jad反编译工具，已经不再更新，且只支持JDK1.4，但并不影响其强大的功能。

基本用法：`jad xxx.class`，会生成直接可读的xxx.jad文件。

自动拆装箱

对于基本类型和包装类型之间的转换，通过xxxValue()和valueOf()两个方法完成自动拆装箱，使用jad进行反编译可以看到该过程：

```
public class Demo {
  public static void main(String[] args) {
    int x = new Integer(10);  // 自动拆箱
    Integer y = x;            // 自动装箱
  }
}
```

反编译后结果：

```
public class Demo
{
    public Demo(){}

    public static void main(String args[])
    {
        int i = (new Integer(10)).intValue();   // intValue()拆箱
        Integer integer = Integer.valueOf(i);   // valueOf()装箱
    }
}
```

foreach语法糖

在遍历迭代时可以foreach语法糖，对于数组类型直接转换成for循环：

```
// 原始代码
int[] arr = {1, 2, 3, 4, 5};
    for(int item: arr) {
        System.out.println(item);
    }
}

// 反编译后代码
int ai[] = {
    1, 2, 3, 4, 5
};
int ai1[] = ai;
int i = ai1.length;
// 转换成for循环
```

```
for(int j = 0; j < i; j++)
{
    int k = ai1[j];
    System.out.println(k);
}
```

对于容器类的遍历会使用iterator进行迭代：

```
import java.io.PrintStream;
import java.util.*;

public class Demo
{
    public Demo() {}
    public static void main(String args[])
    {
        ArrayList arraylist = new ArrayList();
        arraylist.add(Integer.valueOf(1));
        arraylist.add(Integer.valueOf(2));
        arraylist.add(Integer.valueOf(3));
        Integer integer;
        // 使用的for循环+Iterator，类似于链表迭代：
        // for (ListNode cur = head; cur != null; System.out.println(cur.val)){
        //     cur = cur.next;
        // }
        for(Iterator iterator = arraylist.iterator(); iterator.hasNext();
System.out.println(integer))
            integer = (Integer)iterator.next();
    }
}
```

Arrays.asList(T...)

熟悉Arrays.asList(T...)用法的小伙伴都应该知道，asList()方法传入的参数不能是基本类型的数组，必须包装成包装类型再使用，否则对应生成的列表的大小永远是1：

```
import java.util.*;
public class Demo {
  public static void main(String[] args) {
    int[] arr1 = {1, 2, 3};
    Integer[] arr2 = {1, 2, 3};
    List lists1 = Arrays.asList(arr1);
    List lists2 = Arrays.asList(arr2);
    System.out.println(lists1.size()); // 1
    System.out.println(lists2.size()); // 3
  }
}
```

从反编译结果来解释，为什么传入基本类型的数组后，返回的List大小是1：

```
// 反编译后文件
import java.io.PrintStream;
import java.util.Arrays;
import java.util.List;

public class Demo
{
    public Demo() {}

    public static void main(String args[])
    {
        int ai[] = {
            1, 2, 3
        };
        // 使用包装类型，全部元素由int包装为Integer
        Integer ainteger[] = {
            Integer.valueOf(1), Integer.valueOf(2), Integer.valueOf(3)
        };

        // 注意这里被反编译成二维数组，而且是一个1行三列的二维数组
        // list.size()当然返回1
        List list = Arrays.asList(new int[][] { ai });
        List list1 = Arrays.asList(ainteger);
        System.out.println(list.size());
        System.out.println(list1.size());
    }
}
```

从上面结果可以看到，传入基本类型的数组后，会被转换成一个二维数组，而且是**new int[1][arr.length]**这样的数组，调用list.size()当然返回1。

注解

Java中的类、接口、枚举、注解都可以看做是类类型。使用jad来看一下@interface被转换成什么：

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(RetentionPolicy.RUNTIME)
public @interface Foo{
  String[] value();
  boolean bar();
}
```

查看反编译代码可以看出：

- 自定义的注解类Foo被转换成接口Foo，并且继承Annotation接口

- 原来自定义接口中的value()和bar()被转换成抽象方法

```
import java.lang.annotation.Annotation;

public interface Foo
    extends Annotation
{
    public abstract String[] value();

    public abstract boolean bar();
}
```

注解通常和反射配合使用，而且既然自定义的注解最终被转换成接口，注解中的属性被转换成接口中的抽象方法，那么通过反射之后拿到接口实例，在通过接口实例自然能够调用对应的抽象方法：

```
import java.util.Arrays;

@Foo(value={"sherman", "decompiler"}, bar=true)
public class Demo{
    public static void main(String[] args) {
        Foo foo = Demo.class.getAnnotation(Foo.class);
        System.out.println(Arrays.toString(foo.value())); // [sherman, decompiler]
        System.out.println(foo.bar());                    // true
    }
}
```

枚举

通过jad反编译可以很好地理解枚举类。

空枚举

先定义一个空的枚举类：

```
public enum DummyEnum {
}
```

使用jad反编译查看结果：

- 自定义枚举类被转换成final类，并且继承Enum
- 提供了两个参数（name，odinal）的私有构造器，并且调用了父类的构造器。注意即使没有提供任何参数，也会有该该构造器，其中name就是枚举实例的名称，odinal是枚举实例的索引号
- 初始化了一个private static final自定义类型的空数组 **$VALUES**
- 提供了两个public static方法：
    - values()方法通过clone()方法返回内部$VALUES的浅拷贝。这个方法结合私有构造器可以完美实现单例模式，想一想values()方法是不是和单例模式中getInstance()方法功能类似

- valueOf(String s)：调用父类Enum的valueOf方法并强转返回

```
public final class DummyEnum extends Enum
{
    // 功能和单例模式的getInstance()方法相同
    public static DummyEnum[] values()
    {
        return (DummyEnum[])$VALUES.clone();
    }
    // 调用父类的valueOf方法，并强转返回
    public static DummyEnum valueOf(String s)
    {
        return (DummyEnum)Enum.valueOf(DummyEnum, s);
    }
    // 默认提供一个私有的两个参数的构造器，并调用父类Enum的构造器
    private DummyEnum(String s, int i)
    {
        super(s, i);
    }
    // 初始化一个private static final的本类空数组
    private static final DummyEnum $VALUES[] = new DummyEnum[0];

}
```

包含抽象方法的枚举

枚举类中也可以包含抽象方法，但是必须定义枚举实例并且立即重写抽象方法，就像下面这样：

```
public enum DummyEnum {
    DUMMY1 {
        public void dummyMethod() {
            System.out.println("[1]: implements abstract method in enum class");
        }
    },

    DUMMY2 {
        public void dummyMethod() {
            System.out.println("[2]: implements abstract method in enum class");
        }
    };

    abstract void dummyMethod();

}
```

再来反编译看看有哪些变化：

- 原来final class变成了abstract class：这很好理解，有抽象方法的类自然是抽象类

- 多了两个public static final的成员DUMMY1、DUMMY2，这两个实例的初始化过程被放到了static代码块中，并且实例过程中直接重写了抽象方法，类似于匿名内部类的形式。
- 数组**$VALUES[]**初始化时放入枚举实例

还有其它变化么?

在反编译后的DummyEnum类中，是存在抽象方法的，而枚举实例在静态代码块中初始化过程中重写了抽象方法。在Java中，抽象方法和抽象方法重写同时放在一个类中，只能通过内部类形式完成。因此上面第二点应该说成就是以内部类形式初始化。

可以看一下DummyEnum.class存放的位置，应该多了两个文件:

- DummyEnum$1.class
- DummyEnum$2.class

Java中.class文件出现$符号表示有内部类存在，就像OutClass$InnerClass，这两个文件出现也应证了上面的匿名内部类初始化的说法。

```
import java.io.PrintStream;

public abstract class DummyEnum extends Enum
{
    public static DummyEnum[] values()
    {
        return (DummyEnum[])$VALUES.clone();
    }

    public static DummyEnum valueOf(String s)
    {
        return (DummyEnum)Enum.valueOf(DummyEnum, s);
    }

    private DummyEnum(String s, int i)
    {
        super(s, i);
    }

    // 抽象方法
    abstract void dummyMethod();

    // 两个pubic static final实例
    public static final DummyEnum DUMMY1;
    public static final DummyEnum DUMMY2;
    private static final DummyEnum $VALUES[];

    // static代码块进行初始化
    static
    {
        DUMMY1 = new DummyEnum("DUMMY1", 0) {
            public void dummyMethod()
            {
                System.out.println("[1]: implements abstract method in enum
```

```
class");
            }
        }
;
        DUMMY2 = new DummyEnum("DUMMY2", 1) {
            public void dummyMethod()
            {
                System.out.println("[2]: implements abstract method in enum
class");
            }
        }
;
        // 对本类数组进行初始化
        $VALUES = (new DummyEnum[] {
            DUMMY1, DUMMY2
        });
    }
}
```

正常的枚举类

实际开发中，枚举类通常的形式是有两个参数（int code，Sring msg）的构造器，可以作为状态码进行返回。Enum类实际上也是提供了包含两个参数且是protected的构造器，这里为了避免歧义，将枚举类的构造器设置为三个，使用jad反编译：

最大的变化是：现在的private构造器从2个参数变成5个，而且在内部仍然将前两个参数通过super传递给父类，剩余的三个参数才是真正自己提供的参数。可以想象，如果自定义的枚举类只提供了一个参数，最终生成底层代码中private构造器应该有三个参数，前两个依然通过super传递给父类。

```
public final class CustomEnum extends Enum
{
    public static CustomEnum[] values()
    {
        return (CustomEnum[])$VALUES.clone();
    }

    public static CustomEnum valueOf(String s)
    {
        return (CustomEnum)Enum.valueOf(CustomEnum, s);
    }

    private CustomEnum(String s, int i, int j, String s1, Object obj)
    {
        super(s, i);
        code = j;
        msg = s1;
        data = obj;
    }

    public static final CustomEnum FIRST;
    public static final CustomEnum SECOND;
```

```
    public static final CustomEnum THIRD;
    private int code;
    private String msg;
    private Object data;
    private static final CustomEnum $VALUES[];

    static
    {
        FIRST = new CustomEnum("FIRST", 0, 10010, "first", Long.valueOf(100L));
        SECOND = new CustomEnum("SECOND", 1, 10020, "second", "Foo");
        THIRD = new CustomEnum("THIRD", 2, 10030, "third", new Object());
        $VALUES = (new CustomEnum[] {
            FIRST, SECOND, THIRD
        });
    }
}
```