博客： https://www.cnblogs.com/HOsystem/p/14116443.html

2、具体内容

简单Java类主要是由属性所组成，并且提供有相应的setter、getter处理方法，同时简单Java类最大的特征就是通过对象保存相应的类属性内容。但是如果使用传统的简单Java类开发，那么也会面临非常麻烦的困难。

范例：传统的简单Java类操作

```
package cn.mldn.demo;
class Emp {
	private String ename ;
	private String job ;
	//setter getter略
}
```

特别强调，为了方便理解，本次Emp类之中定义的ename、job两个属性的类型使用的都是String类型。按照传统的做法，此时应该首先实例化Emp类的对象，而后通过实例化对象进行setter方法的调用用以设置属性内容。

范例：传统的调用

```
package cn.mldn.demo;
public class JavaAPIDemo {
	public static void main(String[] args) throws Exception {
		Emp emp = new Emp() ; // 更多情况下开发者关注的是内容的设置
		emp.setEname("SMITH");
		emp.setJob("CLERK");
		// 使用为对象设置之后
		System.out.println("姓名：" + emp.getEname() + "、职位：" + emp.getJob());
	}
}
```

在整个进行Emp对象实例化并设置数据的操作过程之中，设置数据的部分是最麻烦的，可以想象一下，如果说现在Emp类里面提供有50个属性，那么对于整个的程序将会出现一堆的setter方法调用。或者再进一步说明，在实际的开发之中，简单Java类的个数是非

常多的，那么如果所有的简单java类都牵扯到属性赋值的时候，这种情况下代码编写的重复性将会非常高。

按照传统的直观的编程方式，所带来的的问题就是代码会存在有大量的重复操作，如果要想解决对象的重复处理操作那么唯一的解决方案就是反射机制，反射机制最大的特征是可以根据其自身的特点（Object类直接操作、可以直接操作属性或方法）实现相同功能类的重复操作的抽象处理。

■属性自动设置解决方案

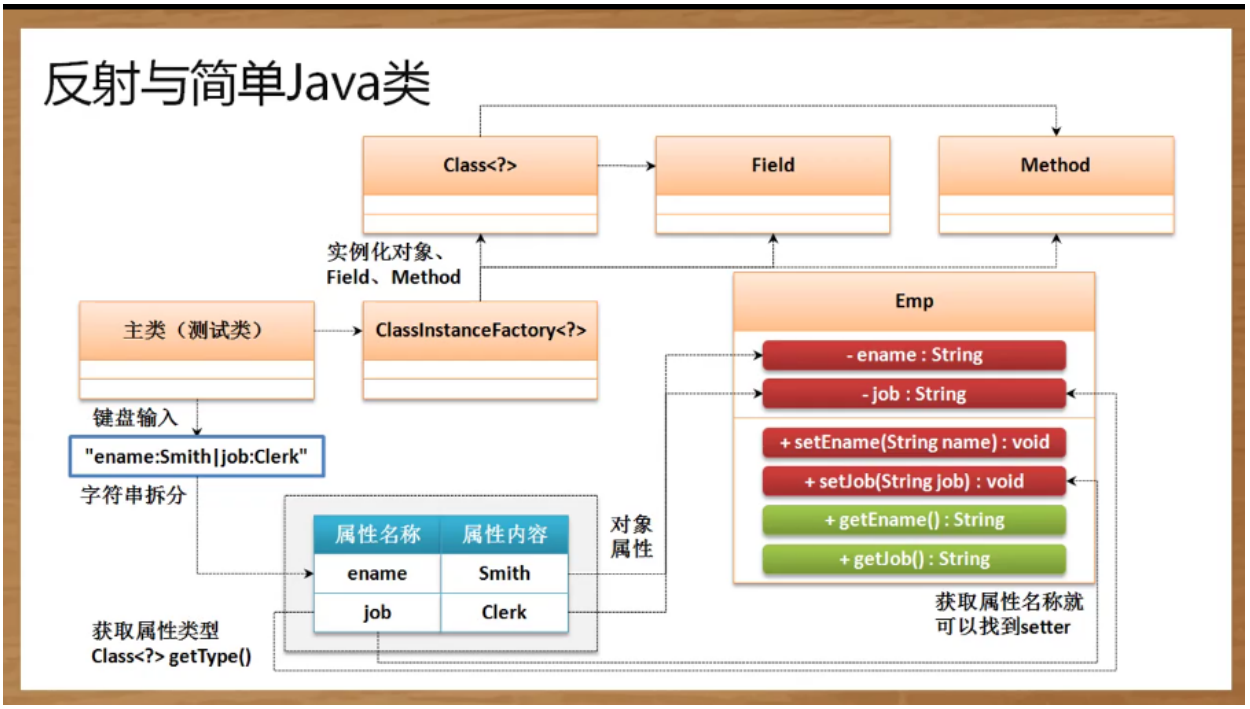
经过了分析之后已经确认了当前简单Java类操作的问题所在，而对于开发者而言就需要想办法通过一种解决方案来实现属性内容的自动设置，那么这个时候的设置强烈建议采用字符串的形式来描述对应的类型。

1、在进行程序开发的时候String字符串可以描述的内容有很多，并且也可以由开发者自行定义字符串的结构，下面就采用“属性:内容|属性:内容|”的形式来为简单Java类中的属性初始化。

```

```

2、类设计的基本结构：应该由一个专门的ClassInstanceFactory类负责所有的反射处理，即：接收反射对象与要设置的属性内容，同时可以获取指定类的实例化对象。



3、设计的基本结构

```
class ClassInstanceFactory {
    private ClassInstanceFactory() {}
    /**
     * 实例化对象的创建方法，该对象可以根据传入的字符串结构“属性:内容|属性:内容”
     * @param clazz 要进行反射实例化的Class类对象，有Class就可以反射实例化对象
```

```
     * @param value 要设置给对象的属性内容
     * @return 一个已经配置好属性内容的Java类对象
     */
    public static <T> T create(Class<?> clazz,String value) {
        return null ;
    }
}
```

```
package cn.mldn.demo;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
        String value = "ename:Smith|job:Clerk" ;
        Emp emp = ClassInstanceFactory.create(Emp.class, value) ;
        System.out.println("姓名：" + emp.getEname() + "、职位：" + emp.getJob());
    }
}
```

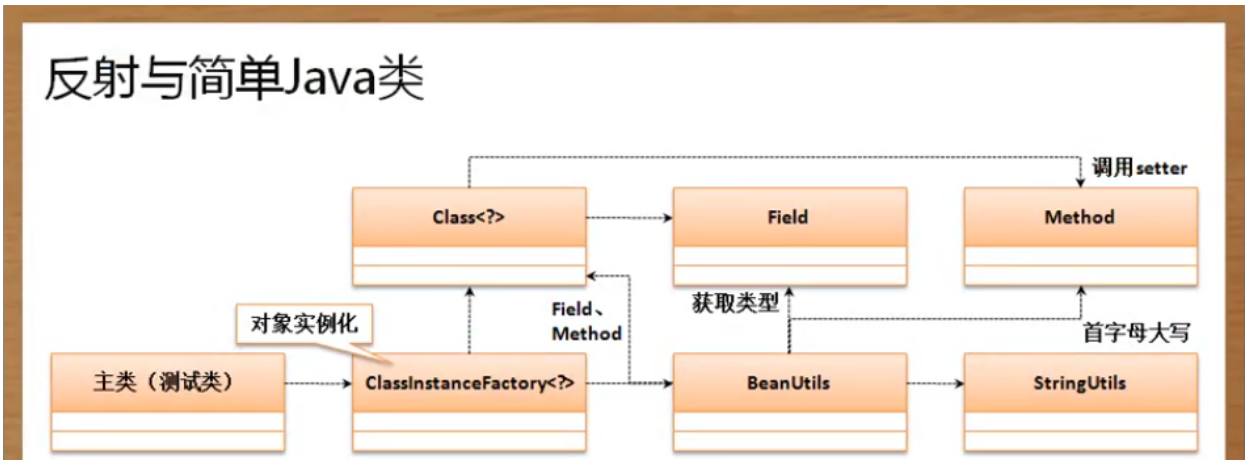
那么在当前的开发之中，所需要留给用户完善的就是ClassInstanceFactory.create()处理方法。

■单级属性配置

对于此时的Emp类里面会发现所给出的数据类型都没有其它的引用关联了，只是描述了Emp本类的对象，所以这样的设置称为单级设置处理，所以此时应该处理两件事情：

·需要通过反射进行指定类对象的实例化处理；

·进行内容的设置（Field属性类型、方法名称、要设置的内容）；



1、定义StirngUtils实现首字母大写功能

```
class StringUtils {
    public static String initcap(String str) {
        if (str == null || "".equals(str)) {
            return str ;
        }
        if (str.length() == 1) {
            return str.toUpperCase() ;
        } else {
            return str.substring(0,1).toUpperCase() + str.substring(1) ;
        }
```

```
    }
}
```

2、定义BeanUtils工具类，该工具类主要实现属性的设置

```
class BeanUtils {     // 进行Bean处理的类
    private BeanUtils() {}
    /**
     * 实现指定对象的属性设置
     * @param obj 要进行反射操作的实例化对象
     * @param value 包含有指定内容的字符串，格式“属性:内容|属性:内容”
     */
    public static void setValue(Object obj,String value) {
        String results [] = value.split("\\|") ; // 按照“|”进行每一组属性的拆分
        for (int x = 0 ; x < results.length ; x ++) {       // 循环设置属性内容
            // attval[0]保存的是属性名称、attval[1]保存的是属性内容
            String attval [] = results[x].split(":") ; // 获取“属性名称”与内容;
            try {
                Field field = obj.getClass().getDeclaredField(attval[0]) ; // 获取成员
                Method setMethod = obj.getClass().getDeclaredMethod("set" +
StringUtils.initcap(attval[0]), field.getType()) ;
                setMethod.invoke(obj, attval[1]) ; // 调用setter方法设置内容
            } catch (Exception e) {}
        }
    }
}
```

3、ClassInstanceFactory负责实例化对象并调用BeanUtils类实现属性内容的设置

```
class ClassInstanceFactory {
    private ClassInstanceFactory() {}
    /**
     * 实例化对象的创建方法，该对象可以根据传入的字符串结构“属性:内容|属性:内容”
     * @param clazz 要进行反射实例化的Class类对象，有Class就可以反射实例化对象
     * @param value 要设置给对象的属性内容
     * @return 一个已经配置好属性内容的Java类对象
     */
    public static <T> T create(Class<?> clazz,String value) {
        try { // 如果要想采用反射进行简单Java类对象属性设置的时候，类中必须要有无参构造
            Object obj = clazz.getDeclaredConstructor().newInstance() ;
            BeanUtils.setValue(obj, value); // 通过反射设置属性
            return (T) obj ; // 返回对象
        } catch (Exception e) {
            e.printStackTrace(); // 如果此时真的出现了错误，本质上抛出异常也没用
            return null ;
        }
    }
}
```

即使现在类中的属性再多，那么也可以轻松的实现setter的调用（类对象实例化处理）。

■设置多种数据类型

现在已经成功的实现了单级的属性配置，但是这里面依然需要考虑一个实际的情况，当前所给定的数据类型只是String，但是在实际的开发之中面对简单的java类中的属性类型一般的可选为：long（Long）、int（Integer）、double（Double）、String、Date（日期、日期时间），所以这个时候对于当前的程序代码久必须做出修改，要求可以实现各种数据类型的配置。

既然要求可以实现不同类型的内容设置，并且BeanUtils类主要是完成属性赋值处理的，那么就可以在这个类之中追加有一系列的处理方法。

```
class BeanUtils {      // 进行Bean处理的类
    private BeanUtils() {}
    /**
     * 实现指定对象的属性设置
     * @param obj 要进行反射操作的实例化对象
     * @param value 包含有指定内容的字符串，格式“属性:内容|属性:内容”
     */
    public static void setValue(Object obj,String value) {
        String results [] = value.split("\\|") ; // 按照“|”进行每一组属性的拆分
        for (int x = 0 ; x < results.length ; x ++) {       // 循环设置属性内容
            // attval[0]保存的是属性名称、attval[1]保存的是属性内容
            String attval [] = results[x].split(":") ; // 获取“属性名称”与内容;
            try {
                Field field = obj.getClass().getDeclaredField(attval[0]) ; // 获取成员
                Method setMethod = obj.getClass().getDeclaredMethod("set" +
StringUtils.initcap(attval[0]), field.getType()) ;
                Object convertValue =
BeanUtils.convertAttributeValue(field.getType().getName(), attval[1]) ;
                setMethod.invoke(obj, convertValue) ; // 调用setter方法设置内容
            } catch (Exception e) {}
        }
    }
    /**
     * 实现属性类型转换处理
     * @param type 属性类型，通过Field获取的
     * @param value 属性的内容，传入的都是字符串，需要将其变为指定类型
     * @return 转换后的数据
     */
    private static Object convertAttributeValue(String type,String value) {
        if ("long".equals(type) || "java.lang.Long".equals(type)) {       // 长整型
            return Long.parseLong(value) ;
        } else if ("int".equals(type) || "java.lang.int".equals(type)) {
            return Integer.parseInt(value) ;
        } else if ("double".equals(type) || "java.lang.double".equals(type)) {
            return Double.parseDouble(value) ;
        } else if ("java.util.Date".equals(type)) {
            SimpleDateFormat sdf = null ;
            if (value.matches("\\d{4}-\\d{2}-\\d{2}")) {      // 日期类型
                sdf = new SimpleDateFormat("yyyy-MM-dd") ;
```

```
                } else if (value.matches("\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}")) {
                        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss") ;
                } else {
                        return new Date() ; // 当前日期
                }
                try {
                        return sdf.parse(value) ;
                } catch (ParseException e) {
                        return new Date() ; // 当前日期
                }
        } else {
                return value ;
        }
    }
}
```

此时只是列举出了常用的几种数据类型，如果要想将其作为一个产品推广，你必须要考虑所有可能出现的类型，同时所有可能出现的日期格式也需要考虑。

■级联对象实例化

如果说现在给定的类对象之中存在有其它的引用的级联关系的情况下，称为多级设置。例如：一个雇员属于一个部门，一个部门属于一个公司，所以这个时候对于简单java类的基本关系定义如下：

Company.java	**Dept.java**	**Emp.java**

class Company {
private String name ;
private Date createdate ;
}
	class Dept {
	private String dname ;
	private String loc ;
	private Company **company** ;
	}
		class Emp {
		private long empno ;
		private String ename ;
		private String job ;
		private double salary ;
		private Date hiredate ;
		private Dept **dept** ;
		}

如果要通过Emp进行操作，则应该使用“.”作为级联关系的处理：

dept.dname:财务部	Emp类实例化对象.**getDept**().setDname("财务部")
dept.company.name:MLDN	Emp类实例化对象.**getDept**().**getComany**().setName("MLDN")

但是考虑到代码的简洁性，所以应该考虑可以通过级联的配置自动实现类中属性的实例化。

```
    String value = "empno:7369|ename:Smith|job:Clerk|salary:750.00|hiredate:1989-10-10|" +
                "dept.dname:财务部|dept.company.name:MLDN" ;
```

现在的属性存在有多级的关系，那么对于多级的关系就必须与单级的配置区分开。

```
class BeanUtils {    // 进行Bean处理的类
```

```
    private BeanUtils() {}
    /**
     * 实现指定对象的属性设置
     * @param obj 要进行反射操作的实例化对象
     * @param value 包含有指定内容的字符串，格式“属性:内容|属性:内容”
     */
    public static void setValue(Object obj,String value) {
        String results [] = value.split("\\|") ; // 按照“|”进行每一组属性的拆分
        for (int x = 0 ; x < results.length ; x ++) {       // 循环设置属性内容
            // attval[0]保存的是属性名称、attval[1]保存的是属性内容
            String attval [] = results[x].split(":") ; // 获取“属性名称”与内容;
            try {
                if (attval[0].contains(".")) {      // 多级配置
                    String temp [] = attval[0].split("\\.") ;
                    Object currentObject = obj ;
                    // 最后一位肯定是指定类中的属性名称，所以不在本次实例化处理的范畴之内
                    for (int y = 0 ; y < temp.length - 1 ; y ++) {       // 实例化
                        // 调用相应的getter方法，如果getter方法返回了null表示该对象未实例化
                        Method getMethod = currentObject.getClass().getDeclaredMethod("get" + StringUtils.initcap(temp[y])) ;
                        Object tempObject = getMethod.invoke(currentObject) ;
                        if (tempObject == null) {     // 该对象现在并没有实例化
                            Field field = currentObject.getClass().getDeclaredField(temp[y]) ;      // 获取属性类型
                            Method method = currentObject.getClass().getDeclaredMethod("set" + StringUtils.initcap(temp[y]), field.getType()) ;
                            Object newObject = field.getType().getDeclaredConstructor().newInstance() ;
                            method.invoke(currentObject, newObject) ;
                            currentObject = newObject ;
                        } else {
                            currentObject = tempObject ;
                        }
                    }
                } else {
                    Field field = obj.getClass().getDeclaredField(attval[0]) ; // 获取成员
                    Method setMethod = obj.getClass().getDeclaredMethod("set" + StringUtils.initcap(attval[0]), field.getType()) ;
                    Object convertValue = BeanUtils.convertAttributeValue(field.getType().getName(), attval[1]) ;
                    setMethod.invoke(obj, convertValue) ; // 调用setter方法设置内容
                }
            } catch (Exception e) {
            }
        }
    }
    /**
     * 实现属性类型转换处理
     * @param type 属性类型，通过Field获取的
```

```
     * @param value 属性的内容，传入的都是字符串，需要将其变为指定类型
     * @return 转换后的数据
     */
    private static Object convertAttributeValue(String type,String value) {
        if ("long".equals(type) || "java.lang.Long".equals(type)) {       // 长整型
            return Long.parseLong(value) ;
        } else if ("int".equals(type) || "java.lang.int".equals(type)) {
            return Integer.parseInt(value) ;
        } else if ("double".equals(type) || "java.lang.double".equals(type)) {
            return Double.parseDouble(value) ;
        } else if ("java.util.Date".equals(type)) {
            SimpleDateFormat sdf = null ;
            if (value.matches("\\d{4}-\\d{2}-\\d{2}")) {      // 日期类型
                sdf = new SimpleDateFormat("yyyy-MM-dd") ;
            } else if (value.matches("\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}")) {
                sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss") ;
            } else {
                return new Date() ; // 当前日期
            }
            try {
                return sdf.parse(value) ;
            } catch (ParseException e) {
                return new Date() ; // 当前日期
            }
        } else {
            return value ;
        }
    }
}
```

执行自动的级联配置的实例化处理操作，在以后进行项目的编写之中一定会使用到。

完整代码

```
package cn.mldn.demo;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
public class JavaAPIDemo {
    public static void main(String[] args) throws Exception {
    String value = "empno:7369|ename:Smith|job:Clerk|salary:750.00|hiredate:1989-10-10|"
+
            "dept.dname:财务部|dept.company.name:MLDN" ;
        Emp emp = ClassInstanceFactory.create(Emp.class, value) ;
        System.out.println("雇员编号：" + emp.getEmpno() + "姓名：" + emp.getEname()
+ "、职位：" + emp.getJob() + "、基本工资："
                + emp.getSalary() + "、雇佣日期：" + emp.getHiredate());
        System.out.println(emp.getDept());
        System.out.println(emp.getDept().getCompany());
    }
```

```
}
class ClassInstanceFactory {
	private ClassInstanceFactory() {}
	/**
	 * 实例化对象的创建方法，该对象可以根据传入的字符串结构“属性:内容|属性:内容”
	 * @param clazz 要进行反射实例化的Class类对象，有Class就可以反射实例化对象
	 * @param value 要设置给对象的属性内容
	 * @return 一个已经配置好属性内容的Java类对象
	 */
	public static <T> T create(Class<?> clazz,String value) {
		try { // 如果要想采用反射进行简单Java类对象属性设置的时候，类中必须要有无参构造
			Object obj = clazz.getDeclaredConstructor().newInstance() ;
			BeanUtils.setValue(obj, value); // 通过反射设置属性
			return (T) obj ; // 返回对象
		} catch (Exception e) {
			e.printStackTrace(); // 如果此时真的出现了错误，本质上抛出异常也没用
			return null ;
		}
	}
}
class StringUtils {
	public static String initcap(String str) {
		if (str == null || "".equals(str)) {
			return str ;
		}
		if (str.length() == 1) {
			return str.toUpperCase() ;
		} else {
			return str.substring(0,1).toUpperCase() + str.substring(1) ;
		}
	}
}
class BeanUtils {	// 进行Bean处理的类
	private BeanUtils() {}
	/**
	 * 实现指定对象的属性设置
	 * @param obj 要进行反射操作的实例化对象
	 * @param value 包含有指定内容的字符串，格式“属性:内容|属性:内容”
	 */
	public static void setValue(Object obj,String value) {
		String results [] = value.split("\\|") ; // 按照“|”进行每一组属性的拆分
		for (int x = 0 ; x < results.length ; x ++) {		// 循环设置属性内容
			// attval[0]保存的是属性名称、attval[1]保存的是属性内容
			String attval [] = results[x].split(":") ; // 获取“属性名称”与内容;
			try {
				if (attval[0].contains(".")) {		// 多级配置
					String temp [] = attval[0].split("\\.") ;
					Object currentObject = obj ;
					// 最后一位肯定是指定类中的属性名称，所以不在本次实例化处理的范畴
之内
					for (int y = 0 ; y < temp.length - 1 ; y ++) {	// 实例化
```

```
                                // 调用相应的getter方法，如果getter方法返回了null表示该对象未实例化
                                Method getMethod = currentObject.getClass().getDeclaredMethod("get" + StringUtils.initcap(temp[y])) ;
                                Object tempObject = getMethod.invoke(currentObject) ;
                                if (tempObject == null) {       // 该对象现在并没有实例化
                                    Field field = currentObject.getClass().getDeclaredField(temp[y]) ;    // 获取属性类型
                                    Method method = currentObject.getClass().getDeclaredMethod("set" + StringUtils.initcap(temp[y]), field.getType()) ;
                                    Object newObject = field.getType().getDeclaredConstructor().newInstance() ;
                                    method.invoke(currentObject, newObject) ;
                                    currentObject = newObject ;
                                } else {
                                    currentObject = tempObject ;
                                }
                            }
                        } else {
                            Field field = obj.getClass().getDeclaredField(attval[0]) ; // 获取成员
                            Method setMethod = obj.getClass().getDeclaredMethod("set" + StringUtils.initcap(attval[0]), field.getType()) ;
                            Object convertValue = BeanUtils.convertAttributeValue(field.getType().getName(), attval[1]) ;
                            setMethod.invoke(obj, convertValue) ; // 调用setter方法设置内容
                        }
                    } catch (Exception e) {
                    }
                }
            }
            /**
             * 实现属性类型转换处理
             * @param type 属性类型，通过Field获取的
             * @param value 属性的内容，传入的都是字符串，需要将其变为指定类型
             * @return 转换后的数据
             */
            private static Object convertAttributeValue(String type,String value) {
                if ("long".equals(type) || "java.lang.Long".equals(type)) {       // 长整型
                    return Long.parseLong(value) ;
                } else if ("int".equals(type) || "java.lang.int".equals(type)) {
                    return Integer.parseInt(value) ;
                } else if ("double".equals(type) || "java.lang.double".equals(type)) {
                    return Double.parseDouble(value) ;
                } else if ("java.util.Date".equals(type)) {
                    SimpleDateFormat sdf = null ;
                    if (value.matches("\\d{4}-\\d{2}-\\d{2}")) {      // 日期类型
                        sdf = new SimpleDateFormat("yyyy-MM-dd") ;
                    } else if (value.matches("\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}")) {
                        sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss") ;
                    } else {
                        return new Date() ; // 当前日期
```

```
                }
                try {
                        return sdf.parse(value) ;
                } catch (ParseException e) {
                        return new Date() ; // 当前日期
                }
        } else {
                return value ;
        }
    }
}
class Company {
    private String name ;
    private Date createdate ;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Date getCreatedate() {
        return createdate;
    }
    public void setCreatedate(Date createdate) {
        this.createdate = createdate;
    }
}
class Dept {
    private String dname ;
    private String loc ;
    private Company company ;
    public String getDname() {
        return dname;
    }
    public void setDname(String dname) {
        this.dname = dname;
    }
    public String getLoc() {
        return loc;
    }
    public void setLoc(String loc) {
        this.loc = loc;
    }
    public Company getCompany() {
        return company;
    }
    public void setCompany(Company company) {
        this.company = company;
    }

}
class Emp {
```

```
    private long empno ;
    private String ename ;
    private String job ;
    private double salary ;
    private Date hiredate ;
    private Dept dept ;

    public Dept getDept() {
        return dept;
    }
    public void setDept(Dept dept) {
        this.dept = dept;
    }
    public void setEname(String ename) {
        this.ename = ename;
    }
    public void setJob(String job) {
        this.job = job;
    }
    public String getEname() {
        return ename;
    }
    public String getJob() {
        return job;
    }
    public long getEmpno() {
        return empno;
    }
    public void setEmpno(long empno) {
        this.empno = empno;
    }
    public double getSalary() {
        return salary;
    }
    public void setSalary(double salary) {
        this.salary = salary;
    }
    public Date getHiredate() {
        return hiredate;
    }
    public void setHiredate(Date hiredate) {
        this.hiredate = hiredate;
    }
}
```

■级联属性设置

现在已经成功的实现级联的对象实例化处理，那么随后就需要去考虑级联的属性的设置了，在之前考虑级联对象实例化处理的时候会发现，循环的时候都是少了一位的。

```
// 最后一位肯定是指定类中的属性名称，所以不在本次实例化处理的范畴之内
for (int y = 0 ; y < temp.length - 1 ; y ++) {    // 实例化
// 调用相应的getter方法，如果getter方法返回了null表示该对象未实例化
      Method getMethod = currentObject.getClass().getDeclaredMethod("get" +
StringUtils.initcap(temp[y])) ;
      Object tempObject = getMethod.invoke(currentObject) ;
      if (tempObject == null) {        // 该对象现在并没有实例化
            Field field = currentObject.getClass().getDeclaredField(temp[y]) ;       // 获取属性类
型
            Method method = currentObject.getClass().getDeclaredMethod("set" +
StringUtils.initcap(temp[y]), field.getType()) ;
            Object newObject = field.getType().getDeclaredConstructor().newInstance() ;
            method.invoke(currentObject, newObject) ;
            currentObject = newObject ;
      } else {
            currentObject = tempObject ;
            }
      }
}
```

单此时代码循环处理完成之后，currentObject表示的就是可以进行setter方法调用的对象了，并且理论上该对象一定不可能为null，随后就可以按照之前的方式利用对象进行setter调用。

范例：实现对象实的级联属性设置

```
// 进行属性内容的设置
Field field = currentObject.getClass().getDeclaredField(temp[temp.length - 1]) ; // 获取成员
Method setMethod = currentObject.getClass().getDeclaredMethod("set" +
StringUtils.initcap(temp[temp.length - 1]), field.getType()) ;
Object convertValue = BeanUtils.convertAttributeValue(field.getType().getName(), attval[1])
;
setMethod.invoke(currentObject, convertValue) ; // 调用setter方法设置内容
```

在以后的开发之中简单Java类的赋值处理将不再重复调用setter操作完成，而这种处理形式是在正规开发中普遍采用的方式。

完整代码

```
package cn.mldn.demo;
import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Date;
public class JavaAPIDemo {
      public static void main(String[] args) throws Exception {
      String value = "empno:7369|ename:Smith|job:Clerk|salary:750.00|hiredate:1989-10-10|"
+
                  "dept.dname:财务部|dept.company.name:MLDN" ;
            Emp emp = ClassInstanceFactory.create(Emp.class, value) ;
```

```
            System.out.println("雇员编号：" + emp.getEmpno() + "姓名：" + emp.getEname()
+ "、职位：" + emp.getJob() + "、基本工资："
                        + emp.getSalary() + "、雇佣日期：" + emp.getHiredate());
            System.out.println(emp.getDept().getDname());
            System.out.println(emp.getDept().getCompany().getName());
        }
}
class ClassInstanceFactory {
        private ClassInstanceFactory() {}
        /**
         * 实例化对象的创建方法，该对象可以根据传入的字符串结构“属性:内容|属性:内容”
         * @param clazz 要进行反射实例化的Class类对象，有Class就可以反射实例化对象
         * @param value 要设置给对象的属性内容
         * @return 一个已经配置好属性内容的Java类对象
         */
        public static <T> T create(Class<?> clazz,String value) {
            try { // 如果要想采用反射进行简单Java类对象属性设置的时候，类中必须要有无参构造
                Object obj = clazz.getDeclaredConstructor().newInstance() ;
                BeanUtils.setValue(obj, value); // 通过反射设置属性
                return (T) obj ; // 返回对象
            } catch (Exception e) {
                e.printStackTrace(); // 如果此时真的出现了错误，本质上抛出异常也没用
                return null ;
            }
        }
}
class StringUtils {
        public static String initcap(String str) {
            if (str == null || "".equals(str)) {
                return str ;
            }
            if (str.length() == 1) {
                return str.toUpperCase() ;
            } else {
                return str.substring(0,1).toUpperCase() + str.substring(1) ;
            }
        }
}
class BeanUtils {      // 进行Bean处理的类
        private BeanUtils() {}
        /**
         * 实现指定对象的属性设置
         * @param obj 要进行反射操作的实例化对象
         * @param value 包含有指定内容的字符串，格式“属性:内容|属性:内容”
         */
        public static void setValue(Object obj,String value) {
            String results [] = value.split("\\|") ; // 按照“|”进行每一组属性的拆分
            for (int x = 0 ; x < results.length ; x ++) {       // 循环设置属性内容
                // attval[0]保存的是属性名称、attval[1]保存的是属性内容
                String attval [] = results[x].split(":") ; // 获取“属性名称”与内容；
                try {
                    if (attval[0].contains(".")) {      // 多级配置
```

```
                        String temp [] = attval[0].split("\\.") ;
                        Object currentObject = obj ;
                        // 最后一位肯定是指定类中的属性名称，所以不在本次实例化处理的范畴之内
                        for (int y = 0 ; y < temp.length - 1 ; y ++) {    // 实例化
                            // 调用相应的getter方法，如果getter方法返回了null表示该对象未实例化
                            Method getMethod = currentObject.getClass().getDeclaredMethod("get" + StringUtils.initcap(temp[y])) ;
                            Object tempObject = getMethod.invoke(currentObject) ;
                            if (tempObject == null) {        // 该对象现在并没有实例化
                                Field field = currentObject.getClass().getDeclaredField(temp[y]) ;    // 获取属性类型
                                Method method = currentObject.getClass().getDeclaredMethod("set" + StringUtils.initcap(temp[y]), field.getType()) ;
                                Object newObject = field.getType().getDeclaredConstructor().newInstance() ;
                                method.invoke(currentObject, newObject) ;
                                currentObject = newObject ;
                            } else {
                                currentObject = tempObject ;
                            }
                        }
                        // 进行属性内容的设置
                        Field field = currentObject.getClass().getDeclaredField(temp[temp.length - 1]) ; // 获取成员
                        Method setMethod = currentObject.getClass().getDeclaredMethod("set" + StringUtils.initcap(temp[temp.length - 1]), field.getType()) ;
                        Object convertValue = BeanUtils.convertAttributeValue(field.getType().getName(), attval[1]) ;
                        setMethod.invoke(currentObject, convertValue) ; // 调用setter方法设置内容
                    } else {
                        Field field = obj.getClass().getDeclaredField(attval[0]) ; // 获取成员
                        Method setMethod = obj.getClass().getDeclaredMethod("set" + StringUtils.initcap(attval[0]), field.getType()) ;
                        Object convertValue = BeanUtils.convertAttributeValue(field.getType().getName(), attval[1]) ;
                        setMethod.invoke(obj, convertValue) ; // 调用setter方法设置内容
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
        /**
         * 实现属性类型转换处理
         * @param type 属性类型，通过Field获取的
         * @param value 属性的内容，传入的都是字符串，需要将其变为指定类型
         * @return 转换后的数据
```

```
	 */
	private static Object convertAttributeValue(String type,String value) {
		if ("long".equals(type) || "java.lang.Long".equals(type)) {	// 长整型
			return Long.parseLong(value) ;
		} else if ("int".equals(type) || "java.lang.int".equals(type)) {
			return Integer.parseInt(value) ;
		} else if ("double".equals(type) || "java.lang.double".equals(type)) {
			return Double.parseDouble(value) ;
		} else if ("java.util.Date".equals(type)) {
			SimpleDateFormat sdf = null ;
			if (value.matches("\\d{4}-\\d{2}-\\d{2}")) {	// 日期类型
				sdf = new SimpleDateFormat("yyyy-MM-dd") ;
			} else if (value.matches("\\d{4}-\\d{2}-\\d{2} \\d{2}:\\d{2}:\\d{2}")) {
				sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss") ;
			} else {
				return new Date() ; // 当前日期
			}
			try {
				return sdf.parse(value) ;
			} catch (ParseException e) {
				return new Date() ; // 当前日期
			}
		} else {
			return value ;
		}
	}
}
class Company {
	private String name ;
	private Date createdate ;
	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
	public Date getCreatedate() {
		return createdate;
	}
	public void setCreatedate(Date createdate) {
		this.createdate = createdate;
	}
}
class Dept {
	private String dname ;
	private String loc ;
	private Company company ;
	public String getDname() {
		return dname;
	}
	public void setDname(String dname) {
		this.dname = dname;
```

```
	}
	public String getLoc() {
		return loc;
	}
	public void setLoc(String loc) {
		this.loc = loc;
	}
	public Company getCompany() {
		return company;
	}
	public void setCompany(Company company) {
		this.company = company;
	}

}
class Emp {
	private long empno ;
	private String ename ;
	private String job ;
	private double salary ;
	private Date hiredate ;
	private Dept dept ;

	public Dept getDept() {
		return dept;
	}
	public void setDept(Dept dept) {
		this.dept = dept;
	}
	public void setEname(String ename) {
		this.ename = ename;
	}
	public void setJob(String job) {
		this.job = job;
	}
	public String getEname() {
		return ename;
	}
	public String getJob() {
		return job;
	}
	public long getEmpno() {
		return empno;
	}
	public void setEmpno(long empno) {
		this.empno = empno;
	}
	public double getSalary() {
		return salary;
	}
	public void setSalary(double salary) {
		this.salary = salary;
```

```
	}
	public Date getHiredate() {
		return hiredate;
	}
	public void setHiredate(Date hiredate) {
		this.hiredate = hiredate;
	}
}
```