

Jakarta Commons 学习

1.	概述.....	4
2.	Commons Lang.....	4
2.1.	commons.lang 包	4
2.1.1.	ArrayUtils.....	5
2.1.2.	StringUtils	7
2.1.3.	BitField.....	10
2.1.4.	BooleanUtils.....	10
2.1.5.	CharRange.....	10
2.1.6.	CharUtils	11
2.1.7.	ClassUtils	11
2.1.8.	ObjectUtils	11
2.1.9.	RandomStringUtils.....	12
2.1.10.	SerializationUtils.....	12
2.1.11.	StringEscapeUtils	12
2.1.12.	SystemUtils.....	12
2.1.13.	Validate	12
2.1.14.	WordUtils.....	13
2.2.	commons.lang.builder 包	13
2.2.1.	CompareToBuilder.....	16
2.2.2.	EqualsBuilder.....	16
2.2.3.	HashCodeBuilder	17
2.2.4.	ToStringBuilder.....	18
2.2.5.	ToStringStyle 和 StandardToStringStyle.....	18
2.3.	commons.lang.math 包	19
2.4.	commons.lang.time 包	21
2.4.1.	DateFormatUtils.....	23
2.4.2.	DateUtils	23
2.4.3.	DurationFormatUtils	23
2.4.4.	SimpleDateFormat	24
2.4.5.	StopWatch	24
2.5.	commons.lang.enums 包	24
2.6.	commons.lang.exception 包	24
2.7.	commons.lang.mutable 包	24
3.	Commons BeanUtils	24
3.1.	commons.beanutils 包	25
3.1.1.	BeanUtils.....	25
3.1.2.	BeanUtilsBean.....	28
3.1.3.	PropertyUtils	29
3.1.4.	PropertyUtilsBean.....	29
3.1.5.	ConvertUtils	29
3.1.6.	ConvertUtilsBean.....	30
3.1.7.	ConstructorUtils	30
3.1.8.	MethodUtils.....	31
3.1.9.	DynaBean/DynaClass/DynaProperty	32
3.1.10.	BasicDynaBean/BasicDynaClass.....	32
3.1.11.	WrapDynaBean/WrapDynaClass/ConvertingWrapDynaBean.....	33
3.1.12.	JDBCDataSource/ResultSet/ResultSetIterator/RowSet.....	33
3.1.13.	LazyDynaBean/LazyDynaClass/LazyDynaMap.....	35
4.	Commons Collections.....	36
4.1.	commons.collections.bag	37
4.2.	commons.collections.buffer	38
4.3.	commons.collections.bidimap.....	39

4.4.	Comparator 组.....	40
4.5.	算子组.....	41
4.5.1.	Predicate.....	44
4.5.2.	Transformer	45
4.5.3.	Closure	46
4.5.4.	Factory	46
4.6.	Collections 组	47
4.6.1.	FastArrayList/FastHashMap/FastTreeMap	47
4.6.2.	ExtendedProperties	47
4.6.3.	BoundedCollection/BoundedMap	50
4.6.4.	MultiKey	50
4.7.	Map 组	51
4.7.1.	MultiMap.....	51
4.7.2.	BeanMap	51
4.7.3.	LazyMap	52
4.7.4.	CaseInsensitiveMap	53
4.7.5.	IdentityMap	53
4.7.6.	LRUMap	53
4.7.7.	MultiKeyMap.....	53
4.7.8.	ReferenceMap/ReferenceIdentityMap	54
4.7.9.	SingletonMap	54
4.8.	List 组	54
4.8.1.	CursorableLinkedList.....	54
4.8.2.	FixedSizeList.....	54
4.8.3.	LazyList.....	54
4.8.4.	NodeCachingLinkedList	55
4.8.5.	TreeList	55
5.	Commons Codec	55
5.1.	commons.codec.binary.....	56
5.1.1.	Base64.....	56
5.1.2.	Hex	56
5.1.3.	BinaryCodec.....	57
5.2.	commons.codec.digest	57
5.2.1.	DigestUtil	57
6.	Commons Betwixt.....	58
7.	Commons Digester.....	59
8.	Commons CLI.....	62
9.	Commons Discovery.....	63

1. 概述

可重用性是 Jakarta Commons 项目的灵魂所在。这些包在设计阶段就已经考虑了可重用性问题。其中一些包，例如 Commons 里面用来记录日志的 Logging 包，最初是为其他项目设计的，例如 Jakarta Struts 项目，当人们发现这些包对于其他项目也非常有用，能够极大地帮助其他项目的开发，他们决定为这些包构造一个"公共"的存放位置，这就是 Jakarta Commons 项目。

为了真正提高可重用性，每一个包都必须不依赖于其他大型的框架或项目。因此，Commons 项目的包基本上都是独立的，不仅是相对于其他项目的独立，而且相对于 Commons 内部的大部分其他包独立。虽然存在一些例外的情况，例如 Betwixt 包要用到 XML API，但绝大部分只使用最基本的 API，其主要目的就是要能够通过简单的接口方便地调用。

2. Commons Lang

2.1.commons.lang 包

Lang 包提供了一些有用的包含 static 方法的 Util 类。除了 6 个 Exception 类和 2 个已经 deprecated 的数字类之外，commons.lang 包共包含了 17 个实用的类：

ArrayUtils	用于对数组的操作，如添加、查找、删除、子数组、倒序、元素类型转换等
BitField	用于操作位元，提供了一些方便而安全的方法
BooleanUtils	用于操作和转换 boolean 或者 Boolean 及相应的数组
CharEncoding	包含了 Java 环境支持的字符编码，提供是否支持某种编码的判断
CharRange	用于设定字符范围并做相应检查
CharSet	用于设定一组字符作为范围并做相应检查
CharSetUtils	用于操作 CharSet
CharUtils	用于操作 char 值和 Character 对象
ClassUtils	用于对 Java 类的操作，不使用反射
ObjectUtils	用于操作 Java 对象，提供 null 安全的访问和其他一些功能
RandomStringUtils	用于生成随机的字符串
SerializationUtils	用于处理对象序列化，提供比一般 Java 序列化更高级的处理能力

StringEscapeUtils	用于正确处理转义字符，产生正确的 Java、JavaScript、HTML、XML 和 SQL 代码
StringUtils	处理 String 的核心类，提供了相当多的功能
SystemUtils	在 java.lang.System 基础上提供更方便的访问，如用户路径、Java 版本、时区、操作系统等判断
Validate	提供验证的操作，有点类似 assert 断言
WordUtils	用于处理单词大小写、换行等

2.1.1. ArrayUtils

给 `int[]`, `String[]` 等基本类型或基本包装类型的 `Array` 加上类似 `CollectionsAPI` 提供的 `add`, `remove`, `reverse`, `subarray`, `indexOf`, `lastIndexOf`, `isEmpty`, `clone`，把 **null** 认为是空数组

函数	参数	说明
add	<code>add(Object[] array, int index, Object element)</code> <code>add(Object[] array, Object element)</code>	在数组的某个位置或者最后插入一个值，相当于对 <code>System.arraycopy</code> 做了个封装(数组可以是 null)
addAll	<code>addAll(Object[] array1, Object[] array2)</code>	两个同类型数组叠加，相当于对 <code>System.arraycopy</code> 做了个封装(数组可以是 null)
clone	<code>clone(Object[] array)</code>	拷贝一个数组(数组可以是 null)
contains	<code>contains (Object[] array, Object objectToFind)</code>	检查值是否在数组内，数组可以是 null，值如果是 null，则查数组内为 null 这一项
getLength	<code>getLength(Object array)</code>	得到数组长度(数组可以是 null)
indexOf	<code>indexOf(Object[] array, Object objectToFind)</code> <code>indexOf(Object[] array, Object objectToFind, int startIndex)</code>	搜索数组的值(数组可以是 null)
isEmpty	<code>isEmpty(Object[] array)</code>	检查数组为 null 或为空
isEqual	<code>isEqual(Object array1, Object array2)</code>	比较两个数组中的内容，支持多维数组
isSameLength	<code>isSameLength(Object[] array1, Object[] array2)</code>	两个数组长度是否相等，null 被视作空数组
lastIndexOf	<code>lastIndexOf(Object[] array, Object objectToFind)</code> <code>lastIndexOf(Object[] array, Object objectToFind, int startIndex)</code>	从下往上找搜索数组的值（数组可以是 null）
remove	<code>remove(Object[] array, int index)</code>	删除数组中某个位置的值
removeElement	<code>removeElement(Object[] array, Object element)</code>	删除数组中第一个匹配的值
reverse	<code>reverse(Object[] array)</code>	反转数组
subarray	<code>subarray(Object[] array, int startIndexInclusive, int endIndexExclusive)</code>	取得数组的一部分
toMap	<code>toMap(Object[] array)</code> <code>Map colorMap = Array.toMap(new String[][] {{</code>	将多维数组转为 map, 第一维为 key，第二维为 value

	<pre> {"RED", "#FF0000"}, {"BLUE", "#0000FF"} </pre>	
<u>toObject</u>		将 int 数组转成 Integer 数组，
<u>toPrimitive</u>		将 Integer 数组转成 int 数组
<u>toString</u>	<pre> toString(Object array) toString(Object array, String stringIfNull) </pre>	将数组转换成类似 {a,b} 文本

这段代码说明了我们可以如何方便的利用 `ArrayUtils` 类帮我们完成数组的打印、查找、克隆、倒序、以及值型/对象数组之间的转换等操作。

```

package sean.study.jakarta.common.lang;

import java.util.Map;
import org.apache.commons.lang.ArrayUtils;

public class ArrayUtilsUsage {

    public static void main(String[] args) {

        // data setup
        int[] intArray1 = { 2, 4, 8, 16 };
        int[][] intArray2 = { { 1, 2 }, { 2, 4 }, { 3, 8 }, { 4, 16 } };
        Object[][] notAMap = {
            { "A", new Double(100) },
            { "B", new Double(80) },
            { "C", new Double(60) },
            { "D", new Double(40) },
            { "E", new Double(20) }
        };

        // printing arrays
        System.out.println("intArray1: " + ArrayUtils.toString(intArray1));
        System.out.println("intArray2: " + ArrayUtils.toString(intArray2));
        System.out.println("notAMap: " + ArrayUtils.toString(notAMap));

        // finding items
        System.out.println("intArray1 contains '8'? "
            + ArrayUtils.contains(intArray1, 8));
        System.out.println("intArray1 index of '8'? "
            + ArrayUtils.indexOf(intArray1, 8));
        System.out.println("intArray1 last index of '8'? "
            + ArrayUtils.lastIndexOf(intArray1, 8));

        // cloning and resversing
        int[] intArray3 = ArrayUtils.clone(intArray1);
        System.out.println("intArray3: " + ArrayUtils.toString(intArray3));
        ArrayUtils.reverse(intArray3);
        System.out.println("intArray3 reversed: "

```

```
        + ArrayUtils.toString(intArray3));

    // primitive to Object array
    Integer[] integerArray1 = ArrayUtils.toObject(intArray1);
    System.out.println("integerArray1: "
        + ArrayUtils.toString(integerArray1));

    // build Map from two dimensional array
    Map map = ArrayUtils.toMap(notAMap);
    Double res = (Double) map.get("C");
    System.out.println("get 'C' from map: " + res);

    }

}
```

以下是运行结果:

```
intArray1: {2,4,8,16}
intArray2: {{1,2},{2,4},{3,8},{4,16}}
notAMap: {{A,100.0},{B,80.0},{C,60.0},{D,40.0},{E,20.0}}
intArray1 contains '8'? true
intArray1 index of '8'? 2
intArray1 last index of '8'? 2
intArray3: {2,4,8,16}
intArray3 reversed: {16,8,4,2}
integerArray1: {2,4,8,16}
get 'C' from map: 60.0
```

2.1.2. StringUtils

StringUtils 类具有简单而强大的处理能力，从检查空串（对 null 的情况处理很得体），到分割子串，到生成格式化的字符串，使用都很简洁，也很直截了当。

IsEmpty/IsBlank	checks if a String contains text
Trim/Strip	removes leading and trailing whitespace
Equals	compares two strings null-safe

IndexOf/LastIndexOf/Contains	null-safe index-of checks
IndexOfAny/LastIndexOfAny/IndexOfAnyBut/LastIndexOfAnyBut	index-of any of a set of Strings
ContainsOnly/ContainsNone	does String contains only/none of these characters
Substring/Left/Right/Mid	null-safe substring extractions
SubstringBefore/SubstringAfter/SubstringBetween	substring extraction relative to other strings
Split/Join	splits a String into an array of substrings and vice versa
Remove/Delete	removes part of a String
Replace/Overlay	Searches a String and replaces one String with another
Chomp/Chop	removes the last part of a String
LeftPad/RightPad/Center/Repeat	pads a String
UpperCase/LowerCase/SwapCase/Capitalize/Uncapitalize	changes the case of a String
CountMatches	counts the number of occurrences of one String in another
IsAlpha/IsNumeric/IsWhitespace/IsAsciiPrintable	checks the characters in a String
DefaultString	protects against a null input String

Reverse/ReverseDelimited	reverses a String
Abbreviate	abbreviates a string using ellipsis
Difference	compares two Strings and reports on their differences
LevenshteinDistance	the number of changes needed to change one String into another

处理 String 的原则:

null	null
empty	a zero-length string ("")
space	the space character (' ', char 32)
whitespace	the characters defined by Character.isWhitespace(char)
trim	the characters <= 32 as in String.trim()

从下面代码中我们可以大致了解到这个 StringUtils 类简单而强大的处理能力,从检查空串(对 null 的情况处理很得体),到分割子串,到生成格式化的字符串,使用都很简洁,也很直截了当。

```
package sean.study.jakarta.common.lang;

import org.apache.commons.lang.StringUtils;

public class StringUtilsAndWordUtilsUsage {

    public static void main(String[] args) {

        // data setup
        String str1 = "";
        String str2 = " ";
        String str3 = "\t";
        String str4 = null;
        String str5 = "123";
        String str6 = "ABCDEFGH";
        String str7 = "It feels good to use Jakarta Commons.\r\n";

        // check for empty strings
```



```

System.out.println("=====");
System.out.println("Is str1 blank? " + StringUtils.isBlank(str1));
System.out.println("Is str2 blank? " + StringUtils.isBlank(str2));
System.out.println("Is str3 blank? " + StringUtils.isBlank(str3));
System.out.println("Is str4 blank? " + StringUtils.isBlank(str4));

// check for numerics
System.out.println("=====");
System.out.println("Is str5 numeric? " + StringUtils.isNumeric(str5));
System.out.println("Is str6 numeric? " + StringUtils.isNumeric(str6));

// reverse strings / whole words
System.out.println("=====");
System.out.println("str6: " + str6);
System.out.println("str6 reversed: " + StringUtils.reverse(str6));
System.out.println("str7: " + str7);
String str8 = StringUtils.chomp(str7);
str8 = StringUtils.reverseDelimited(str8, ' ');
System.out.println("str7 reversed whole words : \r\n" + str8);

// build header (useful to print log messages that are easy to locate)
System.out.println("=====");
System.out.println("print header:");
String padding = StringUtils.repeat("=", 50);
String msg = StringUtils.center(" Customised Header ", 50, "%");
Object[] raw = new Object[]{padding, msg, padding};
String header = StringUtils.join(raw, "\r\n");
System.out.println(header);

}
}

```

输出的结果如下：

```

=====
Is str1 blank? true
Is str2 blank? true
Is str3 blank? true
Is str4 blank? true
=====
Is str5 numeric? true
Is str6 numeric? false
=====
str6: ABCDEFG

```

```
str6 reversed: GFEDCBA
str7: It feels good to use Jakarta Commons.

str7 reversed whole words :
Commons. Jakarta use to good feels It
=====

print header:
=====
%%%%%%%%%% Customised Header %%%%%%%%%%%
=====
```

2.1.3. BitField

提供 C 中常见的置位操作。比如，'0101001b',其 isSet(01b)就是 true。

2.1.4. BooleanUtils

给 boolean 附加一些功能，比如转换成'Yes/No','on/off',或者转换到 int 等。

2.1.5. CharRange

连续的字符容器对象，比如说你要判定 H 是否在 A 和 P 之间，或者要生成一个 A~Z 的字符串，可以用这个。

2.1.6. CharUtils

判定 char 的大小写，是否是数字，是否可打印等等。可以转换为 unicode 的字符串。

2.1.7. ClassUtils

用于在 Class 和 ClassName 之间转换，获得 Class 的 Interfaces,superclass, packageName,短名字，把 int 换为 Integer 等等。

函数	参数	说明
<u>getAllInterfaces</u>	getAllInterfaces(Class cls)	得到一个 class 所有的接口，再通过递归获得这些接口的父接口，会剔重。

<u>getAllSuperclasses</u>	getAllSuperclasses(Class cls)	取得一个 class 所有的父 class
<u>getPackageName</u>	getPackageName(Class cls) getPackageName(Object object, String valueIfNull) getPackageName(String className)	得到一个 class 或 Object 或者 class 名字的包名
<u>getShortClassName</u>	getShortClassName(Class cls) getShortClassName(Object object, String valueIfNull) getShortClassName(String className)	得到一个 class 或 Object 或者 class 名字的类名(不包括包名)
<u>isAssignable</u>	isAssignable(Class[] classArray, Class[] toClassArray) isAssignable(Class cls, Class toClass)	检查一组 class 是否可以转成另一组 class,支持 int 等类型
<u>isInnerClass</u>	isInnerClass(Class cls)	检查某个类是否是内嵌类(an inner class or static nested class)
<u>primitivesToWrappers</u>	primitivesToWrappers(Class[] classes) primitiveToWrapper(Class cls)	将 Integer.TYPE 转成 Integer.class

2.1.8. ObjectUtils

支持 null 的一些 Object 操作。

函数	参数	说明
<u>defaultIfNull</u>	defaultIfNull(Object object, Object defaultValue)	如果第一个为 null，用后一个代替
<u>equals</u>	equals(Object object1, Object object2)	比较两个 object，支持 null
<u>hashCode</u>	hashCode(Object obj)	支持 null,null 返回 0
<u>toString</u>	toString(Object obj) toString(Object obj, String nullStr)	支持 null 的 toString

2.1.9. RandomStringUtils

生成固定长度\类型的随机字符串。

2.1.10. SerializationUtils

通过序列化来完全拷贝一个复杂对象，以及其他一些增强

函数	参数	说明
<u>clone</u>	clone(Serializable object)	通过序列化来完全拷贝一个复杂对象，速度慢但是准确
<u>deserialize</u>	deserialize(byte[] objectData)	deserialize 增强

	deserialize(InputStream inputStream)	
serialize	serialize(Serializable obj) serialize(Serializable obj, OutputStream outputStream)	serialize 增强

2.1.11. StringEscapeUtils

字符串转义工具，支持 Java, Java Script, HTML, XML, 和 SQL

转	反转	说明
escapeHtml (String str)	unescapeHtml (String str)	转义 HTML
escapeJava (String str)	unescapeJava (String str)	转义 java
escapeJavaScript (String str)	unescapeJavaScript (String str)	转义 javascript
escapeSql (String str)		转义 sql
escapeXml (String str)	unescapeXml (String str)	转义 xml

2.1.12. SystemUtils

获得 JavaHome, Version, userDir, userHome 等等.用于判定是否满足某个最小版本限制很合适

2.1.13. Validate

用于简化 if 判定.当不满足条件时,抛出 IllegalArgumentException

2.1.14. WordUtils

对英文单词的处理，包括首字大写，反转等等

2.2.commons.lang.builder 包

lang.builder 包包含了一组用于产生每个 Java 类中都常使用到的 toString()、hashCode()、equals()、compareTo()等等方法的构造器

CompareToBuilder	用于辅助实现 Comparable.compareTo(Object)方法
EqualsBuilder	用于辅助实现 Object.equals()方法
HashCodeBuilder	用于辅助实现 Object.hashCode()方法
ToStringBuilder	用于辅助实现 Object.toString()方法

ReflectionToStringBuilder	使用反射机制辅助实现 <code>Object.toString()</code> 方法
ToStringStyle	辅助 <code>ToStringBuilder</code> 控制输出格式
StandardToStringStyle	辅助 <code>ToStringBuilder</code> 控制标准格式

我们知道，在实际应用中，其实经常需要在运行过程中判定对象的知否相等、比较、取 hash、和获取对象基本信息（一般是产生 log 日志）。然而实现这些 `compareTo`、`equals`、`hashCode`、`toString` 其实并非那么直截了当，甚至稍有不注意就可能造成难以追踪的 bug，而且这些方法手工维护的话，比较繁琐，也容易出错。于是 Commons Lang 在 `builder` 这个包中提供了上述辅助类，为我们简化这些方法的实现和维护。

来看一个例子：

```
package sean.study.jakarta.common.lang;

import java.util.Date;
import org.apache.commons.lang.builder.*;

public class BuilderUsage {
    public static void main(String[] args) {
        Staff staff1 = new Staff(123, "John Smith", new Date());
        Staff staff2 = new Staff(456, "Jane Smith", new Date());
        System.out.println("staff1's info: " + staff1);
        System.out.println("staff2's info: " + staff2);
        System.out.println("staff1's hash code: " + staff1.hashCode());
        System.out.println("staff2's hash code: " + staff2.hashCode());
        System.out.println("staff1 equals staff2? " + staff1.equals(staff2));
    }
}

class Staff implements Comparable {
    private long staffId;
    private String staffName;
    private Date dateJoined;
    public Staff() { }

    public Staff(long staffId, String staffName, Date dateJoined) {
        this.staffId = staffId;
        this.staffName = staffName;
        this.dateJoined = dateJoined;
    }

    public int compareTo(Object o) {
        int res = -1;
        if (o != null && Staff.class.isAssignableFrom(o.getClass())) {
            Staff s = (Staff) o;
            res = new CompareToBuilder()
                .append(dateJoined, s.getDateJoined())
                .append(staffName, s.getStaffName()).toComparison();
        }
    }
}
```

```

        return res;
    }

    public boolean equals(Object o) {
        boolean res = false;
        if (o != null && Staff.class.isAssignableFrom(o.getClass())) {
            Staff s = (Staff) o;
            res = new EqualsBuilder()
                .append(staffId, s.getStaffId())
                .isEquals();
        }
        return res;
    }

    public int hashCode() {
        return new HashCodeBuilder(11, 23).append(staffId).toHashCode();
    }

    public String toString() {
        return new ToStringBuilder(this, ToStringStyle.MULTI_LINE_STYLE)
            .append("staffId", staffId)
            .append("staffName", staffName)
            .append("dateJoined", dateJoined)
            .toString();
    }

    public Date getDateJoined() {
        return dateJoined;
    }

    public void setDateJoined(Date dateJoined) {
        this.dateJoined = dateJoined;
    }

    public long getStaffId() {
        return staffId;
    }

    public void setStaffId(long staffId) {
        this.staffId = staffId;
    }

    public String getStaffName() {
        return staffName;
    }

    public void setStaffName(String staffName) {
        this.staffName = staffName;
    }
}

```

以下是运行结果:

```

staff1's info: sean.study.jakarta.common.lang.Staff@190d11[
    staffId=123

```

```
    staffName=John Smith
    dateJoined=Sat Jul 30 13:18:45 CST 2005
]
staff2's info: sean.study.jakarta.common.lang.Staff@1fb8ee3[
    staffId=456
    staffName=Jane Smith
    dateJoined=Sat Jul 30 13:18:45 CST 2005
]
staff1's hash code: 376
staff2's hash code: 709
staff1 equals staff2? false
```

这些 builder 使用起来都很简单，new 一个实例，append 需要参与的信息，最后加上 toComparison、isEqual、hashCode、toString 这样的结尾即可。相应的，如果你不需要这样级别的控制，也可以使用利用反射机制的版本自动化实现需要的功能，如：

```
public int compareTo(Object o) {
    return CompareToBuilder.reflectionCompare(this, o);
}

public boolean equals(Object o) {
    return EqualsBuilder.reflectionEquals(this, o);
}

public int hashCode() {
    return HashCodeBuilder.reflectionHashCode(this);
}

public String toString() {
    return ReflectionToStringBuilder.toString(this);
}
```

尤其当我们在项目中不希望过多的参与到对这些对象方法的维护时，采用 Commons 提供的利用反射的这些 API 就成了方便而相对安全的一个方案。

2.2.1. CompareToBuilder

在数据对象的排序处理中，我们常常会比较 Object 中的某个值或字符串，可以利用 Comparable 来处理，或是利用下面的方法

```
public class MyClass {
    String field1;
    int field2;
    boolean field3;

    ...

    public int compareTo(Object o) {
```

```

MyClass myClass = (MyClass) o;
return new CompareToBuilder()
    .appendSuper(super.compareTo(o))
    .append(this.field1, myClass.field1)
    .append(this.field2, myClass.field2)
    .append(this.field3, myClass.field3)
    .toComparison();
}
}

```

如果, 你需要对整个 `Object` 做比较呢, 那么就直接写简单的 `reflectionCompare`, 就可以得到每个 `field` 的比较结果了

```

public int compareTo(Object o) {
    return CompareToBuilder.reflectionCompare(this, o);
}

```

2.2.2. EqualsBuilder

`EqualsBuilder` 和 `CompareToBuilder` 类似, 只是加强实现相等的部分, 如果你对两个对象的比较只是限于某些 `field` 得比较, 那么就可以利用以下的方法来确认是否相等..

```

public boolean equals(Object o) {
    if ( !(o instanceof MyClass) ) {
        return false;
    }
    MyClass rhs = (MyClass) o;
    return new EqualsBuilder()
        .appendSuper(super.equals(o))
        .append(field1, rhs.field1)
        .append(field2, rhs.field2)
        .append(field3, rhs.field3)
        .isEquals();
}
}

```

如果, 你需要对整个 `Object` 做比较呢, 那么就直接写简单的 `reflectionCompare`, 就可以得到每个 `field` 的比较结果了

```

public boolean equals(Object o) {
    return EqualsBuilder.reflectionEquals(this, o);
}

```


2.2.3. hashCodeBuilder

在 Class 之中最好存在一个 hashCode 让它根据内存数值生成具有唯一性的 hash 判断值, 不过建立一个 hashCode 生成器可能非常麻烦, 不用烦恼, 使用 commons-lang 将会让程序更简单 ~

```
public class Person {
    String name;
    int age;
    boolean isSmoker;
    ...

    public int hashCode() {
        //你可以写死 ( hard-coded), 随机产生, 只要非零并且为奇数的值就可以了.
        //最好是每个 class 都具有不同的 hashCode 值.
        return new hashCodeBuilder(17, 37).
            append(name).
            append(age).
            append(smoker).
            toHashCode();
    }
}
```

如果, 你需要对整个 Object 做比较呢, 那么就写简单的 reflectionCompare, 就可以得到每个 field 的比较结果了

```
public int hashCode() {
    return hashCodeBuilder.reflectionHashCode(this);
}
```

2.2.4. ToStringBuilder

我们会要求所有的数据对象都要撰写 toString() 覆写 Object.toString(), 因为这样 debug 的时候才不需要重复地去找错误的地方.

```
public class Person {
    String name;
    int age;
    boolean isSmoker;
    ...

    public String toString() {
        return new ToStringBuilder(this).
            append("name", name).
            ...
    }
}
```

```
        append("age", age).
        append("smoker", smoker).
        toString();
    }
}
```

如果，你需要对整个 Object 做比较呢，那么就直接写简单的 reflectionCompare，就可以得到每个 field 的比较结果了

```
public String toString() {
    return ToStringBuilder.reflectionToString(this);
}
```

2.2.5. ToStringStyle 和 StandardToStringStyle

辅助 ToStringBuilder 控制输出格式，在 new ToStringBuilder() 时用

ToStringStyle.DEFAULT_STYLE	The default toString style.
ToStringStyle.MULTI_LINE_STYLE	The multi line toString style.
ToStringStyle.NO_FIELD_NAMES_STYLE	The no field names toString style.
ToStringStyle.SHORT_PREFIX_STYLE	The short prefix toString style.
ToStringStyle.SIMPLE_STYLE	The simple toString style.

2.3. commons.lang.math 包

包中共有 10 个类，这些类可以归纳成四组：

Fraction	处理分数
NumberUtils	处理数值
Range 、 NumberRange 、 IntRange 、 LongRange 、 FloatRange 、 DoubleRange	处理数值范围
JVMRandom 、 RandomUtils	处理随机数

其中 NumberUtils.inNumber(String)方法，它会正确判断出给定的字符串是否是合法的数值，而 StringUtils.isNumeric(String)只能判断一个字符串是否是由纯数字字符组成。

举个例子分别说明上述四组的使用方法：

```
package sean.study.jakarta.common.lang;
import org.apache.commons.lang.*;
import org.apache.commons.lang.math.*;

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

```

demoFraction();
demoNumberUtils();
demoNumberRange();
demoRandomUtils();
}

public static void demoFraction() {
    System.out.println(StringUtils.center(" demoFraction ", 30, "="));
    Fraction myFraction = Fraction.getFraction(144, 90);
    // Fraction myFraction = Fraction.getFraction("1 54/90");
    System.out.println("144/90 as fraction: " + myFraction);
    System.out.println("144/90 to proper: " + myFraction.toProperString());
    System.out.println("144/90 as double: " + myFraction.doubleValue());
    System.out.println("144/90 reduced: " + myFraction.reduce());
    System.out.println("144/90 reduced proper: "
        + myFraction.reduce().toProperString());
    System.out.println();
}

public static void demoNumberUtils() {
    System.out.println(StringUtils.center(" demoNumberUtils ", 30, "="));
    System.out.println("Is 0x3F a number? "
        + StringUtils.capitalize(BooleanUtils.toStringYesNo(NumberUtils
            .isNumber("0x3F"))) + ".");
    double[] array = { 1.0, 3.4, 0.8, 7.1, 4.6 };
    double max = NumberUtils.max(array);
    double min = NumberUtils.min(array);
    String arrayStr = ArrayUtils.toString(array);
    System.out.println("Max of " + arrayStr + " is: " + max);
    System.out.println("Min of " + arrayStr + " is: " + min);
    System.out.println();
}

public static void demoNumberRange() {
    System.out.println(StringUtils.center(" demoNumberRange ", 30, "="));
    Range normalScoreRange = new DoubleRange(90, 120);
    double score1 = 102.5;
    double score2 = 79.9;
    System.out.println("Normal score range is: " + normalScoreRange);
    System.out.println("Is "
        + score1
        + " a normal score? "
        + StringUtils
            .capitalize(BooleanUtils.toStringYesNo(normalScoreRange
                .containsDouble(score1))) + ".");
    System.out.println("Is "
        + score2

```

```

        + " a normal score? "
        + StringUtils
            .capitalize(BooleanUtils.toStringYesNo(normalScoreRange
                .containsDouble(score2))) + ".");
    System.out.println();
}

public static void demoRandomUtils() {
    System.out.println(StringUtils.center(" demoRandomUtils ", 30, "="));
    for (int i = 0; i < 5; i++) {
        System.out.println(RandomUtils.nextInt(100));
    }
    System.out.println();
}
}

```

以下是运行结果：

```

===== demoFraction =====
144/90 as fraction: 144/90
144/90 to proper: 1 54/90
144/90 as double: 1.6
144/90 reduced: 8/5
144/90 reduced proper: 1 3/5

===== demoNumberUtils =====
Is 0x3F a number? Yes.
Max of {1.0,3.4,0.8,7.1,4.6} is: 7.1
Min of {1.0,3.4,0.8,7.1,4.6} is: 0.8

===== demoNumberRange =====
Normal score range is: Range[90.0,120.0]
Is 102.5 a normal score? Yes.
Is 79.9 a normal score? No.

===== demoRandomUtils =====
75
63
40
21
92

```

2.4.commons.lang.time 包

包中共有 10 个类，这些类可以归纳成四组：

DateFormatUtils	提供格式化日期和时间的功能及相关常量
DateUtils	在 Calendar 和 Date 的基础上提供更方便的访问
DurationFormatUtils	提供格式化时间跨度的功能及相关常量
FastDateFormat	为 SimpleDateFormat 提供一个的线程安全的替代类
StopWatch	一个方便的计时器

举例：

```
package sean.study.jakarta.commons.lang;
import java.util.*;
import org.apache.commons.lang.*;
import org.apache.commons.lang.time.*;

public class DateTimeUsage {
    public static void main(String[] args) {
        demoDateUtils();
        demoStopWatch();
    }

    public static void demoDateUtils() {
        System.out.println(StringUtils.center(" demoDateUtils ", 30, "="));
        Date date = new Date();
        String isoDateTime = DateFormatUtils.ISO_DATETIME_FORMAT.format(date);
        String isoTime = DateFormatUtils.ISO_TIME_NO_T_FORMAT.format(date);
        FastDateFormat fdf = FastDateFormat.getInstance("yyyy-MM");
        String customDateTime = fdf.format(date);
        System.out.println("ISO_DATETIME_FORMAT: " + isoDateTime);
        System.out.println("ISO_TIME_NO_T_FORMAT: " + isoTime);
        System.out.println("Custom FastDateFormat: " + customDateTime);
        System.out.println("Default format: " + date);
        System.out.println("Round HOUR: " + DateUtils.round(date, Calendar.HOUR));
        System.out.println("Truncate HOUR: " + DateUtils.truncate(date,
Calendar.HOUR));
        System.out.println();
    }

    public static void demoStopWatch() {
        System.out.println(StringUtils.center(" demoStopWatch ", 30, "="));
        StopWatch sw = new StopWatch();
        sw.start();
        operationA();
        sw.stop();
    }
}
```

```
        System.out.println("operationA used " + sw.getTime() + " milliseconds.");
        System.out.println();
    }

    public static void operationA() {
        try {
            Thread.sleep(999);
        }
        catch (InterruptedException e) {
            // do nothing
        }
    }
}
```

以下是运行结果：

```
===== demoDateUtils =====
ISO_DATETIME_FORMAT: 2005-08-01T12:41:51
ISO_TIME_NO_T_FORMAT: 12:41:51
Custom FastDateFormat: 2005-08
Default format: Mon Aug 01 12:41:51 CST 2005
Round HOUR: Mon Aug 01 13:00:00 CST 2005
Truncate HOUR: Mon Aug 01 12:00:00 CST 2005

===== demoStopWatch =====
operationA used 1000 milliseconds.
```

2.4.1. DateFormatUtils

将日期转换成文本，方法是 static 的，还有几个缺省格式的 FastDateFormat，

2.4.2. DateUtils

对 Date 和 Calendar 一些相当有用的操作

函数	参数	说明
isSameDay	isSameDay(Calendar cal1, Calendar cal2) isSameDay(Date date1, Date date2)	检查两个时间是否同一天
isSameInstant	isSameInstant(Calendar cal1, Calendar cal2) isSameInstant(Date date1, Date date2)	是否同一时间
isSameLocalTime	isSameLocalTime(Calendar cal1, Calendar cal2)	
parseDate	parseDate(String str, String[] parsePatterns)	用多个模式来解析时间
round	round(Calendar date, int field) round(Date date, int field) round(Object date, int field)	对日期某个字段作四舍五入

<u>truncate</u>	truncate(Calendar date, int field) truncate(Date date, int field) truncate(Object date, int field)	对日期某个字段取整
---------------------------------	--	-----------

2.4.3. DurationFormatUtils

格式化一个时间段：如：

```
DurationFormatUtils.formatPeriod(
    new SimpleDateFormat("yymmdd").parse("050101").getTime()
    , new SimpleDateFormat("yymmdd").parse("070121").getTime(), "间隔 y 年 m 月
d 天")
， 返回 “间隔 2 年 0 月 20 天”
```

函数	参数	说明
<u>formatDuration</u>	formatDuration(long durationMillis, String format) formatDuration(long durationMillis, String format, boolean padWithZeros)	对以毫秒为单位的时间段进行格式化
<u>formatDurationHMS</u>	formatDurationHMS(long durationMillis)	"H:mm:ss.SSS"格式
<u>formatPeriod</u>	formatPeriod(long startMillis, long endMillis, String format)	对开始和结束进行格式化

2.4.4. SimpleDateFormat

为 SimpleDateFormat 提供一个的线程安全的替代类，只作 format，不作 parse

2.4.5. Stopwatch

计时器，可用在记录操作时间

函数	参数	说明
<u>start</u> ()		开始计时
<u>stop</u> ()		停止计时
<u>suspend</u> ()		暂停
<u>resume</u> ()		继续
<u>reset</u> ()		重置
<u>getTime</u> ()		得到时间

2.5.commons.lang.enums 包

提供对枚举的支持

2.6.commons.lang.exception 包

lang.exception 包用于处理 Java 标准 API 中的 exception, 为 1.4 之前版本提供 Nested Exception 功能

2.7.commons.lang.mutable 包

lang.mutable 用于包装值型变量

3. Commons BeanUtils

BeanUtils 最核心的好处在于: 我们在编码时, 并不需要知道我们处理的 JavaBeans 具体是什么类型, 有哪些属性, 这些信息是可以动态获取的, 甚至我们都可以不必去关心事实上是否存在这样一个具体的 JavaBean 类。我们只需要知道有一个 JavaBean 的实例, 我们需要从中取得某个属性, 设定某个属性的值, 或者仅仅是需要一个属性表。要做到这些, 依靠 Sun 提供的 JavaBean 规范似乎找不到一个很直接的方式, 除非硬编码, 将 getXxxx()和 setXxxx()直接写进我们的程序。但是这样就大大增加了代码的复杂度、耦合性和维护成本。

Commons BeanUtils 一共包括如下 5 个包:

<code>commons.beanutils</code>	核心包, 定义一组 Utils 类和需要用到的接口规范
<code>commons.beanutils.converters</code>	转换 String 到需要类型的类, 实现 Converter 接口
<code>commons.beanutils.locale</code>	beanutils 的 locale 敏感版本
<code>commons.beanutils.locale.converters</code>	converters 的 locale 敏感版本
<code>commons.collection</code>	beanutils 使用到的 Collection 类

3.1.commons.beanutils 包

4 个基础接口:

<code>Converter</code>	只要实现了这个 Converter 接口并注册到 ConvertUtils 类即可
------------------------	---

	被我们的 BeanUtils 包所使用，它的主要目的是提供将给定的 Object 实例转换为目标类型的算法。我们可以在 beanutils.converters 包中找到相当多的已经实现的转换器。
DynaBean	该接口定义的是一个动态的 JavaBean，它的属性类型、名称和值都是可以动态改变的。
DynaClass	该接口定义的是针对实现了 DynaBean 接口的类的 java.lang.Class 对象，提供如 getName()、newInstance()等方法。
MutableDynaClass	该接口是对 DynaClass 的扩展，使得动态 bean 的属性可以动态增加或删除。

3.1.1. BeanUtils

提供通过反射机制填写 JavaBeans 属性的工具/静态方法，BeanUtils 会对用到的 class 所有信息作缓存，所以该 class 第一次使用速度比较慢。

getProperty 和 setProperty 使用方法：

BeanUtils 可以直接 get 和 set 一个属性的值。它将 property 分成 5 种类型：

Simple	简单类型，如 String、Int
Indexed	索引类型，如数组、arrayList，使用时类似 name[index]
Mapped	是指 Map，比如 HashMap，使用时类似 name(key)
Nested	使用时类似 name1.name2.name3
Combined	使用时类似 name1.name2[index].name3(key)

访问不同类型的数据可以直接调用函数 getProperty 和 setProperty。它们都只有 2 个参数，第一个是 JavaBean 对象，第二个是要操作的属性名。

```
public class Company{
    private String name;
    private HashMap address = new HashMap();
    private String[] otherInfo;
    private ArrayList product;
    private ArrayList employee;
    private HashMap telephone;
    ...
}
Company c = new Company();
c.setName("Simple");
```

对于 Simple 类型，参数二直接是属性名即可

```
System.out.println(BeanUtils.getProperty(c, "name"));
```

对于 Map 类型，则需要以“属性名（key 值）”的形式

```
System.out.println(BeanUtils.getProperty(c, "address (A2)"));
HashMap am = new HashMap();
am.put("1", "234-222-122211");
```

```
am.put("2", "021-086-1232323");
BeanUtils.setProperty(c, "telephone", am);
System.out.println(BeanUtils.getProperty(c, "telephone (2)"));
```

对于 **Indexed**, 则为“属性名[索引值]”, 注意这里对于 **ArrayList** 和数组都可以用一样的方式进行操作。

```
System.out.println(BeanUtils.getProperty(c, "otherInfo[2]"));
BeanUtils.setProperty(c, "product[1]", "NOTES SERVER");
System.out.println(BeanUtils.getProperty(c, "product[1]"));
```

当然这 3 种类也可以组合使用

```
System.out.println(BeanUtils.getProperty(c, "employee[1].name"))
```

copyProperties 使用方法:

copyProperties 可直接进行 Bean 之间的 clone, 但是这种 copy 都是浅拷贝, 复制后的 2 个 Bean 的同一个属性可能拥有同一个对象的 ref, 这个在使用时要小心, 特别是对于属性为自定义类的情况。

例如 **Teacher** 和 **TeacherForm** 两个对象, 传统的方式是使用类似下面的语句对属性逐个赋值:

```
//得到 TeacherFormTeacherForm
teacherForm=(TeacherForm) form;
//构造 Teacher 对象
Teacher teacher=new Teacher();
//赋值
teacher.setName(teacherForm.getName());
teacher.setAge(teacherForm.getAge());
teacher.setGender(teacherForm.getGender());
teacher.setMajor(teacherForm.getMajor());
teacher.setDepartment(teacherForm.getDepartment());
```

而使用 **BeanUtils** 后, 代码就大大改观了

```
//得到 TeacherFormTeacherForm
teacherForm=(TeacherForm) form;
//构造 Teacher 对象
Teacher teacher=new Teacher();
//赋值
BeanUtils.copyProperties(teacher, teacherForm);
```

如果 **Teacher** 和 **TeacherForm** 间存在名称不相同的属性, 则 **BeanUtils** 不对这些属性进行处理, 需要程序员手动处理。例如: **Teacher** 包含 **modifyDate** (该属性记录最后修改日期, 不需要用户在界面中输入) 属性而 **TeacherForm** 无此属性, 那么在上面代码的 **copyProperties()** 后还要加上一句: **teacher.setModifyDate(new Date());**

除 **BeanUtils** 外还有一个名为 **PropertyUtils** 的工具类, 它也提供 **copyProperties()** 方法, 作用与 **BeanUtils** 的同名方法十分相似, 主要的区别在于后者提供类型转换功能, 即发现两个 **JavaBean** 的同名属性为不同类型时, 在支持的数据类型范围内进行转换, 而前者不支持这个功能, 但是速度会更快一些。 **BeanUtils** 支持的转换类型如下:

- java.lang.BigDecimal

- java.lang.BigInteger
- boolean and java.lang.Boolean
- byte and java.lang.Byte
- char and java.lang.Character
- java.lang.Class
- double and java.lang.Double
- float and java.lang.Float
- int and java.lang.Integer
- long and java.lang.Long
- short and java.lang.Short
- java.lang.String
- java.io.File
- java.net.URL
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

这里要注意一点，java.util.Date 是不被支持的，而它的子类 java.sql.Date 是被支持的。因此如果对象包含时间类型的属性，且希望被转换的时候，一定要使用 java.sql.Date 类型。否则在转换时会提示 argument mistype 异常。

populate

populate 用于将一个 map 的值填充到一个 bean 中，在 struts 中这个函数被用于从 http request 中取得参数添加到 FormBean

3.1.2. BeanUtilsBean

区别于 BeanUtils 的静态方法方式，使得自定义的配置得以保持。

使用方法：BeanUtilsBean.getInstance()

函数	参数	说明
初始化	getInstance() setInstance(BeansBean newInstance)	
cloneBean	cloneBean(java.lang.Object bean)	拷贝任何一个 Object，使用 PropertyUtils.copyProperties 来做浅拷贝
copyProperties	copyProperties(java.lang.Object dest, java.lang.Object orig)	拷贝所有属性，可自动转换属性
describe	describe(java.lang.Object bean)	这个方法返回一个 Object 中所有的可读属性，并将属性名/属性值放入一个 Map 中，另外还有一个名为 class 的属性，属性值是 Object 的类名，所有 Map 中的 key/value 都是 String，而不管 object 中实际的值是多少
populate	populate(java.lang.Object bean, java.util.Map properties)	describe()的反操作
copyProperty	copyProperty(java.lang.Object bean,	拷贝某个属性

	java.lang.String name, java.lang.Object value)	
<u>setProperty</u>	setProperty(java.lang.Object bean, java.lang.String name, java.lang.Object value)	设置某个属性
<u>getProperty</u>	getProperty(java.lang.Object bean, java.lang.String name)	取得某个属性，转换成 String 返回
<u>getNestedProperty</u>	getNestedProperty(java.lang.Object bean, java.lang.String name)	同上
<u>getSimpleProperty</u>	getSimpleProperty(java.lang.Object bean, java.lang.String name)	得到 simple 类型属性
<u>getIndexedProperty</u>	getIndexedProperty(java.lang.Object bean, java.lang.String name, int index)	得到 index 类型属性
<u>getMappedProperty</u>	getMappedProperty(java.lang.Object bean, java.lang.String name, java.lang.String key)	得到 map 类型属性
<u>getArrayProperty</u>	getArrayProperty(java.lang.Object bean, java.lang.String name)	得到数组类型属性，返回 String[]

3.1.3. PropertyUtils

提供利用 Java 反射 API 调用具体对象的 getter 和 setter 的工具/静态方法, 和 BeanUtils 类似, 但是不做属性转换

这个类和 BeanUtils 类很多的方法在参数上都是相同的, 但返回值不同。BeanUtils 着重于 "Bean", 返回值通常是 String, 而 PropertyUtils 着重于属性, 它的返回值通常是 Object。

3.1.4. PropertyUtilsBean

PropertyUtils 类的实例化实现, 区别于 PropertyUtils 的静态方法方式, 使得自定义的配置得以保持, 和 BeanUtils 类似, 但是不做属性转换

3.1.5. ConvertUtils

提供工具/静态方法, 用于将 String 对象及其数组转换为指定的类型的对象及其数组。可以用如下代码注册自己的实现:

```
Converter myConverter =
    new org.apache.commons.beanutils.converter.IntegerConverter();
ConvertUtils.register(myConverter, Integer.TYPE);    // Native type
ConvertUtils.register(myConverter, Integer.class);   // Wrapper class
```

支持的类型如下:

- java.lang.BigDecimal
- java.lang.BigInteger

- boolean and java.lang.Boolean
- byte and java.lang.Byte
- char and java.lang.Character
- java.lang.Class
- double and java.lang.Double
- float and java.lang.Float
- int and java.lang.Integer
- long and java.lang.Long
- short and java.lang.Short
- java.lang.String
- java.io.File
- java.net.URL
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

3.1.6. ConvertUtilsBean

ConvertUtils 类的实例化实现，区别于 ConvertUtils 的静态方法方式，使得自定义的配置得以保持

ConvertUtilsBean 通过一个 HashMap 管理所有的 XXXConverter。这个 HashMap 的 key 为 XXX 的类全名，值为相应的 XXXConverter 对象。通过 deregister() 方法，初始化这个 HashMap。这个初始化方法会为每一个 XXXConverter 类提供一个缺省的值。用户可以动过 setDefaultXXX(...) 方法来自行设置 XXXConverter 对象的缺省值。这个类还提供了 convert(...) 方法，对 String value 进行相应的转化。

函数	参数	说明
初始化	getInstance()	
convert	convert(java.lang.Object value) convert(java.lang.String[] values, java.lang.Class clazz) convert(java.lang.String value, java.lang.Class clazz)	相互转换
lookup	lookup(java.lang.Class clazz)	根据 class 查到注册的 converter
register	register(java.lang.Class clazz, Converter converter) register(Converter converter, java.lang.Class clazz)	注册 converter
deregister	deregister() deregister(java.lang.Class clazz)	重注册 converter

3.1.7. ConstructorUtils

调用对象中的构造方法，这个类中的方法主要分成两种，一种是得到构造方法，一种是创建对象。事实上多数时候得到构造方法的目的就是创建对象

代码：

```
Month month=new Month(1, "Jan");
```

可以写成：

```
Object obj=ConstructorUtils.invokeConstructor(Month.class, {new Integer(1), "Jan"});
Month month=(Month)obj;
```

如果需要强制指定构造方法的参数类型，可以这样调用：

```
Object[] args={new Integer(1), "Jan"};
Class[] argsType={int.class, String.class};
Object obj;
obj = ConstructorUtils.invokeExactConstructor(Month.class, args, argsType);
Month month=(Month)obj;
```

`invokeExactConstructor`，该方法对参数要求更加严格，传递进去的参数必须严格符合构造方法的参数列表。

函数	参数	说明
<u><code>getAccessibleConstructor</code></u>	<code>getAccessibleConstructor(java.lang.Class klass, java.lang.Class parameterType)</code> <code>getAccessibleConstructor(java.lang.Class klass, java.lang.Class[] parameterTypes)</code> <code>getAccessibleConstructor(java.lang.reflect.Constructor ctor)</code>	得到可以访问的 Constructor 对象
<u><code>invokeConstructor</code></u>	<code>invokeConstructor(java.lang.Class klass, java.lang.Object arg)</code> <code>invokeConstructor(java.lang.Class klass, java.lang.Object[] args)</code> <code>invokeConstructor(java.lang.Class klass, java.lang.Object[] args, java.lang.Class[] parameterTypes)</code>	调用构造方法
<u><code>invokeExactConstructor</code></u>	<code>invokeExactConstructor(java.lang.Class klass, java.lang.Object arg)</code> <code>invokeExactConstructor(java.lang.Class klass, java.lang.Object[] args)</code> <code>invokeExactConstructor(java.lang.Class klass, java.lang.Object[] args, java.lang.Class[] parameterTypes)</code>	调用构造方法

3.1.8. MethodUtils

调用对象的方法。与 `ConstructorUtils` 类似，不过调用的时候，通常需要再指定一个 `methodName` 的参数。

函数	参数	说明
<u><code>getAccessibleMethod</code></u>	<code>getAccessibleMethod(java.lang.Class clazz, java.lang.String methodName, java.lang.Class parameterType)</code> <code>getAccessibleMethod(java.lang.Class clazz, java.lang.String methodName, java.lang.Class[] parameterTypes)</code> <code>getAccessibleMethod(java.lang.reflect.Method method)</code>	得到可以访问的 Method 对象
<u><code>getMatchingAccessibleMethod</code></u>	<code>getMatchingAccessibleMethod(java.lang.Class clazz, java.lang.String methodName,</code>	得到匹配的方法对象，根据方法名

	java.lang.Class[] parameterTypes)	称和参数来匹配
<u>getPrimitiveType</u>	getPrimitiveType(java.lang.Class wrapperType)	送入 Integer.class, 返回 int.class 等等
<u>getPrimitiveWrapper</u>	getPrimitiveWrapper(java.lang.Class primitiveType)	上面的反操作
<u>invokeExactMethod</u>	invokeExactMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object arg) invokeExactMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object[] args) invokeExactMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object[] args, java.lang.Class[] parameterTypes)	准确调用方法
<u>invokeMethod</u>	invokeMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object arg) invokeMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object[] args) invokeMethod(java.lang.Object object, java.lang.String methodName, java.lang.Object[] args, java.lang.Class[] parameterTypes)	调用方法
<u>isAssignmentCompatible</u>	isAssignmentCompatible(java.lang.Class parameterType, java.lang.Class parameterization)	检查参数的类型是否匹配

3.1.9. DynaBean/DynaClass/DynaProperty

DynaBean 并不是 Java 中所定义的 Bean，而是一种“假”的 Bean。因为它并不是通过 getXXX 和 setXXX 方法，对 XXX 属性进行取值和设值的。它通过一个实现了 DynaClass 接口的类，帮助管理其所有的属性的类别，而自己则管理对 XXX 属性值的设定和获取。在设值的时候会通过与 name 对应的 DynaProperty 对象，检查赋值的类别是否正确。

DynaProperty 类描述的是 DynaBean 中所包含的属性的类型。DynaProperty 类有三个属性：属性的名称：name，属性的类型：type，属性内容类型：contentType，如果 DynaProperty 描述的是个容器对象(List 或者 Map)，那么这个 contentType 就代表这个容器内元素的类别。这个类值得关注的地方是 writeObject 和 readObject 方法的实现。它会首先判断自己的 type 是否是一个 primitive 的类，如果是，则先写入 true 标志，再写入对应的 primitive 类的编号；否则写入 false 标志，再写入 type。因为在调用 readObject 方法时，如果得出的是 primitive 类型，则 type 的值为 XXX.TYPE 而不是 XXX.class。

DynaClass 是一个接口，用来管理 DynaBean 中所有的 DynaProperty 属性。

3.1.10. BasicDyanBean/BasicDynaClass

BasicDyanBean 实现自 DynaBean 接口。它包含一个实现了 DynaClass 接口的类的对象，和一个用来存放值的 HashMap。这个 HashMap 的 key 与 DynaClass 中 HashMap 的 key 是一一对应的。

BasicDynaClass 实现了 DynaClass 接口，以 DynaProperty 的 name 为 key 保存所有这些

DynaProperty 对象。它通过 newInstance 方法动态生成实现了 DynaBean 接口的类的对象；注意这个类是可以动态指定的，如果没有，那么就是默认的 BasicDynaBean 类。动态指定类是通过反射实现的，程序如下：

```
//dynaBeanClass 为任意的实现了 DynaBean 接口的类，constructorTypes 为这个
//类的构造方法所需要的参数的类型
constructor = dynaBeanClass.getConstructor(constructorTypes);
//constructorValues 为构造方法的参数值，实际上它的值为当前的 BasicDynaClass
return ((DynaBean) constructor.newInstance(constructorValues));
```

3.1.11. WrapDynaBean/WrapDynaClass/ConvertingWrapDynaBean

WrapDynaBean: DynaBean 的一种实现，包含一个标准的 JavaBean 实例，以便我们可以使用 DynaBean 的 API 去访问它的属性，区别于 ConvertingWrapDynaBean，它不做专门的类型转换

WrapDynaClass: DynaClass 的一种实现，针对那些包装标准 JavaBean 实例的 DynaBeans

ConvertingWrapDynaBean: 包含了标准 JavaBean 实例的 DynaBean 实现，使得我们可以使用 DynaBean 的 API 来访问起属性，同时提供设定属性时的类型转换，继承自并区别于 WrapDynaBean

使用如下代码来包装一个普通 Bean:

```
Object aJavaBean = ...;
DynaBean db = new WrapDynaBean(aJavaBean);
```

3.1.12. JDBCClass/ResultSet/ResultSetIterator/RowSet

JDBCClass: 为 DynaClass 的 JDBC 实现提供公用的逻辑

ResultSetDynaClass: 包装 java.sql.ResultSet 中的 java.sql.Row 实例的 DynaBean 所对应的 DynaClass 实现

ResultSetIterator: 针对 ResultSetDynaClass 的 java.util.Iterator 实现

RowSetDynaClass: DynaClass 的一种实现，用于在内存中创建一组表示 SQL 查询结果的 DynaBeans，区别于 ResultSetDynaClass，它不需要保持 ResultSet 打开

```
ResultSet rs = ...;
ResultSetDynaClass rsdc = new ResultSetDynaClass(rs);
Iterator rows = rsdc.iterator();
while (rows.hasNext()) {
    DynaBean row = (DynaBean) rows.next();
    ... process this row ...
}
```



```
rs.close();
```

ResultSetDynaClass 原来是一个 ResultSet 的包装器，ResultSetDynaClass 实现了 DynaClass，它的 iterator 方法返回一个 ResultSetIterator，则是实现了 DynaBean 接口。在获得一个 DynaBean 之后，我们就可以用：

```
DynaBean row = (DynaBean) rows.next();  
//field1 是其中一个字段的名称  
System.out.println(row.get("field1"));
```

再看另一个类 RowSetDynaClass 的用法，代码如下：

```
String driver="com.mysql.jdbc.Driver";  
String  
url="jdbc:mysql://localhost/2hu?useUnicode=true&characterEncoding=GBK";  
String username="root";  
String password="";  
  
java.sql.Connection con=null;  
PreparedStatement ps=null;  
ResultSet rs=null;  
try {  
Class.forName(driver).newInstance();  
con = DriverManager.getConnection(url);  
ps=con.prepareStatement("select * from forumlist");  
rs=ps.executeQuery();  
//先打印一下，用于检验后面的结果。  
while(rs.next()){  
System.out.println(rs.getString("name"));  
}  
//这里必须用 beforeFirst，因为 RowSetDynaClass 只从当前位置向前滚动  
rs.beforeFirst();  
RowSetDynaClass rsdc = new RowSetDynaClass(rs);  
rs.close();  
ps.close();  
List rows = rsdc.getRows();//返回一个标准的 List，存放的是 DynaBean  
for (int i = 0; i <rows.size(); i++) {  
DynaBean b=(DynaBean) rows.get(i);  
System.out.println(b.get("name"));  
}} catch (Exception e) {  
e.printStackTrace();  
}finally{  
try {con.close();} catch (Exception e) {}  
}
```

是不是很有趣？封装了 ResultSet 的数据，代价是占用内存。如果一个表有 10 万条记录，rsdc.getRows()就会返回 10 万个记录。

需要注意的是 ResultSetDynaClass 和 RowSetDynaClass 的不同之处：

1. ResultSetDynaClass 是基于 Iterator 的，一次只返回一条记录，而 RowSetDynaClass 是基

于 List 的，一次性返回全部记录。直接影响是在数据比较多时 `ResultSetDynaClass` 会比较的快速，而 `RowSetDynaClass` 需要将 `ResultSet` 中的全部数据都读出来（并存储在其内部），会占用过多的内存，并且速度也会比较慢。

2. `ResultSetDynaClass` 一次只处理一条记录，在处理完成之前，`ResultSet` 不可以关闭。
3. `ResultSetIterator` 的 `next()` 方法返回的 `DynaBean` 其实是指向其内部的一个固定对象，在每次 `next()` 之后，内部的值都会被改变。这样做的目的是节约内存，如果你需要保存每次生成的 `DynaBean`，就需要创建另一个 `DynaBean`，并将数据复制过去，下面也是 java doc 中的代码：

```
ArrayList results = new ArrayList(); // To hold copied list
ResultSetDynaClass rsdc = ...;
DynaProperty properties[] = rsdc.getDynaProperties();
BasicDynaClass bdc =
    new BasicDynaClass("foo", BasicDynaBean.class,
        rsdc.getDynaProperties());
Iterator rows = rsdc.iterator();
while (rows.hasNext()) {
    DynaBean oldRow = (DynaBean) rows.next();
    DynaBean newRow = bdc.newInstance();
    PropertyUtils.copyProperties(newRow, oldRow);
    results.add(newRow);
}
```

3.1.13. LazyDynaBean/LazyDynaClass/LazyDynaMap

LazyDynaBean: 懒载入 `DynaBean`，自动往 `DynaClass` 添加属性并提供懒载入 List 和懒载入 Map 的功能

LazyDynaClass: 实现 `MutableDynaClass` 接口的类

LazyDynaMap: 为 Map 实例提供一个轻量级的 `DynaBean` 包装

`LazyDynaBean` 实现一个动态的 `Bean`，可以直接往里面加入属性，作为一个 `JavaBean` 一样使用，也可以用上面的 `BeanUtils` 或 `get/set` 方法进行操作，而不用事先定义一个标准的 `JavaBean` 类。

记得在 J2ee 设计模式中有一种 `Value Object` 的模式，用于在 MVC 各层之间传递数据，避免直接传递大业务对象引起的性能问题，为了避免在项目中出现很多 `Bean` 类，在书中提供了一个动态 `Value Object` 的实现（通过扩展 Map）。这里 `LazyDynaBean` 则可以作为一种更加成熟、稳定的实现来使用。

```
//这里使用 LazyDynaMap，它是 LazyBean 的一个轻量级实现
LazyDynaMap dynaBean1 = new LazyDynaMap();
dynaBean1.set("foo", "bar"); // simple
dynaBean1.set("customer", "title", "Mr"); // mapped
dynaBean1.set("address", 0, "address1"); // indexed
System.out.println(dynaBean1.get("address", 0));
Map myMap = dynaBean1.getMap(); // retrieve the Map
System.out.println(myMap.toString());
```

上面的例子可以看到，它可以在 `set` 时自动增加 `bean` 的 `property`（既赋值的同时增加 `Bean` 中的 `property`），同时也支持 3 中类型的 `property`，并且 `LazyDynaMap` 还可以导出为 `map`。

对于这个类还有两个重要的 Field 要注意：

returnnull——指定在 **get** 方法使用了一个没有定义过的 **property** 时，**DynaBean** 的行为。

```
//取的字段的值
//设为 true。若 Bean 中没有此字段，返回 null
//默认为 false。若 Bean 中没有此字段，自动增加一个：
dynaBean1.setReturnNull(true);
System.out.println(dynaBean1.get("aaa")); //此时返回 null
```

Restricted——指定是否允许改变这个 **bean** 的 **property**。设为 **true** 后，字段不可再增删和修改

```
//默认为 false，允许增删和修改
dynaBean1.setRestricted(true);
dynaBean1.set("test", "error"); //这里会出错！
```

通过设置这两个属性，可以防止意外修改 **DynaBean** 的 **property**。在设计架构时，你可以在后台从数据表或 **xml** 文件自动产生 **DynaBean**，在传到控制层和表示层之前设置上述属性使其 **Bean** 结构不允许修改，如此就不可能无意中修改 **Bean** 包含的属性……这样既可以享用它的便利，又可以防止由此引入的错误可能，设计者实在深得偷懒的精髓啊！

4. Commons Collections

Java 1.4 Collections API 大致可以发现三种主要的类别：

1. 容器类：如 **Collection**、**List**、**Map** 等，用于存放对象和进行简单操作的；
2. 操作类：如 **Collections**、**Arrays** 等，用于对容器类的实例进行相对复杂操作如排序等；
3. 辅助类：如 **Iterator**、**Comparator** 等，用于辅助操作类以及外部调用代码实现对容器类的操作，所谓辅助，概括而通俗的来讲，就是这些类提供一种算法，你给它一个对象或者一组对象，或者仅仅是按一定的规则调用它，它给你一个运算后的答案，帮助你正确处理容器对象。比如 **Iterator** 会告诉你容器中下一个对象有没有、是什么，而 **Comparator** 将对对象大小/先后次序的算法逻辑独立出来。

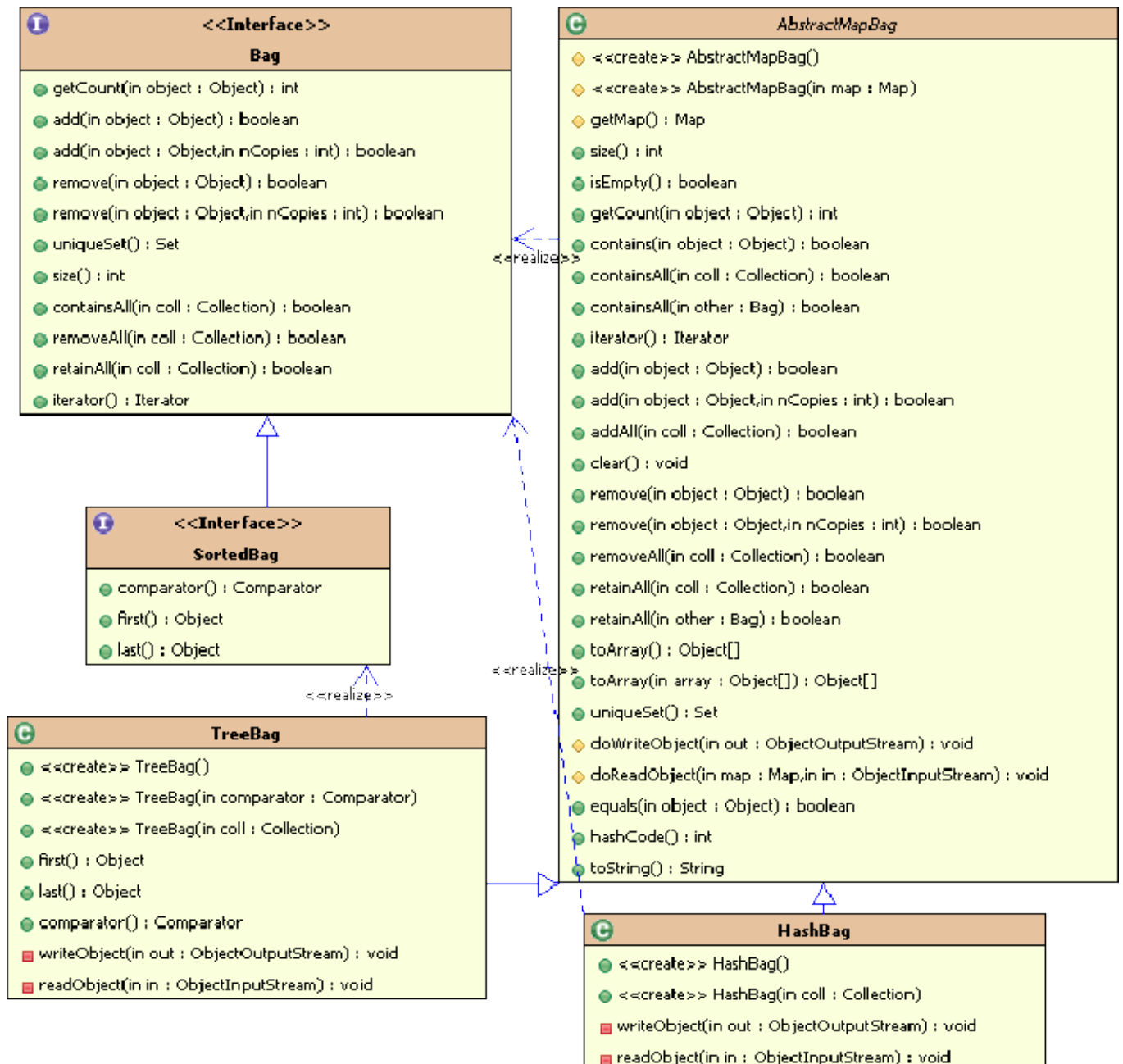
接口 Interfaces	实现 Implementations				
	Hash Table	Array	Balanced Tree	Linked List	Hash +Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

同样，**Jakarta Commons Collections** 我们细细看来，也能够找出类似的划分：

1. 作为容器类的补充，我们可以找到 **Bag**、**Buffer**、**BidiMap**、**OrderedMap** 等等；
2. 作为操作类的补充，我们可以找到 **CollectionUtils**、**IteratorUtils**、**ListUtils**、**SetUtils** 等等；
3. 作为辅助类的补充，我们可以找到 **MapIterator**、**Closure**、**Predicate**、**Transformer** 等等；

4.1.commons.collections.bag

我们有时候需要在 Collection 中存放多个相同对象的拷贝，并且需要很方便的取得该对象拷贝的个数，Bag 提供这样的功能。例如你有一个 Bag 包含了 {a, a, b, c}，你就可以使用 getCount(a)，取得数量值为 2，你也可以使用 uniqueSet() 将会回传 {a, b, c} 这个 Set。需要注意的一点是它虽然 extends Collection，但是如果真把它完全当作 java.util.Collection 来用会遇到语义上的问题。



4.2. commons.collections.buffer

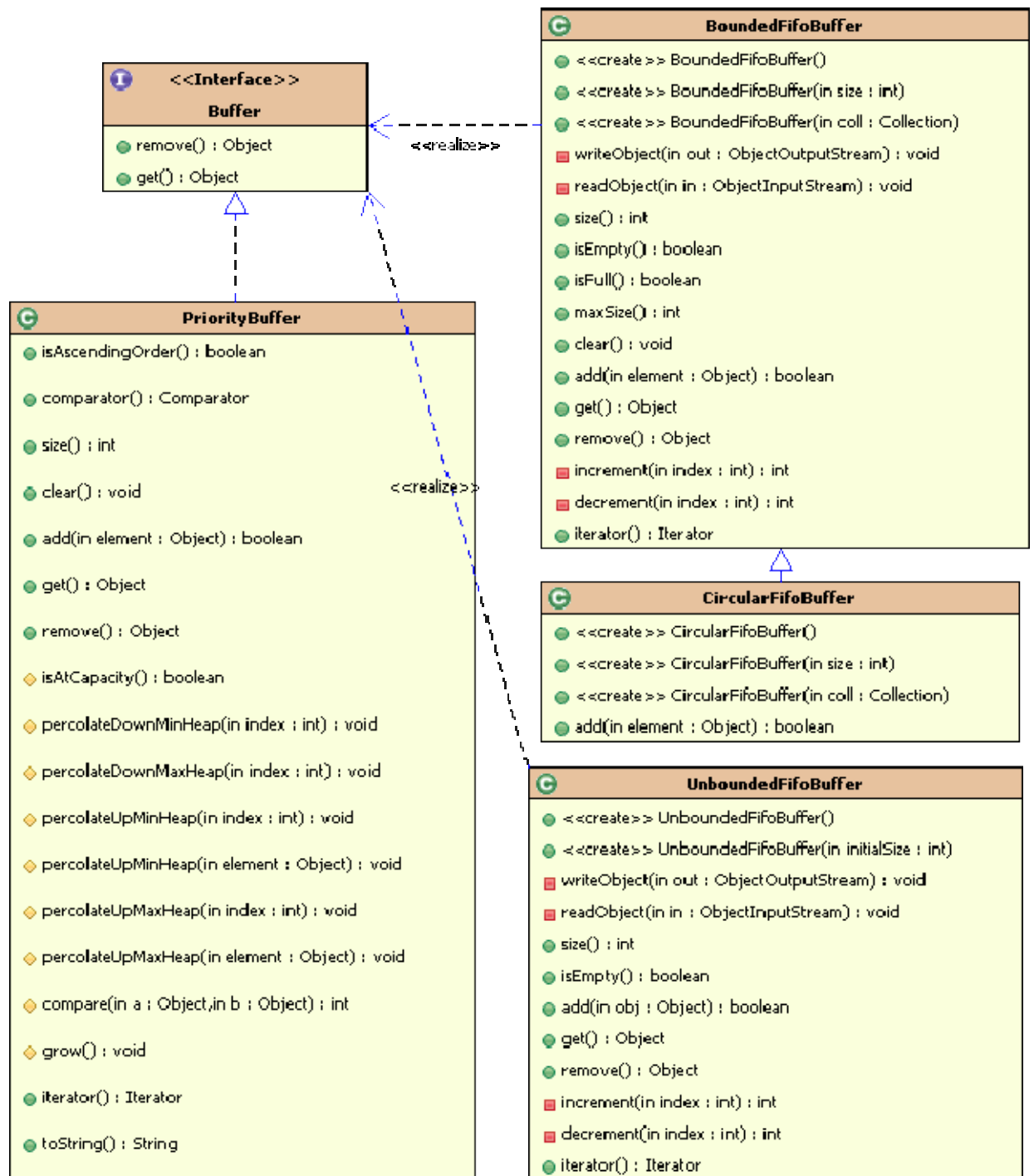
实现按一定规则添加删除的序列，如 FIFO。

UnboundedFifoBuffer，提供先进先出的大小可变的队列。

BoundedFifoBuffer 则是对其大小进行了限制，是固定大小的先进先出队列。

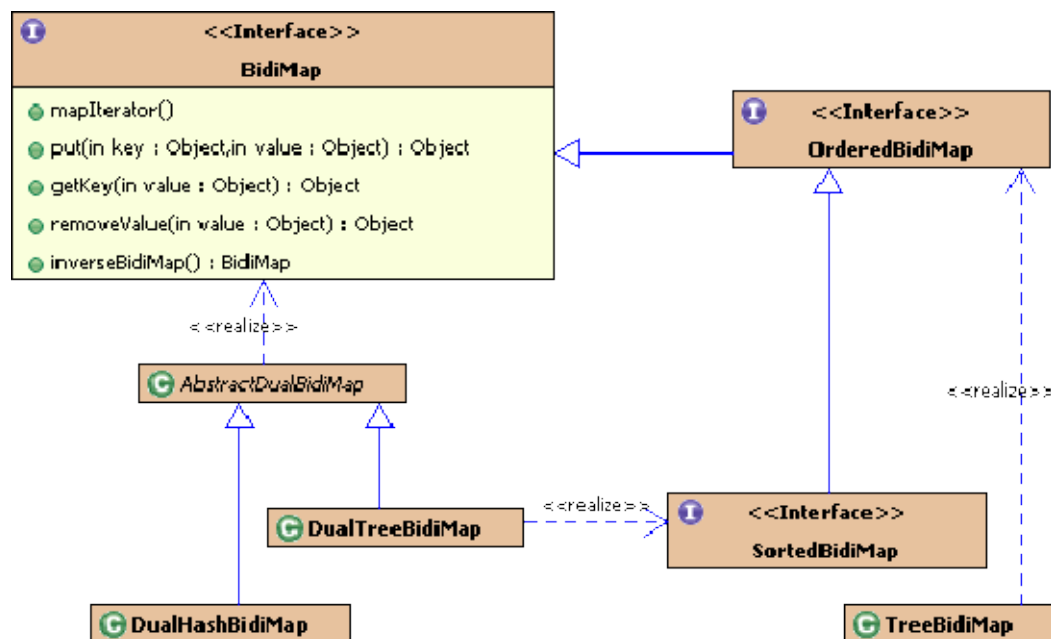
BlockingBuffer 要在多线程的环境中才能体现出它的价值，尤其是当我们需要实现某种流水线时这个 **BlockingBuffer** 很有用：每个流水线上的组件从上游的 **BlockingBuffer** 获取数据，处理后放到下一个 **BlockingBuffer** 中依次传递。**BlockingBuffer** 的核心特色通俗点说就是如果你向它要东西，而它暂时还没有的话，你可以一直等待直至拿到为止。

PriorityBuffer 则提供比一般的先进先出 **Buffer** 更强的控制力：我们可以自定义 **Comparator** 给它，告诉它怎么判定它的成员的先后顺序，优先级最高的最先走。



4.3. commons.collections.bidimap

所谓 BidiMap，直译就是双向 Map，可以通过 key 找到 value，也可以通过 value 找到 key，这在我们日常的代码-名称匹配的时候很方便：因为我们除了需要通过代码找到名称之外，往往也需要处理用户输入的名称，然后获取其代码。需要注意的是 BidiMap 当中不光 key 不能重复，value 也不可以。



4.4. Comparator 组

其实 Comparator 这个概念并不是 Commons Collections 引入的, 在标准的 Java Collections API 中, 已经确定了一个 `java.util.Comparator` 接口, 只是有很多人并不了解, Commons Collections 也只是扩展了这个接口而已。这个 `java.util.Comparator` 定义如下核心方法:

```
public int compare(Object arg0, Object arg1)
```

传给它两个对象, 它要告诉我们这两个对象哪一个在特定的语义下更“大”, 或者两者相等。如果 $arg0 > arg1$, 返回大于 0 的整数; 如果 $arg0 = arg1$, 返回 0; 如果 $arg0 < arg2$, 返回小于 0 的整数。

我们看看 Commons Collections 给我们提供了哪些 Comparator 的实现类 (都在 `org.apache.commons.collections.comparators` 包下面):

BooleanComparator - 用于排序一组 Boolean 对象, 指明先 true 还是先 false;

ComparableComparator - 用于排序实现了 `java.lang.Comparable` 接口的对象 (我们常用的 Java 类如 String、Integer、Date、Double、File、Character 等等都实现了 Comparable 接口);

ComparatorChain - 定义一组 Comparator 链, 链中的 Comparator 对象会被依次执行;

FixedOrderComparator - 用于定义一个特殊的顺序, 对一组对象按照这样的自定义顺序进行排序;

NullComparator - 让 null 值也可参与比较, 可以设定为先 null 或者后 null;

ReverseComparator - 将原有的 Comparator 效果反转;

TransformingComparator - 将一个 Comparator 装饰为具有 Transformer 效果的 Comparator。

s

以上除了 ComparatorChain 之外, 似乎都是实现一些很基本的比较方法, 但是当我们用 ComparatorChain 将一组 Comparator 串起来之后, 就可以实现非常灵活的比较操作。

4.5. 算子组

算子成为 Commons Collections 3.1 中的有趣的部分有两个原因：它们没有得到应得的重视并且它们有改变你编程的方式的潜力。算子只是一个奇特的名字，它代表了一个包装了函数的对象——一个“函数对象”。当然，它们不是一回事。如果你曾经使用过 C 和 C++ 的方法指针，你就会理解算子的威力。

一个算子是一个对象——一个 Predicate, 一个 Closure, 一个 Transformer。

Predicates 求对象的值并返回一个 **boolean**, **Transformer** 求对象的值并返回新对象, **Closure** 接受对象并执行代码。算子可以被组合成组合算子来模仿循环, 逻辑表达式, 和控制结构, 并且算子也可以被用来过滤和操作集合中的元素。在这么短的篇幅中解释清楚算子是不可能的, 所以跳过介绍, 我将会通过使用和不使用算子来解决同一问题 (解释算子)。在这个例子中, 从一个 **ArrayList** 中而来的 **Student** 对象会被排序到两个 **List** 中, 如果他们符合某种标准的话。

成绩为 A 的学生会被加到 **honorRollStudents**(光荣榜)中, 得 D 和 F 的学生被加到 **problemStudents** (问题学生)list 中。学生分开以后, 系统将会遍历每个 list, 给加入到光荣榜中学生一个奖励, 并安排与问题学生的家长谈话的时间表。下面的代码不使用算子实现了这个过程:

```
List allStudents = getAllStudents();
// 创建两个 ArrayList 来存放荣誉学生和问题学生
List honorRollStudents = new ArrayList();
List problemStudents = new ArrayList();
// 遍历所有学生, 将荣誉学生放入一个 List, 问题学生放入另一个
Iterator allStudentsIter = allStudents.iterator();
while (allStudentsIter.hasNext()) {
    Student s = (Student) allStudentsIter.next();
    if (s.getGrade().equals("A")) {
        honorRollStudents.add(s);
    } else if (s.getGrade().equals("B") && s.getAttendance() == PERFECT) {
        honorRollStudents.add(s);
    } else if (s.getGrade().equals("D") || s.getGrade().equals("F")) {
        problemStudents.add(s);
    } else if (s.getStatus() == SUSPENDED) {
        problemStudents.add(s);
    }
}
// 对于的有荣誉学生, 增加一个奖励并存储到数据库中
Iterator honorRollIter = honorRollStudents.iterator();
while (honorRollIter.hasNext()) {
    Student s = (Student) honorRollIter.next();
    // 给学生记录增加一个奖励
    s.addAward("honor roll", 2005);
    Database.saveStudent(s);
}
```



```

// 对所有问题学生，增加一个注释并存储到数据库中
Iterator problemIter = problemStudents.iterator();
while (problemIter.hasNext()) {
    Student s = (Student) problemIter.next();
    // 将学生标记为需特别注意
    s.addNote("talk to student", 2005);
    s.addNote("meeting with parents", 2005);
    Database.saveStudent(s);
}

```

上述例子是非常过程化的;要想知道 **Student** 对象发生了什么事必须遍历每一行代码。例子的第一部分是基于成绩和考勤对 **Student** 对象进行逻辑判断。

第二部分对 **Student** 对象进行操作并存储到数据库中。像上述这个有着 50 行代码程序也是大多程序所开始的一可管理的过程化的复杂性。但是当需求变化时，问题出现了。一旦判断逻辑改变，你就需要在第一部分中增加更多的逻辑表达式。

举例来说，如果一个有着成绩 **B** 和良好出勤记录，但有五次以上的留堂记录的学生被判定为问题学生，那么你的逻辑表达式将会如何处理?或者对于第二部分中，只有在上一年度不是问题学生的学生才能进入光荣榜的话，如何处理?当例外和需求开始改变进而影响到过程代码时，可管理的复杂性就会变成不可维护的面条式的代码。

从上面的例子中回来，考虑一下那段代码到底在做什么。它在一个 **List** 遍历每一个对象，检查标准，如果适用该标准，对此对象进行某些操作。上述例子可以进行改进的关键一处在从代码中将标准与动作解藕开来。下面的两处代码引用以一种非常不同的方法解决了上述的问题。首先，荣誉榜和问题学生的标准被两个 **Predicate** 对象模型化了，并且加之于荣誉学生和问题学生上的动作也被两个 **Closure** 对象模型化了。这四个对象如下定义：

```

// 匿名的 Predicate 决定一个学生是否加入荣誉榜
Predicate isHonorRoll = new Predicate() {
    public boolean evaluate(Object object) {
        Student s = (Student) object;
        return ((s.getGrade().equals("A")) ||
(s.getGrade().equals("B") && s.getAttendance() == PERFECT));
    }
};

// 匿名的 Predicate 决定一个学生是否是问题学生
Predicate isProblem = new Predicate() {
    public boolean evaluate(Object object) {
        Student s = (Student) object;
        return ((s.getGrade().equals("D") ||
s.getGrade().equals("F")) || s.getStatus() == SUSPENDED);
    }
};

// 匿名的 Closure 将一个学生加入荣誉榜
Closure addToHonorRoll = new Closure() {
    public void execute(Object object) {
        Student s = (Student) object;
    }
};

```

```

        // 对学生增加一个荣誉记录
        s.addAward("honor roll", 2005);
        Database.saveStudent(s);
    }
};

// 匿名的 Closure 将学生标记为需特别注意
Closure flagForAttention = new Closure() {
    public void execute(Object object) {
        Student s = (Student) object;
        // 标记学生为需特别注意
        s.addNote("talk to student", 2005);
        s.addNote("meeting with parents", 2005);
        Database.saveStudent(s);
    }
};

```

这四个匿名的 **Predicate** 和 **Closure** 是从作为一个整体互相分离的。**flagForAttention**(标记为注意)并不知道什么是确定一个问题学生的标准。现在需要的是将正确的 **Predicate** 和正确的 **Closure** 结合起来的方法，这将在下面的例子中展示：

```

Map predicateMap = new HashMap();
predicateMap.put( isHonorRoll, addToHonorRoll );
predicateMap.put( isProblem, flagForAttention );
predicateMap.put( null, ClosureUtils.nopClosure() );
Closure processStudents = ClosureUtils.switchClosure( predicateMap );
CollectionUtils.forAllDo( allStudents, processStudents );

```

在上面的代码中，**predicateMap** 将 **Predicate** 与 **Closure** 进行了配对；如果一个学生满足作为键值的 **Predicate** 的条件，那么它将把它的值传到作为 **Map** 的值的 **Closure** 中。通过提供一个 **NOPClosure** 值和 **null** 键对，我们将把不符合任何 **Predicate** 条件的 **Student** 对象传给由 **ClosureUtils** 调用创建的“不做任何事”或者“无操作”的 **NOPClosure**。

一个 **SwitchClosure**，**processStudents**，从 **predicateMap** 中创建。并且通过使用 **CollectionUtils.forAllDo()** 方法，将 **processStudents Closure** 应用到 **allStudents** 中的每一个 **Student** 对象上。这是非常不一样的处理方法；记住，你并没有遍历任何队列。而是通过设置规则和因果关系，以及 **CollectionUtils** 和 **SwitchClosure** 来完成了这些操作。

当你将使用 **Predicate** 的标准与使用 **Closure** 的动作将分离开来时，你的代码的过程式处理就少了，而且更容易测试了。**isHonorRoll Predicate** 能够与 **addToHonorRoll Closure** 分离开来来进行独立的单元测试，它们也可以合起来通过使用 **Student** 类的模仿对象进行测试。第二个例子也会演示 **CollectionUtils.forAllDo()**，它将一个 **Closure** 应用到了一个 **Collection** 的每一个元素中。

你也许注意到了使用算子并没减少代码行数，实际上，使用算子还增加了代码量。但是，通过算子，你得到了将到了标准与动作的模块性与封装性的好处。如果你的代码题已经接近于几百行，那么请考虑一下更少过程化处理，更多面向对象的解决方案——通过使用算子。

所有的算子-- **Closure**，**Predicate**，和 **Transformer**——能够被合并为合并算子来处理任何种类的逻辑问题。**switch**，**while** 和 **for** 结构能够被 **SwitchClosure**，**WhileClosure**，和 **ForClosure** 模型化。

复合的逻辑表达式可以被多个 **Predicate** 构建，通过使用 **OrPredicate**，**AndPredicate**，

AllPredicate, 和 NonePredicate 将它们相互联接。Commons BeanUtils 也包含了算子的实现被用来将算子应用到 bean 的属性中 -- BeanPredicate, BeanComparator, 和 BeanPropertyValueChangeClosure。算子是考虑底层的应用架构的不一样的方法, 它们可以很好地改造你编码实现的方法。

4.5.1. Predicate

Predicate 是 Commons Collections 中定义的一个接口, 可以在 org.apache.commons.collections 包中找到。其中定义的方法签名如下:

```
public boolean evaluate(Object object)
```

它以一个 Object 对象为参数, 处理后返回一个 boolean 值, 检验某个对象是否满足某个条件。其实这个 Predicate 和 Comparator 还有我们即将看到的 Transformer 和 Closure 等都有些类似 C/C++ 中的函数指针, 它们都只是提供简单而明确定义的函数功能而已。

跟其他组类似, Commons Collections 也提供了一组定义好的 Predicate 类供我们使用, 这些类都放在 org.apache.commons.collections.functors 包中。当然, 我们也可以自定义 Predicate, 只要实现这个 Predicate 接口即可。在 Commons Collections 中我们也可以很方便使用的一组预定义复合 Predicate, 我们提供 2 个或不定数量个 Predicate, 然后交给它, 它可以帮我们处理额外的逻辑, 如 AndPredicate 处理两个 Predicate, 只有当两者都返回 true 它才返回 true; AnyPredicate 处理多个 Predicate, 当其中一个满足就返回 true, 等等。

Class	初始化方法	说明
AllPredicate	getInstance(Collection predicates) getInstance(Predicate[] predicates)	当所有内部 predicates 成功后, 才返回成功
AnyPredicate	getInstance(Collection predicates) getInstance(Predicate[] predicates)	当任一内部 predicates 成功后, 才返回成功
NonePredicate	getInstance(Collection predicates) getInstance(Predicate[] predicates)	当所有内部 predicates 失败后, 才返回成功
OnePredicate	getInstance(Collection predicates) getInstance(Predicate[] predicates)	只有一个 true, 才返回 true
AndPredicate	getInstance(Predicate predicate1, Predicate predicate2)	两个 predicates 做 and
OrPredicate	getInstance(Predicate predicate1, Predicate predicate2)	两个 predicates 做 or
EqualPredicate	getInstance(Object object)	检查的 object 和 predicates 里面一样时成功
NotPredicate	getInstance(Predicate predicate)	反
IdentityPredicate	getInstance(Object object)	检查引用
InstanceofPredicate	getInstance(Class type)	检查类类型
UniquePredicate	getInstance()	对象第一次检查的时候返回 true
NotNullPredicate	getInstance()	检查非空
NullPredicate	getInstance()	检查空
ExceptionPredicate	getInstance()	执行时抛 Exception
NullIsExceptionPredicate	getInstance(Predicate predicate)	输入是 null 抛 Exception

<u>NullIsFalsePredicate</u>	getInstance(Predicate predicate)	输入是 null 返回 true
<u>NullIsTruePredicate</u>	getInstance(Predicate predicate)	输入是 null 返回 false
<u>TruePredicate</u>	getInstance()	永远返回 true
<u>FalsePredicate</u>	getInstance()	永远返回 false
<u>TransformerPredicate</u>	getInstance(Transformer transformer)	返回 transformer 转换后结果
<u>TransformedPredicate</u>	getInstance(Transformer transformer, Predicate predicate)	交给另外一个 predicate 处理前, 先做 transform

4.5.2. Transformer

我们有时候需要将某个对象转换成另一个对象供另一组方法调用,而这两类对象的类型有可能并不是出于同一个继承体系的,或者说出了很基本的 `Object` 之外没有共同的父类,或者我们根本不关心他们是不是有其他继承关系,甚至就是同一个类的实例只是对我们而言无所谓,我们为了它能够被后续的调用者有意义的识别和处理,在这样的情形,我们就可以利用 `Transformer`。除了基本的转型 `Transformer` 之外, `Commons Collections` 还提供了 `Transformer` 链和带条件的 `Transformer`,使得我们很方便的组装出有意义的转型逻辑。

假定我们在处理员工聘用时,需要将原来的 `Applicant` 对象转换为 `Employee` 对象,而 `Applicant` 类和 `Employee` 类无论继承关系、字段内容、具体业务职能等等都不是同一派系的,只是某些字段是相关的,且要求必要的转换,那么这个时候我们使用 `Transformer` 就可以比较方便的实现这项功能,并且由于它的实现是灵活的、模块化的,使得今后的维护也变得清晰和易于处理。

Class	初始化方法	说明
<u>ChainedTransformer</u>	getInstance(Transformer[] transformers)	按顺序逐个处理
<u>CloneTransformer</u>	getInstance()	拷贝一个输入,使用 <code>PrototypeFactory</code>
<u>ClosureTransformer</u>	getInstance(Closure closure)	调用 closure, 返回 input
<u>ConstantTransformer</u>	getInstance(Object constantToReturn)	不做转换, 返回常数
<u>ExceptionTransformer</u>	getInstance()	执行时抛 Exception
<u>FactoryTransformer</u>	getInstance(Factory factory)	执行 factory 并返回结果
<u>InstantiateTransformer</u>	getInstance(Class[] paramTypes, Object[] args)	通过反射创建 new object
<u>InvokerTransformer</u>	getInstance(String methodName, Class[] paramTypes, Object[] args)	调用值的某个方法
<u>MapTransformer</u>	getInstance(Map map)	根据送入的 input 查 map,返回 map 的 value
<u>NOPTransformer</u>	getInstance()	不操作
<u>PredicateTransformer</u>	getInstance(Predicate predicate)	调用 predicate, 返回 input
<u>StringValueTransformer</u>	getInstance()	调用 <code>String.valueOf</code>
<u>SwitchTransformer</u>	getInstance(Predicate[] predicates, Transformer[] transformers, Transformer defaultTransformer) getInstance(Map predicatesAndTransformers)	当 predicates 为 true 是执行 transformer

4.5.3. Closure

Class	初始化方法	说明
ChainedClosure	getInstance(Closure[] closures)	串行执行所有 closure
ExceptionClosure		执行时抛 Exception
ForClosure	getInstance(int count, Closure closure)	执行 closure 指定次数
IfClosure	getInstance(Predicate predicate, Closure trueClosure, Closure falseClosure)	执行 if else
NOPClosure	getInstance()	不操作
SwitchClosure	getInstance(Predicate[] predicates, Closure[] closures, Closure defaultClosure)	当 predicates 为 true 时执行 closures
TransformerClosure	getInstance(Transformer transformer)	执行 transformer
WhileClosure	getInstance(Predicate predicate, Closure closure, boolean doLoop)	执行 closure 直到 predicate 返回 true

4.5.4. Factory

Class	初始化方法	说明
ConstantFactory	getInstance(Object constantToReturn)	返回一个常数
ExceptionFactory		执行时抛 Exception
InstantiateFactory		通过反射创建 new object
PrototypeFactory	getInstance(Object prototype)	基于原型 clone object

4.6. Collections 组

4.6.1. FastArrayList/FastHashMap/FastTreeMap

FastArrayList, FastHashMap, 和 FastTreeMap。它们分别继承了标准的 ArrayList、HashMap、和 TreeMap，且都提供了在多线程下安全的同步读写访问。然而，安全性是需要成本的，这样便降低了写操作的速度。当读操作是非同步时，写操作在现存结构被替代时是在一份数据备份上进行地。他有几个步骤。

1. Clone 已经存在的 collection
2. 对 clone 执行修改的动作
3. 把已经被修改过的 clone 更换现在存在的 collection

举例来说, struts 就是先把 form-bean 整个 load 到 FastHashMap 中, 来减少每次读取配置文件的 io 时间。

上述三个类在初始化的时候是 slow 模式，需要用 setFast(true)方法来修改执行模式。

4.6.2. ExtendedProperties

对 Properties 的扩展，支持一个属性有多个值，支持分行，支持类型转换，支持转义属性文件范例：

```
# lines starting with # are comments

# This is the simplest property
key = value

# A long property may be separated on multiple lines
longvalue = aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa \
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

















































# This is a property with many tokens
tokens_on_a_line = first token, second token

# This sequence generates exactly the same result
tokens_on_multiple_lines = first token
tokens_on_multiple_lines = second token

# commas may be escaped in tokens
commas.escaped = Hi\, what'up?
```

属性文件规则：

1. 基本语法：key = value
2. 键值可以用除了 '=' 以外所有字符
3. 可以用反斜线来对长字符串做分行
4. 用逗号来区分多个值
5. 内容中的逗号用反斜线 + 逗号来转义
6. 反斜线自己用两个反斜线来转义 '\\'
7. 同样键值出现多次，相当于多值
8. # 是注释符号
9. 键值为 include 代表从包括另外一个配置文件，相同的键值将被覆盖。
10. 可以通过 getFloat 这样的方法读取转换后的属性。

 ExtendedProperties
 <<create>> ExtendedProperties()
 <<create>> ExtendedProperties(in file : String)
 <<create>> ExtendedProperties(in file : String,in defaultFile : String)
 isInitialized() : boolean
 getInclude() : String
 setInclude(in inc : String) : void
 load(in input : InputStream) : void
 load(in input : InputStream,in enc : String) : void
 getProperty(in key : String) : Object
 addProperty(in key : String,in value : Object) : void
 setProperty(in key : String,in value : Object) : void
 save(in output : OutputStream,in header : String) : void
 combine(in props : ExtendedProperties) : void
 clearProperty(in key : String) : void
 getKeys() : Iterator
 getKeys(in prefix : String) : Iterator
 subset(in prefix : String) : ExtendedProperties
 display() : void
 getString(in key : String) : String
 getString(in key : String,in defaultValue : String) : String
 getProperties(in key : String) : Properties
 getProperties(in key : String,in defaults : Properties) : Properties
 getStringArray(in key : String) : String[]
 getVector(in key : String) : Vector
 getVector(in key : String,in defaultValue : Vector) : Vector
 getBoolean(in key : String) : boolean
 getBoolean(in key : String,in defaultValue : boolean) : boolean
 getBoolean(in key : String,in defaultValue : Boolean) : Boolean
 testBoolean(in value : String) : String
 getByte(in key : String) : byte
 getByte(in key : String,in defaultValue : byte) : byte
 getByte(in key : String,in defaultValue : Byte) : Byte
 getShort(in key : String) : short
 getShort(in key : String,in defaultValue : short) : short
 getShort(in key : String,in defaultValue : Short) : Short
 getInt(in name : String) : int
 getInt(in name : String,in def : int) : int
 getInteger(in key : String) : int
 getInteger(in key : String,in defaultValue : int) : int
 getInteger(in key : String,in defaultValue : Integer) : Integer
 getLong(in key : String) : long
 getLong(in key : String,in defaultValue : long) : long
 getLong(in key : String,in defaultValue : Long) : Long
 getFloat(in key : String) : float
 getFloat(in key : String,in defaultValue : float) : float
 getFloat(in key : String,in defaultValue : Float) : Float
 getDouble(in key : String) : double
 getDouble(in key : String,in defaultValue : double) : double
 getDouble(in key : String,in defaultValue : Double) : Double
 convertProperties(in props : Properties) : ExtendedProperties

4.6.3. BoundedCollection/BoundedMap

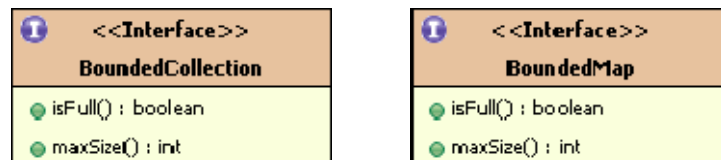
该集合的 size 可以变化，但是不能超过一个最大值。

BoundedCollection 的实现有： [BoundedFifoBuffer](#)

BoundedMap 的实现有： [LRUMap](#)

LRUMap(Least Recently Used)是一个可维护最大容量的 Map，而且使用了至少一个运算法则来定义在这一 Map 已满时要移去的节点。

BoundedFifoBuffer 是一个固定大小的 Buffer，当 buffer 满时，移除最先进入的对象。



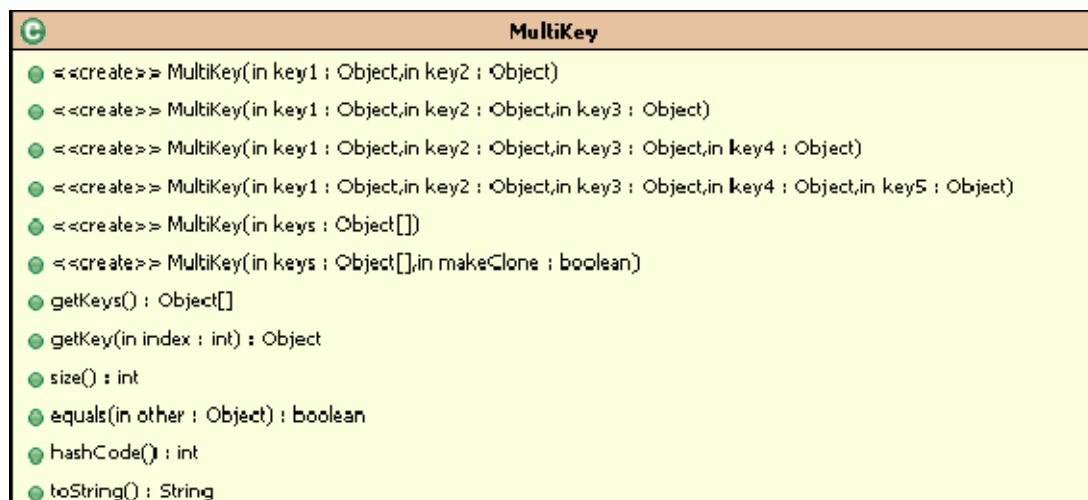
4.6.4. MultiKey

对 Key 进行包装，支持复合 Key(类似表中的(复合主键))。

范例代码：

```
// populate map with data mapping key+locale to localizedText
Map map = new HashMap();
MultiKey multiKey = new MultiKey(key, locale);
map.put(multiKey, localizedText);

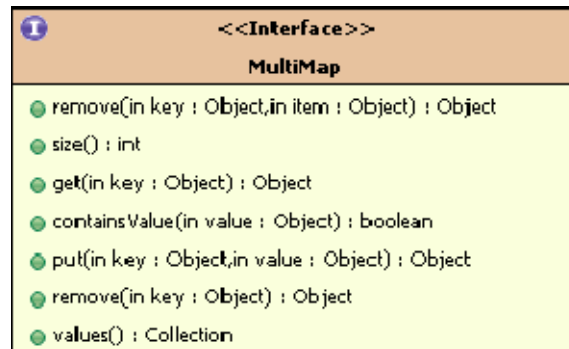
// later retrieve the localized text
MultiKey multiKey = new MultiKey(key, locale);
String localizedText = (String) map.get(multiKey);
```



4.7. Map 组

4.7.1. MultiMap

所谓 MultiMap,就是说一个 key 不在是简单的指向一个对象,而是一组对象,add()和 remove()的时候跟普通的 Map 无异,只是在 get()时返回一个 Collection,利用 MultiMap,我们就可以很方便的往一个 key 上放数量不定的对象,也就实现了一对多。



MultiMap 的实现有: commons.collections.MultiHashMap, 代码:

```
MultiMap mhm = new MultiHashMap();
mhm.put(key, "A");
mhm.put(key, "B");
mhm.put(key, "C");
List list = (List) mhm.get(key);
```

返回的 list 将包含内容"A", "B", "C"。

4.7.2. BeanMap

BeanMap 使一个 bean 的所有属性和值可以按照 map 的方式来读取和修改。Map 的 key 是属性名, value 是属性值。

BeanMap
<ul style="list-style-type: none"> • <<create>> BeanMap() • <<create>> BeanMap(in bean : Object) • toString() : String • clone() : Object • putAllWritable(in map : BeanMap) : void • clear() : void • containsKey(in name : Object) : boolean • containsValue(in value : Object) : boolean • get(in name : Object) : Object • put(in name : Object,in value : Object) : Object • size() : int • keySet() : Set • entrySet() : Set • values() : Collection • getType(in name : String) : Class • keyIterator() : Iterator • valueIterator() : Iterator • entryIterator() : Iterator • getBean() : Object • setBean(in newBean : Object) : void • getReadMethod(in name : String) : Method • getWriteMethod(in name : String) : Method

4.7.3. LazyMap

所谓 LazyMap，意思就是这个 Map 中的键/值对一开始并不存在，当被调用到时才创建，这样的解释初听上去是不是有点不可思议？这样的 LazyMap 有用吗？我们这样来理解：我们需要一个 Map，但是由于创建成员的方法很“重”（比如数据库访问），或者我们只有在调用 get() 时才知道如何创建，或者 Map 中出现的可能性很多很多，我们无法在 get() 之前添加所有可能出现的键/值对，或者任何其它解释得通的原因，我们觉得没有必要去初始化一个 Map 而又希望它可以在必要时自动处理数据生成的话，LazyMap 就变得很有用了。

范例：

```
Factory factory = new Factory() {
    public Object create() {
        return new Date();
    }
}

Map lazy = Lazy.map(new HashMap(), factory);

Object obj = lazy.get("NOW");
```

在上述代码执行后，obj 将包含一个新的 Date 对象，同时这个对象通过 lazy.get("NOW") 参数关联到了“NOW”关键字。

4.7.4. CaseInsensitiveMap

一个关键字大小写不敏感(转换成小写)的 HashMap，支持 null
范例代码：

```
Map map = new CaseInsensitiveMap();  
map.put("One", "One");  
map.put("Two", "Two");  
map.put(null, "Three");  
map.put("one", "Four");
```

返回的 list 将包含内容"A", "B", "C"。

使用 map.get(null) 会返回"Three"

使用 map.get("ONE") 会返回 "Four"

调用 keySet() 返回{"one", "two", null}.

4.7.5. IdentityMap

普通 HashMap 通过调用 key 的 equals()方法来搜索，而 IdentityMap 直接用引用“==”来搜索 key，在特定条件下，速度比 HashMap 快

4.7.6. LRUMap

LRUMap(Least Recently Used)是一个可维护最大容量的 Map，Map 已满时将移去最少使用的节点。

4.7.7. MultiKeyMap

MuliKeyMap 表示一个 value 可以使用多个 key，map 的 size 是按照 value 的数量来计算的。考虑到一个 size=50 的 cache:

```
private MultiKeyMap cache = MultiKeyMap.decorate(new LRUMap(50));  
  
public String getAirlineName(String code, String locale) {  
    String name = (String) cache.get(code, locale);  
    if (name == null) {  
        name = getAirlineNameFromDB(code, locale);  
        cache.put(code, locale, name);  
    }  
    return name;  
}
```

4.7.8. ReferenceMap/ReferenceIdentityMap

允许 Map 中的值被回收站回收。ReferenceIdentityMap 使用引用做比较。

4.7.9. SingletonMap

Map 被创建后，不能再被增加或删除里面的值，但是可以单个修改。

4.8. List 组

4.8.1. CursorableLinkedList

A List implementation with a ListIterator that allows concurrent modifications to the underlying list.

4.8.2. FixedSizeList

Decorates another List to fix the size preventing add/remove.

4.8.3. LazyList

List 中的值当被调用到时才创建

范例：

```
Factory factory = new Factory() {
    public Object create() {
        return new Date();
    }
}

List lazy = LazyList.decorate(new ArrayList(), factory);
Object obj = lazy.get(3);
```

4.8.4. NodeCachingLinkedList

A List implementation that stores a cache of internal Node objects in an effort to reduce wasteful object creation.

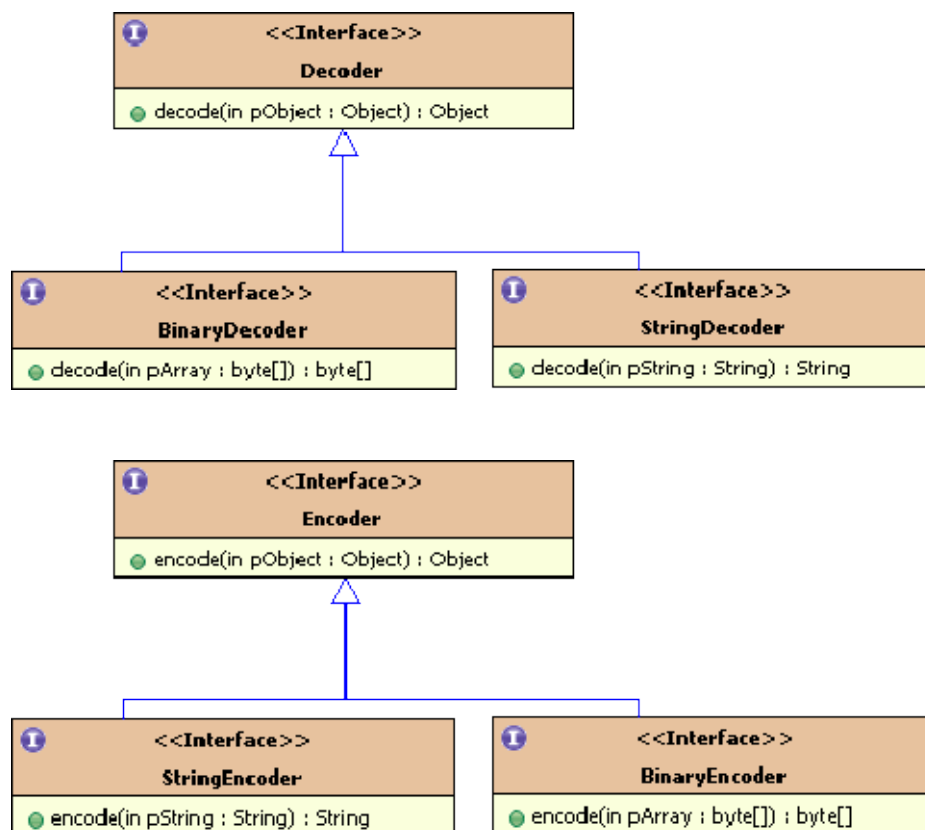
4.8.5. TreeList

TreeList 提供快速的增加、删除功能。和其他 List 的性能比较：

	get	add	insert	iterate	remove
TreeList	3	5	1	2	1
ArrayList	1	1	40	1	40
LinkedList	5800	1	350	2	325

5. Commons Codec

Codec 包含一些通用的编码解码算法。包括一些语音编码器， Hex, Base64, 以及 URL encoder。定义的基础接口：



5.1.commons.codec.binary

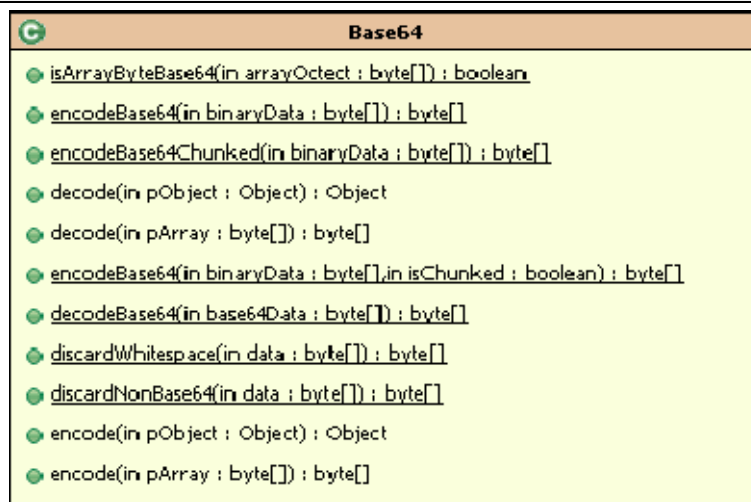
5.1.1. Base64

Base64 编码主要用于 Email 传输。定义 MIME 文档传输的 RFC 规定了 Base 64 编码，从而使得任何二进制数据都可以转换成可打印的 ASCII 字符集安全地传输。例如，假设要通过 Email 传输一个图形文件，Email 客户端软件就会利用 Base64 编码把图形文件的二进制数据转换成 ASCII 码。在 Base64 编码中，每三个 8 位的字节被编码成一个 4 个字符的组，每个字符包含原来 24 位中的 6 位，编码后的字符串大小是原来的 1.3 倍，文件的末尾追加"="符号。除了 MIME 文档之外，Base64 编码技术还用于 BASIC 认证机制中 HTTP 认证头的"用户：密码"字符串。

Base64 类的使用相当简单，最主要的两个静态方法是：Base64.encodeBase64(byte[] byteArray)，用于对字节数组中指定的内容执行 Base64 编码；Base64.decodeBase64(byte[] byteArray)，用于对字节数组中指定的内容执行 Base64 解码。另外，Base64 还有一个静态方法 Base64.isArrayByteBase64(byte[] byteArray)，用于检测指定的字节数组是否可通过 Base64 测试（即是否包含了经过 Base64 编码的数据，如前所述，Base64 编码的结果只包含可打印的 ASCII 字符）。









范例：

```
byte[] encodedBytes=Base64.encodeBase64(testString.getBytes());
String decodedString=new String(Base64.decodeBase64(encodedBytes));
System.err.println("'^'是一个合法的Base64字符吗？"
    + Base64.isArrayByteBase64(invalidBytes));
```













5.1.2. Hex

Hex 编码/解码就是执行字节数据和等价的十六进制表示形式之间的转换。

 Hex
 <code>decodeHex(in data : char[]) : byte[]</code>
 <code>toDigit(in ch : char, in index : int) : int</code>
 <code>encodeHex(in data : byte[]) : char[]</code>
 <code>decode(in array : byte[]) : byte[]</code>
 <code>decode(in object : Object) : Object</code>
 <code>encode(in array : byte[]) : byte[]</code>
 <code>encode(in object : Object) : Object</code>

5.1.3. BinaryCodec










在字节数据和等价的二进制表示形式之间的转换。

 BinaryCodec
 <code>encode(in raw : byte[]) : byte[]</code>
 <code>encode(in raw : Object) : Object</code>
 <code>decode(in ascii : Object) : Object</code>
 <code>decode(in ascii : byte[]) : byte[]</code>
 <code>toByteArray(in ascii : String) : byte[]</code>
 <code>fromAscii(in ascii : char[]) : byte[]</code>
 <code>fromAscii(in ascii : byte[]) : byte[]</code>
 <code>toAsciiBytes(in raw : byte[]) : byte[]</code>
 <code>toAsciiChars(in raw : byte[]) : char[]</code>
 <code>toAsciiString(in raw : byte[]) : String</code>

5.2. commons.codec.digest

5.2.1. DigestUtil

对 java.security.MessageDigest 的一些有用的封装

 DigestUtils
 <code>getDigest(in algorithm : String) : MessageDigest</code>
 <code>md5(in data : byte[]) : byte[]</code>
 <code>md5(in data : String) : byte[]</code>
 <code>md5Hex(in data : byte[]) : String</code>
 <code>md5Hex(in data : String) : String</code>
 <code>sha(in data : byte[]) : byte[]</code>
 <code>sha(in data : String) : byte[]</code>
 <code>shaHex(in data : byte[]) : String</code>
 <code>shaHex(in data : String) : String</code>

6. Commons Betwixt

- 概况：实现 XML 和 JavaBean 的映射。

- 何时适用：当你想要以灵活的方式实现 XML 和 Bean 的映射，需要一个数据绑定框架之时。

- 示例应用：BetwixtDemo.java, Mortgage.java, mortgage.xml。要求 CLASSPATH 中必须包含 commons-betwixt-1.0-alpha-1.jar、commons-logging.jar、commons-beanutils.jar、commons-collections.jar、以及 commons-digester.jar。

- 说明：

如果你以前曾经用 Castor 绑定数据，一定会欣赏 Betwixt 的灵活性。Castor 适合在一个预定义模式（Schema）的基础上执行 Bean 和 XML 之间的转换；但如果你只想执行数据和 XML 之间的转换，最好的选择就是 Betwixt。Betwixt 的特点就是灵活，能够方便地将数据输出成为人类可阅读的 XML。

Betwixt 的用法相当简单。如果要把 Bean 转换成 XML，首先创建一个 BeanWriter 的实例，设置其属性，然后输出；如果要把 XML 转换成 Bean，首先创建一个 BeanReader 的实例，设置其属性，然后用 Digester 执行转换。

将 Bean 转换成 XML：

```
// 用 Betwixt 将 Bean 转换成 XML 必须有 BeanWriter 的实例。
// 由于 BeanWriter 的构造函数要求有一个写入器对象，
// 所以从创建一个 StringWriter 开始
StringWriter outputWriter = new StringWriter();
// 注意输出结果并不是格式良好的，所以需要在开始位置
// 写入下面的内容：
outputWriter.write("<?xml version='1.0' ?>");
// 创建一个 BeanWriter
BeanWriter writer = new BeanWriter(outputWriter);
// 我们可以设置该写入器的各种属性。
// 下面的第一行禁止写入 ID，
// 第二行允许格式化输出
writer.setWriteIDs(false);
writer.enablePrettyPrint();
// 创建一个 Bean 并将其输出
Mortgage mortgage = new Mortgage(6.5f, 25);
// 将输出结果写入输出设备
try {
    writer.write("mortgage", mortgage);
    System.err.println(outputWriter.toString());
}
```



```
} catch(Exception e) {  
    System.err.println(e);  
}
```

将 XML 转换成 Bean:

```
// 用 Betwixt 来读取 XML 数据并以此为基础创建  
// Bean, 必须用到 BeanReader 类。注意 BeanReader 类扩展了  
// Digester 包的 Digester 类。  
BeanReader reader = new BeanReader();  
  
// 注册类  
try {  
    reader.registerBeanClass(Mortgage.class);  
    // 并解析它...  
    Mortgage mortgageConverted =  
        (Mortgage)reader.parse(new File("mortgage.xml"));  
    // 检查转换得到的 mortgage 是否包含文件中的值  
    System.err.println("Rate: " + mortgageConverted.getRate() +  
        ", Years: " + mortgageConverted.getYears());  
} catch(Exception ee) {  
    ee.printStackTrace();  
}
```

注意, 通过 `BeanReader` 注册类时, 如果顶层元素的名称和类的名称不同, 必须用另一个方法注册并指定准确的路径, 如 `reader.registerBeanClass("toplevelementname", Mortgage.class)`。

7. Commons Digester

- 概况: 提供友好的、事件驱动的高级 XML 文档处理 API。
- 何时适用: 当你想要处理 XML 文档, 而且希望能够根据 XML 文档中特定的模式所触发的一组规则来执行某些操作时。
- 示例应用: `DigesterDemo.java`、`Employee.java`、`Company.java`、`rules.xml` 以及 `company.xml`。要求 CLASSPATH 中必须包含 `commons-digester.jar`、`commons-logging.jar`、`commons-beanutils.jar` 以及 `commons-collections.jar`。
- 说明:
Digester 在解析配置文件的时候最为有用。实际上, Digester 最初就是为读取 Struts 配置文件而开发的, 后来才移到 Commons 包。

Digester 是一个强大的模式匹配工具，允许开发者在一个比 SAX 或 DOM API 更高的层次上处理 XML 文档，当找到特定的模式(或找不到模式)时能够触发一组规则。使用 **Digester** 的基本思路是：首先创建一个 **Digester** 的实例，然后用它注册一系列模式和规则，最后将 XML 文档传递给它。此后，**Digester** 就会分析 XML 文档，按照注册次序来触发规则。如果 XML 文档中的某个元素匹配一条以上的规则，所有的规则会按照注册次序被依次触发。

Digester 本身带有 12 条预定义的规则。当 XML 文档中找到一个特定的模式时，想要调用某个方法吗？很简单，使用预定义的 **CallMethodRule**！另外，你不一定要使用预定的规则，**Digester** 允许用户通过扩展 **Rule** 类定义自己的规则。

在指定模式时，元素必须用绝对名称给出。例如，根元素直接用名称指定，下一层元素则通过"/"符号引出。例如，假设 **company** 是根元素，**company/employee** 就是匹配其中一个子元素的模式。**Digester** 允许使用通配符，例如 ***/employee** 将匹配 XML 文档内出现的所有 **employee** 元素。

找到匹配的模式时，关联到该匹配模式的规则内有四个回调方法会被调用，它们是：**begin**，**end**，**body**，和 **finish**。这些方法被调用的时刻正如其名字所示，例如调用 **begin** 和 **end** 的时刻分别是遇到元素的开始标记和结束标记之时，**body** 是在遇到了匹配模式之内的文本时被调用，**finish** 则是在全部对匹配模式的处理工作结束后被调用。

最后，模式可以在一个外部的规则 XML 文档内指定（利用 **digester-rules.dtd**），或者在代码之内指定，下面要使用的是第一种办法，因为这种办法比较常用。

使用 **Digester** 之前要创建两个 XML 文档。第一个就是数据或配置文件，也就是我们准备对其应用规则的文件。下面是一个例子（**company.xml**）

```
<?xml version="1.0" encoding="gb2312"?>
<company>
  <name>我的公司</name>
  <address>中国浙江</address>
  <employee>
    <name>孙悟空</name>
    <employeeNo>10000</employeeNo>
  </employee>
  <employee>
    <name>猪八戒</name>
    <employeeNo>10001</employeeNo>
  </employee>
</company>
```

第二个文件是规则文件 `rules.xml`。`rules.xml` 告诉 `Digester` 要在 `company.xml` 中查找什么、找到了之后执行哪些操作：

```
<?xml version="1.0" encoding="gb2312"?>
<digester-rules>
  <!-- 创建顶层的 Company 对象 -->
  <object-create-rule pattern="company" classname="Company" />
  <call-method-rule pattern="company/name" methodname="setName"
    paramcount="0" />
  <call-method-rule pattern="company/address"
    methodname="setAddress" paramcount="0" />
  <pattern value="company/employee">
    <object-create-rule classname="Employee" />
    <call-method-rule pattern="name" methodname="setName"
      paramcount="0" />
    <call-method-rule pattern="employeeNo" methodname="setEmployeeNo"
      paramcount="0" />
    <set-next-rule methodname="addEmployee" />
  </pattern>
</digester-rules>
```

这个文件有哪些含义呢？第一条规则，`<object-create-rule pattern="company" classname="Company" />`，告诉 `Digester` 如果遇到了模式 `company`，则必须遵从 `object-create-rule`，也就是要创建一个类的实例！那么要创建的是哪一个类的实例呢？`classname="Company"` 属性指定了类的名称。因此，解析 `company.xml` 的时候，当遇到顶级的 `company` 元素，等到 `object-create-rule` 规则执行完毕，我们就拥有了一个 `Digester` 创建的 `Company` 类的实例。

现在要理解 `call-method-rule` 规则也应该不那么困难了，这里 `call-method-rule` 的功能是在遇到 `company/name` 或 `company/address` 模式时调用一个方法（方法的名字通过 `methodname` 属性指定）。

最后一个模式匹配值得注意，它把规则嵌套到了匹配模式之中。两种设定规则和模式的方式都是 `Digester` 接受的，我们可以根据自己的需要任意选择。在这个例子中，模式里面定义的规则在遇到 `company/employee` 模式时创建一个 `Employee` 类的对象，设置其属性，最后用 `set-next-rule` 将这个雇员加入到顶层的 `Company`。

创建好上面两个 XML 文件之后，只要用两行代码就可以调用 `Digester` 了：

```
Digester digester = DigesterLoader.createDigester(rules.toURL());
Company company = (Company)digester.parse(inputXMLFile);
```

第一行代码装入规则文件，创建一个 **Digester**。第二行代码利用该 **Digester** 来应用规则。

8. Commons CLI

■ 概况：**CLI** 即 **Command Line Interface**，也就是"命令行接口"，它为 **Java** 程序访问和解析命令行参数提供了一种统一的接口。

■ 何时适用：当你需要以一种一致的、统一的方式访问命令行参数之时。

■ 示例应用：**CLIDemo.java**。**CLASSPATH** 中必须包含 **commons-cli-1.0.jar**。

■ 说明：

有多少次你不得不为一个新的应用程序重新设计新的命令行参数处理方式？如果能够只用某个单一的接口，统一完成诸如定义输入参数（是否为强制参数，数值还是字符串，等等）、根据一系列规则分析参数、确定应用要采用的路径等任务，那该多好！答案就在 **CLI**。

在 **CLI** 中，每一个想要在命令中指定的参数都是一个 **Option** 对象。首先创建一个 **Options** 对象，将各个 **Option** 对象加入 **Options** 对象，然后利用 **CLI** 提供的方法来解析用户的输入参数。**Option** 对象可以要求用户必须输入某个参数，例如必须在命令行提供文件名。如果某个参数是必须的，创建 **Option** 对象的时候就要显式地指定。

下面是使用 **CLI** 的步骤。

```
// ...
// ① 创建一个 Options:
Options options = new Options();
options.addOption("t", false, "current time");
// ...
// ② 创建一个解析器，分析输入:
CommandLineParser parser = new BasicParser();
CommandLine cmd;
try {
    cmd = parser.parse(options, args);
} catch (ParseException pe) {
    usage(options);
}
```

```
        return;
    }
    // ...
    // ③ 最后就可以根据用户的输入，采取相应的操作：
    if (cmd.hasOption("n")) {
        System.err.println("Nice to meet you: " +
            cmd.getOptionValue('n'));
    }
}
```

这就是使用 CLI 的完整过程了。当然，CLI 还提供了其他高级选项，例如控制格式和解析过程等，但基本的使用思路仍是一致的。

9. Commons Discovery

- 概况：Discovery 组件是发现模式（Discovery Pattern）的一个实现，它的目标是按照一种统一的方式定位和实例化类以及其他资源。

- 何时适用：当你想用最佳的算法在 Java 程序中查找 Java 接口的各种实现之时。

- 应用实例：DiscoveryDemo.java, MyInterface.java, MyImpl1.java, MyImpl2.java, MyInterface。要求 CLASSPATH 中必须包含 commons-discovery.jar 和 commons-logging.jar。

- 说明：

Discovery 的意思就是"发现"，它试图用最佳的算法查找某个接口的所有已知的实现。在使用服务的场合，当我们想要查找某个服务的所有已知的提供者时，Discovery 组件尤其有用。

考虑一下这种情形：我们为某个特别复杂的任务编写了一个接口，所有该接口的实现都用各不相同的方式来完成这个复杂任务，最终用户可以根据需要来选择完成任务的具体方式。那么，在这种情形下，最终用户应该用什么办法来找出接口的所有可用实现（即可能的完成任务的方式）呢？

上面描述的情形就是所谓的服务-服务提供者体系。服务的功能由接口描述，服务提供者则提供具体的实现。现在的问题是最终用户要用某种办法来寻找系统中已经安装了哪些服务提供者。在这种情形下，Discovery 组件就很有用了，它不仅可以用来查找那些实现了特定接口的类，而且还可以用来查找资源，例如图片或其他文件等。在执行这些操作时，Discovery 遵从 Sun 的服务提供者体系所定义的规则。

由于这个原因，使用 **Discovery** 组件确实带来许多方便。请读者参阅本文后面示例程序中的接口 **MyInterface.java** 和两个实现类 **MyImpl1.java**、**MyImpl2.java**，了解下面例子的细节。在使用 **Discovery** 的时候要提供 **MyInterface** 文件，把它放入 **META-INF/services** 目录，注意该文件的名称对应接口的完整限定名称（**Fully Qualified Name**），如果接口属于某个包，该文件的名称也必须相应地改变。

```
// ...
// ① 创建一个类装入器的实例。
ClassLoaders loaders =
    ClassLoaders.getAppLoaders(MyInterface.class, getClass(), false);
// ...
// ② 用 DiscoverClass 的实例来查找实现类。
DiscoverClass discover = new DiscoverClass(loaders);
// ...
// ③ 查找实现了指定接口的类：
Class implClass = discover.find(MyInterface.class);
System.err.println("Implementing Provider: " + implClass.getName());
```

运行上面的代码，就可以得到在 **MyInterface** 文件中注册的类。再次提醒，如果你的实现是封装在包里面的，在这里注册的名字也应该作相应地修改，如果该文件没有放在正确的位置，或者指定名字的实现类不能找到或实例化，程序将抛出 **DiscoverException**，表示找不到符合条件的实现。下面是 **MyInterface** 文件内容的一个例子：**MyImpl2 # Implementation 2**。

当然，实现类的注册办法并非只有这么一种，否则的话 **Discovery** 的实用性就要大打折扣了！实际上，按照 **Discovery** 内部的类查找机制，按照这种方法注册的类将是 **Discovery** 最后找到的类。另一种常用的注册方法是通过系统属性或用户定义的属性来传递实现类的名字，例如，放弃 **META-INF/services** 目录下的文件，改为执行 **java -DMyInterface=MyImpl1 DiscoveryDemo** 命令来运行示例程序，这里的系统属性是接口的名字，值是该接口的提供者，运行的结果是完全一样的。

Discovery 还可以用来创建服务提供者的(**singleton**)实例并调用其方法，语法如下：**((MyInterface)discover.newInstance(MyInterface.class)).myMethod();**。注意在这个例子中，我们并不知道到底哪一个服务提供者实现了 **myMethod**，甚至我们根本不必关心这一点。具体的情形与运行这段代码的方式以及运行环境中已经注册了什么服务提供者有关，在不同的环境下运行，实际得到的服务提供者可能不同。

