimerTest02();
    }
}

public class TimerTaskTest02 extends TimerTask{

    @Override
    public void run() {
        System.out.println("指定时间执行线程任务...");
    }
}
```

当时间到达 11:39:00 时就会执行该线程任务，当然大于该时间也会执行！！执行结果为：

```
指定时间 time=Tue Jun 10 11:39:00 CST 2014
指定时间执行线程任务...
```

2.3、在延迟指定时间后以指定的间隔时间循环执行定时任务

```
public class TimerTest03 {
    Timer timer;

    public TimerTest03(){
        timer = new Timer();
        timer.schedule(new TimerTaskTest03(), 1000, 2000);
    }
```

```
    public static void main(String[] args) {
        new TimerTest03();
    }
}

public class TimerTaskTest03 extends TimerTask{

    @Override
    public void run() {
        Date date = new Date(this.scheduledExecutionTime());
        System.out.println("本次执行该线程的时间为：" + date);
    }
}
```

运行结果:

```
本次执行该线程的时间为：Tue Jun 10 21:19:47 CST 2014
本次执行该线程的时间为：Tue Jun 10 21:19:49 CST 2014
本次执行该线程的时间为：Tue Jun 10 21:19:51 CST 2014
本次执行该线程的时间为：Tue Jun 10 21:19:53 CST 2014
本次执行该线程的时间为：Tue Jun 10 21:19:55 CST 2014
本次执行该线程的时间为：Tue Jun 10 21:19:57 CST 2014
.................
```

对于这个线程任务,如果我们不将该任务停止,他会一直运行下去。

对于上面三个实例，LZ 只是简单的演示了一下，同时也没有讲解 scheduleAtFixedRate 方法的例子，其实该方法与 schedule 方法一样!

2.4、分析 schedule 和 scheduleAtFixedRate

1、schedule(TimerTask task, Date time)、schedule(TimerTask task, long delay)

对于这两个方法而言，如果指定的计划执行时间 scheduledExecutionTime<= systemCurrentTime，则 task 会被立即执行。scheduledExecutionTime 不会因为某一个 task 的过度执行而改变。

2、schedule(TimerTask task, Date firstTime, long period)、schedule(TimerTask task, long delay, long period)

这两个方法与上面两个就有点儿不同的，前面提过 Timer 的计时器任务会因为前一个任务执行时间较长而延时。在这两个方法中，每一次执行的 task 的计划时间会随着前一个 task 的实际时间而发生改变，也就是 sched

uledExecutionTime(n+1)=realExecutionTime(n)+periodTime。也就是说如果第 n 个 task 由于某种情况导致这次的执行时间过程，最后导致 systemCurrentTime>= scheduledExecutionTime(n+1)，这是第 n+1 个 task 并不会因为到时了而执行，他会等待第 n 个 task 执行完之后再执行，那么这样势必会导致 n+2 个的执行实现 scheduledExecutionTime 放生改变即 scheduledExecutionTime(n+2) = realExecutionTime(n+1)+periodTime。所以这两个方法更加注重保存间隔时间的稳定。

3、scheduleAtFixedRate(TimerTask task, Date firstTime, long period)、scheduleAtFixedRate(TimerTask task, long delay, long period)

在前面也提过 scheduleAtFixedRate 与 schedule 方法的侧重点不同，schedule 方法侧重保存间隔时间的稳定，而 scheduleAtFixedRate 方法更加侧重于保持执行频率的稳定。为什么这么说，原因如下。在 schedule 方法中会因为前一个任务的延迟而导致其后面的定时任务延时，而 scheduleAtFixedRate 方法则不会，如果第 n 个 task 执行时间过长导致 systemCurrentTime>= scheduledExecutionTime(n+1)，则不会做任何等待他会立即执行第 n+1 个 task，所以 scheduleAtFixedRate 方法执行时间的计算方法不同于 schedule，而是 scheduledExecutionTime(n)=firstExecuteTime +n*periodTime，该计算方法永远保持不变。所以 scheduleAtFixedRate 更加侧重于保持执行频率的稳定。

三、Timer 的缺陷

3.1、Timer 的缺陷

Timer 计时器可以定时（指定时间执行任务）、延迟（延迟 5 秒执行任务）、周期性地执行任务（每隔个 1 秒执行任务），但是，Timer 存在一些缺陷。首先 Timer 对调度的支持是基于绝对时间的，而不是相对时间，所以它对系统时间的改变非常敏感。其次 Timer 线程是不会捕获异常的，如果 TimerTask 抛出的了未检查异常则会导致 Timer 线程终止，同时 Timer 也不会重新恢复线程的执行，他会错误的认为整个 Timer 线程都会取消。同时，已经被安排单尚未执行的 TimerTask 也不会再执行了，新的任务也不能被调度。故如果 TimerTask 抛出未检查的异常，Timer 将会产生无法预料的行为。

1、Timer 管理时间延迟缺陷

前面 Timer 在执行定时任务时只会创建一个线程任务，如果存在多个线程，若其中某个线程因为某种原因而导致线程任务执行时间过长，超过了两个任务的间隔时间，会发生一些缺陷：

```
public class TimerTest04 {
    private Timer timer;
    public long start;

    public TimerTest04(){
        this.timer = new Timer();
        start = System.currentTimeMillis();
    }

    public void timerOne(){
        timer.schedule(new TimerTask() {
            public void run() {
                System.out.println("timerOne invoked ,the time:" + (System.currentTimeMillis() - start));
                try {
                    Thread.sleep(4000);   //线程休眠3000
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }, 1000);
    }
```

```
    public void timerTwo(){
        timer.schedule(new TimerTask() {
            public void run() {
                System.out.println("timerOne invoked ,the time:" + (System.currentTimeMillis() - start));
            }
        }, 3000);
    }

    public static void main(String[] args) throws   Exception {
        TimerTest04 test = new TimerTest04();

        test.timerOne();
        test.timerTwo();
    }
}
```

按照我们正常思路，timerTwo 应该是在 3s 后执行，其结果应该是：

```
timerOne invoked ,the time:1001
timerOne invoked ,the time:3001
```

但是事与愿违，timerOne 由于 sleep(4000)，休眠了 4S，同时 Timer 内部是一个线程，导致 timeOne 所需的时间超过了间隔时间，结果：

```
timerOne invoked ,the time:1000
timerOne invoked ,the time:5000
```

2、Timer 抛出异常缺陷

如果 TimerTask 抛出 RuntimeException，Timer 会终止所有任务的运行。如下：

```
public class TimerTest04 {
    private Timer timer;

    public TimerTest04(){
        this.timer = new Timer();
    }

    public void timerOne(){
        timer.schedule(new TimerTask() {
            public void run() {
                throw new RuntimeException();
```

```
            }
        }, 1000);
    }

    public void timerTwo(){
        timer.schedule(new TimerTask() {

            public void run() {
                System.out.println("我会不会执行呢？？");
            }
        }, 1000);
    }

    public static void main(String[] args) {
        TimerTest04 test = new TimerTest04();
        test.timerOne();
        test.timerTwo();
    }
}
```

运行结果：timerOne 抛出异常，导致 timerTwo 任务终止。

```
Exception in thread "Timer-0" java.lang.RuntimeException
    at com.chenssy.timer.TimerTest04$1.run(TimerTest04.java:25)
    at java.util.TimerThread.mainLoop(Timer.java:555)
    at java.util.TimerThread.run(Timer.java:505)
```

对于 Timer 的缺陷，我们可以考虑 ScheduledThreadPoolExecutor 来替代。Timer 是基于绝对时间的，对系统时间比较敏感，而 ScheduledThreadPoolExecutor 则是基于相对时间；Timer 是内部是单一线程，而 ScheduledThreadPoolExecutor 内部是个线程池，所以可以支持多个任务并发执行。

3.2、用 ScheduledExecutorService 替代 Timer

1、解决问题一：

```
public class ScheduledExecutorTest {
    private  ScheduledExecutorService scheduExec;

    public long start;

    ScheduledExecutorTest(){
        this.scheduExec =  Executors.newScheduledThreadPool(2);
```

```
        this.start = System.currentTimeMillis();
    }

    public void timerOne(){
        scheduExec.schedule(new Runnable() {
            public void run() {
                System.out.println("timerOne,the time:" + (System.currentTimeMillis() - start));
                try {
                    Thread.sleep(4000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        },1000,TimeUnit.MILLISECONDS);
    }

    public void timerTwo(){
        scheduExec.schedule(new Runnable() {
            public void run() {
                System.out.println("timerTwo,the time:" + (System.currentTimeMillis() - start));
            }
        },2000,TimeUnit.MILLISECONDS);
    }

    public static void main(String[] args) {
        ScheduledExecutorTest test = new ScheduledExecutorTest();
        test.timerOne();
        test.timerTwo();
    }
}
```

运行结果：

```
timerOne,the time:1003
timerTwo,the time:2005
```

2、解决问题二

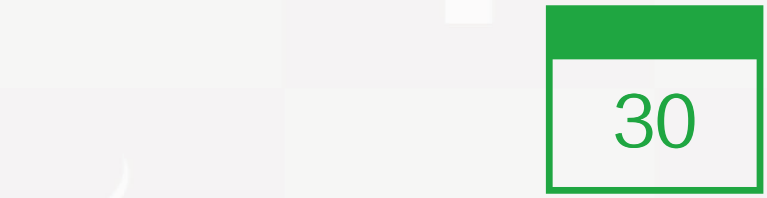
```
public class ScheduledExecutorTest {
    private  ScheduledExecutorService scheduExec;

    public long start;
```

```
    ScheduledExecutorTest(){
        this.scheduExec =  Executors.newScheduledThreadPool(2);
        this.start = System.currentTimeMillis();
    }

    public void timerOne(){
        scheduExec.schedule(new Runnable() {
            public void run() {
                throw new RuntimeException();
            }
        },1000,TimeUnit.MILLISECONDS);
    }

    public void timerTwo(){
        scheduExec.scheduleAtFixedRate(new Runnable() {
            public void run() {
                System.out.println("timerTwo invoked .....");
            }
        },2000,500,TimeUnit.MILLISECONDS);
    }

    public static void main(String[] args) {
        ScheduledExecutorTest test = new ScheduledExecutorTest();
        test.timerOne();
        test.timerTwo();
    }
}
```

运行结果：

```
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
timerTwo invoked .....
.....................
```

参考文献:http://blog.csdn.net/lmj623565791/article/details/27109467



30
Vector

在 Java提高篇（二一）—ArrayList 、Java 提高篇（二二）—LinkedList ，详细讲解了 ArrayList、linkedList 的原理和实现过程，对于 List 接口这里还介绍一个它的实现类 Vector，Vector 类可以实现可增长的对象数组。

一、Vector 简介

Vector 可以实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。不过，Vector 的大小是可以增加或者减小的，以便适应创建 Vector 后进行添加或者删除操作。

Vector 实现 List 接口，继承 AbstractList 类，所以我们可以将其看做队列，支持相关的添加、删除、修改、遍历等功能。

Vector 实现 RandmoAccess 接口，即提供了随机访问功能，提供提供快速访问功能。在 Vector 我们可以直接访问元素。

Vector 实现了 Cloneable 接口，支持 clone() 方法，可以被克隆。

```
public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

Vector 提供了四个构造函数：

```
/**
     * 构造一个空向量，使其内部数据数组的大小为 10，其标准容量增量为零。
     */
     public Vector() {
            this(10);
     }

    /**
     * 构造一个包含指定 collection 中的元素的向量，这些元素按其 collection 的迭代器返回元素的顺序排列。
     */
    public Vector(Collection<? extends E> c) {
        elementData = c.toArray();
        elementCount = elementData.length;
        // c.toArray might (incorrectly) not return Object[] (see 6260652)
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, elementCount,
                    Object[].class);
    }

    /**
     * 使用指定的初始容量和等于零的容量增量构造一个空向量。
     */
```

```
 public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}

/**
 *  使用指定的初始容量和容量增量构造一个空的向量。
 */
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    this.elementData = new Object [initialCapacity];
    this.capacityIncrement = capacityIncrement;
}
```

在成员变量方面，Vector 提供了 elementData , elementCount， capacityIncrement 三个成员变量。其中

elementData：”Object[] 类型的数组”，它保存了 Vector 中的元素。按照 Vector 的设计 elementData 为一个动态数组，可以随着元素的增加而动态的增长，其具体的增加方式后面提到（ensureCapacity 方法）。如果在初始化 Vector 时没有指定容器大小，则使用默认大小为 10.

elementCount：`Vector` 对象中的有效组件数。

capacityIncrement：向量的大小大于其容量时，容量自动增加的量。如果在创建 Vector 时，指定了 capacityIncrement 的大小；则，每次当 Vector 中动态数组容量增加时>，增加的大小都是 capacityIncrement。如果容量的增量小于等于零，则每次需要增大容量时，向量的容量将增大一倍。

同时 Vector 是线程安全的!

二、源码解析

对于源码的解析，LZ 在这里只就增加（add）删除（remove）两个方法进行讲解。

2.1增加：add(E e)

add(E e)：将指定元素添加到此向量的末尾。

```
public synchronized boolean add(E e) {
        modCount++;
        ensureCapacityHelper(elementCount + 1);   //确认容器大小，如果操作容量则扩容操作
        elementData[elementCount++] = e;   //将e元素添加至末尾
        return true;
    }
```

这个方法相对而言比较简单，具体过程就是先确认容器的大小，看是否需要进行扩容操作，然后将E元素添加到此向量的末尾。

```
private void ensureCapacityHelper(int minCapacity) {
        //如果
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

    /**
     * 进行扩容操作
     * 如果此向量的当前容量小于minCapacity，则通过将其内部数组替换为一个较大的数组俩增加其容量。
     * 新数据数组的大小姜维原来的大小 + capacityIncrement，
     * 除非 capacityIncrement 的值小于等于零，在后一种情况下，新的容量将为原来容量的两倍，不过，如果此大小仍然小于 minC
     */
    private void grow(int minCapacity) {
        int oldCapacity = elementData.length;     //当前容器大小
        /*
         * 新容器大小
         * 若容量增量系数(capacityIncrement) > 0，则将容器大小增加到capacityIncrement
         * 否则将容量增加一倍
         */
        int newCapacity = oldCapacity + ((capacityIncrement > 0) ?
                                         capacityIncrement : oldCapacity);
```

```
        if (newCapacity - minCapacity < 0)
            newCapacity = minCapacity;

        if (newCapacity - MAX_ARRAY_SIZE > 0)
            newCapacity = hugeCapacity(minCapacity);

        elementData = Arrays.copyOf(elementData, newCapacity);
    }

    /**
     * 判断是否超出最大范围
     * MAX_ARRAY_SIZE：private static final int  MAX_ARRAY_SIZE = Integer.MAX_VALUE - 8;
     */
    private static int hugeCapacity(int minCapacity) {
        if (minCapacity < 0)
            throw new OutOfMemoryError();
        return (minCapacity > MAX_ARRAY_SIZE) ? Integer.MAX_VALUE : MAX_ARRAY_SIZE;
    }
```

对于 Vector 整个的扩容过程，就是根据 capacityIncrement 确认扩容大小的，若 capacityIncrement <= 0 则扩大一倍，否则扩大至 capacityIncrement 。当然这个容量的最大范围为 Integer.MAX_VALUE即，2^32 - 1，所以 Vector 并不是可以无限扩充的。

2.2、remove(Object o)

```
/**
     * 从Vector容器中移除指定元素E
     */
    public boolean remove(Object o) {
        return removeElement(o);
    }

    public synchronized boolean removeElement(Object obj) {
        modCount++;
        int i = indexOf(obj);   //计算obj在Vector容器中位置
        if (i >= 0) {
            removeElementAt(i);   //移除
            return true;
        }
        return false;
    }
```

```
public synchronized void removeElementAt(int index) {
    modCount++;    //修改次数+1
    if (index >= elementCount) {   //删除位置大于容器有效大小
        throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
    }
    else if (index < 0) {    //位置小于 < 0
        throw new ArrayIndexOutOfBoundsException(index);
    }
    int j = elementCount - index - 1;
    if (j > 0) {
        //从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。
        //也就是数组元素从j位置往前移
        System.arraycopy(elementData, index + 1, elementData, index, j);
    }
    elementCount--;  //容器中有效组件个数 - 1
    elementData[elementCount] = null;   //将向量的末尾位置设置为null
}
```

因为 Vector 底层是使用数组实现的，所以它的操作都是对数组进行操作，只不过其是可以随着元素的增加而动态的改变容量大小，其实现方法是是使用 Arrays.copyOf 方法将旧数据拷贝到一个新的大容量数组中。 Vector 的整个内部实现都比较简单，这里就不在重述了。

三、Vector 遍历

Vector 支持 4 种遍历方式。

3.1、随机访问

因为 Vector 实现了 RandmoAccess 接口，可以通过下标来进行随机访问。

```
for(int i = 0 ; i < vec.size() ; i++){
      value = vec.get(i);
   }
```

3.2、迭代器

```
Iterator it = vec.iterator();
   while(it.hasNext()){
      value = it.next();
      //do something
   }
```

3.3、for 循环

```
for(Integer value:vec){
      //do something
   }
```

3.4、Enumeration 循环

```
Vector vec = new Vector<>();
   Enumeration enu = vec.elements();
   while (enu.hasMoreElements()) {
      value = (Integer)enu.nextElement();
   }
```

31

Iterator

迭代对于我们搞 Java 的来说绝对不陌生。我们常常使用 JDK 提供的迭代接口进行 Java 集合的迭代。

```
Iterator iterator = list.iterator();
       while(iterator.hasNext()){
          String string = iterator.next();
          //do something
       }
```

迭代其实我们可以简单地理解为遍历，是一个标准化遍历各类容器里面的所有对象的方法类，它是一个很典型的设计模式。Iterator 模式是用于遍历集合类的标准访问方法。它可以把访问逻辑从不同类型的集合类中抽象出来，从而避免向客户端暴露集合的内部结构。 在没有迭代器时我们都是这么进行处理的。如下：

对于数组我们是使用下标来进行处理的:

```
int[] arrays = new int[10];
  for(int i = 0 ; i < arrays.length ; i++){
     int a = arrays[i];
     //do something
  }
```

对于 ArrayList 是这么处理的:

```
List<String> list = new ArrayList<String>();
  for(int i = 0 ; i < list.size() ;  i++){
    String string = list.get(i);
    //do something
  }
```

对于这两种方式，我们总是都事先知道集合的内部结构，访问代码和集合本身是紧密耦合的，无法将访问逻辑从集合类和客户端代码中分离出来。同时每一种集合对应一种遍历方法，客户端代码无法复用。 在实际应用中如何需要将上面将两个集合进行整合是相当麻烦的。所以为了解决以上问题， Iterator 模式腾空出世，它总是用同一种逻辑来遍历集合。使得客户端自身不需要来维护集合的内部结构，所有的内部状态都由 Iterator 来维护。客户端从不直接和集合类打交道，它总是控制 Iterator，向它发送”向前”，”向后”，”取当前元素”的命令，就可以间接遍历整个集合。

上面只是对 Iterator 模式进行简单的说明，下面我们看看 Java 中 Iterator 接口，看他是如何来进行实现的。

一、java.util.Iterator

在 Java 中 Iterator 为一个接口，它只提供了迭代了基本规则，在 JDK 中他是这样定义的：对 collection 进行迭代的迭代器。迭代器取代了 Java Collections Framework 中的 Enumeration。迭代器与枚举有两点不同：

1、迭代器允许调用者利用定义良好的语义在迭代期间从迭代器所指向的 collection 移除元素。

2、方法名称得到了改进。

其接口定义如下：

```
public interface Iterator {
        boolean hasNext();
        Object next();
        void remove();
}
```

其中：

Object next()：返回迭代器刚越过的元素的引用，返回值是 Object，需要强制转换成自己需要的类型

boolean hasNext()：判断容器内是否还有可供访问的元素

void remove()：删除迭代器刚越过的元素

对于我们而言，我们只一般只需使用 next()、hasNext() 两个方法即可完成迭代。如下：

```
for(Iterator it = c.iterator(); it.hasNext(); ) {
        Object o = it.next();
         //do something
}
```

前面阐述了 Iterator 有一个很大的优点,就是我们不必知道集合的内部结果,集合的内部结构、状态由 Iterator 来维持，通过统一的方法 hasNext()、next() 来判断、获取下一个元素，至于具体的内部实现我们就不用关心了。但是作为一个合格的程序员我们非常有必要来弄清楚 Iterator 的实现。下面就 ArrayList 的源码进行分析分析。

二、各个集合的 Iterator 的实现

下面就 ArrayList 的 Iterator 实现来分析，其实如果我们理解了 ArrayList、Hashset、TreeSet 的数据结构，内部实现，对于他们是如何实现 Iterator 也会胸有成竹的。因为 ArrayList 的内部实现采用数组，所以我们只需要记录相应位置的索引即可，其方法的实现比较简单。

2.1、ArrayList 的 Iterator 实现

在 ArrayList 内部首先是定义一个内部类 Itr，该内部类实现 Iterator 接口，如下：

```
private class Itr implements Iterator<E> {
    //do something
}
```

而 ArrayList 的 iterator() 方法实现：

```
public Iterator<E> iterator() {
        return new Itr();
    }
```

所以通过使用 ArrayList.iterator() 方法返回的是 Itr() 内部类，所以现在我们需要关心的就是 Itr() 内部类的实现：

在 Itr 内部定义了三个 int 型的变量：cursor、lastRet、expectedModCount。其中 cursor 表示下一个元素的索引位置，lastRet 表示上一个元素的索引位置

```
int cursor;
        int lastRet = -1;
        int expectedModCount = modCount;
```

从 cursor、lastRet 定义可以看出，lastRet 一直比 cursor 少一所以 hasNext() 实现方法异常简单，只需要判断 cursor 和 lastRet 是否相等即可。

```
public boolean hasNext() {
            return cursor != size;
        }
```

对于 next() 实现其实也是比较简单的，只要返回 cursor 索引位置处的元素即可，然后修改 cursor、lastRet 即可，

```
public E next() {
        checkForComodification();
        int i = cursor;   //记录索引位置
        if (i >= size)   //如果获取元素大于集合元素个数，则抛出异常
            throw new NoSuchElementException();
        Object[] elementData =  ArrayList.this.elementData;
        if (i >= elementData.length)
            throw new ConcurrentModificationException();
        cursor = i + 1;     //cursor + 1
        return (E) elementData[lastRet = i];  //lastRet + 1 且返回cursor处元素
    }
```

checkForComodification() 主要用来判断集合的修改次数是否合法，即用来判断遍历过程中集合是否被修改过。在 Java 提高篇（二一）—- ArrayList 中已经阐述了。modCount 用于记录 ArrayList 集合的修改次数，初始化为 0，，每当集合被修改一次（结构上面的修改，内部update不算），如 add、remove 等方法，modCount + 1，所以如果 modCount 不变，则表示集合内容没有被修改。该机制主要是用于实现 ArrayList 集合的快速失败机制，在 Java 的集合中，较大一部分集合是存在快速失败机制的，这里就不多说，后面会讲到。所以要保证在遍历过程中不出错误，我们就应该保证在遍历过程中不会对集合产生结构上的修改（当然 remove 方法除外），出现了异常错误，我们就应该认真检查程序是否出错而不是 catch 后不做处理。

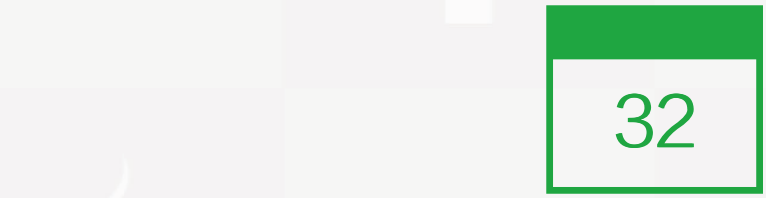
```
final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
```

对于 remove() 方法的是实现，它是调用 ArrayList 本身的 remove() 方法删除 lastRet 位置元素，然后修改 modCount 即可。

```
public void remove() {
        if (lastRet < 0)
            throw new IllegalStateException();
        checkForComodification();

        try {
            ArrayList.this.remove(lastRet);
            cursor = lastRet;
            lastRet = -1;
```

```
            expectedModCount = modCount;
        } catch (IndexOutOfBoundsException ex) {
            throw new ConcurrentModificationException();
        }
    }
```

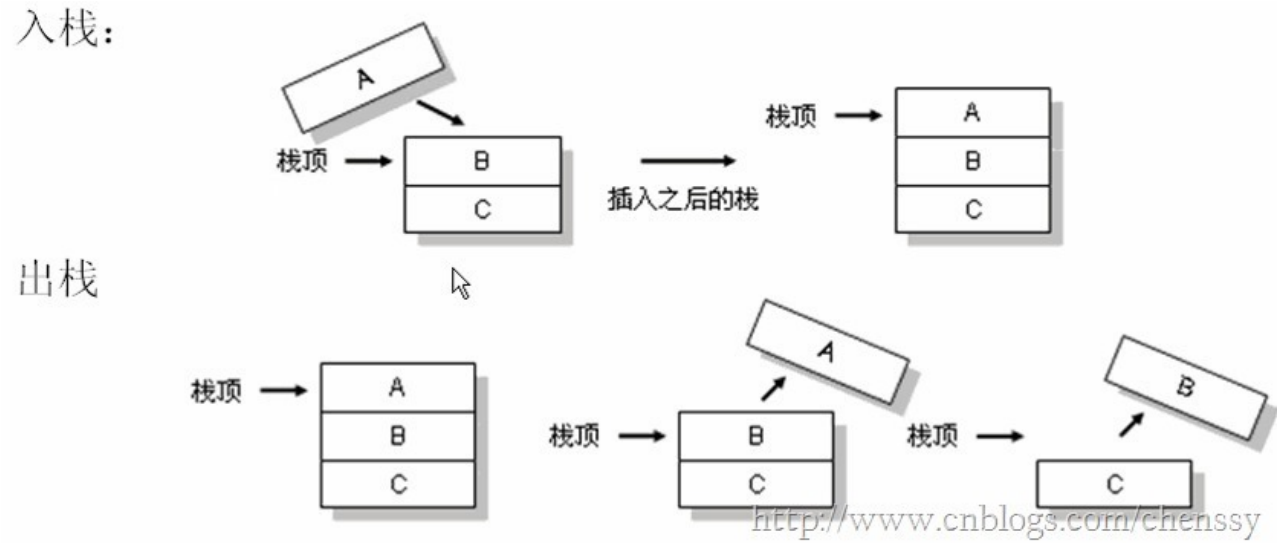
这里就对 ArrayList 的 Iterator 实现讲解到这里，对于 Hashset、TreeSet 等集合的 Iterator 实现，各位如果感兴趣可以继续研究，个人认为在研究这些集合的源码之前，有必要对该集合的数据结构有清晰的认识，这样会达到事半功倍的效果！！！！



32

Stack

在 Java 中 Stack 类表示后进先出（LIFO）的对象堆栈。栈是一种非常常见的数据结构，它采用典型的先进后出的操作方式完成的。每一个栈都包含一个栈顶，每次出栈是将栈顶的数据取出，如下：



图片 32.1 fig.1

Stack 通过五个操作对 Vector 进行扩展，允许将向量视为堆栈。这个五个操作如下：

操作	说明
empty()	测试堆栈是否为空。
peek()	查看堆栈顶部的对象，但不从堆栈中移除它。
pop()	移除堆栈顶部的对象，并作为此函数的值返回该对象。
push(E item)	把项压入堆栈顶部。
search(Object o)	返回对象在堆栈中的位置，以 1 为基数。

Stack 继承 Vector，他对 Vector 进行了简单的扩展：

```
public class Stack<E> extends Vector<E>
```

Stack 的实现非常简单，仅有一个构造方法，五个实现方法（从Vector继承而来的方法不算与其中），同时其实现的源码非常简单

```
/**
     * 构造函数
     */
    public Stack() {
    }

    /**
     *  push函数：将元素存入栈顶
     */
```

```
public E push(E item) {
    // 将元素存入栈顶。
    // addElement()的实现在Vector.java中
    addElement(item);

    return item;
}

/**
 * pop函数：返回栈顶元素，并将其从栈中删除
 */
public synchronized E pop() {
    E    obj;
    int    len = size();

    obj = peek();
    // 删除栈顶元素，removeElementAt()的实现在Vector.java中
    removeElementAt(len - 1);

    return obj;
}

/**
 * peek函数：返回栈顶元素，不执行删除操作
 */
public synchronized E peek() {
    int    len = size();

    if (len == 0)
        throw new EmptyStackException();
    // 返回栈顶元素，elementAt()具体实现在Vector.java中
    return elementAt(len - 1);
}

/**
 * 栈是否为空
 */
public boolean empty() {
    return size() == 0;
}

/**
 *  查找“元素o”在栈中的位置：由栈底向栈顶方向数
 */
public synchronized int search(Object o) {
```

```
        // 获取元素索引，elementAt()具体实现在Vector.java中
        int i = lastIndexOf(o);

        if (i >= 0) {
            return size() - i;
        }
        return -1;
    }
```

33

List 总结

前面 LZ 已经充分介绍了有关于 List 接口的大部分知识，如 ArrayList、LinkedList、Vector、Stack，通过这几个知识点可以对 List 接口有了比较深的了解了。只有通过归纳总结的知识才是你的知识。所以下面 LZ 就 List 接口做一个总结。推荐阅读：

Java提高篇（二一）—-ArrayList

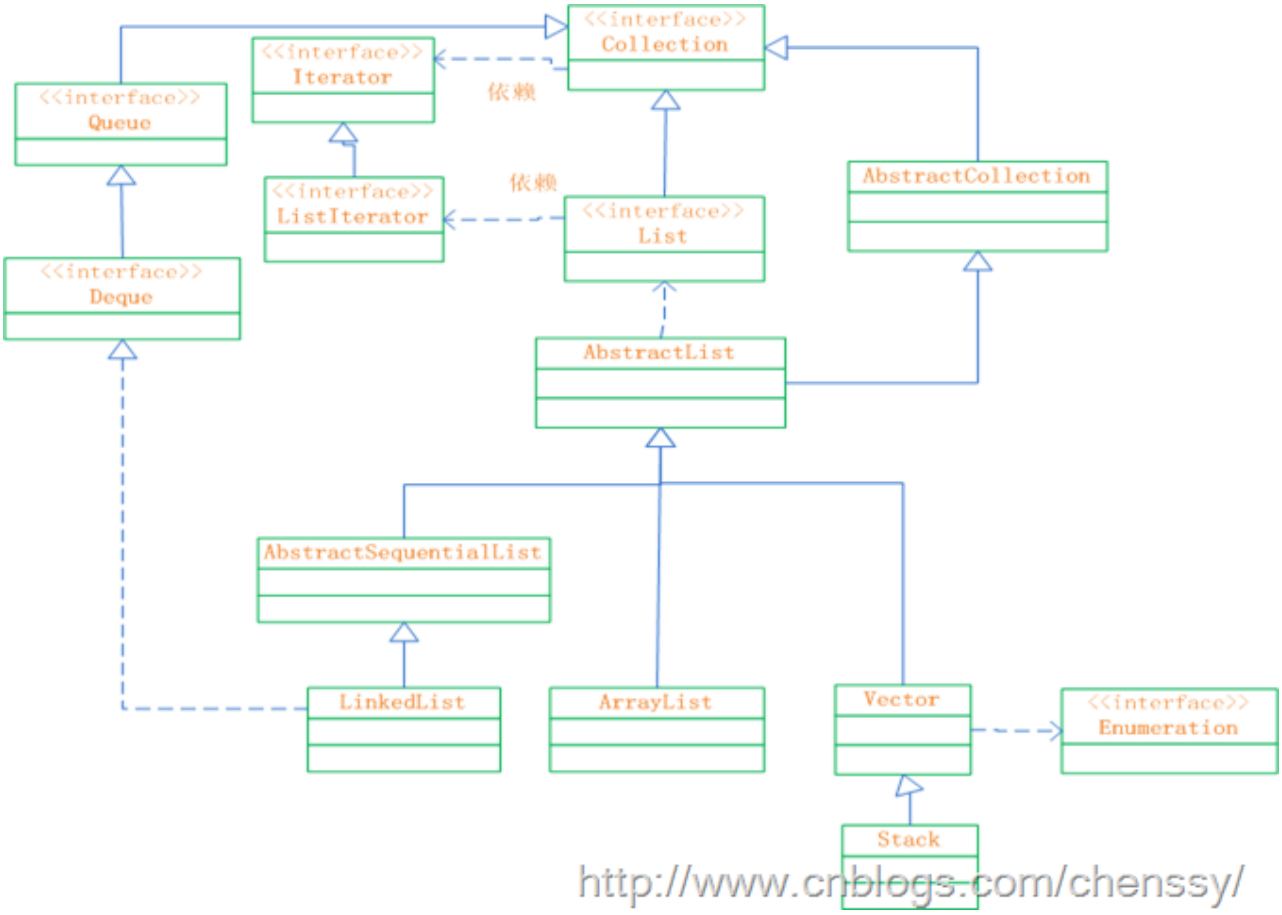
Java提高篇（二二）—-LinkedList

Java提高篇（二九）—-Vector

Java提高篇（三一）—-Stack

一、List 接口概述

List 接口，成为有序的 Collection 也就是序列。该接口可以对列表中的每一个元素的插入位置进行精确的控制，同时用户可以根据元素的整数索引（在列表中的位置）访问元素，并搜索列表中的元素。 下图是 List 接口的框架图：



图片 33.1 fig.1

通过上面的框架图，可以对 List 的结构了然于心，其各个类、接口如下：

Collection：Collection 层次结构 中的根接口。它表示一组对象，这些对象也称为 collection 的元素。对于 Collection 而言，它不提供任何直接的实现，所有的实现全部由它的子类负责。

AbstractCollection：提供 Collection 接口的骨干实现，以最大限度地减少了实现此接口所需的工作。对于我们而言要实现一个不可修改的 collection，只需扩展此类，并提供 iterator 和 size 方法的实现。但要实现可修改的 collection，就必须另外重写此类的 add 方法（否则，会抛出 UnsupportedOperationException），iterator 方法返回的迭代器还必须另外实现其 remove 方法。

Iterator：迭代器。

ListIterator：系列表迭代器，允许程序员按任一方向遍历列表、迭代期间修改列表，并获得迭代器在列表中的当前位置。

List：继承于 Collection 的接口。它代表着有序的队列。

AbstractList：List 接口的骨干实现，以最大限度地减少实现“随机访问”数据存储（如数组）支持的该接口所需的工作。

Queue：队列。提供队列基本的插入、获取、检查操作。

Deque：一个线性 collection，支持在两端插入和移除元素。大多数 Deque 实现对于它们能够包含的元素数没有固定限制，但此接口既支持有容量限制的双端队列，也支持没有固定大小限制的双端队列。

AbstractSequentialList：提供了 List 接口的骨干实现，从而最大限度地减少了实现受“连续访问”数据存储（如链接列表）支持的此接口所需的工作。从某种意义上说，此类与在列表的列表迭代器上实现“随机访问”方法。

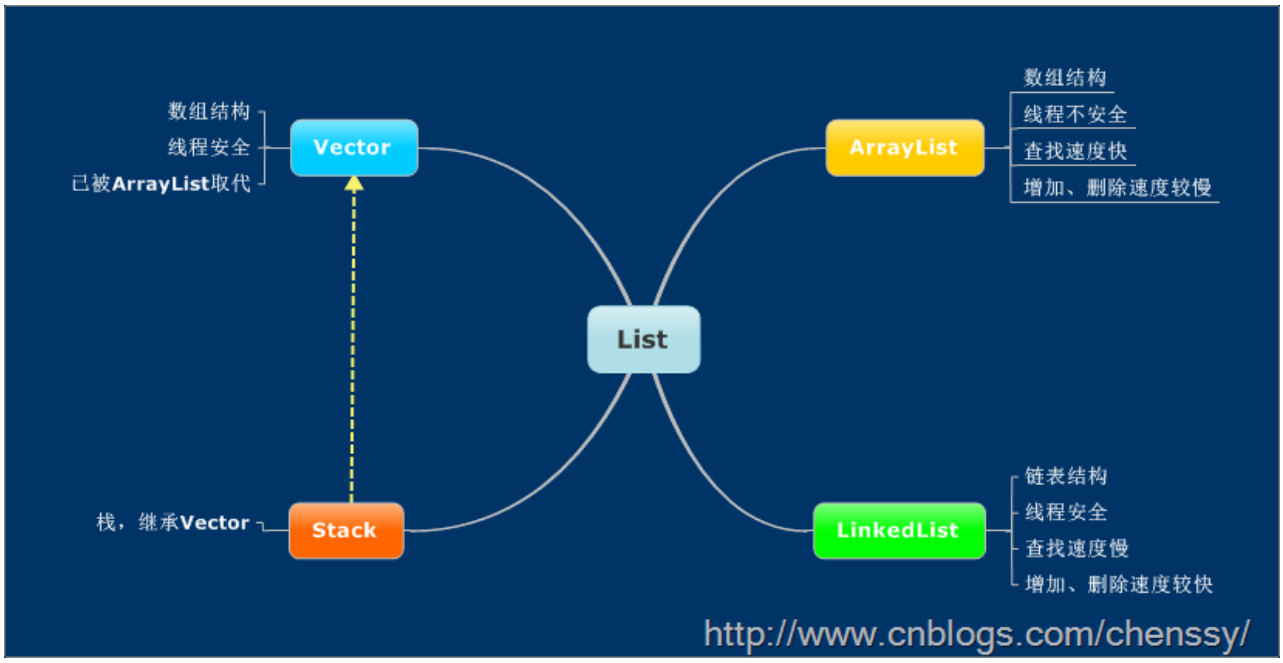
LinkedList：List 接口的链接列表实现。它实现所有可选的列表操作。

ArrayList：List 接口的大小可变数组的实现。它实现了所有可选列表操作，并允许包括 null 在内的所有元素。除了实现 List 接口外，此类还提供一些方法来操作内部用来存储列表的数组的大小。

Vector：实现可增长的对象数组。与数组一样，它包含可以使用整数索引进行访问的组件。

Stack：后进先出（LIFO）的对象堆栈。它通过五个操作对类 Vector 进行了扩展 ，允许将向量视为堆栈。

Enumeration：枚举，实现了该接口的对象，它生成一系列元素，一次生成一个。连续调用 nextElement 方法将返回一系列的连续元素。


数组结构
线程安全
已被ArrayList取代
Vector
ArrayList
数组结构
线程不安全
查找速度快
增加、删除速度较慢
List
栈，继承Vector
Stack
LinkedList
链表结构
线程安全
查找速度慢
增加、删除速度较快
http://www.cnblogs.com/chenssy/

图片 33.2 fig.2

二、使用场景

学习知识的根本目的就是使用它。每个知识点都有它的使用范围。集合也是如此，在 Java 中集合的家族非常庞大，每个成员都有最适合的使用场景。在刚刚接触 List 时，LZ 就说过**如果涉及到“栈”、“队列”、“链表”等操作，请优先考虑用 List。**至于是那个 List 则分如下：

1、对于需要快速插入、删除元素，则需使用 LinkedList。

2、对于需要快速访问元素，则需使用 ArrayList。

3、对于“单线程环境”或者“多线程环境，但是 List 仅被一个线程操作”，需要考虑使用非同步的类，如果是“多线程环境，切 List 可能同时被多个线程操作”，考虑使用同步的类（如Vector）。

2.1ArrayList、LinkedList 性能分析

在 List 中我们使用最普遍的就是 LinkedList 和 ArrayList，同时我们也了解了他们两者之间的使用场景和区别。

```
public class ListTest {
    private static final int COUNT = 100000;

    private static ArrayList arrayList = new ArrayList<>();
    private static LinkedList linkedList = new LinkedList<>();
    private static Vector vector = new Vector<>();

    public static void insertToList(List list){
        long startTime = System.currentTimeMillis();

        for(int i = 0 ; i < COUNT ; i++){
            list.add(0,i);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("插入 " + COUNT + "元素" + getName(list) + "花费 " + (endTime - startTime) + " 毫秒");
    }

    public static void deleteFromList(List list){
        long startTime = System.currentTimeMillis();

        for(int i = 0 ; i < COUNT ; i++){
          list.remove(0);
```

```
        }

        long endTime = System.currentTimeMillis();
        System.out.println("删除" + COUNT + "元素" + getName(list) + "花费 " + (endTime - startTime) + " 毫秒");
    }

    public static void readList(List list){
        long startTime = System.currentTimeMillis();

        for(int i = 0 ; i < COUNT ; i++){
            list.get(i);
        }

        long endTime = System.currentTimeMillis();
        System.out.println("读取" + COUNT + "元素" +  getName(list) + "花费 " + (endTime - startTime) + " 毫秒");
    }

    private static String getName(List list) {
        String name = "";
        if(list instanceof ArrayList){
            name = "ArrayList";
        }
        else if(list instanceof LinkedList){
            name = "LinkedList";
        }
        else if(list instanceof Vector){
            name = "Vector";
        }
        return name;
    }

    public static void main(String[] args) {
        insertToList(arrayList);
        insertToList(linkedList);
        insertToList(vector);

        System.out.println("------------------------------------");

        readList(arrayList);
        readList(linkedList);
        readList(vector);

        System.out.println("------------------------------------");

        deleteFromList(arrayList);
```

```
        deleteFromList(linkedList);
        deleteFromList(vector);
    }
}
```

运行结果:

```
插入 100000元素ArrayList花费 3900 毫秒
插入 100000元素LinkedList花费 15 毫秒
插入 100000元素Vector花费 3933 毫秒
--------------------------------------
读取100000元素ArrayList花费 0 毫秒
读取100000元素LinkedList花费 8877 毫秒
读取100000元素Vector花费 16 毫秒
--------------------------------------
删除100000元素ArrayList花费 4618 毫秒
删除100000元素LinkedList花费 16 毫秒
删除100000元素Vector花费 4759 毫秒
```

从上面的运行结果我们可以清晰的看出 ArrayList、LinkedList、Vector 增加、删除、遍历的效率问题。下面我就插入方法 add(int index, E element),delete、get 方法各位如有兴趣可以研究研究。

首先我们先看三者之间的源码:

ArrayList

```
public void add(int index, E element) {
        rangeCheckForAdd(index);  //检查是否index是否合法

        ensureCapacityInternal(size + 1);  //扩容操作
        System.arraycopy(elementData, index, elementData, index + 1, size - index);   //数组拷贝
        elementData[index] = element;  //插入
        size++;
    }
```

rangeCheckForAdd、ensureCapacityInternal 两个方法没有什么影响，真正产生影响的是 System.arraycopy 方法，该方法是个 JNI 函数，是在 JVM 中实现的。声明如下:

```
public static native void arraycopy(Object src, int srcPos, Object dest, int destPos, int length);
```

目前 LZ 无法看到源码，具体的实现不是很清楚，不过 System.arraycopy 源码分析对其进行了比较清晰的分析。但事实上我们只需要了解该方法会移动 index 后面的所有元素即可，这就意味着 ArrayList 的 add(int index, E element) 方法会引起 index 位置之后所有元素的改变，这真是牵一处而动全身。

LinkedList

```
public void add(int index, E element) {
    checkPositionIndex(index);

    if (index == size)    //插入位置在末尾
        linkLast(element);
    else
        linkBefore(element, node(index));
}
```

该方法比较简单，插入位置在末尾则调用 linkLast 方法，否则调用 linkBefore 方法，其实 linkLast、linkBefore 都是非常简单的实现，就是在 index 位置插入元素，至于 index 具体为知则有 node 方法来解决，同时 node 对 index 位置检索还有一个加速作用，如下：

```
Node<E> node(int index) {
    if (index < (size >> 1)) {    //如果index 小于 size/2 则从头开始查找
        Node<E> x = first;
        for (int i = 0; i < index; i++)
            x = x.next;
        return x;
    } else {    //如果index 大于 size/2 则从尾部开始查找
        Node<E> x = last;
        for (int i = size - 1; i > index; i--)
            x = x.prev;
        return x;
    }
}
```

所以 linkedList 的插入动作比 ArrayList 动作快就在于两个方面。1：linkedList 不需要执行元素拷贝动作，没有牵一发而动全身的大动作。2:查找插入位置有加速动作即：若 index < 双向链表长度的 1/2，则从前向后查找; 否则，从后向前查找。

Vector

Vector 的实现机制和 ArrayList 一样，同样是使用动态数组来实现的，所以他们两者之间的效率差不多，add 的源码也一样，如下：

```
public void add(int index, E element) {
        insertElementAt(element, index);
    }

    public synchronized void insertElementAt(E obj, int index) {
        modCount++;
        if (index > elementCount) {
            throw new ArrayIndexOutOfBoundsException(index
                                        + " > " + elementCount);
        }
        ensureCapacityHelper(elementCount + 1);
        System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
        elementData[index] = obj;
        elementCount++;
    }
```

上面是针对 ArrayList、LinkedList、Vector 三者之间的 add（int index,E element) 方法的解释，解释了 LinkedList 的插入动作要比 ArrayList、Vector 的插入动作效率为什么要高出这么多！至于 delete、get 两个方法 LZ 就不多解释了。

同时 LZ 在写上面那个例子时发现了一个非常有趣的现象，就是 linkedList 在某些时候执行 add 方法时比 ArrayList 方法会更慢！至于在什么情况？为什么会慢 LZ 下篇博客解释，当然不知道这个情况各位是否也遇到过？？

2.2、Vector 和 ArrayList 的区别

图片 33.3 fig.3

34

Map 总结

在前面 LZ 详细介绍了 HashMap 、HashTable 、TreeMap 的实现方法，从数据结构、实现原理、源码分析三个方面进行阐述，对这个三个类应该有了比较清晰的了解,下面 LZ 就 Map 做一个简单的总结。

推荐阅读：

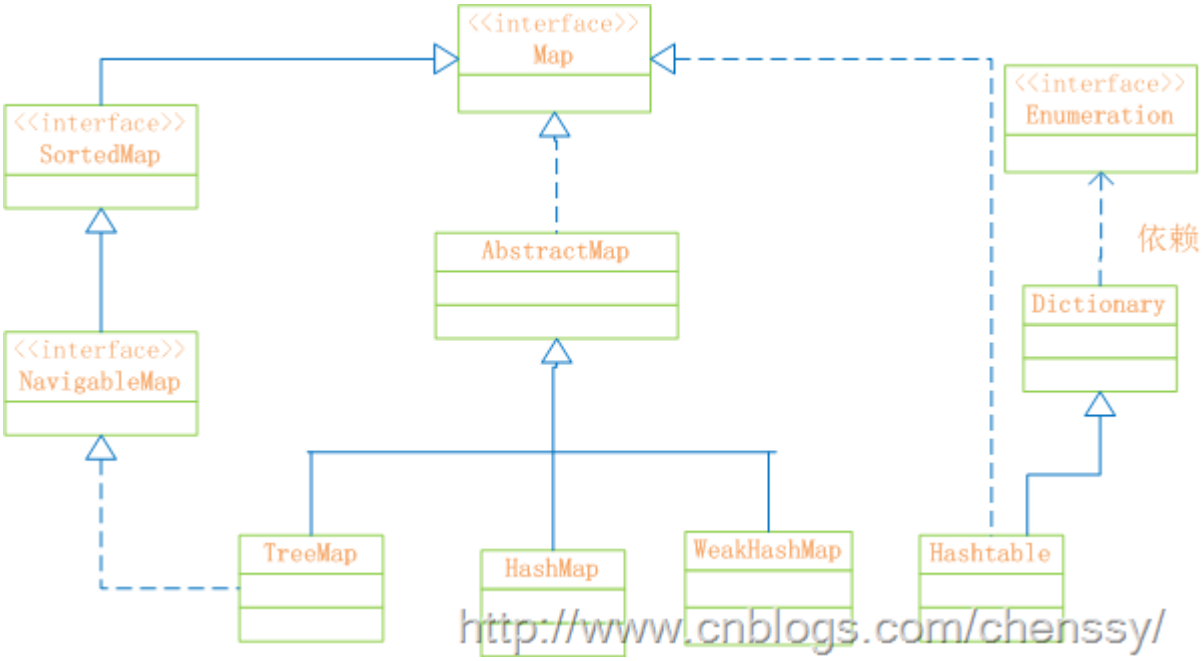
Java提高篇（二三）—-HashMap

Java提高篇（二五）—-HashTable

Java提高篇（二七）—-TreeMap

一、Map 概述

首先先看 Map 的结构示意图



图片 34.1 fig.1

Map：“键值”对映射的抽象接口。该映射不包括重复的键，一个键对应一个值。

SortedMap：有序的键值对接口，继承 Map 接口。

NavigableMap：继承 SortedMap，具有了针对给定搜索目标返回最接近匹配项的导航方法的接口。

AbstractMap：实现了 Map 中的绝大部分函数接口。它减少了“Map 的实现类”的重复编码。

Dictionary：任何可将键映射到相应值的类的抽象父类。目前被Map接口取代。

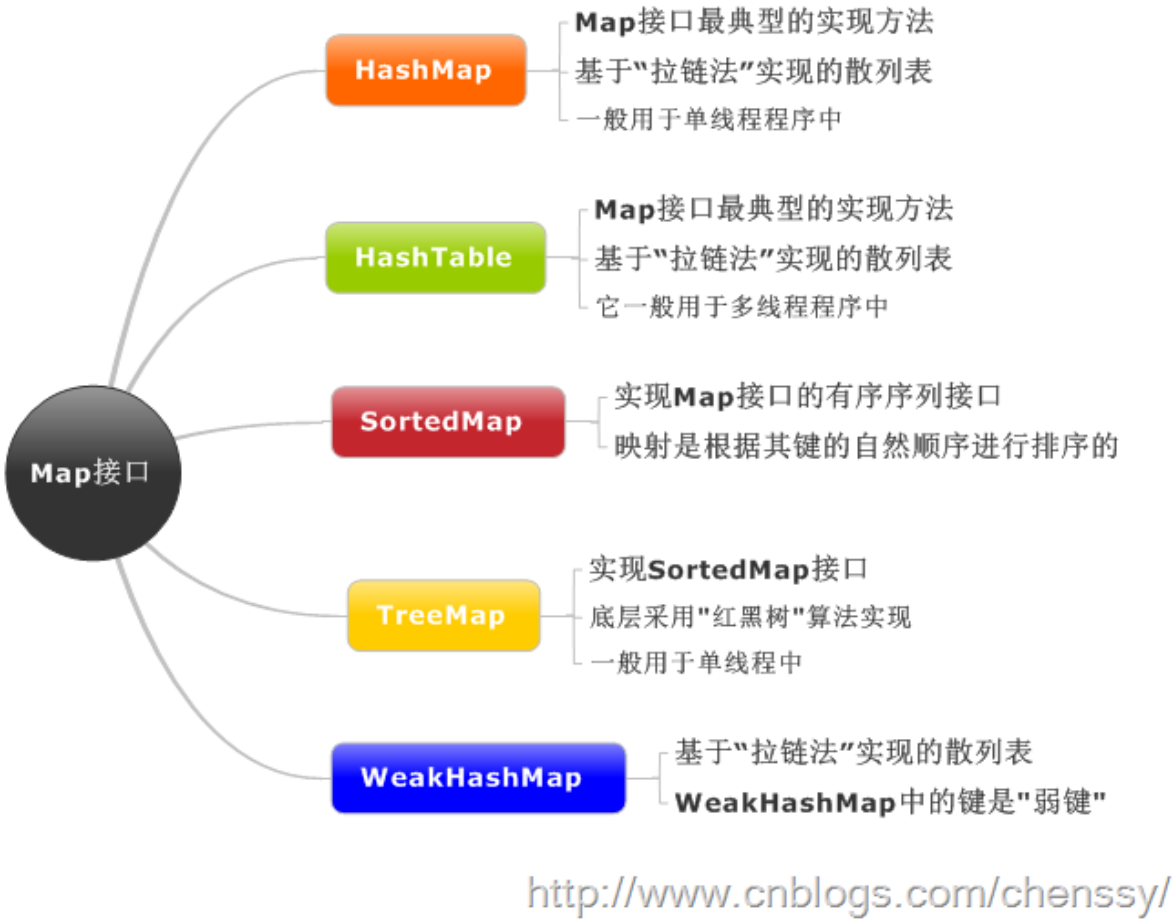
TreeMap：有序散列表，实现 SortedMap 接口，底层通过红黑树实现。

HashMap：是基于“拉链法”实现的散列表。底层采用“数组+链表”实现。

WeakHashMap：基于“拉链法”实现的散列表。

HashTable：基于“拉链法”实现的散列表。

总结如下：



图片 34.2 fig.2

他们之间的区别：


比较
HashMap与WeakHashMap
共同点
基于"拉链发"实现的散列表，存储的是"键值对"映射
都继承于AbstractMap，并且实现Map接口
"键"和"值"都允许为null
都是非同步
不同点
继承接口不同
HashMap实现了Cloneable和Serializable接口
WeakHashMap没有
键不同
HashMap的键是"强引用"
WeakHashMap的键是"弱引用"
HashMap与HashTable
共同点
都实现了Map、Cloneable、Serializable接口
底层都是基于"拉链法"实现
不同点
线程安全不同
HashMap单线程安全
HashTable多线程安全
null的处理不同
HashMap的key、value都可为null
Hashtable的key、value都不可为null
遍历
HashMap只支持Iterator
HashTable支持Iterator和Enumeration
基类不同
HashMap基于AbstractMap
HashTable基于Dictionary
http://www.cnblogs.com/chenssy/

图片 34.3 fig.3

二、内部哈希： 哈希映射技术

几乎所有通用 Map 都使用哈希映射技术。对于我们程序员来说我们必须要对其有所了解。

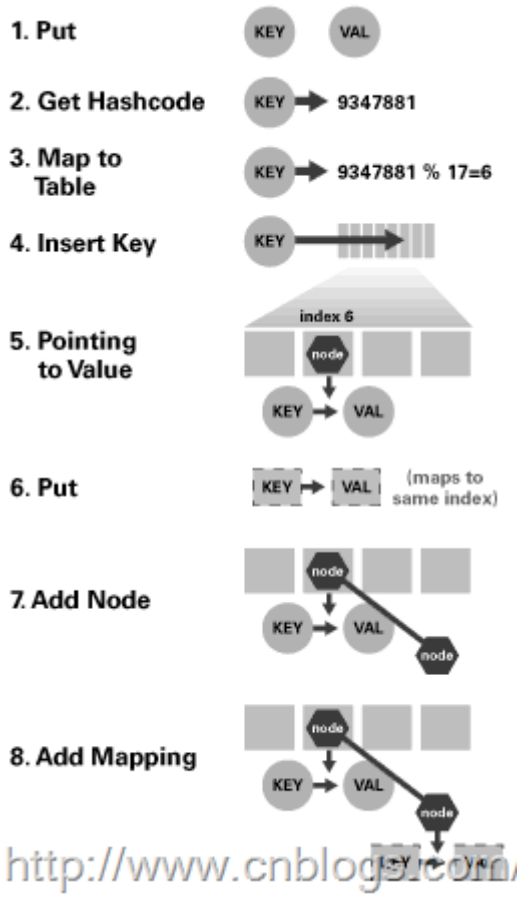
哈希映射技术是一种就元素映射到数组的非常简单的技术。由于哈希映射采用的是数组结果，那么必然存在一中用于确定任意键访问数组的索引机制，该机制能够提供一个小于数组大小的整数，我们将该机制称之为哈希函数。在 Java 中我们不必为寻找这样的整数而大伤脑筋，因为每个对象都必定存在一个返回整数值的 hashCode 方法，而我们需要做的就是将其转换为整数，然后再将该值除以数组大小取余即可。如下

```
int hashValue = Maths.abs(obj.hashCode()) % size;
```

下面是 HashMap、HashTable 的：

```
----------HashMap------------
//计算hash值
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}
//计算key的索引位置
static int indexFor(int h, int length) {
        return h & (length-1);
}
-----HashTable--------------
int index = (hash & 0x7FFFFFFF) % tab.length;    //确认该key的索引位置
```

位置的索引就代表了该节点在数组中的位置。下图是哈希映射的基本原理图：



图片 34.4 fig.4

在该图中 1-4 步骤是找到该元素在数组中位置，5-8 步骤是将该元素插入数组中。在插入的过程中会遇到一点点小挫折。在众多肯能存在多个元素他们的 hash 值是一样的，这样就会得到相同的索引位置，也就说多个元素会映射到相同的位置，这个过程我们称之为“冲突”。解决冲突的办法就是在索引位置处插入一个链接列表，并简单地将元素添加到此链接列表。当然也不是简单的插入，在 HashMap 中的处理过程如下：获取索引位置的链表，如果该链表为 null，则将该元素直接插入，否则通过比较是否存在与该 key 相同的 key，若存在则覆盖原来 key 的 value 并返回旧值，否则将该元素保存在链头（最先保存的元素放在链尾）。下面是 HashMap 的 put 方法，该方法详细展示了计算索引位置，将元素插入到适当的位置的全部过程：

```
public V put(K key, V value) {
        //当key为null，调用putForNullKey方法，保存null与table第一个位置中，这是HashMap允许为null的原因
        if (key == null)
            return putForNullKey(value);
        //计算key的hash值
        int hash = hash(key.hashCode());
        //计算key hash 值在 table 数组中的位置
        int i = indexFor(hash, table.length);
        //从i出开始迭代 e,判断是否存在相同的key
        for (Entry<K, V> e = table[i]; e != null; e = e.next) {
```

```
            Object k;
            //判断该条链上是否有hash值相同的(key相同)
            //若存在相同，则直接覆盖value，返回旧value
            if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
                V oldValue = e.value;   //旧值 = 新值
                e.value = value;
                e.recordAccess(this);
                return oldValue;     //返回旧值
            }
        }
        //修改次数增加1
        modCount++;
        //将key、value添加至i位置处
        addEntry(hash, key, value, i);
        return null;
    }
```

HashMap 的 put 方法展示了哈希映射的基本思想，其实如果我们查看其它的 Map，发现其原理都差不多！

三、Map 优化

首先我们这样假设，假设哈希映射的内部数组的大小只有1，所有的元素都将映射该位置（0），从而构成一条较长的链表。由于我们更新、访问都要对这条链表进行线性搜索，这样势必会降低效率。我们假设，如果存在一个非常大数组，每个位置链表处都只有一个元素，在进行访问时计算其 index 值就会获得该对象，这样做虽然会提高我们搜索的效率，但是它浪费了控件。诚然，虽然这两种方式都是极端的，但是它给我们提供了一种优化思路：**使用一个较大的数组让元素能够均匀分布。**在 Map 有两个会影响到其效率，一是容器的初始化大小、二是负载因子。

3.1、调整实现大小

在哈希映射表中，内部数组中的每个位置称作“存储桶”(bucket)，而可用的存储桶数（即内部数组的大小）称作容量 (capacity)，我们为了使 Map 对象能够有效地处理任意数的元素，将 Map 设计成可以调整自身的大小。我们知道当 Map 中的元素达到一定量的时候就会调整容器自身的大小，但是这个调整大小的过程其开销是非常大的。调整大小需要将原来所有的元素插入到新数组中。我们知道 index = hash(key) % length。这样可能会导致原先冲突的键不在冲突，不冲突的键现在冲突的，重新计算、调整、插入的过程开销是非常大的，效率也比较低下。所以，**如果我们开始知道 Map 的预期大小值，将 Map 调整的足够大，则可以大大减少甚至不需要重新调整大小，这很有可能会提高速度。**下面是 HashMap 调整容器大小的过程，通过下面的代码我们可以看到其扩容过程的复杂性：

```
void resize(int newCapacity) {
        Entry[] oldTable = table;   //原始容器
        int oldCapacity = oldTable.length;   //原始容器大小
        if (oldCapacity == MAXIMUM_CAPACITY) {    //是否超过最大值：1073741824
           threshold = Integer.MAX_VALUE;
           return;
        }

        //新的数组：大小为 oldCapacity * 2
        Entry[] newTable = new Entry[newCapacity];
        transfer(newTable, initHashSeedAsNeeded(newCapacity));
        table = newTable;
        /*
        * 重新计算阀值 =  newCapacity * loadFactor >  MAXIMUM_CAPACITY + 1 ?
        *                    newCapacity * loadFactor :MAXIMUM_CAPACITY + 1
        */
        threshold = (int)Math.min(newCapacity * loadFactor, MAXIMUM_CAPACITY + 1);
```

```
    }

    //将元素插入到新数组中
    void transfer(Entry[] newTable, boolean rehash) {
        int newCapacity = newTable.length;
        for (Entry<K,V> e : table) {
            while(null != e) {
                Entry<K,V> next = e.next;
                if (rehash) {
                    e.hash = null == e.key ? 0 : hash(e.key);
                }
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            }
        }
    }
```

3.2、负载因子

为了确认何时需要调整 Map 容器，Map 使用了一个额外的参数并且粗略计算存储容器的密度。在 Map 调整大小之前，使用”负载因子”来指示 Map 将会承担的“负载量”，也就是它的负载程度，当容器中元素的数量达到了这个“负载量”，则 Map 将会进行扩容操作。负载因子、容量、Map 大小之间的关系如下：负载因子 * 容量 > map 大小 —–>调整 Map 大小。

例如：如果负载因子大小为 0.75（HashMap 的默认值），默认容量为 11，则 11 * 0.75 = 8.25 = 8，所以当我们容器中插入第八个元素的时候，Map 就会调整大小。

负载因子本身就是在控件和时间之间的折衷。当我使用较小的负载因子时，虽然降低了冲突的可能性，使得单个链表的长度减小了，加快了访问和更新的速度，但是它占用了更多的控件，使得数组中的大部分控件没有得到利用，元素分布比较稀疏，同时由于 Map 频繁的调整大小，可能会降低性能。但是如果负载因子过大，会使得元素分布比较紧凑，导致产生冲突的可能性加大，从而访问、更新速度较慢。所以我们一般推荐不更改负载因子的值，采用默认值 0.75.

35

fail-fast 机制

在 JDK 的 Collection 中我们时常会看到类似于这样的话：

例如，ArrayList:

注意，迭代器的快速失败行为无法得到保证，因为一般来说，不可能对是否出现不同步并发修改做出任何硬性保证。快速失败迭代器会尽最大努力抛出 ConcurrentModificationException。因此，为提高这类迭代器的正确性而编写一个依赖于此异常的程序是错误的做法：迭代器的快速失败行为应该仅用于检测 bug。

HashMap 中：

注意，迭代器的快速失败行为不能得到保证，一般来说，存在非同步的并发修改时，不可能作出任何坚决的保证。快速失败迭代器尽最大努力抛出 ConcurrentModificationException。因此，编写依赖于此异常的程序的做法是错误的，正确做法是：迭代器的快速失败行为应该仅用于检测程序错误。

在这两段话中反复地提到”快速失败”。那么何为”快速失败”机制呢?

“快速失败”也就是 fail-fast，它是 Java 集合的一种错误检测机制。当多个线程对集合进行结构上的改变的操作时，有可能会产生 fail-fast 机制。记住是有可能，而不是一定。例如：假设存在两个线程（线程 1、线程 2），线程 1 通过 Iterator 在遍历集合 A 中的元素，在某个时候线程 2 修改了集合 A 的结构（是结构上面的修改，而不是简单的修改集合元素的内容），那么这个时候程序就会抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。

一、fail-fast 示例

```
public class FailFastTest {
    private static List<Integer> list = new ArrayList<>();

    /**
     * @desc:线程one迭代list
     * @Project:test
     * @file:FailFastTest.java
     * @Authro:chenssy
     * @data:2014年7月26日
     */
    private static class threadOne extends Thread{
        public void run() {
            Iterator<Integer> iterator = list.iterator();
            while(iterator.hasNext()){
                int i = iterator.next();
                System.out.println("ThreadOne 遍历:" + i);
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    /**
     * @desc:当i == 3时，修改list
     * @Project:test
     * @file:FailFastTest.java
     * @Authro:chenssy
     * @data:2014年7月26日
     */
    private static class threadTwo extends Thread{
        public void run(){
            int i = 0 ;
            while(i < 6){
                System.out.println("ThreadTwo run：" + i);
                if(i == 3){
                    list.remove(i);
                }
```

```
            i++;
          }
        }
    }

    public static void main(String[] args) {
        for(int i = 0 ; i < 10;i++){
            list.add(i);
        }
        new threadOne().start();
        new threadTwo().start();
    }
}
```

运行结果：

```
ThreadOne 遍历:0
ThreadTwo run：0
ThreadTwo run：1
ThreadTwo run：2
ThreadTwo run：3
ThreadTwo run：4
ThreadTwo run：5
Exception in thread "Thread-0" java.util.ConcurrentModificationException
    at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
    at java.util.ArrayList$Itr.next(Unknown Source)
    at test.ArrayListTest$threadOne.run(ArrayListTest.java:23)
```

二、fail-fast 产生原因

通过上面的示例和讲解，我初步知道 fail-fast 产生的原因就在于程序在对 collection 进行迭代时，某个线程对该 collection 在结构上对其做了修改，这时迭代器就会抛出 ConcurrentModificationException 异常信息，从而产生 fail-fast。

要了解 fail-fast 机制，我们首先要对 ConcurrentModificationException 异常有所了解。当方法检测到对象的并发修改，但不允许这种修改时就抛出该异常。同时需要注意的是，该异常不会始终指出对象已经由不同线程并发修改，如果单线程违反了规则，同样也有可能会抛出改异常。

诚然，迭代器的快速失败行为无法得到保证，它不能保证一定会出现该错误，但是快速失败操作会尽最大努力抛出 ConcurrentModificationException 异常，所以因此，为提高此类操作的正确性而编写一个依赖于此异常的程序是错误的做法，正确做法是：ConcurrentModificationException 应该仅用于检测 bug。下面我将以 ArrayList 为例进一步分析 fail-fast 产生的原因。

从前面我们知道 fail-fast 是在操作迭代器时产生的。现在我们来看看 ArrayList 中迭代器的源代码：

```
private class Itr implements Iterator<E> {
        int cursor;
        int lastRet = -1;
        int expectedModCount = ArrayList.this.modCount;

        public boolean hasNext() {
            return (this.cursor != ArrayList.this.size);
        }

        public E next() {
            checkForComodification();
            /** 省略此处代码 */
        }

        public void remove() {
            if (this.lastRet < 0)
                throw new IllegalStateException();
            checkForComodification();
            /** 省略此处代码 */
        }

        final void checkForComodification() {
            if (ArrayList.this.modCount == this.expectedModCount)
```

```
            return;
        throw new ConcurrentModificationException();
    }
}
```

从上面的源代码我们可以看出，迭代器在调用 next()、remove() 方法时都是调用 checkForComodification() 方法，该方法主要就是检测 modCount == expectedModCount？若不等则抛出 ConcurrentModificationException 异常，从而产生 fail-fast 机制。所以要弄清楚为什么会产生 fail-fast 机制我们就必须要用弄明白为什么 modCount != expectedModCount ，他们的值在什么时候发生改变的。

expectedModCount 是在 Itr 中定义的：int expectedModCount = ArrayList.this.modCount;所以他的值是不可能会修改的，所以会变的就是 modCount。modCount 是在 AbstractList 中定义的，为全局变量：

```
protected transient int modCount = 0;
```

那么他什么时候因为什么原因而发生改变呢？请看 ArrayList 的源码：

```
public boolean add(E paramE) {
    ensureCapacityInternal(this.size + 1);
    /** 省略此处代码 */
}

private void ensureCapacityInternal(int paramInt)  {
    if (this.elementData == EMPTY_ELEMENTDATA)
        paramInt = Math.max(10, paramInt);
    ensureExplicitCapacity(paramInt);
}

private void ensureExplicitCapacity(int paramInt) {
    this.modCount += 1;    //修改modCount
    /** 省略此处代码 */
}

public boolean remove(Object paramObject) {
    int i;
    if (paramObject == null)
        for (i = 0; i < this.size; ++i) {
            if (this.elementData[i] != null)
                continue;
            fastRemove(i);
            return true;
        }
    else
```

```
            for (i = 0; i < this.size; ++i) {
                if (!(paramObject.equals(this.elementData[i])))
                    continue;
                fastRemove(i);
                return true;
            }
        return false;
    }

    private void fastRemove(int paramInt) {
        this.modCount += 1;   //修改modCount
        /** 省略此处代码 */
    }

    public void clear() {
        this.modCount += 1;    //修改modCount
        /** 省略此处代码 */
    }
```

从上面的源代码我们可以看出，ArrayList 中无论 add、remove、clear 方法只要是涉及了改变 ArrayList 元素的个数的方法都会导致 modCount 的改变。所以我们这里可以初步判断由于 expectedModCount 得值与 modCount 的改变不同步，导致两者之间不等从而产生 fail-fast 机制。知道产生 fail-fast 产生的根本原因了，我们可以有如下场景：

有两个线程（线程 A，线程 B），其中线程 A 负责遍历 list、线程B修改 list。线程 A 在遍历 list 过程的某个时候（此时 expectedModCount = modCount=N），线程启动，同时线程B增加一个元素，这是 modCount 的值发生改变（modCount + 1 = N + 1）。线程 A 继续遍历执行 next 方法时，通告 checkForComodification 方法发现 expectedModCount = N，而 modCount = N + 1，两者不等，这时就抛出ConcurrentModificationException 异常，从而产生 fail-fast 机制。

所以，直到这里我们已经完全了解了 fail-fast 产生的根本原因了。知道了原因就好找解决办法了。

三、fail-fast 解决办法

通过前面的实例、源码分析，我想各位已经基本了解了 fail-fast 的机制，下面我就产生的原因提出解决方案。这里有两种解决方案：

方案一：在遍历过程中所有涉及到改变 modCount 值得地方全部加上 synchronized 或者直接使用 Collections.synchronizedList，这样就可以解决。但是不推荐，因为增删造成的同步锁可能会阻塞遍历操作。

方案二：使用 CopyOnWriteArrayList 来替换 ArrayList。推荐使用该方案。

CopyOnWriteArrayList 为何物？ArrayList 的一个线程安全的变体，其中所有可变操作（add、set 等等）都是通过对底层数组进行一次新的复制来实现的。 该类产生的开销比较大，但是在两种情况下，它非常适合使用。1：在不能或不想进行同步遍历，但又需要从并发线程中排除冲突时。2：当遍历操作的数量大大超过可变操作的数量时。遇到这两种情况使用 CopyOnWriteArrayList 来替代 ArrayList 再适合不过了。那么为什么 CopyOnWriterArrayList 可以替代 ArrayList 呢?

第一、CopyOnWriterArrayList 的无论是从数据结构、定义都和 ArrayList 一样。它和 ArrayList 一样，同样是实现 List 接口，底层使用数组实现。在方法上也包含 add、remove、clear、iterator 等方法。

第二、CopyOnWriterArrayList 根本就不会产生 ConcurrentModificationException 异常，也就是它使用迭代器完全不会产生 fail-fast 机制。请看：

```
private static class COWIterator<E> implements ListIterator<E> {
    /** 省略此处代码 */
    public E next() {
       if (!(hasNext()))
          throw new NoSuchElementException();
       return this.snapshot[(this.cursor++)];
    }

    /** 省略此处代码 */
  }
```

CopyOnWriterArrayList 的方法根本就没有像 ArrayList 中使用 checkForComodification 方法来判断 expectedModCount 与 modCount 是否相等。它为什么会这么做，凭什么可以这么做呢？我们以 add 方法为例：

```
public boolean add(E paramE) {
    ReentrantLock localReentrantLock = this.lock;
    localReentrantLock.lock();
    try {
       Object[] arrayOfObject1 = getArray();
       int i = arrayOfObject1.length;
       Object[] arrayOfObject2 = Arrays.copyOf(arrayOfObject1, i + 1);
       arrayOfObject2[i] = paramE;
       setArray(arrayOfObject2);
       int j = 1;
       return j;
    } finally {
       localReentrantLock.unlock();
    }
  }
```

```
final void setArray(Object[] paramArrayOfObject) {
    this.array = paramArrayOfObject;
}
```

CopyOnWriterArrayList 的 add 方法与 ArrayList 的 add 方法有一个最大的不同点就在于，下面三句代码：

```
Object[] arrayOfObject2 = Arrays.copyOf(arrayOfObject1, i + 1);
arrayOfObject2[i] = paramE;
setArray(arrayOfObject2);
```

就是这三句代码使得 CopyOnWriterArrayList 不会抛 ConcurrentModificationException 异常。他们所展现的魅力就在于 copy 原来的 array，再在 copy 数组上进行 add 操作，这样做就完全不会影响 COWIterator 中的 array 了。

所以 CopyOnWriterArrayList 所代表的核心概念就是：任何对 array 在结构上有所改变的操作（add、remove、clear 等），CopyOnWriterArrayList 都会 copy 现有的数据，再在 copy 的数据上修改，这样就不会影响 COWIterator 中的数据了，修改完成之后改变原有数据的引用即可。同时这样造成的代价就是产生大量的对象，同时数组的 copy 也是相当有损耗的。

36

Java 集合细节（一）：请为集合指定初始容量

集合是我们在 Java 编程中使用非常广泛的，它就像大海，海纳百川，像万能容器，盛装万物，而且这个大海，万能容器还可以无限变大（如果条件允许）。当这个海、容器的量变得非常大的时候，它的初始容量就会显得很重要了，因为挖海、扩容是需要消耗大量的人力物力财力的。同样的道理，Collection 的初始容量也显得异常重要。所以：**对于已知的情景，请为集合指定初始容量。**

```
public static void main(String[] args) {
        StudentVO student = null;
        long begin1 = System.currentTimeMillis();
        List<StudentVO> list1 = new ArrayList<>();
        for(int i = 0 ; i < 1000000; i++){
            student = new StudentVO(i,"chenssy_"+i,i);
            list1.add(student);
        }
        long end1 = System.currentTimeMillis();
        System.out.println("list1 time：" + (end1 - begin1));

        long begin2 = System.currentTimeMillis();
        List<StudentVO> list2 = new ArrayList<>(1000000);
        for(int i = 0 ; i < 1000000; i++){
            student = new StudentVO(i,"chenssy_"+i,i);
            list2.add(student);
        }
        long end2 = System.currentTimeMillis();
        System.out.println("list2 time：" + (end2 - begin2));
    }
```

上面代码两个 list 都是插入 1000000 条数据，只不过 list1 没有没有申请初始化容量，而 list2 初始化容量 1000000。那运行结果如下：

```
list1 time：1638
list2 time：921
```

从上面的运行结果我们可以看出 list2 的速度是 list1 的两倍左右。在前面 LZ 就提过，ArrayList 的扩容机制是比较消耗资源的。我们先看 ArrayList 的 add 方法：

```
public boolean add(E e) {
        ensureCapacity(size + 1);
        elementData[size++] = e;
        return true;
    }
```

```
public void ensureCapacity(int minCapacity) {
    modCount++;        //修改计数器
    int oldCapacity = elementData.length;
    //当前需要的长度超过了数组长度，进行扩容处理
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        //新的容量 = 旧容量 * 1.5 + 1
        int newCapacity = (oldCapacity * 3)/2 + 1;
            if (newCapacity < minCapacity)
                newCapacity = minCapacity;
      //数组拷贝，生成新的数组
      elementData = Arrays.copyOf(elementData,  newCapacity);
    }
}
```

ArrayList 每次新增一个元素，就会检测 ArrayList 的当前容量是否已经到达临界点，如果到达临界点则会扩容 1.5 倍。然而 ArrayList 的扩容以及数组的拷贝生成新的数组是相当耗资源的。所以若我们事先已知集合的使用场景，知道集合的大概范围，我们最好是指定初始化容量，这样对资源的利用会更加好，尤其是大数据量的前提下，效率的提升和资源的利用会显得更加具有优势。

Java 集合细节一：请为集合指定初始容量

37

Java 集合细节（二）：asList 的缺陷

在实际开发过程中我们经常使用 asList 讲数组转换为 List，这个方法使用起来非常方便，但是 asList 方法存在几个缺陷：

一、避免使用基本数据类型数组转换为列表

使用 8 个基本类型数组转换为列表时会存在一个比较有味的缺陷。先看如下程序：

```
public static void main(String[] args) {
        int[] ints = {1,2,3,4,5};
        List list = Arrays.asList(ints);
        System.out.println("list'size：" + list.size());
    }
    ----------------------------------
    outPut：
    list'size：1
```

程序的运行结果并没有像我们预期的那样是 5 而是逆天的 1，这是什么情况？先看源码：

```
public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }
```

asList 接受的参数是一个泛型的变长参数，我们知道基本数据类型是无法发型化的，也就是说 8 个基本类型是无法作为 asList 的参数的， 要想作为泛型参数就必须使用其所对应的包装类型。但是这个这个实例中为什么没有出错呢？因为该实例是将 int 类型的数组当做其参数，而在 Java 中数组是一个对象，它是可以泛型化的。所以该例子是不会产生错误的。既然例子是将整个 int 类型的数组当做泛型参数，那么经过 asList 转换就只有一个 int 的列表了。如下：

```
public static void main(String[] args) {
    int[] ints = {1,2,3,4,5};
    List list = Arrays.asList(ints);
    System.out.println("list 的类型:" + list.get(0).getClass());
    System.out.println("list.get(0) == ints：" + list.get(0).equals(ints));
}
-------------------------------------------
outPut:
list 的类型:class [I
list.get(0) == ints：true
```

从这个运行结果我们可以充分证明 list 里面的元素就是 int 数组。弄清楚这点了，那么修改方法也就一目了然了：将 int 改变为 Integer。

```
public static void main(String[] args) {
        Integer[] ints = {1,2,3,4,5};
        List list = Arrays.asList(ints);
        System.out.println("list'size：" + list.size());
        System.out.println("list.get(0) 的类型:" +  list.get(0).getClass());
        System.out.println("list.get(0) == ints[0]：" + list.get(0).equals(ints[0]));
    }
    ----------------------------------------
    outPut:
    list'size：5
    list.get(0) 的类型:class java.lang.Integer
    list.get(0) == ints[0]：true
```

Java 细节（2.1）：在使用 asList 时不要将基本数据类型当做参数。

二、asList 产生的列表不可操作

对于上面的实例我们再做一个小小的修改：

```
public static void main(String[] args) {
        Integer[] ints = {1,2,3,4,5};
        List list = Arrays.asList(ints);
        list.add(6);
    }
```

该实例就是讲 ints 通过 asList 转换为 list 类别，然后再通过 add 方法加一个元素，这个实例简单的不能再简单了，但是运行结果呢？打出我们所料：

```
Exception in thread "main" java.lang.UnsupportedOperationException
    at java.util.AbstractList.add(Unknown Source)
    at java.util.AbstractList.add(Unknown Source)
    at com.chenssy.test.arrayList.AsListTest.main(AsListTest.java:10)
```

运行结果尽然抛出 UnsupportedOperationException 异常，该异常表示 list 不支持 add 方法。这就让我们郁闷了，list 怎么可能不支持 add 方法呢？难道 JDK 脑袋堵塞了？我们再看 asList 的源码：

```
public static <T> List<T> asList(T... a) {
        return new ArrayList<>(a);
    }
```

asList 接受参数后，直接 new 一个 ArrayList，到这里看应该是没有错误的啊？别急，再往下看：

```
private static class ArrayList<E> extends AbstractList<E>
    implements RandomAccess, java.io.Serializable{
        private static final long serialVersionUID = -2764017481108945198L;
        private final E[] a;

        ArrayList(E[] array) {
            if (array==null)
                throw new NullPointerException();
            a = array;
        }
        //.....................
    }
```

这是 ArrayList 的源码,从这里我们可以看出,此 ArrayList 不是 java.util.ArrayList，他是 Arrays 的内部类。该内部类提供了 size、toArray、get、set、indexOf、contains 方法，而像 add、remove 等改变 list 结果的方法从 AbstractList 父类继承过来，同时这些方法也比较奇葩，它直接抛出 UnsupportedOperationException 异常：

```
public boolean add(E e) {
        add(size(), e);
        return true;
    }

    public E set(int index, E element) {
        throw new UnsupportedOperationException();
    }

    public void add(int index, E element) {
        throw new UnsupportedOperationException();
    }

    public E remove(int index) {
        throw new UnsupportedOperationException();
    }
```

通过这些代码可以看出 asList 返回的列表只不过是一个披着 list 的外衣，它并没有 list 的基本特性（变长）。该 list 是一个长度不可变的列表，传入参数的数组有多长，其返回的列表就只能是多长。所以：

Java 细节（2.2）：不要试图改变 asList 返回的列表，否则你会自食苦果。

38

Java 集合细节（三）：subList 的缺陷

我们经常使用 subString 方法来对 String 对象进行分割处理，同时我们也可以使用 subList、subMap、subSet 来对 List、Map、Set 进行分割处理，但是这个分割存在某些瑕疵。

一、subList 返回仅仅只是一个视图

首先我们先看如下实例：

```
public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<Integer> ();
        list1.add(1);
        list1.add(2);

        //通过构造函数新建一个包含list1的列表 list2
        List<Integer> list2 = new ArrayList<Integer>(list1);

        //通过subList生成一个与list1一样的列表 list3
        List<Integer> list3 = list1.subList(0, list1.size());

        //修改list3
        list3.add(3);

        System.out.println("list1 == list2：" + list1.equals(list2));
        System.out.println("list1 == list3：" + list1.equals(list3));
    }
```

这个例子非常简单，无非就是通过构造函数、subList 重新生成一个与 list1 一样的 list，然后修改 list3，最后比较 list1 == list2?、list1 == list3?。按照我们常规的思路应该是这样的：因为 list3 通过 add 新增了一个元素，那么它肯定与 list1 不等，而 list2 是通过 list1 构造出来的，所以应该相等，所以结果应该是：

```
list1 == list2：true
list1 == list3: false
```

首先我们先不论结果的正确与否，我们先看 subList 的源码：

```
public List<E> subList(int fromIndex, int toIndex) {
        subListRangeCheck(fromIndex, toIndex, size);
        return new SubList(this, 0, fromIndex,  toIndex);
    }
```

subListRangeCheck 方式是判断 fromIndex、toIndex 是否合法，如果合法就直接返回一个 subList 对象，**注意在产生该 new 该对象的时候传递了一个参数 this ，该参数非常重要，因为他代表着原始 list。**

```
/**
    * 继承AbstractList类，实现RandomAccess接口
    */
   private class SubList extends AbstractList<E> implements RandomAccess {
      private final AbstractList<E> parent;   //列表
      private final int parentOffset;
      private final int offset;
      int size;

      //构造函数
      SubList(AbstractList<E> parent,
            int offset, int fromIndex, int toIndex) {
         this.parent = parent;
         this.parentOffset = fromIndex;
         this.offset = offset + fromIndex;
         this.size = toIndex - fromIndex;
         this.modCount = ArrayList.this.modCount;
      }

      //set方法
      public E set(int index, E e) {
         rangeCheck(index);
         checkForComodification();
         E oldValue = ArrayList.this.elementData(offset + index);
         ArrayList.this.elementData[offset + index] = e;
         return oldValue;
      }

      //get方法
      public E get(int index) {
         rangeCheck(index);
         checkForComodification();
         return ArrayList.this.elementData(offset + index);
      }

      //add方法
      public void add(int index, E e) {
         rangeCheckForAdd(index);
         checkForComodification();
         parent.add(parentOffset + index, e);
         this.modCount = parent.modCount;
         this.size++;
      }
```

```
        //remove方法
        public E remove(int index) {
            rangeCheck(index);
            checkForComodification();
            E result = parent.remove(parentOffset + index);
            this.modCount = parent.modCount;
            this.size--;
            return result;
        }
    }
```

该 SubLsit 是 ArrayList 的内部类，它与 ArrayList 一样，都是继承 AbstractList 和实现 RandomAccess 接口。同时也提供了 get、set、add、remove 等 list 常用的方法。但是它的构造函数有点特殊，在该构造函数中有两个地方需要注意：

1、this.parent = parent;而 parent 就是在前面传递过来的 list，也就是说 this.parent 就是原始 list 的引用。

2、this.offset = offset + fromIndex;this.parentOffset = fromIndex;。同时在构造函数中它甚至将 modCount（fail-fast机制）传递过来了。

我们再看 get 方法，在 get 方法中 return ArrayList.this.elementData(offset + index);这段代码可以清晰表明 get 所返回就是原列表 offset + index位置的元素。同样的道理还有 add 方法里面的：

```
parent.add(parentOffset + index, e);
this.modCount = parent.modCount;
```

remove 方法里面的

```
E result = parent.remove(parentOffset + index);
this.modCount = parent.modCount;
```

诚然，到了这里我们可以判断 subList 返回的 SubList 同样也是 AbstractList 的子类，同时它的方法如 get、set、add、remove 等都是在原列表上面做操作，它并没有像 subString 一样生成一个新的对象。所以 **subList 返回的只是原列表的一个视图，它所有的操作最终都会作用在原列表上。**

那么从这里的分析我们可以得出上面的结果应该恰恰与我们上面的答案相反：

```
list1 == list2：false
list1 == list3：true
```

Java 细节（3.1）：subList 返回的只是原列表的一个视图，它所有的操作最终都会作用在原列表上

二、subList 生成子列表后，不要试图去操作原列表

从上面我们知道 subList 生成的子列表只是原列表的一个视图而已，如果我们操作子列表它产生的作用都会在原列表上面表现，但是如果我们操作原列表会产生什么情况呢?

```
public static void main(String[] args) {
        List<Integer> list1 = new ArrayList<Integer>();
        list1.add(1);
        list1.add(2);

        //通过subList生成一个与list1一样的列表 list3
        List<Integer> list3 = list1.subList(0, list1.size());
        //修改list3
        list1.add(3);

        System.out.println("list1'size：" + list1.size());
        System.out.println("list3'size：" + list3.size  ());
    }
```

该实例如果不产生意外，那么他们两个 list 的大小都应该都是 3，但是偏偏事与愿违，事实上我们得到的结果是这样的：

```
list1'size：3
Exception in thread "main" java.util.ConcurrentModificationException
    at java.util.ArrayList$SubList.checkForComodification(Unknown Source)
    at java.util.ArrayList$SubList.size(Unknown Source)
    at com.chenssy.test.arrayList.SubListTest.main(SubListTest.java:17)
```

list1 正常输出，但是 list3 就抛出 ConcurrentModificationException 异常，看过我另一篇博客的同仁肯定对这个异常非常，fail-fast? 不错就是 fail-fast 机制，在 fail-fast 机制中，LZ 花了很多力气来讲述这个异常，所以这里 LZ 就不对这个异常多讲了（更多请点这里：Java 提高篇（三四）—- fail-fast 机制 ）。我们再看 size 方法：

```
public int size() {
            checkForComodification();
            return this.size;
        }
```

size 方法首先会通过 checkForComodification 验证，然后再返回this.size。

```
private void checkForComodification() {
        if (ArrayList.this.modCount != this.modCount)
            throw new ConcurrentModificationException();
    }
```

该方法表明当原列表的 modCount 与 this.modCount 不相等时就会抛出 ConcurrentModificationException。同时我们知道 modCount 在 new 的过程中“继承”了原列表 modCount，只有在修改该列表（子列表）时才会修改该值（先表现在原列表后作用于子列表）。而在该实例中我们是操作原列表，原列表的 modCount 当然不会反应在子列表的 modCount 上啦，所以才会抛出该异常。

对于子列表视图，它是动态生成的，生成之后就不要操作原列表了，否则必然都导致视图的不稳定而抛出异常。最好的办法就是将原列表设置为只读状态，要操作就操作子列表：

```
//通过subList生成一个与list1一样的列表 list3
List<Integer> list3 = list1.subList(0, list1.size());

//对list1设置为只读状态
list1 = Collections.unmodifiableList(list1);
```

Java 细节（3.2）：生成子列表后，不要试图去操作原列表，否则会造成子列表的不稳定而产生异常

三、推荐使用 subList 处理局部列表

在开发过程中我们一定会遇到这样一个问题：获取一堆数据后，需要删除某段数据。例如，有一个列表存在 1000 条记录，我们需要删除 100-200 位置处的数据，可能我们会这样处理：

```
for(int i = 0 ; i < list1.size() ; i++){
  if(i >= 100 && i <= 200){
     list1.remove(i);
     /*
      * 当然这段代码存在问题，list remove之后后面的元素会填充上来，
      * 所以需要对i进行简单的处理，当然这个不是这里讨论的 问题。
      */
  }
}
```

这个应该是我们大部分人的处理方式吧，其实还有更好的方法，利用 subList。在前面 LZ 已经讲过，子列表的操作都会反映在原列表上。所以下面一行代码全部搞定：

```
list1.subList(100, 200).clear();
```

简单而不失华丽！！！！！

参考资料：编写高质量代码：改善 Java 程序的 151 个建议

39

Java 集合细节（四）：保持 compareTo 和 equals 同步

在 Java 中我们常使用 Comparable 接口来实现排序，其中 compareTo 是实现该接口方法。我们知道 compareTo 返回 0 表示两个对象相等，返回正数表示大于，返回负数表示小于。同时我们也知道 equals 也可以判断两个对象是否相等，那么他们两者之间是否存在关联关系呢?

```
public class Student implements Comparable<Student>{
    private String id;
    private String name;
    private int age;

    public Student(String id,String name,int age){
        this.id = id;
        this.name = name;
        this.age = age;
    }

    public boolean equals(Object obj){
        if(obj == null){
        return false;
        }

        if(this == obj){
            return true;
        }

        if(obj.getClass() != this.getClass()){
            return false;
        }

        Student student = (Student)obj;
        if(!student.getName().equals(getName())){
            return false;
        }

        return true;
    }

    public int compareTo(Student student) {
        return this.age - student.age;
    }

    /** 省略getter、setter方法 */
}
```

Student 类实现 Comparable 接口和实现 equals 方法，其中 compareTo 是根据 age 来比对的，equals 是根据 name 来比对的。

```
public static void main(String[] args){
    List<Student> list = new ArrayList<>();
    list.add(new Student("1", "chenssy1", 24));
    list.add(new Student("2", "chenssy1", 26));

    Collections.sort(list);  //排序

    Student student = new Student("2", "chenssy1", 26);

    //检索student在list中的位置
    int index1 = list.indexOf(student);
    int index2 = Collections.binarySearch(list, student);

    System.out.println("index1 = " + index1);
    System.out.println("index2 = " + index2);
 }
```

按照常规思路来说应该两者 index 是一致的，因为他们检索的是同一个对象，但是非常遗憾，其运行结果：

```
index1 = 0
index2 = 1
```

为什么会产生这样不同的结果呢？这是因为 indexOf 和 binarySearch 的实现机制不同，indexOf 是基于 equals 来实现的只要 equals 返回 TRUE 就认为已经找到了相同的元素。而 binarySearch 是基于 compareTo 方法的，当 compareTo 返回 0 时就认为已经找到了该元素。在我们实现的 Student 类中我们覆写了 compareTo 和 equals 方法，但是我们的 compareTo、equals 的比较依据不同，一个是基于 age、一个是基于 name。比较依据不同那么得到的结果很有可能会不同。所以知道了原因，我们就好修改了：将两者之间的比较依据保持一致即可。

对于 compareTo 和 equals 两个方法我们可以总结为：**compareTo 是判断元素在排序中的位置是否相等，equals 是判断元素是否相等，既然一个决定排序位置，一个决定相等，所以我们非常有必要确保当排序位置相同时，其 equals 也应该相等。**

细节（4.1）：实现了 compareTo 方法，就有必要实现 equals 方法，同时还需要确保两个方法同步。