day01

MyEclipse 介绍

1 debug 断点调试

- 设置断点；
- 测试跳入（F5）；
- 测试跳过（F6）；
- 测试跳出（F7）；
- 添加断点；
- 测试进入下一断点；
- 测试返回当前方法栈的头部（Drop To Frame）；
- 清除断点；y
- 清除表达式；

注意，停止程序！

2 常用快捷键

必须的：

- Alt + /（内容助理）：补全；
- Ctrl + 1（快速定位）：出错时定位错误，与点击“红 X”效果一样；
- Ctrl + Shift + O：导包；
- Ctrl + Shift + F：格式化代码块；

不是必须的（自己读，不讲）：

- Ctrl + Shift + T：查看源代码；
- Ctrl + 点击源代码：查看源代码；
- F3：查看选中类的源代码；
- Alt + 左键：查看源代码时的“原路返回”；
- Ctrl + Shift + X：把小写修改为大写；
- Ctrl + Shift + Y：把小写修改为小写；
- Ctrl + Alt + 下键：复制当前行；
- Ctrl + /：添加或撤销行注释；
- Ctrl + Shift + /：对选中代码添加段注释；
- Ctrl + Shift + \：撤销当前段注释；
- Alt + 上键：向上移动当前行；
- Alt + 下键：向上移动当前行；

- Ctrl + D：删除当前行；

JUnit

1 JUnit 的概述

当我们写完了一个类后，总是要执行一下类中的方法，查看是否符合我们的意图，这就是单元测试了。而 Junit 就是单元测试工具。

- 导包：导入 Junit4 或以上版本；
- 编写一个类：Person，它就是要被测试的类；
- 编写测试类：PersonTest，给出测试方法，在测试方法上使用@Test 注解；
- 执行测试方法。

Person

```
package cn.itcast;

public class Person {
    public void run() {
        System.out.println("run");
    }
    public void eat() {
        System.out.println("eat");
    }
}
```

包资源管理器→选中 Person 类→右键→new→JUnit TestCase→修改包名为 junit.test→下一步→选中要测试的方法。

PersonTest

```
package junit.test;

import org.junit.Test;
import cn.itcast.Person;

public class PersonTest {
    @Test
    public void testRun() {
        Person person = new Person();
        person.run();
    }
    @Test
    public void testEat() {
```

```
        Person person = new Person();
        person.eat();
    }
}
```

选中某个测试方法，鼠标右键→Run as→Junit Test，即执行测试方法。

@Test 注解的作用是指定方法为测试方法，测试方法必须是 public、void、无参的!!!

2 @Before 和@After(了解)

如果你需要某个方法在每个测试方法之前先执行，那么你需要写一个方法，然后使用@Before 来标记这个方法。例如在 testRun()和 testEat()方法之前需要创建一个 Person 对象。

PersonTest

```
package junit.test;

import org.junit.Before;
import org.junit.Test;
import cn.itcast.Person;

public class PersonTest {
    private Person person;
    @Before
    public void setUp() {
        person = new Person();
    }
    @Test
    public void testRun() {
        person.run();
    }
    @Test
    public void testEat() {
        person.eat();
    }
}
```

@After 注解标记的方法会在每个执行方法之后执行

@Before 和@After 标记的方法必须是 public、void、无参。

JDK5.0 新特性

1 自动拆装箱

自动拆装箱是 JDK5.0 的新特性之一，这一特性可以使基本类型，与之对应的包装器类型之间直接转换，例如 int 的包装器类型是 Integer！在 JDK5.0 之后，你甚至可以把 int 当作成 Integer 来使用，把 Integer 当成 int 来使用。当然，这不是 100%的！

1.1 自动拆装箱概述

在 JDK5.0 之后，Java 允许把基本类型与其对应的包装器类型之间自动相互转换。

- 自动装箱：Integer i = 100，把 int 类型直接赋值给 Integer 类型；
- 自动拆装：int a = new Integer(100)，把 Integer 类型直接赋值给 int 类型。

1.2 自动拆装箱原理

其实自动拆装箱是由编译器完成的！我们写的代码，再由编译器“二次加工”，然后再编译成.class 文件！那么编译器是怎么“二次加工”的呢？

- Integer i = 100：编译器加工为：Integer i = Integer.valueOf(100)；
- int a = i：编译器加载为：int a = i.intValue()；

这也说明一个道理：JVM 并不知道什么是自动拆装箱，JVM 还是原来的 JVM（JDK1.4 之前），只是编译器在 JDK5.0 时“强大”了！

1.3 自动拆装箱演变

大家来看看下面代码：

```
Integer i = 100;//这是自动装箱
Object o = i;//这是向上转型
```

上面代码是没有问题的，我们是否可以修改上面代码为：

```
Object o = 100;
```

ok，这是可以的！通过编译器的处理后上面代码为：

```
Object o = Integer.valueOf(100);
```

在来看下面代码：

```
Object o = Integer.valueOf(100);
int a = o;//编译失败！
```

上面代码是不行的，因为 o 不是 Integer 类型，不能自动拆箱，你需要先把 o 转换成 Integer 类型后，才能赋值给 int 类型。

```
Object o = Integer.valueOf(100);
int a = (Integer)o;
```

1.4 变态小题目

来看下面代码：

```
Integer i1 = 100;
Integer i2 = 100;
boolean b1 = i1 == i2;//结果为 true

Integer i3 = 200;
Integer i4 = 200;
boolean b2 = i3 == i4;//结果为 false
```

你可能对上面代码的结果感到费解，那么我们来打开这个疑团。第一步，我们先把上面代码通过编译器的“二次加工”处理一下：

```
Integer i1 = Integer.valueOf(100);
Integer i2 = Integer.valueOf(100);
boolean b1 = i1 == i2;//结果为 true

Integer i3 = Integer.valueOf(200);
Integer i4 = Integer.valueOf(200);
boolean b2 = i3 == i4;//结果为 false
```

这时你应该可以看到，疑团在 Integer.valueOf()方法身上。传递给这个方法 100 时，它返回的 Integer 对象是同一个对象，而传递给这个方法 200 时，返回的却是不同的对象。这是我们需要打开 Integer 的源码（这里就不粘贴 Integer 的源代码了），查看它的 valueOf()方法内容。

Integer 类的内部缓存了-128~127 之间的 256 个 Integer 对象，如果 valueOf()方法需要把这个范围之内的整数转换成 Integer 对象时，valueOf()方法不会去 new 对象，而是从缓存中直接获取，这就会导致 valueOf(100)两次，都是从缓存中获取的同一个 Integer 对象！如果 valueOf()方法收到的参数不在缓存范围之内，那么 valueOf()方法会 new 一个新对象！这就是为什么 Integer.valueOf(200)两次返回的对象不同的原因了。

2 可变参数

可变参数就是一个方法可以接收任意多个参数！例如：fun()、fun(1)、fun(1,1)、fun(1,1,1)。你可能认为这是方法重载，但这不是重载，你想想重载能重载多少个方法，而 fun()方法是可以传递任何个数的参数，你能重载这么多个方法么？

2.1 定义可变参数方法

```
public void fun(int... arr) {}
```

上面方法 fun()的参数类型为 int...，其中“...”不是省略号，而是定义参数类型的方式。参数 arr 就是可变参数类型。你可以把上面代码理解为：public void fun(int[] arr)。

```
public int sum1(int[] arr) {
    int sum = 0;
```

```
        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
```

```
    public int sum2(int... arr) {
        int sum = 0;
        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
```

你可能会想，“int[]”和“int…”没有什么不同，只是“int…”是一种新的定义数组形参的方式罢了。那么我应该恭喜你了！没错，这么理解就对了！但要注意，只有在方法的形参中可以使用 int…来代替 int[]。

2.2 调用带有可变参数的方法

sum1()和 sum2()两个方法的调用：

```
sum1(new int[]{1,2,3});
sum2(new int[]{1,2,3});
```

这看起来没什么区别！但是对于 sum2 还有另一种调用方式：

```
sum2();
sum2(1);
sum2(1,2);
sum2(1,2,3);
```

这看起来好像是使用任意多个参数来调用 sum2()方法，这就是调用带有可变参数方法的好处了。

2.3 编译器“二次加工”

编译器对 sum2 方法定义的“二次加工”结果为：

```
    public int sum2(int[] arr) {
        int sum = 0;
        for(int i = 0; i < arr.length; i++) {
            sum += arr[i];
        }
        return sum;
    }
```

即把“int…”修改为“int[]”类型。

编译器对 sum2 方法调用的二次加载结果为：

```
sum2(new int[0]);
sum2(new int[] {1});
sum2(new int[] {1, 2});
sum2(new int[] {1, 2, 3});
```

结论：可变参数其实就是数组类型，只不过在调用方法时方便一些，由编译器帮我们把多个实参放到一个数组中传递给形参。

2.4 可变参数方法的限制

- 一个方法最多只能有一个可变参数；
- 可变参数必须是方法的最后一个参数。

3 增强 for 循环

增强 for 循环是 for 的一种新用法！用来循环遍历数组和集合。

3.1 增强 for 的语法

for(元素类型 e : 数组或集合对象) {
}
例如：

```
int[] arr = {1,2,3};
for(int i : arr) {
    System.out.println(i);
}
```

增强 for 的冒号左边是定义变量，右边必须是数组或集合类型。例如上例中循环遍历的主浊 arr 这个 int 数组，增强 for 内部会依次把 arr 中的元素赋给变量 i。

3.2 增强 for 的优缺点

- 只能从头到尾的遍历数组或集合，而不能只遍历部分；
- 在遍历 List 或数组时，不能获取当前元素下标；
- 增强 for 使用便简单，这是它唯一的优点了；
- 增强 for 比使用迭代器方便一点！

3.3 增强 for 原理

其实增强 for 内部是使用迭代器完成的！也就是说，任何实现了 Iterable 接口的对象都可以被增强 for 循环遍历！这也是为什么增强 for 可以循环遍历集合的原因（Collection 是 Iterable 的子接口）。

但要注意，Map 并没有实现 Iterable 接口，所以你不能直接使用增强 for 来遍历它！

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

```
        map.put("1", "one");
        map.put("2", "two");
        map.put("3", "three");

        for(String key : map.keySet()) {
            String value = map.get(key);
            System.out.println(key + "=" + value);
        }
```

泛型

1 泛型概述

泛型是 JDK5.0 新特性，它主要应用在集合类上。有了泛型之后，集合类与数组就越来越像了。例如：Object[] objs = new Object[10]，可以用来存储任何类型的对象。String[] strs = new String[10]只能用来存储 String 类型的对象。

ArrayList list = new ArrayList()，可以用来存储任何类型的对象。ArrayList<String> list = new ArrayList<String>()只有用来存储 String 类型的对象。

1.1 理解泛型类

泛型类具有一到多个泛型变量，在创建泛型类对象时，需要为泛型变量指定值。泛型变量只能赋值为引用类型，而不能是基本类型。例如 ArrayList 类中有一个泛型变量 E，在创建 ArrayList 类的对象时需要为 E 这个泛型变量指定值。

list<String> list = new ArrayList<String>();，

其中 String 就是给 List 的泛型变量 E 赋值了。查阅 ArrayList 的 API 你会知道，泛型变量 E 出现在很多方法中：

boolean add(E e)

E get(int index)

因为我们在创建 list 对象时给泛型类型赋值为 String，所以对于 list 对象而言，所有 API 中的 E 都会被 String 替换。

boolean add(String e)

String get(int index)

也就是说，在使用 list.add()时，只能传递 String 类型的参数，而 list.get()方法返回的一定是 String 类型。

list.add(“hello”);

String s = list.get(0);

1.2 使用泛型对象

创建泛型对象时，引用和 new 两端的泛型类型需要一致，例如上面的引用是 List<String>，而 new 一端是 new ArrayList<String>，两端都是 String 类型！如果不一致就会出错：

List<Object> list = new ArrayList<String>();//编译失败！

Collection 的 addAll()方法中使用了通配符（这个概念在下一个基础加强中讲解），所以使用起来比较方便：

```
List<Object> list = new ArrayList<Object>();
List<String> list1 = new ArrayList<String>();
List<Integer> list2 = new ArrayList<Integer>();
list.addAll(list1);
list.addAll(list2);
```

2 泛型的好处

- 将运行期遇到的问题转移到了编译期；
- 泛型不可能去除所有类型转换，但可以减少了类型转换操作；
- 在循环遍历集合类时，方便了很多；
- 一定程度上提高了安全性。

3 自定义泛型类的语法

3.1 自定义泛型类的语法

public class 类型<一到多个泛型变量的声明> {...}

例如：

public class A<T> {...}

A 类中有一个泛型变量的声明（或称之为定义）。这就相当于创建了一个变量一样，然后就可以在类内使用它了。

3.2 泛型类内使用泛型变量

声明的泛型变量可以在类内使用，例如创建实例变量时使用泛型变量指定类型，可以在实例方法中指定参数类型或返回值类型，**但不能在 static 变量或 static 方法中使用泛型变量**。

```
public class A<T> {
    private T bean;
    public T getBean() {
        return bean;
    }
    public void setBean(T bean) {
        this.bean = bean;
    }
}
```

4 泛型方法定义

不只可以创建泛型类，还可以创建泛型方法。可能你会有个误区，下面的方法不是泛型方法：

```
class A<T> {
    public void fun(T t) {...}
}
```

上面 fun()方法是泛型类中的方法，它不是泛型方法。

泛型方法是可以自己创建泛型变量，泛型方法中创建的泛型变量只能在本方法内使用。泛型方法可以是实例方法，也可以是静态方法。

```
public <T> T get(T[] ts, int index) {
   return ts[index];
}
```

我们必须要区别开什么是创建泛型变量，什么是使用泛型变量。其中<T>是创建泛型变量，它必须在返回值前面给出。

泛型方法中的泛型变量只有两个可以使用的点：返回值和参数，而且所有有意义的泛型方法中都会在返回值和参数两个位置上使用泛型变量，但语法上没有强制的要求。

调用泛型方法，通常无需为泛型变量直接赋值，而是通过传递的参数类型间接为泛型变量赋值，例如：

```
String[] strs = {“hello”, “world”};
String s = get(strs, 0);
```

因为给 T[] ts 参数赋值为 strs，而 strs 的变量为 String[]，所以等同与给 get 方法的泛型变量赋值为 String 类型。所以返回值类型 T 为 String 类型。

枚举

1 什么是枚举类型

我们学习过单例模式，即一个类只有一个实例。而枚举其实就是多例，一个类有多个实例，但实例的个数不是无穷的，是有限个数的。例如 word 文档的对齐方式有几种：左对齐、居中对齐、右对齐。开车的方向有几种：前、后、左、右！

我们称呼枚举类中实例为枚举项！一般一个枚举类的枚举项的个数不应该太多，如果一个枚举类有 30 个枚举项就太多了！

2 定义枚举类型

定义枚举类型需要使用 enum 关键字，例如：

```
public enum Direction {
```

```
        FRONT, BEHIND, LEFT, RIGHT;
    }
    Direction d = Direction.FRONT;
```

注意，定义枚举类的关键字是 enum，而不是 Enum，所有关键字都是小写的！

其中 FRONT、BEHIND、LEFT、RIGHT 都是枚举项，它们都是本类的实例，本类一共就只有四个实例对象。

在定义枚举项时，多个枚举项之间使用逗号分隔，最后一个枚举项后需要给出分号！但如果枚举类中只有枚举项（没有构造器、方法、实例变量），那么可以省略分号！**建议不要省略分号！**

不能使用 new 来创建枚举类的对象，因为枚举类中的实例就是类中的枚举项，所以在类外只能使用类名.枚举项。

3 枚举与 switch

枚举类型可以在 switch 中使用

```
Direction d = Direction.FRONT;
switch(d) {
case FRONT: System.out.println("前面");break;
case BEHIND:System.out.println("后面");break;
case LEFT:  System.out.println("左面");break;
case RIGHT: System.out.println("右面");break;
    default:System.out.println("错误的方向");
}
Direction d1 = d;
System.out.println(d1);
```

注意，在 switch 中，不能使用枚举类名称，例如：“case Direction.FRONT：”这是错误的，因为编译器会根据 switch 中 d 的类型来判定每个枚举类型，在 case 中必须直接给出与 d 相同类型的枚举选项，而不能再有类型。

4 所有枚举类都是 Enum 的子类

所有枚举类都默认是 Enum 类的子类，无需我们使用 extends 来继承。这说明 Enum 中的方法所有枚举类都拥有。

- int compareTo(E e)：比较两个枚举常量谁大谁小，其实比较的就是枚举常量在枚举类中声明的顺序，例如 FRONT 的下标为 0，BEHIND 下标为 1，那么 FRONT 小于 BEHIND；
- boolean equals(Object o)：比较两个枚举常量是否相等；
- int hashCode()：返回枚举常量的 hashCode；
- String name()：返回枚举常量的名字；
- int ordinal()：返回枚举常量在枚举类中声明的序号，第一个枚举常量序号为 0；
- String toString()：把枚举常量转换成字符串；
- static T valueOf(Class enumType, String name)：把字符串转换成枚举常量。

5 枚举类的构造器

枚举类也可以有构造器，**构造器默认都是 private 修饰，而且只能是 private**。因为枚举类的实例不能让外界来创建！

```
enum Direction {
    FRONT, BEHIND, LEFT, RIGHT;

    Direction() {
        System.out.println("hello");
    }
}
```

其实创建枚举项就等同于调用本类的无参构造器，所以 FRONT、BEHIND、LEFT、RIGHT 四个枚举项等同于调用了四次无参构造器，所以你会看到四个 hello 输出。

6 枚举类可以有成员

其实枚举类和正常的类一样，可以有实例变量，实例方法，静态方法等等，只不过它的实例个数是有限的，不能再创建实例而已。

```
enum Direction {
    FRONT("front"), BEHIND("behind"), LEFT("left"), RIGHT("right");

    private String name;

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

    public String getName() {
        return name;
    }
}
```

```
        Direction d = Direction.FRONT;
        System.out.println(d.getName());
```

因为 Direction 类只有唯一的构造器，并且是有参的构造器，所以在创建枚举项时，必须为构造器赋值：FRONT(“front”)，其中”front”就是传递给构造器的参数。你不要鄙视这种语法，你应该做的是接受这种语法！

Direction 类中还有一个实例域：String name，我们在构造器中为其赋值，而且本类还提供了 getName()这个实例方法，它会返回 name 的值。

7 枚举类中还可以有抽象方法（了解）

还可以在枚举类中给出抽象方法，然后在创建每个枚举项时使用“特殊”的语法来重复抽象方法。所谓“特殊”语法就是匿名内部类！也就是说每个枚举项都是一个匿名类的子类对象！

通常 fun()方法应该定义为抽象的方法，因为每个枚举常量都会去重写它。
你无法把 Direction 声明为抽象类，但需要声明 fun()方法为抽象方法。

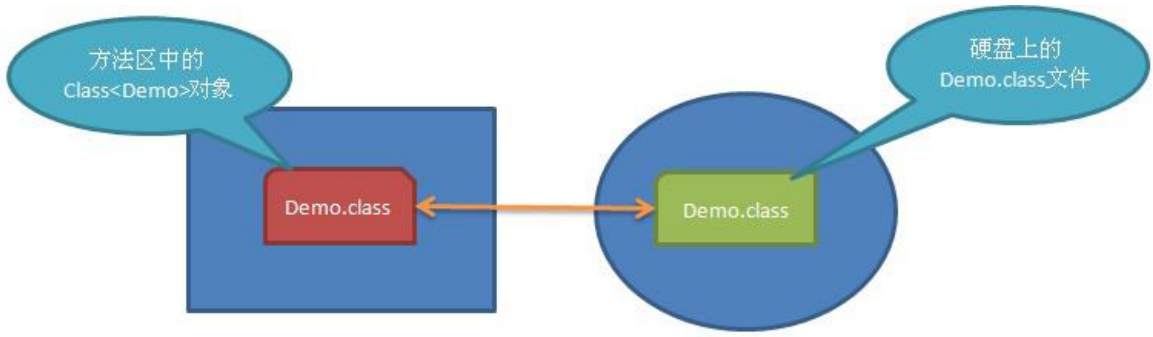
```
enum Direction {
    FRONT() {
        public void fun() {
            System.out.println("FROND: 重写了fun()方法");
        }
    },
    BEHIND() {
        public void fun() {
            System.out.println("BEHIND: 重写了fun()方法");
        }
    },
    LEFT() {
        public void fun() {
            System.out.println("LEFT: 重写了fun()方法");
        }
    },
    RIGHT() {
        public void fun() {
            System.out.println("RIGHT: 重写了fun()方法");
        }
    };

    public abstract void fun();
}
```

反射概述

1 反射的概述

1.1 什么是反射

每个.class 文件被加载到内存后都是一个 Class 类的对象！例如 Demo.class 加载到内存后它是 Class<Demo>类型的一个对象。



反射就是通过 Class 对象获取类型相关的信息。一个 Class 对象就表示一个.class 文件，可以通过 Class 对象获取这个类的构造器、方法，以及成员变量等。

反射是 Java 的高级特性，在框架中大量被使用！我们必须要了解反射，不然无法学好 JavaWeb 相关的知识！

1.2 反射相关类

与反射相关的类：

- Class：表示类；
- Field：表示成员变量；
- Method：表示方法；
- Constructor：表示构造器。

2 Class 类

2.1 获取 Class 类

获取 Class 类的三种基本方式：

- 通过类名称.class，对基本类型也支持；
  - Class c = int.class；
  - Class c = int[].class；
  - Class c = Demo.class
- 通过对象.getClass()方法
  - Class c = obj.getClass();
- Class.forName()通过类名称加载类，这种方法只要有类名称就可以得到 Class；

- Class c = Class.forName("cn.itcast.Demo");

2.2 Class 类的常用方法

- String getName()：获取类名称，包含包名；
- String getSimpleName()：获取类名称，不包含包名；
- Class getSupperClass()：获取父类的 Class，例如：new Integer(100).getClass().getSupperClass()返回的是 Class<Number>！但 new Object().getSupperClass()返回的是 null，因为 Object 没有父类；
- T newInstance()：使用本类无参构造器来创建本类对象；
- boolean isArray()：是否为数组类型；
- boolean isAnnotation()：是否为注解类型；
- boolean isAnnotationPresent(Class annotationClass)：当前类是否被 annotationClass 注解了；
- boolean isEnum()：是否为枚举类型；
- boolean isInterface()：是否为接口类型；
- boolean isPrimitive()：是否为基本类型；
- boolean isSynthetic()：是否为引用类型；

2.3 通过反射创建对象

```
public class Demo1 {
    @Test
    public void fun1() throws Exception {
        String className = "cn.itcast.User";
        Class clazz = Class.forName(className);
        User user = (User)clazz.newInstance();
        System.out.println(user);
    }
}

class User {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
```

```
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

```
User [username=null, password=null]
```

3 Constructor

Constructor 表示一个类的构造器。即构造器的反射对象！

3.1 获取 Constructor 对象

获取 Construcator 对象需要使用 Class 对象，下面 API 来自 Class 类：

- Constructor getConstructor(Class... parameterTypes)：通过指定的参数类型获取公有构造器反射对象；
- Constructor[] getConstructors()：获取所有公有构造器对象；
- Constructor getDeclaredConstructor(Class... parameterTypes)：通过指定参数类型获取构造器反射对象。可以是私有构造器对象；
- Constructor[] getDeclaredConstructors()：获取所有构造器对象。包含私有构造器；

3.2 Construcator 类常用方法

- String getName()：获取构造器名；
- Class getDeclaringClass()：获取构造器所属的类型；
- Class[] getParameterTypes()：获取构造器的所有参数的类型；
- Class[] getExceptionTypes()：获取构造器上声明的所有异常类型；
- T newInstance(Object... initargs)：通过构造器反射对象调用构造器。

3.3 练习：通过 Construcator 创建对象

User.java

```
public class User {
    private String username;
    private String password;
```

```
    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

Demo1.java

```
public class Demo1 {
    @Test
    public void fun1() throws Exception {
        String className = "cn.itcast.User";
        Class clazz = Class.forName(className);
        Constructor c = clazz.getConstructor(String.class, String.class);
        User user = (User)c.newInstance("zhangSan", "123");
        System.out.println(user);
    }
}
```

4 Method

Method 表示方法的反射对象

4.1 获取 Method

获取 Method 需要通过 Class 对象，下面是 Class 类的 API：

- Method getMethod(String name, Class... parameterTypes)：通过方法名和方法参数类型获取方法反射对象，包含父类中声明的公有方法，但不包含所有私有方法；
- Method[] getMethods()：获取所有公有方法，包含父类中的公有方法，但不包含任何私有方法；
- Method getDeclaredMethod(String name, Class... parameterTypes)：通过方法名和方法参数类型获取本类中声明的方法的反射对象，包含本类中的私有方法，但不包含父类中的任何方法；
- Method[] getDeclaredMethods()：获取本类中所有方法，包含本类中的私有方法，但不包含父类中的任何方法。

4.2 Method 常用方法

- String getName()：获取方法名；
- Class getDeclaringClass()：获取方法所属的类型；
- Class[] getParameterTypes()：获取方法的所有参数的类型；
- Class[] getExceptionTypes()：获取方法上声明的所有异常类型；
- Class getReturnType()：获取方法的返回值类型；
- Object invode(Object obj, Object... args)：通过方法反射对象调用方法，如果当前方法是实例方法，那么当前对象就是 obj，如果当前方法是 static 方法，那么可以给 obj 传递 null。args 表示是方法的参数；

4.3 练习：通过 Method 调用方法

```
public class Demo1 {
    @Test
    public void fun1() throws Exception {
        String className = "cn.itcast.User";
        Class clazz = Class.forName(className);
        Constructor c = clazz.getConstructor(String.class, String.class);
        User user = (User)c.newInstance("zhangSan", "123");

        Method method = clazz.getMethod("toString");
        String result = (String)method.invoke(user);
        System.out.println(result);
    }
```

```
}
```

5 Field

Field 表示类的成员变量，可以是实例变量，也可以是静态变量。

5.1 获取 Field 对象

获取 Field 对象需要使用 Class 对象，下面是 Class 类的 API：

- Field getField(String name)：通过名字获取公有成员变量的反射对象，包含父类中声明的公有成员变量；
- Field[] getFields()：获取所有公有成员变量反射对象，包含父类中声明的公有成员变量；
- Field getDeclaredField(String name)：通过名字获取本类中某个成员变量，包含本类的 private 成员变量，但父类中声明的任何成员变量都不包含；
- Field[] getDeclaredFields()：获取本类中声明的所有成员变量，包含 private 成员变量，但不包含父类中声明的任何成员变量；

5.2 Field 类的常用方法

- String getName()：获取成员变量名；
- Class getDeclaringClass()：获取成员变量的类型；
- Class getType()：获取当前成员变量的类型；
- Object get(Object obj)：获取 obj 对象的成员变量的值；
- void set(Object obj, Object value)：设置 obj 对象的成员变量值为 value；

5.3 练习：通过 Field 读写成员

User.java

```
public class User {
    public String username;
    public String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
```

```
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

Demo1.java

```
public class Demo1 {
    @Test
    public void fun1() throws Exception {
        String className = "cn.itcast.User";
        Class clazz = Class.forName(className);
        User user = new User("zhangSan", "123");

        Field field1 = clazz.getField("username");
        Field field2 = clazz.getField("password");

        String username = (String)field1.get(user);
        String password = (String)field2.get(user);

        System.out.println(username + ", " + password);

        field1.set(user, "liSi");
        field2.set(user, "456");

        System.out.println(user);
    }
}
```

6 AccessibleObject

AccessibleObject 类是 Constructor、Method、Field 三个类的父类。AccessibleObject 最为重要的方法如下：

- boolean isAccessible()：判断当前成员是否可访问；
- void setAccessible(boolean flag)：设置当前成员是否可访问。

当 Constructor、Method、Field 为私有时，如果我们想反射操作，那么就必须先调用反射对象的 setAccessible(true)方法，然后才能操作。

User.java

```
public class User {
    private String username;
    private String password;

    public User() {
    }

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
```

```
}
```

注意，User 类的 username 和 password 成员变量为 private 的，这时再通过 Field 来反射操作这两个成员变量就必须先通过 setAccessible(true)设置后才行。

```
public class Demo1 {
    @Test
    public void fun1() throws Exception {
        String className = "cn.itcast.User";
        Class clazz = Class.forName(className);
        User user = new User("zhangSan", "123");

        Field field1 = clazz.getDeclaredField("username");
        Field field2 = clazz.getDeclaredField("password");

        field1.setAccessible(true);
        field2.setAccessible(true);

        String username = (String)field1.get(user);
        String password = (String)field2.get(user);

        System.out.println(username + ", " + password);

        field1.set(user, "liSi");
        field2.set(user, "456");

        System.out.println(user);
    }
}
```

day02-XML

XML

1 XML 的概述

1.1 什么是 XML

XML 全称为 Extensible Markup Language，意思是可扩展的标记语言。XML 语法上和 HTML 比较相似，但 HTML 中的元素是固定的，而 XML 的标签是可以由用户自定义的。

W3C 在 1998 年 2 月发布 1.0 版本，2004 年 2 月又发布 1.1 版本，但因为 1.1 版本不能向下兼容 1.0 版本，所以 1.1 没有人用。同时，在 2004 年 2 月 W3C 又发布了 1.0 版本的第三版。我们要学习的还是 1.0 版本!!!

1.2 XML 的应用场景

保存关系型数据：

```
<student number="1001">
    <name>zhangSan</name>
    <age>23</age>
    <sex>male</sex>
    <teacher name=”liSi”>
      <wife id=”xxx”><name>xxx</name></wife>
    </teacher>
</student>
```

配置文件：

```
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>cn.itcast.servlet.MyServlet</servlet-class>
</servlet>
```

2 XML 语法

来看一个 XML 文档

students.xml

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<students>
    <student number="1001">
        <name>zhangSan</name>
        <age>23</age>
        <sex>male</sex>
    </student>
    <student number="1002">
        <name>liSi</name>
        <age>32</age>
        <sex>female</sex>
    </student>
    <student number="1003">
        <name>wangWu</name>
        <age>55</age>
        <sex>male</sex>
    </student>
</students>
```

2.1 XML 文档声明

- 文档声明必须为<?xml 开头，以?>结束；
- 文档声明必须从文档的 0 行 0 列位置开始；
- 文档声明只有三个属性：
  - versioin：指定 XML 文档版本。必须属性，因为我们不会选择 1.1，只会选择 1.0；
  - encoding：指定当前文档的编码。可选属性，默认值是 utf-8；
  - standalone：指定文档独立性。可选属性，默认值为 yes，表示当前文档是独立文档。如果为 no 表示当前文档不是独立的文档，会依赖外部文件。

2.2 元素

元素是 XML 文档中最重要的组成部分：

- 普通元素的结构：开始标签、元素体、结束标签，例如：<hello>大家好</hello>；
- 元素体：元素体可以是元素，也可以是文本，例如：<a>你好，其中元素的元素体是<a>元素，而<a>元素的元素体是文本；
- 空元素：空元素只有开始标签，而没有结束标签，例如：<c/>，但元素必须自己闭合。

3 属性

```
<student number="1001">
    <name>zhangSan</name>
    <age>23</age>
```

```
    <sex>male</sex>
</student>
```

- 属性是元素的一部分，它必须出现在元素的开始标签中；
- 属性的定义格式：属性名=属性值，其中属性值必须使用单引或双引；
- 一个元素可以有 0~N 个属性，但一个元素中不能出现同名属性；

4 注释

XML 的注释与 HTML 相同，即以“<!--”开始，以“-->”结束。注释内容会被 XML 解析器忽略！

5 转义字符和 CDATA 段

5.1 转义字符

XML 中的转义字符与 HTML 一样。

因为很多符号已经被 XML 文档结构所使用，所以在元素体或属性值中想使用这些符号就必须使用转义字符，例如：“<”、“>”、“‘”、“””、“&”。

字符	字符引用（十进制代码）	字符引用（十六进制代码）	预定义实体引用
<	&#60;	&#x3c;	&lt;
>	&#62;	&#x3e;	&gt;
"	&#34;	&#x22;	&quot;
'	&#39;	&#x27;	&apos;
&	&#38;	&#x26;	&amp;

例如：<a><hello>，<a>元素内部会被解释为：<hello>！

5.2 CDATA 段

当大量的转义字符出现在 xml 文档中时，会使 xml 文档的可读性大幅度降低。这时如果使用 CDATA 段就会好一些。

在 CDATA 段中出现的“<”、“>”、“”“、“‘”、“&”，都无需使用转义字符。这可以提高 xml 文档的可读性。

```
<a><![CDATA[<a>]]></a>
```

在 CDATA 段中不能包含“]]>”，即 CDATA 段的结束定界符。

6 处理指令

处理指令，简称 PI（Processing instruction）。处理指令用来指挥解析器如何解析 XML 文档内容。

例如，在 XML 文档中可以使用 xml-stylesheet 指令，通知 XML 解析器，应用 css 文件显示 xml 文档内容。

```
<?xml-stylesheet type="text/css" href="a.css"?>
```

处理指令以“<?”开头，以“?>”结束，这一点与 xml 文档声明相同。

```
gj1 {font-size: 200px; color: red;}
gj2 {font-size: 100px; color: green;}
gj3 {font-size: 10px;}
gj4 {font-size: 50px; color: blue;}
```

```
<?xml version="1.0" encoding="gbk"?>
<?xml-stylesheet type="text/css" href="a.css" ?>
<gjm>
  <gj1>中国</gj1>
  <gj2>美国</gj2>
  <gj3>日本</gj3>
  <gj4>英国</gj4>
</gjm>
```

7 格式良好的 XML 文档

格式良好的 XML 就是格式正确的 XML 文档，只有 XML 的格式是良好的，XML 解释器才能解释它。下面是对格式良好 XML 文档的要求：

- 必须要有 XML 文档声明；
- 必须且仅能有一个根元素；
- 元素和属性的命名必须遵循 XML 要求：
  - XML 命名区分大小写，例如<a>和<A>是两上不同的元素；
  - 名称中可以包含：字母、数字、下划线、减号，但不能以数字、减号开头；
  - 不能以 xml 开头，无论是大写还是小写都不可以，例如<xml>、<Xml>、<XML>都是错误的；
  - 不能包含空格，例如<ab cd>是错误的。
- 元素之间必须合理包含，例如：<a>xxx是合理的，而<a>xxx就是错误的包含。

DTD

1 DTD 概述

1.1 什么是 DTD

DTD（Document Type Definition），文档类型定义，用来约束 XML 文档。或者可以把 DTD 理解为创建 XML 文档的结构！例如可以用 DTD 要求 XML 文档的根元素名为<students>，<students>中可以有 1~N 个<student>，<student>子元素为<name>、<age>和<sex>，<student>元素还有 number 属性。

DTD 不是 XML 文档，它是 XML 文档的约束文件！就像法律与人一样！

展示 DTD 文档

```
<!ELEMENT students (student+)>
<!ELEMENT student (name,age,sex)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT sex (#PCDATA)>
```

1.2 DTD 分类

- 内部 DTD：在 XML 文档内部嵌入 DTD，只对当前 XML 文档有效；
- 外部 DTD：独立的 DTD 文件，扩展名为.dtd；
  - 本地 DTD：DTD 文件在本地，不在网络上。自己项目，或本公司内部使用的；
  - 公共 DTD：DTD 文件在网络上，不在本地。都是大公司或组织发布的，共大家使用！

1.3 内部 DTD

内部 DTD：

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
<!DOCTYPE students [
<!ELEMENT students (student+)>
<!ELEMENT student (name, age, sex)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT sex (#PCDATA)>
]>
<students>
    <student>
        <name>zhangSan</name>
        <age>23</age>
        <sex>male</sex>
    </student>
</students>
```

- 位置：内部 DTD 在文档声明下面，在根元素上面；

- 语法格式：放到“<!DOCTYPE 根元素名称 [”和“]>”之间；
- 只对当前 XML 文档有效；

1.4 SYSTEM DTD

本地 DTD

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE students SYSTEM "students.dtd">
<students>
    <student>
        <name>zhangSan</name>
        <age>23</age>
        <sex>male</sex>
    </student>
</students>
```

students.dtd

```
<!ELEMENT students (student+)>
<!ELEMENT student (name, age, sex)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT sex (#PCDATA)>
```

- 位置：本地硬盘上；
- 语法格式：直接定义元素或属性即可；
- 本地所有 XML 文档都可以引用这个 dtd 文件；

1.5 公共 DTD

公共 DTD

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE students PUBLIC "-//qdmmy6//DTD ST 1.0//ZH" "http://www.qdmmy6.com/xml/dtds/st.dtd">
<students>
    <student>
        <name>zhangSan</name>
        <age>23</age>
        <sex>male</sex>
    </student>
</students>
```

公共 DTD 是说，DTD 由某个公司或权威组织发布的，供大家使用的 DTD。其格式如下：

<!DOCTYPE 根元素 PUBLIC "DTD 名称" "DTD 网址">

当然你需要知道要使用的 DTD 的名称和网址。如果某个机构公布了 DTD，那么一定也会公布 DTD 的名称和网址。

2 DTD 语法之定义元素

2.1 定义元素语法

定义元素语法：<!ELEMENT 元素名 元素描述>

- <!ELEMENT name (#PCDATA)>，定义名为 name 的元素，内容为文本类型。
- <!ELEMENT student (name,age,sex)>，定义名为 student 元素，内容依次为 name、age、sex 元素；
- <!ELEMENT student ANY>，定义名为 student 元素，内容任意；
- <!ELEMENT student EMPTY>，定义名为 student 元素，不能有内容，即空元素，注意空元素是可以有属性的。

2.2 子元素出现次数

可以使用*、+、?来指定子元素出现的次数

- *：可以出现 0~N 次；
- +：可以出现 1~N 次；
- ?：可以出现 0~1 次。

例如：<!ELEMENT student(name,age?,hobby*,grade+) >，定义 student 元素，第一子元素为 name，必须且仅能出现一次，age 是可有可无的，hobby 可以出现 0~N 次，grade 可以出现 1~N 次。

2.3 枚举类型子元素

<!ELEMENT student (name | age | sex)>，表示 student 子元素为 name、age、sex 其中之一，必须仅且能选择其一。

2.4 练习

- 根元素为 students，可以包含 1~N 个 student 元素；
- student 元素依次包含：name、age、sex 元素；
- name、age、sex 元素的内容类型为文本内容；
- 要求 SYSTEM 外部 DTD。

3 DTD 语法之定义属性

3.1 定义属性的语法

<!ATTLIST 元素名 属性名 属性类型 设置说明>

例如：<!ATTLIST student number CDATA #REQUIRED>，给 student 元素定义属性 number，类型为文本，这个默认是必须的。

3.2 属性设置说明

- #REQUIRED：说明属性是必须的；
- #IMPLIED：说明属性是可选的；
- 默认值：在不给出属性值时，使用默认值。

3.3 属性的类型

- CDATA：文本类型；
- Enumerated：枚举类型；
- ID：ID 类型，ID 类型的属性用来标识元素的唯一性，即元素的 ID 属性值不能与其他元素的 ID 属性值相同；
- IDREF：ID 引用类型，用来指定另一个元素，与另一个元素建立关联关系，IDREF 类型的属性值必须是另一个元素的 ID。

```
<!ELEMENT students (student+) >
<!ELEMENT student EMPTY>
<!ATTLIST student number ID #REQUIRED>
<!ATTLIST student name CDATA #REQUIRED>
<!ATTLIST student sex (male | female) "male" >
<!ATTLIST student friend IDREF #IMPLIED>
```

```
<?xml version="1.0" ?>
<!DOCTYPE students SYSTEM "students.dtd">

<students>
  <student number="itcast_001" name="zhangSan"/>
  <student number="itcast_002" name="liSi" sex="male"/>
  <student number="itcast_003" name="wangWu" sex="female" friend="itcast_002"/>
</students>
```

4 DTD 语法之定义实体（了解）

4.1 什么是实体

有时在 XML 中会出现很多相同的内容，例如“北京传智播客教育科技有限公司”，这个名称太长了，我们希望把这个值与一个“符号”绑定，然后在需要这个名称时使用它绑定的“符号”即可。这个符号就是实体了。例如：“&传智;”!

其中“传智”是实体名，而“北京传智播客教育科技有限公司”是实体值，XML 被解析时，所有实体会被替换成实体名。

4.2 实体分类

实体分为两种：一般实体和参数实体。

- 一般实体：在 XML 文档中使用；
- 参数实体：在 DTD 使用。

4.3 一般实体

- 定义一般实体：<!ENTITY 实体名 “实体值”>，例如：<!ENTITY 大美女 “白冰”>；
- 一般实体引用：&实体名;，例如<xxx>&大美女;</xxx>。

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE students SYSTEM "a.dtd">

<students>
    <student number="itcast_001" name="zhangSan"/>
    <student number="itcast_002" name="liSi" sex="male"/>
    <student number="itcast_003" name="wangWu" sex="female" friend="itcast_002"/>
    <student number="itcast_004" name="&itcast;"/>
</students>
```

```
<!ELEMENT students (student+) >
<!ELEMENT student EMPTY>
<!ATTLIST student number ID #REQUIRED>
<!ATTLIST student name CDATA #REQUIRED>
<!ATTLIST student sex (male | female) "male">
<!ATTLIST student friend IDREF #IMPLIED>
<!ENTITY itcast "北京传智播客教育科技有限公司">
```

4.4 参数实体

- 定义参数实体：<!ENTITY % 实体名 "实体值">，“%”与实体名之间的空格是必须的；
  - 例如：<!ENTITY % friend "student friend IDREF #IMPLIED">
- 参数实体引用：%实体名;；
  - 例如：<!ATTLIST %friend;>

参数实体是在 DTD 内部使用，而不是在 XML 中使用。
在内部 DTD 中使用参数实体会有诸多限制。

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<!DOCTYPE students SYSTEM "a.dtd">

<students>
    <student number="itcast_001" name="zhangSan"/>
```

```
    <student number="itcast_002" name="liSi" sex="male"/>
    <student number="itcast_003" name="wangWu" sex="female" friend="itcast_002"/>
    <student number="itcast_004" name="&itcast;"/>
</students>
```

```
<!ELEMENT students (student+) >
<!ELEMENT student EMPTY>
<!ATTLIST student number ID #REQUIRED>
<!ATTLIST student name CDATA #REQUIRED>
<!ATTLIST student sex (male | female) "male">
<!ENTITY % friend "<!ATTLIST student friend IDREF #IMPLIED>">
%friend;
<!ENTITY itcast "北京传智播客教育科技有限公司">
```

Schema

1 Schema 概述

1.1 什么是 Schema

- Schema 是新的 XML 文档约束；
- Schema 要比 DTD 强大很多；
- Schema 本身也是 XML 文档，但 Schema 文档的扩展名为 xsd，而不是 xml。

1.2 Schema 简介

本课程中不对 Schema 深入探讨，我们只对 Schema 有个了解即可。

students.xsd

```
<?xml version="1.0"?>
<xsd:schema xmlns="http://www.itcast.cn/xml"
                xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                                        targetNamespace="http://www.itcast.cn/xml"
elementFormDefault="qualified">
        <xsd:element name="students" type="studentsType"/>
        <xsd:complexType name="studentsType">
                <xsd:sequence>
                        <xsd:element name="student" type="studentType" minOccurs="0"
maxOccurs="unbounded"/>
                </xsd:sequence>
        </xsd:complexType>
```

```
        <xsd:complexType name="studentType">
                <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="age" type="ageType" />
                        <xsd:element name="sex" type="sexType" />
                </xsd:sequence>
                <xsd:attribute name="number" type="numberType"  use="required"/>
        </xsd:complexType>
        <xsd:simpleType name="sexType">
                <xsd:restriction base="xsd:string">
                        <xsd:enumeration value="male"/>
                        <xsd:enumeration value="female"/>
                </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType name="ageType">
                <xsd:restriction base="xsd:integer">
                        <xsd:minInclusive value="0"/>
                        <xsd:maxInclusive value="100"/>
                </xsd:restriction>
        </xsd:simpleType>
        <xsd:simpleType name="numberType">
                <xsd:restriction base="xsd:string">
                        <xsd:pattern value="ITCAST_\d{4}"/>
                </xsd:restriction>
        </xsd:simpleType>
</xsd:schema>
```

students.xml

```
<?xml version="1.0"?>

<students xmlns="http://www.itcast.cn/xml"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.itcast.cn/xml students.xsd" >
        <student number="ITCAST_1001">
                <name>zhangSan</name>
                <age>20</age>
                <sex>male</sex>
        </student>
        <student number="ITCAST_1002">
                <name>liSi</name>
                <age>25</age>
                <sex>female</sex>
        </student>
```

```
</students>
```

2 Schema 名称空间

2.1 什么是名称空间

如果一个 XML 文档中使用多个 Schema 文件，而这些 Schema 文件中定义了相同名称的元素时就会出现名字冲突。这就像一个 Java 文件中使用了 import java.util.*和 import java.sql.*时，在使用 Date 类时，那么就不明确 Date 是哪个包下的 Date 了。

总之名称空间就是用来处理元素和属性的名称冲突问题，与 Java 中的包是同一用途。如果每个元素和属性都有自己的名称空间，那么就不会出现名字冲突问题，就像是每个类都有自己所在的包一样，那么类名就不会出现冲突。

2.2 目标名称空间

在 XSD 文件中为定义的元素指定名称，即指定目标名称空间。这需要给<xsd:schema>元素添加 targetNamespace 属性。

- <xsd:schema targetNamespace="http://www.itcast.cn/xml">

名称空间可以是任意字符串，但通常我们会使用公司的域名作为名称空间，这与 Java 中的包名使用域名的倒序是一样的！千万不要以为这个域名是真实的，它可以是不存在的域名。

如果每个公司发布的 Schema 都随意指定名称空间，如 a、b 之类的，那么很可能会出现名称空间的名字冲突，所以还是使用域名比较安全，因为域名是唯一的。

当使用了 targetNamespace 指定目标名称空间后，那么当前 XSD 文件中定义的元素和属性就在这个名称空间之中了。

2.3 XML 指定 XSD 文件

在 XML 文件中需要指定 XSD 约束文件，这需要使用在根元素中使用 schemaLocation 属性来指定 XSD 文件的路径，以及目标名称空间。格式为：schemaLocation=”目标名称空间 XSD 文件路径”

- <students schemaLocation="http://www.itcast.cn/xml students.xsd">

schemaLocation 是用来指定 XSD 文件的路径，也就是说为当前 XML 文档指定约束文件。但它不只要指定 XSD 文件的位置，还要指定 XSD 文件的目标名称空间。

其中 http://www.itcast.cn/xml 为目标名称空间，students.xsd 为 XSD 文件的位置，它们中间使用空白符（空格或换行）分隔。

也可以指定多个 XSD 文件，格式为：

- schemaLocation=”目标名称空间 1 XSD 文件路径 1 目标名称空间 2 XSD 文件路径 2”

下面是 spring 配置文件的例子，它一共指定两个 XSD 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
                                        http://www.springframework.org/schema/beans
s/spring-beans-3.0.xsd
                                        http://www.springframework.org/schema/aop
                                        http://www.springframework.org/schema/aop/
spring-aop-3.0.xsd">
</beans>
```

下面是 JavaWeb 项目的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                                                    http://java.sun.com/xml/ns/jav
aee/web-app_2_5.xsd">
</web-app>
```

2.4 定义名称空间

现在我们已经知道一个 XML 中可以指定多个 XSD 文件，例如上面 Spring 的配置文件中就指定了多个 XSD 文件，那么如果我在<beans>元素中给出一个子元素<bean>，你知道它是哪个名称空间中的么？

```
<beans
  xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
  <bean></bean>
</beans>
```

所以只是使用 schemaLocation 指定 XSD 是不够的，它只是导入了这个 XSD 及 XSD 的名称空间而已。schemaLocation 的作用就相当于 Java 中导入 Jar 包的作用！最终还是在 Java 文件中使用 import 来指定包名的。

xmlns 是用来指定名称空间前缀的，所谓前缀就是“简称”，例如中华人发共和国简称中国一样，然后我们在每个元素前面加上前缀，就可以处理名字冲突了。

格式为：xmln:前缀=”名称空间”

注意，使用 xmlns 指定的名称空间必须是在 schemaLocation 中存在的名称空间。

```
<beans
  xmlns:b="http://www.springframework.org/schema/beans"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
            http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
  <b:bean></b:bean>
```

```
    <aop:scoped-proxy/>
</beans>
```

2.5 默认名称空间

在一个 XML 文件中，可以指定一个名称空间没有前缀，那么在当前 XML 文档中没有前缀的元素就来自默认名称空间。

```
<beans
    xmlns="http://www.springframework.org/schema/beans"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
    <bean></bean>
    <aop:scoped-proxy/>
</beans>
```

2.6 W3C 的元素和属性

如果我们的 XML 文件中需要使用 W3C 提供的元素和属性，那么可以不在 schemaLocation 属性中指定 XSD 文件的位置，但一定要定义名称空间，例如：

```
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                http://www.springframework.org/schema/aop
                http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
</beans>
```

上面定义了一个名称空间，前缀为 xsi，名称空间为 http://www.w3.org/2001/XMLSchema-instance。这个名称空间无需在 schemaLocation 中不存在这个名称空间。

你可能已经发现了，schemaLocation 这个属性其实是 w3c 定义的属性，与元素一定，属性也需要指定“出处”，xsi:schemaLocation 中的 xsi 就是名称空间前缀。也就是说，上面我们在没有指定 xsi 名称空间时，就直接使用 schemaLocation 是错误的。

day02-XML 解析器

解析器概述

1 什么是解析器

XML 是保存数据的文件，XML 中保存的数据也需要被程序读取然后使用。那么程序使用什么来读取 XML 文件中的数据呢？XML 解析器！例如.properties 文件的解析器是 Properties 类一样！

XML 不只被 Java 语言使用，还被 C++、C#、Javascript 等等语言使用，所以解析 XML 不是一门语言的工作！

2 DOM 和 SAX 介绍

主流的 XML 解析有两种标准：DOM 和 SAX。它们是标准，是思想，不是真正的解析器，它们是跨语言的!!!

- DOM（Document Object Model）：W3C 组织提供的解析 XML 文档的标准接口；
- SAX（Simple API for XML）：社区讨论的产物，是一种事实上的标准。

Apache 的 xerces 组件实现了 DOM 和 SAX，所以在我们在 Java 中解析 XML 需要使用 xerces。所以我们称 xerces 是 DOM、SAX 解析器。

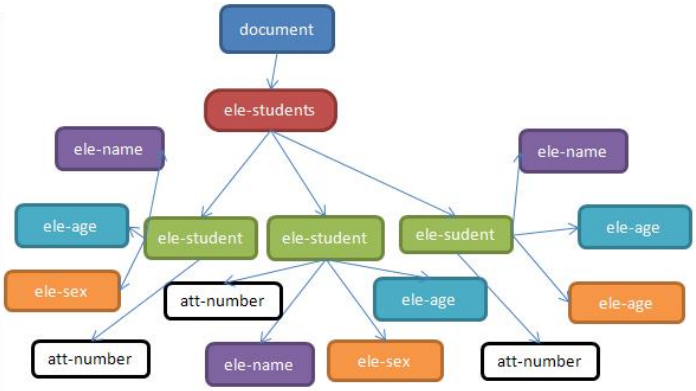
2.1 DOM 解析原理

DOM 要求解析器把整个 XML 文档装载到一个 Document 对象中。即使用 DOM 解析器解析 XML 文档的结果就是一个 Document 对象。

一个 XML 文档解析后对应一个 Document 对象，可以通过 Document 对象获取根元素，然后在通过根元素获取根元素的子元素…，这说明 DOM 解析方式保留了元素之间的结构关系！

- 优点：元素与元素之间的结构关系保留了下来；
- 缺点：如果 XML 文档过大，那么把整个 XML 文档装载进内存，可能会出现内存溢出的现象！

```
<?xml version="1.0" encoding="utf-8" standalone="no" ?>
<students>
 <student number="1001">
  <name>zhangSan</name>
  <age>23</age>
  <sex>male</sex>
 </student>
 <student number="1002">
  <name>liSi</name>
  <age>32</age>
  <sex>female</sex>
 </student>
 <student number="1003">
  <name>wangWu</name>
  <age>55</age>
  <sex>male</sex>
 </student>
</students>
```



2.2 SAX 解析原理

DOM 解析后的结果是一个 Document 对象，而 SAX 解析没有结果！SAX 要求在开始解析之前用户提供一个接口的实现对象，然后把接口实现对象传递给 SAX 解析器，然后在 SAX 解析器的过程中不调用实现对象的方法。

- DOM 是解析时把数据放到了 Document 对象中，然后用户从 Document 中获取需要的数据；
- SAX 要求用户参与到解析过程中来，把想要做的事情写到接口实现对象中，然后 SAX 在解析过程中来调用接口实现对象的方法。

你看过三国么？听说过三气周瑜么？其中二气周瑜你知道么？故事情节大致如下：

倒霉的周瑜让孙权把自己的妹妹嫁给刘备，让刘备来吴国完婚！目的是把刘备软禁在吴国，或者把刘备干掉。如果刘备不同意，那么孙权就有了打刘备，抢荆州的借口了。

孔明识破了这一计，最终让刘备与赵云去吴国。但孔明给了赵云三个锦囊：

- 到了吴国南徐开第一个锦囊；
- 在吴国住到年终开第二个锦囊；
- 回家途中被吴军阻拦开第三个锦囊。

锦囊的内容是什么呢？赵云和刘备性命如何呢？诸葛亮又是如何二气周瑜的呢？这里没有了，自己去买本《三国演义》看吧！

其中赵云就是 SAX 解析器，赵负责去吴国，SAX 负责解析 XML；

其中孔明的三个锦囊就是接口实现中的三个方法；

赵云会在特定时刻打开锦囊，依计而行，SAX 解析器会在解析过程中特定时刻调用接口实现中的某一方法。

取亲之旅 ←→ 被解析的 XML

赵云 ←→ SAX 解析器

锦囊 ←→接口在三个方法（由我们完成）

赵云在取亲之旅中，会在发生特定事件时，执行特定的锦囊。例如在到达吴国南徐时，执行第一个锦囊上的妙计。

SAX 解析器会在解析 XML 文档的过程中，在发生特定事件时，调用接口中特定的方法。例如在 SAX 解析到某个元素的开始标签时，输出元素名称！其中解析到开始标签就是特定的事件，而输出元素名称，就是接口中方法的实现。

接口中方法如下：

```
public void startDocument();
public void endDocument();
public void startElement(String uri, String localName, String qName,
        Attributes atts);
public void endElement(String uri, String localName, String qName)
public void characters(char[] ch, int start, int length);
public void ignorableWhitespace(char[] ch, int start, int length);
public void processingInstruction(String target, String data);
```

接口的实现由我们来完成，然后我们需要把接口实现类对象“交给”SAX 解析器，然后让 SAX 开始解析。SAX 会在特定事件发生时，调用接口中的方法，完成我们交给它的任务。

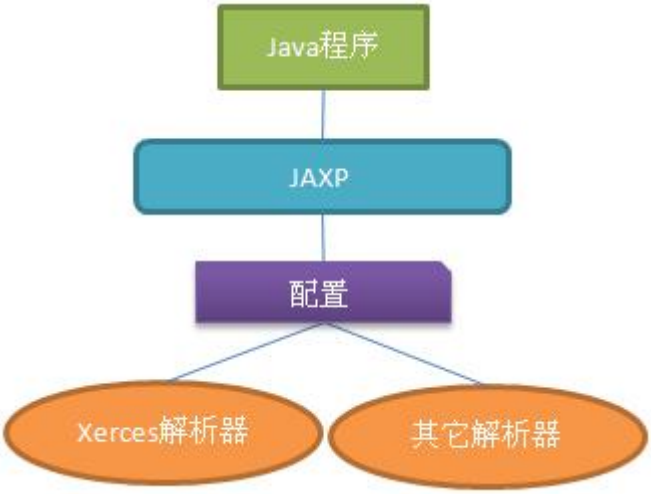
- 优点：适合解析大 XML 文件（内存空间占用小），因为是解析一行处理一行，处理完了就不需要在保留数据了；
- 缺点：因为是解析一行处理一行，解析之后数据就丢失了，所以元素与元素之间的结构关系没有保留下来。

3 JAXP 介绍

我们知道有很多像 xerces 一样的解析器，都对 DOM 和 SAX 提供了实现，那么如果我们在项目中一开始使用了解析 A，然后因为某些原因想更换成解析 B，那么就需要修改项目。

JAXP（Java API for XML Processing）是由 Java 提供的，JAXP 是对所有像 xerces 一样的解析的提供统一接口的 API。

当我们使用 JAXP 完成解析工作时，还需要为 JAXP 指定 xerces 或其他解析器，当需要更换解析器时，无需修改代码，只需要修改配置即可。



JAXP 不是解析器，但使用它可以方便的切换解析器。所以在我们的程序中只会使用 JAXP，而不会直接使用 Xeces。

4 JDOM 和 DOM4j 介绍

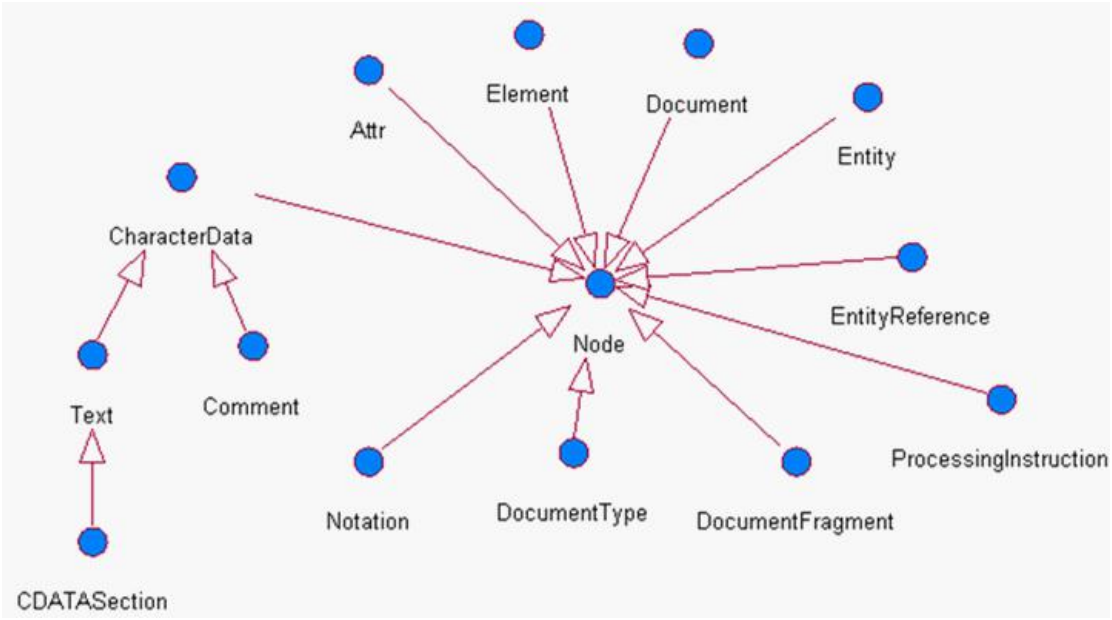
DOM 和 SAX 是跨语言的 XML 解析准备，在 Java 中使用并不方便。而 JDOM 和 DOM4j 是专门为 Java 语言提供的解析工具！使用起来很方便，所以真实开发中使用 JDOM 或 DOM4J 比较多。

又因为 DOM4J 与 JDOM 比较结果为 DOM4j 完胜，所以我们这里只会对 DOM4j 介绍，而不会介绍 JDOM。

DOM 和 SAX 解析

1 DOM 结构模型

DOM 中的核心概念就是节点，在 XML 文档中的元素、属性、文本、处理指令，在 DOM 中都是节点！



2 JAXP 之 DOM 解析

使用 DOM 解析 XML 的目标就是获取到 Document 对象，然后在从 Document 中获取到需要的数据。Document 对象就是 XML 文档在内存中的样子。

1. 获取 Document 三步：

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
```

```
Document document = builder.parse(new File("students.xml"));
```

2. 遍历 Document：

```
    @Test
    public void fun1() throws SAXException, IOException,
ParserConfigurationException {
        DocumentBuilderFactory factory =
DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        Document document = builder.parse(new File("src/students.xml"));

        // 获取根元素
        Element rootEle = document.getDocumentElement();
        // 获取元素的所有子节点
        NodeList nodeList = rootEle.getChildNodes();
        // 循环遍历所有节点
        for(int i = 0; i < nodeList.getLength(); i++) {
            // 获取其中每个节点
            Node node = nodeList.item(i);
            // 判断节点的类型是否为元素
            if(node.getNodeType() == Node.ELEMENT_NODE) {
                // 强转成元素类型
                Element stuEle = (Element) node;
                // 获取元素的名称
                String eleName = stuEle.getNodeName();
                // 获取元素的number属性值
                String number = stuEle.getAttribute("number");
                // 获取名为name的子元素，因为返回值为NodeList，
                // 所以需要使用item(0)方法获取第一个name子元素
                // getTextContent()是获取节点的文本内容
                String name =
stuEle.getElementsByTagName("name").item(0).getTextContent();
                String age =
stuEle.getElementsByTagName("age").item(0).getTextContent();
                String sex =
stuEle.getElementsByTagName("sex").item(0).getTextContent();

                System.out.println(eleName + ":[number=" + number + ", name="
+ name + ", age=" + age + ", sex=" + sex + "]");
            }
        }
    }
```

3 JAXP 之 SAX 解析

使用 SAX 解析 XML 文档需要先给出 DefualtHandler 的子类，重写其中的方法。然后在使用 SAX 开始解析时把 DefaultHandler 子类对象传递给 SAX 解析器。

```
class MyContentHandler extends DefaultHandler {
    public void startDocument() throws SAXException {
        System.out.println("开始解析...");
    }

    public void endDocument() throws SAXException {
        System.out.println("解析结束...");
    }

    public void startElement(String uri, String localName, String qName,
            Attributes atts) throws SAXException {
        System.out.println(qName + "元素解析开始");
    }

    public void endElement(String uri, String localName, String qName)
            throws SAXException {
        System.out.println(qName + "元素解析结束");
    }

    public void characters(char[] ch, int start, int length)
            throws SAXException {
        System.out.print(new String(ch, start, length).trim());
    }
}
```

使用 SAX 解析首先需要获取工厂，再通过工厂获取解析器对象，然后使用解析对象完成解析工作：

```
        SAXParserFactory factory = SAXParserFactory.newInstance();
        SAXParser parser = factory.newSAXParser();
        parser.parse(new File("src/students.xml"), new MyContentHandler());
```

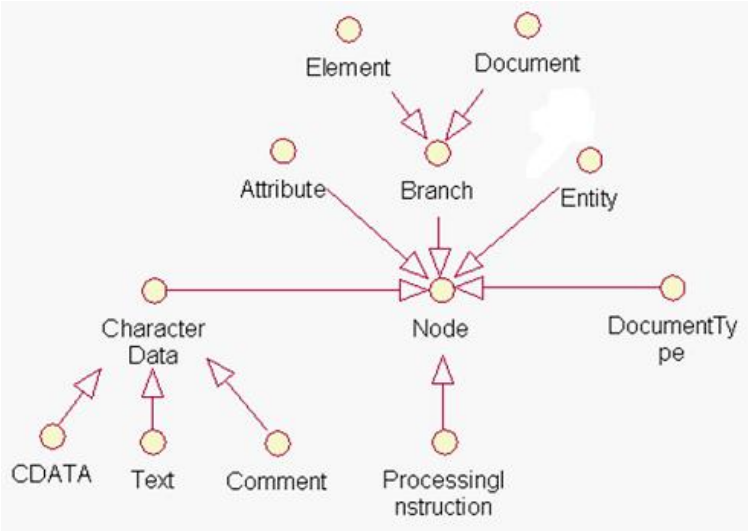
DOM4J 解析

1 DOM4J 概述

DOM4J 是针对 Java 开发人员专门提供的 XML 文档解析规范，它不同与 DOM，但与 DOM 相似。DOM4J 针对 Java 开发人员而设计，所以对于 Java 开发人员来说，使用 DOM4J 要比使用 DOM 更加

方便。

在 DOM4J 中，也有 Node、Document、Element 等接口，结构上与 DOM 中的接口比较相似。但它们是不同的类：



- Node
  - Attribute：表示属性节点；
  - Branch：表示可以包含子元素的节点：
    - Document：表示整个文档；
    - Element：表示元素节点；
  - CharacterData：表示文本节点：
    - Text：表示文本内容；
    - CDATA：表示 CDATA 段内容；
    - Comment：表示注释内容。

我们再次强调，DOM 和 DOM4J 是不同的，DOM 中的 Document 是 org.w3c.Document，而 DOM4J 中的 Document 是 org.dom4j.Document，它们是不同的类，其他 Node、Element 也是一样。

2 读取、保存、创建 Document

使用 dom4j 需要导入：

- dom4j.jar
- jaxen.jar

1. 读取

```
SAXReader reader = new SAXReader();
Document doc = reader.read("src/students.xml");
```

2. 保存 XML 文档

```
// 创建格式化器，使用\t缩进，添加换行
```

```
        OutputFormat format = new OutputFormat("\t", true);
        // 清空数据中原有的换行
        format.setTrimText(true);
        // 创建XML输出流对象
        XMLWriter writer = new XMLWriter(new FileWriter("src/a.xml"), format);
        // 输出Document
        writer.write(doc);
        // 关闭流
        writer.close();
```

3. 创建 Document

```
Document doc = DocumentHelper.createDocument();
```

3 遍历 Document

```
    public void fun1() throws DocumentException {
        SAXReader reader = new SAXReader();
        Document doc = reader.read("src/students.xml");

        // 获取根元素
        Element rootEle = doc.getRootElement();
        // 获取根元素的所有子元素
        List<Element> eleList = rootEle.elements();
        // 遍历元素集合
        for (Element stuEle : eleList) {
            // 获取元素名称
            String eleName = stuEle.getName();

            // 获取元素的number属性值
            String number = stuEle.attributeValue("number");
            // 获取元素的name子元素内容
            String name = stuEle.elementText("name");
            // 获取元素的age子元素内容
            String age = stuEle.elementText("age");
            // 获取元素的sex子元素内容
            String sex = stuEle.elementText("sex");
            System.out.println(eleName + ": [number=" + number + ", name="
                    + name + ", age=" + age + ", sex=" + sex + "]");
        }
    }
```

4 添加 student 元素

```
SAXReader reader = new SAXReader();
Document doc = reader.read("src/students.xml");

// 获取根元素<students>
Element root = doc.getRootElement();
// 为root添加名为student的子元素，并返回这个新添加的子元素
Element stuEle = root.addElement("student");
// 给元素添加属性number，值为123
stuEle.addAttribute("number", "123");
// 添加子元素name，并设置name子元素的文本内容为wangWu
stuEle.addElement("name").setText("wangWu");
stuEle.addElement("age").setText("30");
stuEle.addElement("sex").setText("male");

//////////////////

// 创建格式化器，使用\t缩进，添加换行
OutputFormat format = new OutputFormat("\t", true);
// 清空数据中原有的换行
format.setTrimText(true);
// 创建XML输出流对象
XMLWriter writer = new XMLWriter(new FileWriter("src/a.xml"), format);
// 输出Document
writer.write(doc);
// 关闭流
writer.close();
```

5 查询元素

```
SAXReader reader = new SAXReader();
Document doc = reader.read("src/students.xml");

/*
 * selectSingNode()方法的参数是XPath
 * XPath是在XML文档中查找的一门表达式语言
 * “//”表示查找整个XML文档
 * student表示查找名为student的元素
 * []表示条件
 * @number表示number属性
 * @number='ITCAST_1001'表示条件为number属性等于ITCAST_1001
 *
```

```
         * selectSingNode()方法在查找到多个满足XPath的元素时，只返回第一个。
         */
        Element stuEle1 =
(Element)doc.selectSingleNode("//student[@number='ITCAST_1001']");
        // 把元素转换成字符串
        System.out.println(stuEle1.asXML());

        // 查找name子元素内容为liSi的student元素
        Element stuEle2 = (Element)
doc.selectSingleNode("//student[name='liSi']");
        System.out.println(stuEle2.asXML());
```

6 修改元素

```
        SAXReader reader = new SAXReader();
        Document doc = reader.read("src/students.xml");

        // 查找元素
        Element stuEle =
(Element)doc.selectSingleNode("//student[@number='ITCAST_1001']");
        // 修改student元素的name子元素内容为"张三"
        stuEle.element("name").setText("张三");

        /////////////////

        // 创建格式化器，使用\t缩进，添加换行
        OutputFormat format = new OutputFormat("\t", true);
        // 清空数据中原有的换行
        format.setTrimText(true);
        // 创建XML输出流对象
        XMLWriter writer = new XMLWriter(new FileWriter("src/a.xml"), format);
        // 输出Document
        writer.write(doc);
        // 关闭流
        writer.close();
```

7 删除学生元素

```
        SAXReader reader = new SAXReader();
        Document doc = reader.read("src/students.xml");

        // 查找元素
```

```
        Element stuEle =
(Element)doc.selectSingleNode("//student[@number='ITCAST_1001']");
        // 获取父元素来删除元素
        stuEle.getParent().remove(stuEle);

        //////////////////

        // 创建格式化器，使用\t缩进，添加换行
        OutputFormat format = new OutputFormat("\t", true);
        // 清空数据中原有的换行
        format.setTrimText(true);
        // 创建XML输出流对象
        XMLWriter writer = new XMLWriter(new FileWriter("src/a.xml"), format);
        // 输出Document
        writer.write(doc);
        // 关闭流
        writer.close();
```

8 dom4j API 介绍

Node 方法：

- String asXML()：把当前节点转换成字符串，如果当前 Node 是 Document，那么就会把整个 XML 文档返回；
- String getName()：获取当前节点名字；Document 的名字就是绑定的 XML 文档的路径；Element 的名字就是元素名称；Attribute 的名字就是属性名；
- Document getDocument()：返回当前节点所在的 Document 对象；
- short getNodeType()：获取当前节点的类型；
- String getNodeTypeName()：获取当前节点的类型名称，例如当前节点是 Document 的话，那么该方法返回 Document；
- String getStringValue()：获取当前节点的子孙节点中所有文本内容连接成的字符串；
- String getText()：获取当前节点的文本内容。如果当前节点是 Text 等文本节点，那么本方法返回文本内容；例如当前节点是 Element，那么当前节点的内容不是子元素，而是纯文本内容，那么返回文本内容，否则返回空字符串；
- void setDocument(Document doc)：给当前节点设置文档元素；
- void setParent(Element parent)：给当前节点设置父元素；
- void setText(String text)：给当前节点设置文本内容；

Branch 方法：

- void add(Element e)：添加子元素；
- void add(Node node)：添加子节点；
- void add(Comment comment)：添加注释；

- Element addElement(String eleName)：通过名字添加子元素，返回值为子元素对象；
- void clearContent()：清空所有子内容；
- List content()：获取所有子内容，与获取所有子元素的区别是，<name>liSi</name>元素没有子元素，但有子内容；
- Element elementById(String id)：如果元素有名为“ID”的属性，那么可以使用这个方法来查找；
- int indexOf(Node node)：查找子节点在子节点列表中的下标位置；
- Node node(int index)：通过下标获取子节点；
- int nodeCount()：获取子节点的个数；
- Iterator nodeIterator()：获取子节点列表的迭代器对象；
- boolean remove(Node node)：移除指定子节点；
- boolean remove(Commont commont)：移除指定注释；
- boolean remove(Element e)：移除指定子元素；
- void setContent(List content) ：设置子节点内容；

Document 方法：

- Element getRootElement()：获取根元素；
- void setRootElement()：设置根元素；
- String getXmlEncoding()：获取 XML 文档的编码；
- void setXmlEncoding()：设置 XML 文档的编码；

Element 方法：

- void add(Attribute attr)：添加属性节点；
- void add(CDATA cdata)：添加 CDATA 段节点；
- void add(Text Text)：添加 Text 节点；
- Element addAttribute(String name, String value)：添加属性，返回值为当前元素本身；
- Element addCDATA(String cdata)：添加 CDATA 段节点；
- Element addComment(String comment)：添加属性节点；
- Element addText(String text)：添加 Text 节点；
- void appendAttributes(Element e)：把参数元素 e 的所有属性添加到当前元素中；
- Attribute attribute(int index)：获取指定下标位置上的属性对象；
- Attribute attribute(String name)：通过指定属性名称获取属性对象；
- int attributeCount()：获取属性个数；
- Iterator attributeIterator()：获取当前元素属性集合的迭代器；
- List attributes()：获取当前元素的属性集合；
- String attributeValue(String name)：获取当前元素指定名称的属性值；
- Element createCopy()：clone 当前元素对象，但不会 copy 父元素。也就是说新元素没有父元素，但有子元素；
- Element element(String name)：获取当前元素第一个名称为 name 的子元素；
- Iterator elementIterator()：获取当前元素的子元素集合的迭代器；
- Iterator elementIterator(String name)：获取当前元素中指定名称的子元素集合的迭代器；

- List elements()：获取当前元素子元素集合；
- List elements(String name)：获取当前元素指定名称的子元素集合；
- String elementText(String name)：获取当前元素指定名称的第一个元素文件内容；
- String elementTextTrime(String name)：同上，只是去除了无用空白；
- boolean isTextOnly()：当前元素是否为纯文本内容元素；
- boolean remove(Attribute attr)：移除属性；
- boolean remove(CDATA cdata)：移除 CDATA；
- boolean remove(Text text)：移除 Text。

DocumentHelper 静态方法介绍：

- static Document createDocument()：创建 Dcoument 对象；
- static Element createElement(String name)：创建指定名称的元素对象；
- static Attribute createAttrbute(Element owner, String name, String value)：创建属性对象；
- static Text createText(String text)：创建属性对象；
- static Document parseText(String text)：通过给定的字符串生成 Document 对象；

day03

软件体系结构

1 常见软件体系结构 B/S、C/S

1.1 C/S

- C/S 结构即客户端/服务器（Client/Server），例如 QQ；
- 需要编写服务器端程序，以及客户端程序，例如我们安装的就是 QQ 的客户端程序；
- 缺点：软件更新时需要同时更新客户端和服务器端两端，比较麻烦；
- 优点：安全性比较好。

1.2 B/S

- B/S 结构即浏览器/服务器（Browser/Server）；
- 优点：只需要编写服务器端程序；
- 缺点：安全性较差。

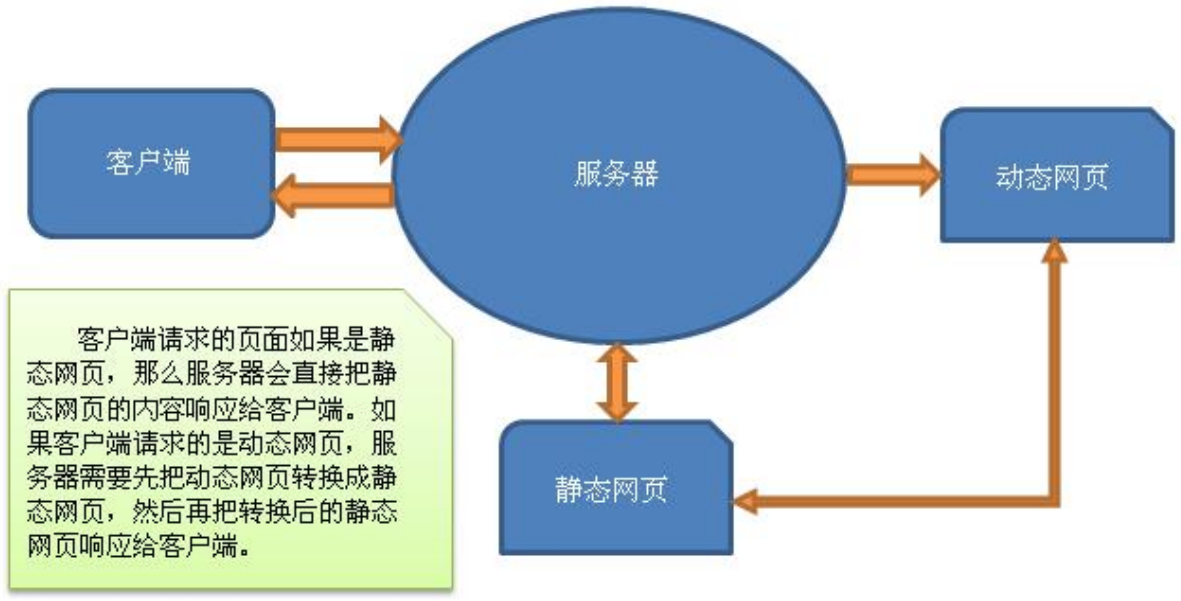
2 WEB 资源

2.1 Web 资源介绍

- html：静态资源；
- JSP/Servlet：动态资源。

当然，除了 JavaWeb 程序，还有其他 Web 程序，例如：ASP、PHP 等。

2.2 静态资源和静态资源区别



2.3 访问 Web 资源

打开浏览器，输入 URL：

- 协议名://域名:端口/路径，例如：http://www.itcast.cn:80/index.html

3 Web 服务器

Web 服务器的作用是接收客户端的请求，给客户端作出响应。

对于 JavaWeb 程序而已，还需要有 JSP/Servlet 容器，JSP/Servlet 容器的基本功能是把动态资源转换成静态资源，当然 JSP/Servlet 容器不只这些功能，我们会在后面一点一点学习。

我们需要使用的是 Web 服务器和 JSP/Servlet 容器，通常这两者会集于一身。下面是对 JavaWeb 服务器：

- Tomcat（Apache）：当前应用最广的 JavaWeb 服务器，；
- JBoss（Redhat 红帽）：支持 JavaEE，应用比较广；
- GlassFish（Orcale）：Oracle 开发 JavaWeb 服务器，应用不是很广；
- Resin（Caucho）：支持 JavaEE，应用越来越广；
- Weblogic（Orcale）：要钱的！支持 JavaEE，适合大型项目；
- Websphere（IBM）：要钱的！支持 JavaEE，适合大型项目；

Tomcat

1 Tomcat 概述

Tomcat 服务器由 Apache 提供，开源免费。由于 Sun 和其他公司参与到了 Tomcat 的开发中，所以最新的 JSP/Servlet 规范总是能在 Tomcat 中体现出来。当前最新版本是 Tomcat8，我们课程中使用 Tomcat7。Tomcat7 支持 Servlet3.0，而 Tomcat6 只支持 Servlet2.5！

2 安装、启动、配置 Tomcat

下载 Tomcat 可以到 http://tomcat.apache.org 下载。

Tomcat 分为安装版和解压版：

- 安装版：一台电脑上只能安装一个 Tomcat；
- 解压版：无需安装，解压即可用，解压多少份都可以，所以我们选择解压版。

2.1 Tomcat 目录结构

安装版 Tomcat 的安装过程请参考 day03_res/Tomcat 安装.doc 文件。

把解压版 Tomcat 解压到一个没有中文，没有空格的路径中即可，建议路径不要太深，因为我们需要经常进入 Tomcat 安装目录。例如：F:\apache-tomcat-7.0.42

2.2 启动和关闭 Tomcat

在启动 Tomcat 之前，我们必须要配置环境变量：

- JAVA_HOME：必须先配置 JAVA_HOME，因为 Tomcat 启动需要使用 JDK；
- CATALANA_HOME：如果是安装版，那么还需要配置这个变量，这个变量用来指定 Tomcat 的安装路径，例如：F:\apache-tomcat-7.0.42。
- 启动：进入%CATALINA_HOME%\bin 目录，找到 startup.bat，双击即可；
- 关闭：进入%CATALINA_HOME%\bin 目录，找到 shutdown.bat，双击即可；

startup.bat 会调用 catalina.bat，而 catalina.bat 会调用 setclasspath.bat，setclasspath.bat 会使用 JAVA_HOME 环境变量，所以我们必须在启动 Tomcat 之前把 JAVA_HOME 配置正确。

启动问题：

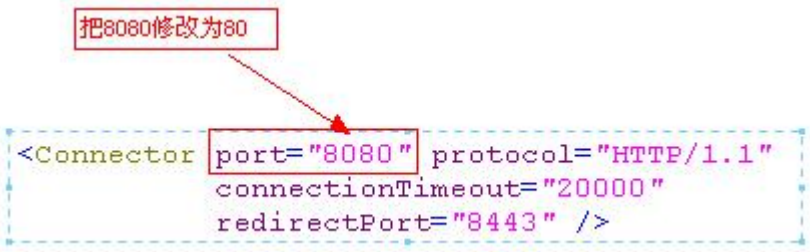
- 点击 startup.bat 后窗口一闪即消失：检查 JAVA_HOME 环境变量配置是否正确；

2.3 进入 Tomcat 主页

访问：http://localhost:8080

2.4 配置端口号

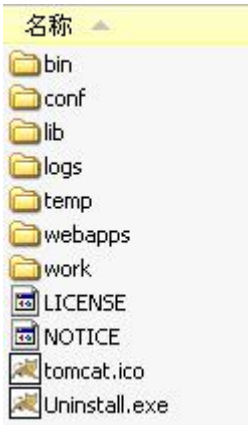
打开%CATALANA_HOME%\conf\server.xml 文件：



http 默认端口号为 80，也就是说在 URL 中不给出端口号时就表示使用 80 端口。当然你也可以修改为其它端口号。

当把端口号修改为 80 后，在浏览器中只需要输入：http://localhost 就可以访问 Tomcat 主页了。

2.5 Tomcat 的目录结构



- bin：该目录下存放的是二进制可执行文件，如果是安装版，那么这个目录下会有两个 exe 文件：tomcat6.exe、tomcat6w.exe，前者是在控制台下启动 Tomcat，后者是弹出 UGI 窗口启动 Tomcat；如果是解压版，那么会有 startup.bat 和 shutdown.bat 文件，startup.bat 用来启动 Tomcat，但需要先配置 JAVA_HOME 环境变量才能启动，shutdawn.bat 用来停止 Tomcat；
- conf：这是一个非常非常重要的目录，这个目录下有四个最为重要的文件：
  - server.xml：配置整个服务器信息。例如修改端口号，添加虚拟主机等；下面会详细介绍这个文件；
  - tomcatusers.xml：存储 tomcat 用户的文件，这里保存的是 tomcat 的用户名及密码，以及用户的角色信息。可以按着该文件中的注释信息添加 tomcat 用户，然后就可以在 Tomcat 主页中进入 Tomcat Manager 页面了；
  - web.xml：部署描述符文件，这个文件中注册了很多 MIME 类型，即文档类型。这些 MIME 类型是客户端与服务器之间说明文档类型的，如用户请求一个 html 网页，那么服务器还会告诉客户端浏览器响应的文档是 text/html 类型的，这就是一个 MIME 类型。客户端浏览器通过这个 MIME 类型就知道如何处理它了。当然是在浏览器中显示这个 html 文件了。但如果服务器响应的是一个 exe 文件，那么浏览器就不可能显示它，而是应该弹出下载窗口才对。MIME 就是用来说明文档的内容是什么类型的！
  - context.xml：对所有应用的统一配置，通常我们不会去配置它。
- lib：Tomcat 的类库，里面是一大堆 jar 文件。如果需要添加 Tomcat 依赖的 jar 文件，可以把它放到这个目录中，当然也可以把应用依赖的 jar 文件放到这个目录中，这个目录中的 jar

所有项目都可以共享之，但这样你的应用放到其他 Tomcat 下时就不能再共享这个目录下的 Jar 包了，所以建议只把 Tomcat 需要的 Jar 包放到这个目录下；

- logs：这个目录中都是日志文件，记录了 Tomcat 启动和关闭的信息，如果启动 Tomcat 时有错误，那么异常也会记录在日志文件中。
- temp：存放 Tomcat 的临时文件，这个目录下的东西可以在停止 Tomcat 后删除！
- webapps：存放 web 项目的目录，其中每个文件夹都是一个项目；如果这个目录下已经存在了目录，那么都是 tomcat 自带的。项目。其中 ROOT 是一个特殊的项目，在地址栏中没有给出项目目录时，对应的就是 ROOT 项目。http://localhost:8080/examples，进入示例项目。其中 examples 就是项目名，即文件夹的名字。
- work：运行时生成的文件，最终运行的文件都在这里。通过 webapps 中的项目生成的！可以把这个目录下的内容删除，再次运行时会生再次生成 work 目录。当客户端用户访问一个 JSP 文件时，Tomcat 会通过 JSP 生成 Java 文件，然后再编译 Java 文件生成 class 文件，生成的 java 和 class 文件都会存放到这个目录下。
- LICENSE：许可证。
- NOTICE：说明文件。

Web 应用

1 创建静态应用

- 在 webapps 下创建一个 hello 目录；
- 在 webapps\hello\下创建 index.html；
- 启动 tomcat；
- 打开浏览器访问 http://localhost:8080/hello/index.html

index.html

```
<html>
  <head>
    <title>hello</title>
  </head>
  <body>
    <h1>Hello World!</h1>
  </body>
</html>
```

2 创建动态应用

- 在 webapps 下创建 hello1 目录；
- 在 webapps\hello1\下创建 WEB-INF 目录；
- 在 webapps\hello1\WEB-INF\下创建 web.xml；

- 在 webapps\hello1\下创建 index.html。
- 打开浏览器访问 http://localhost:8080/hello/index.html

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
</web-app>
```

完整的 Web 应用还需要在 WEB-INF 目录下创建：

- classes；
- lib 目录；

```
webapps
|- hello
   |-index.html
   |-WEB-INF
      |-web.xml
      |-classes
      |-lib
```

- hello：应用目录，hello 就是应用的名称；
- index.html：应用资源。应用下可以有多个资源，例如 css、js、html、jsp 等，也可以把资源放到文件夹中，例如：hello\html\index.html，这时访问 URL 为：http://localhost:8080/hello/html/index.html；
- WEB-INF：这个目录名称必须是大写，这个目录下的东西是无法通过浏览器直接访问的，也就是说放到这里的东西是安全的；
- web.xml：应用程序的部署描述符文件，可以在该文件中对应用进行配置，例如配置应用的首页：

```
<welcome-file-list>
   <welcome-file>index.html</welcome-file>
</welcome-file-list>
```

- classes：存放 class 文件的目录；
- lib：存放 jar 包的目录；

3 配置外部应用

也可以把应用放到 Tomcat 之外，这就是外部应用了。例如我们把上面写的 hello 应用从 webapps 目录中剪切到 C 盘下，即 C:/hello。现在 hello 这个 Web 应用已经不在 Tomcat 中了，这时我们需要在 tomcat 中配置外部应用的位置，配置的方式一共有两种：

- conf/server.xml：打开 server.xml 文件，找到<Host>元素，在其中添加<Context>元素，代码如下：

server.xml

```
<Host name="localhost"  appBase="webapps"
      unpackWARs="true" autoDeploy="true">
      <Context path="itcast_hello" docBase="C:/hello"/>
</Heost>
```

1) path：指定当前应用的名称；
2) docBase：指定应用的物理位置；
3) **浏览器访问路径：http://localhost:8080/itcast_hello/index.html**。

- conf/catalana/localhost：在该目录下创建 itcast_hello.xml 文件，在该文件中编写<Context>元素，代码如下：

```
1 <Context docBase="C:/hello"/>
```

1) 文件名：指定当前应用的名称；
2) docBase：指定应用的物理位置；
3) 浏览器访问路径：http://localhost:8080/itcast_hello/index.html。

4 理解 server.xml（了解）

<Server>
  <Servier>
    <Connector>
    <Engine>
      <Host>
        <Context>

- <Server>：根元素，表示整个服务器的配置信息；
- <Service>：<Server>的子元素，在<Server>中只能有一个<Service>元素，它表示服务；
- <Connector>：<Service>的子元素，在<Service>中可以有 N 个<Connector>元素，它表示连接。
- <Engine>：<Service>的子元素，在<Service>中只能有一<Engine>元素，该元素表示引擎，它是<Service>组件的核心。
- <Host>：<Engine>的子元素，在<Engine>中可以有 N 个<Host>元素，每个<Host>元素表示一个虚拟主机。所谓虚拟主机就像是真的主机一样，每个主机都有自己的主机名和项目目录。例如<Host name="localhost" appBase="webapps">表示主机名为 localhost，这个主机的项目存放在 webapps 目录中。访问这个项目下的主机时，需要使用 localhost 主机名，项目都存放在 webapps 目录下。
- <Context>：<Host>元素的子元素，在<Host>中可以有 N 个<Context>元素，每个<Context>元素表示一个应用。如果应用在<Host>的 appBase 指定的目录下，那么可以不配置<Context>元素，如果是外部应用，那么就必须配置<Context>。如果要为应用指定资源，也需要配置<Context>元素。

我们可以把<Server>看作是一个大酒店：

- <Service>：酒店的服务部门；
- <Connector>：服务员；
- <Engine>：后厨；
- <Host>：后厨中的一个区，例如川菜区是一个<Host>、粤菜区是一个<Host>；
- <Context>：后厨的一个厨师。

用户发出一个请求：http://localhost:8080/hello/index.jsp。发现是 http/1.1 协议，而且还是 8080 端口，所以就交给了处理这一请求的“服务员（处理 HTTP 请求的<Connector>）”，“服务员”再把请求交给了“后厨（<Engine>）”，因为请求是要一盘水煮鱼，所以由“川菜区（<Host>）”负责，因为“大老王师傅<Context>”做水煮鱼最地道，所以由它完成。

- <Connector>：关心请求中的 http、和 8080；
- < Host>：关心 localhost；
- <Context>：关心 hello

5 映射虚拟主机

我们的目标是，在浏览器中输出：http://www.itcast.cn 就可以访问我们的项目。

完成这一目标，我们需要做三件事：

- 修改端口号为 80，这一点应该没有问题吧；
- 在本机上可以解析域名为 127.0.0.1，这需要修改 C:\WINDOWS\system32\drivers\etc\hosts 文件，添加对 http://www.itcast.cn 和 127.0.0.01 的绑定关系；
- 在 server.xml 文件中添加一个<Host>（主机）。

1) 修改端口号为 80

```
<Connector port="80" protocol="HTTP/1.1"
           connectionTimeout="20000"
           redirectPort="8443" />
```

2) 绑定 http://www.itcast.cn 和 127.0.0.01 的绑定关系

```
23  127.0.0.1       www.itcast.cn
```

3) server.xml 文件中添加一个<Host>

```
<Host name="www.itcast.cn"  appBase="F:/itcastapps"
      unpackWARs="true" autoDeploy="true">
</Host>
```

- name=”www.itcast.cn”：指定虚拟主机名为 www.itcast.cn；
- appBase=”F:/itcastapps”：指定当前虚拟主机的应用程序存放目录为 F:/itcastapps。
- 在 itcastapps 目录下创建名为 ROOT 的应用，因为一个主机只可以有一个名为 ROOT 的应用，名为 ROOT 的应用在浏览器中访问是可以不给出应用名称。

现在访问：http://www.itcast.cn 看看是什么页面！
请注意，只有本机可以通过 http://www.itcast.cn 来访问，而其他电脑不可以！

6 MyEclipse 创建 JavaWeb 应用

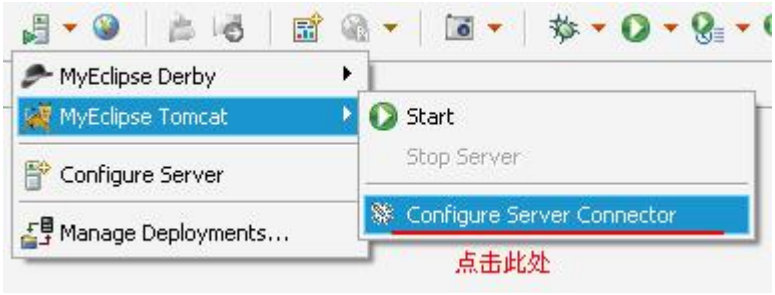
6.1 配置 Tomcat

使用 MyEclipse 配置服务器后，就可以使用 MyEclipse 来启动和停止服务器了。当然，你需要先安装好服务器（Tomcat），才能配置。

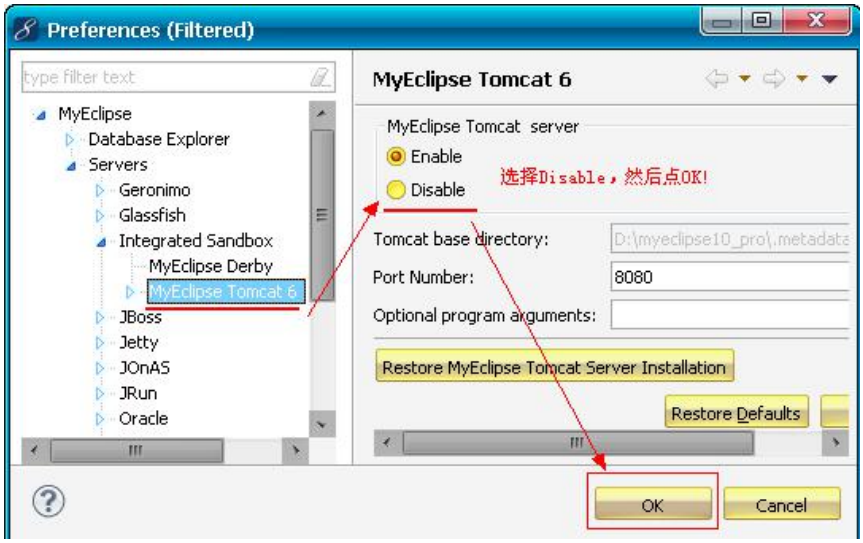
MyEclipse 自带了一个 Tomcat，强烈建议不要使用它。所以，我们需要先把 MyEclipse 自带的 Tomcat 关闭，然后再来配置我们自己的 Tomcat。

- 关闭 MyEclipse 自带 Tomcat。

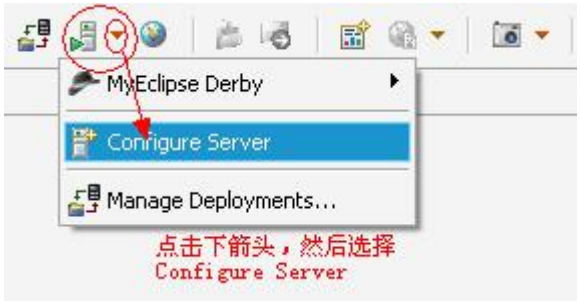
在工具栏中找到  ，点击下箭头，点击 Configure Server Connector。

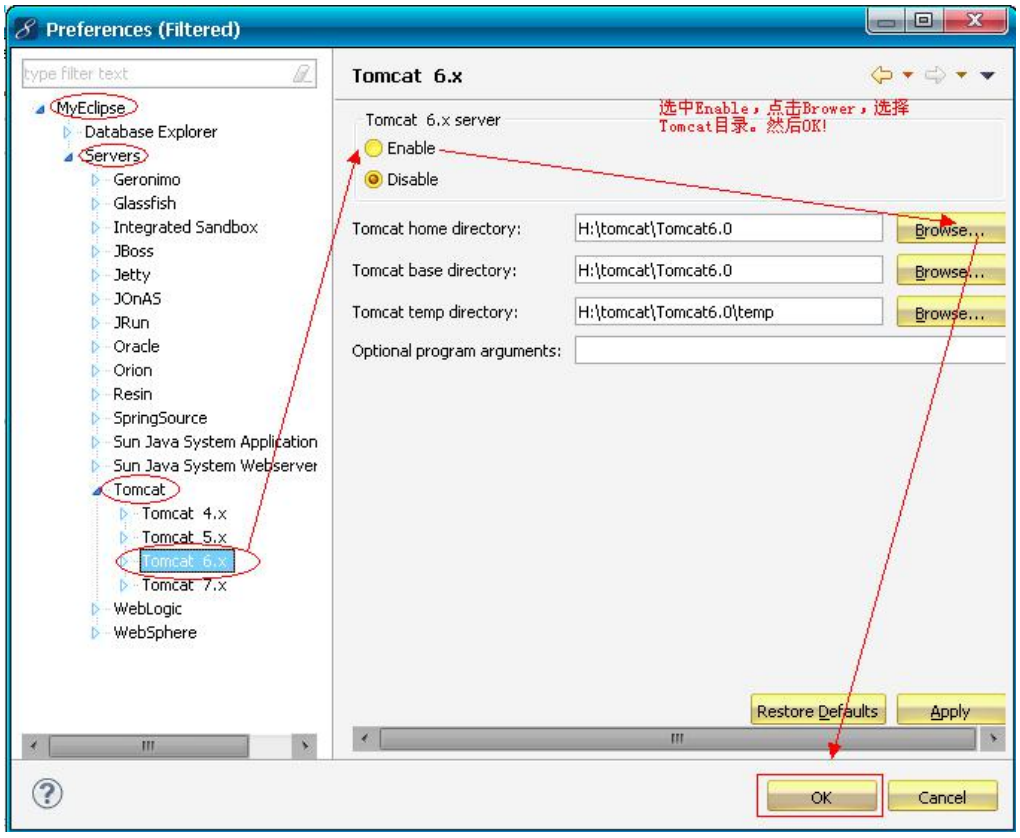


弹出对话框

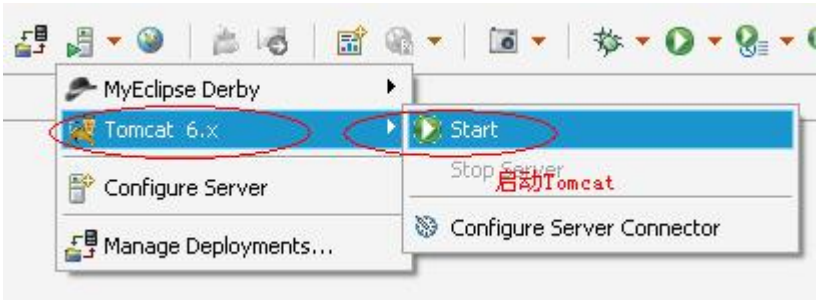


- 配置我们自己的 Tomcat

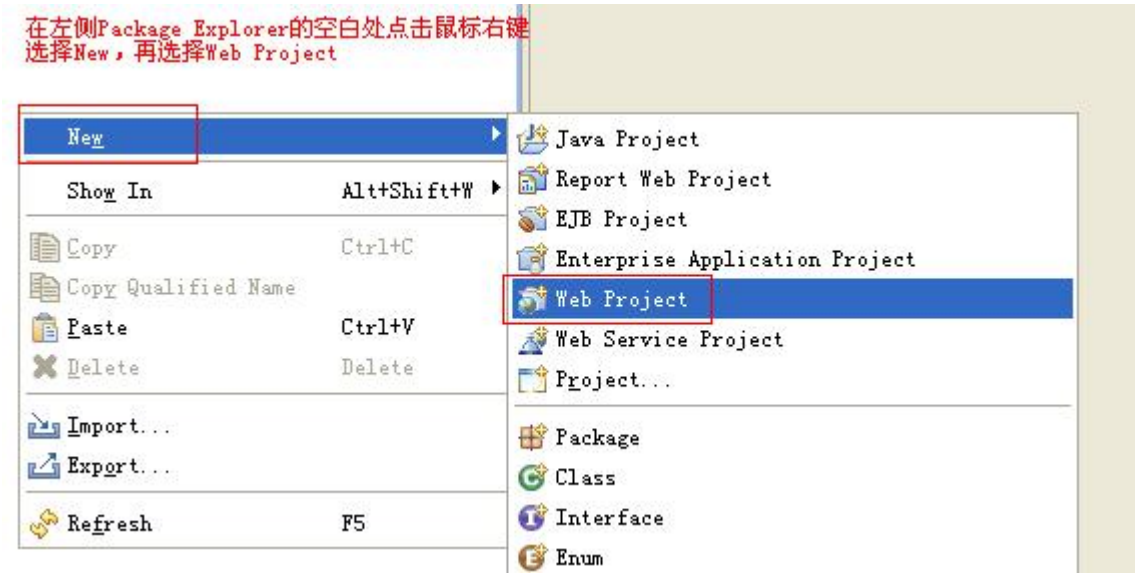


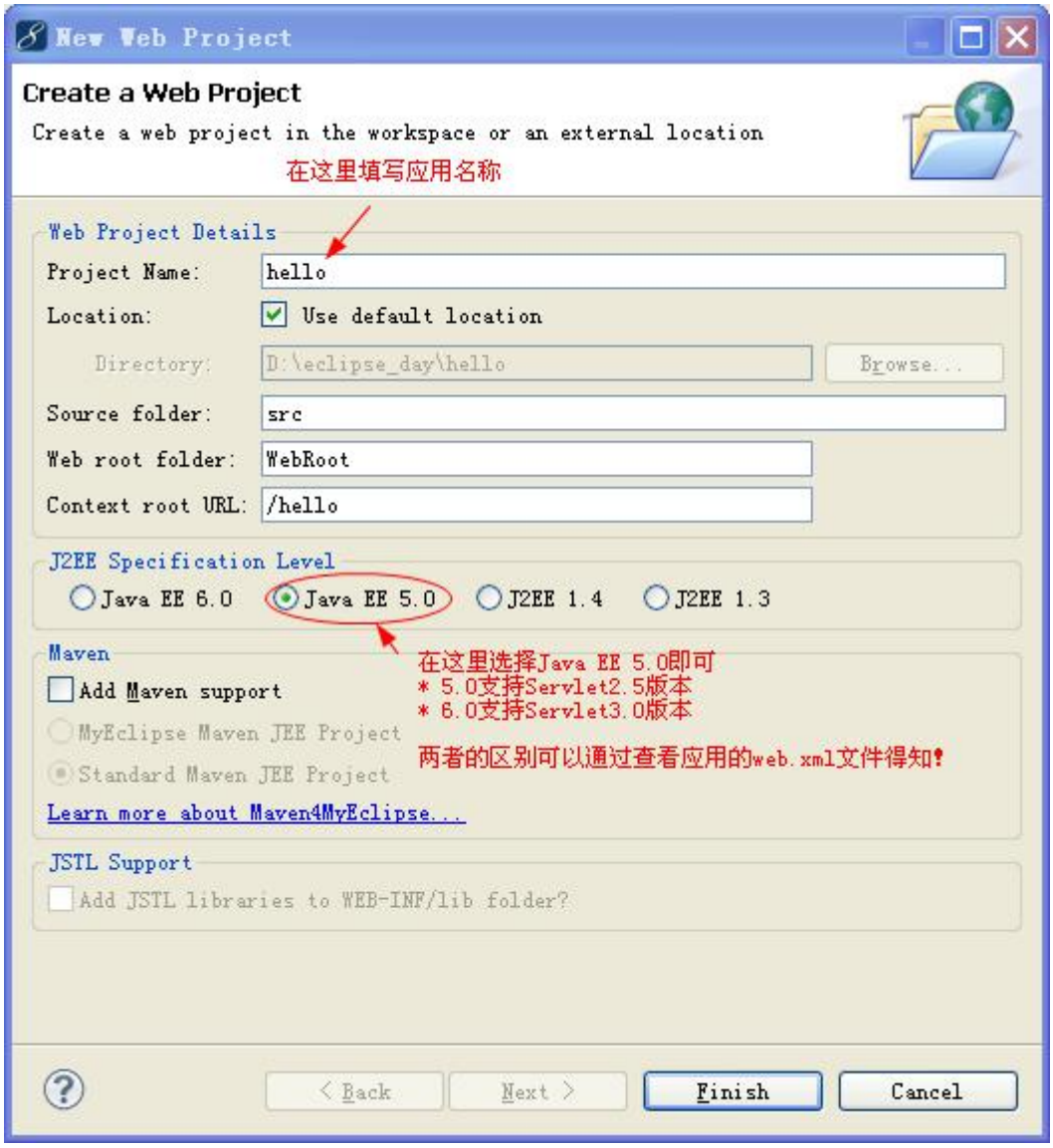


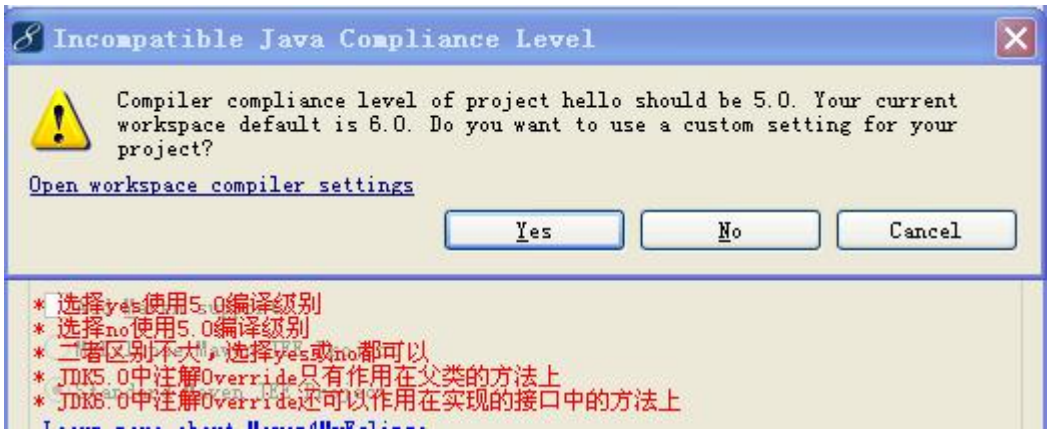
使用 MyEclipse 启动 Tomcat



6.2 创建 JavaWeb 应用

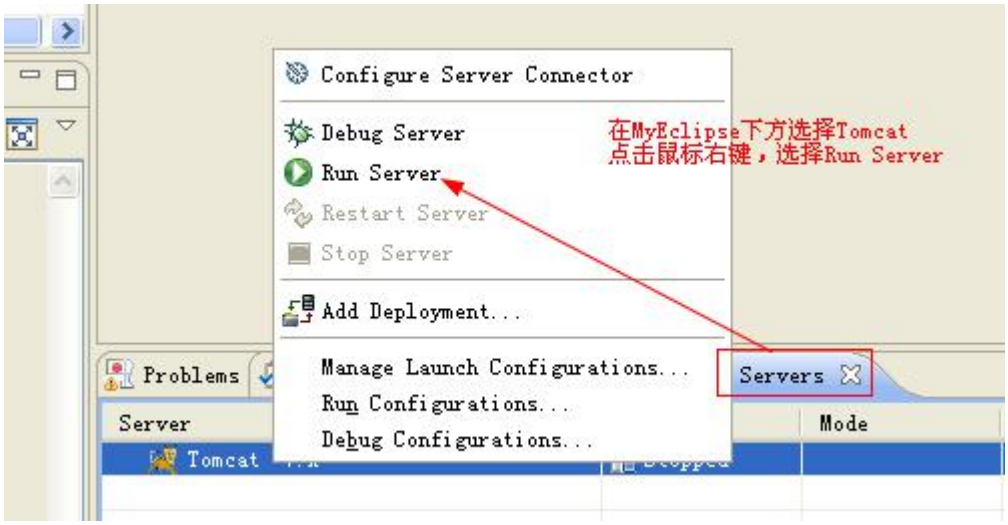


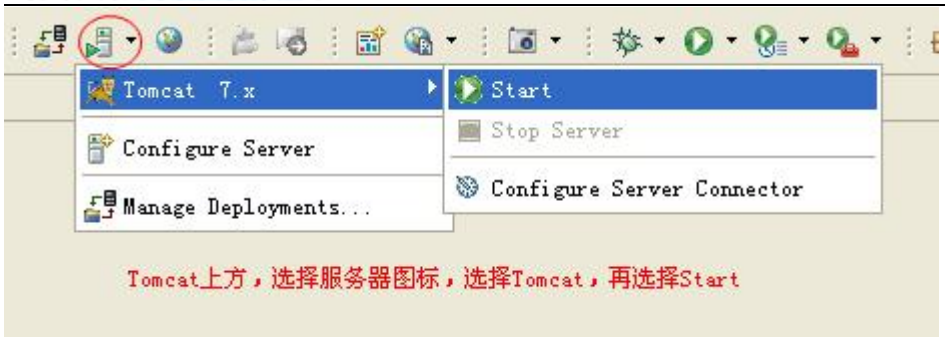



Incompatible Java Compliance Level
Compiler compliance level of project hello should be 5.0. Your current workspace default is 6.0. Do you want to use a custom setting for your project?
Open workspace compiler settings
Yes
No
Cancel
* 选择yes使用5.0编译级别
* 选择no使用5.0编译级别
* 二者区别不大，选择yes或no都可以
* JDK5.0中注解Override只有作用在父类的方法上
* JDK6.0中注解Override还可以作用在实现的接口中的方法上


Package
Type Hier
hello
src
JRE System Library [jdk1.7.0_09]
Java EE 5 Libraries
WebRoot
META-INF
WEB-INF
lib
web.xml
index.jsp
存放项目源代码
JDK
javaEE相关jar
web项目的根，当项目部署时，就是把这个目录下的所有内容发布到Tomcat的wabapps下，发布到Tomcat下后，这个目录名就是以项目名来命名了
清单目录，与jar文件中的清单目录是一样的
在WebRoot下的目录结构与我们自己写的Web项目是一致。
MyEclipse会创建一个index.jsp页面
lib目录下存放本项目所依赖的jar文件，如果src下创建了java文件，那么还会在WEB-INF目录下多出一个classes目录，这个目录会存放src源文件编译后的class文件

6.3 启动 Tomcat


Configure Server Connector
Debug Server
Run Server
Restart Server
Stop Server
Add Deployment...
Manage Launch Configurations...
Run Configurations...
Debug Configurations...
在MyEclipse下方选择Tomcat
点击鼠标右键，选择Run Server
Problems
Servers
Server
Mode
Tomcat

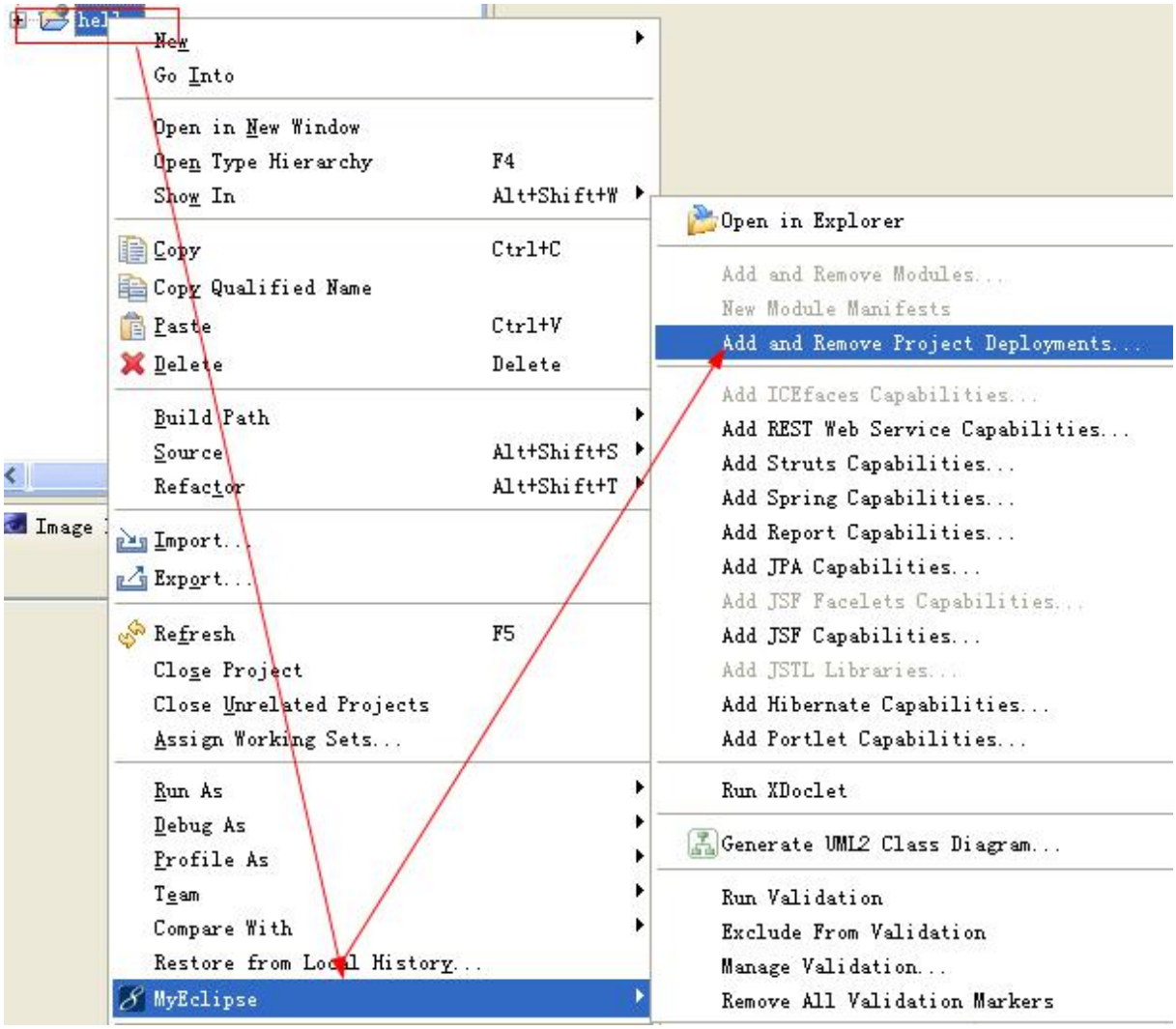


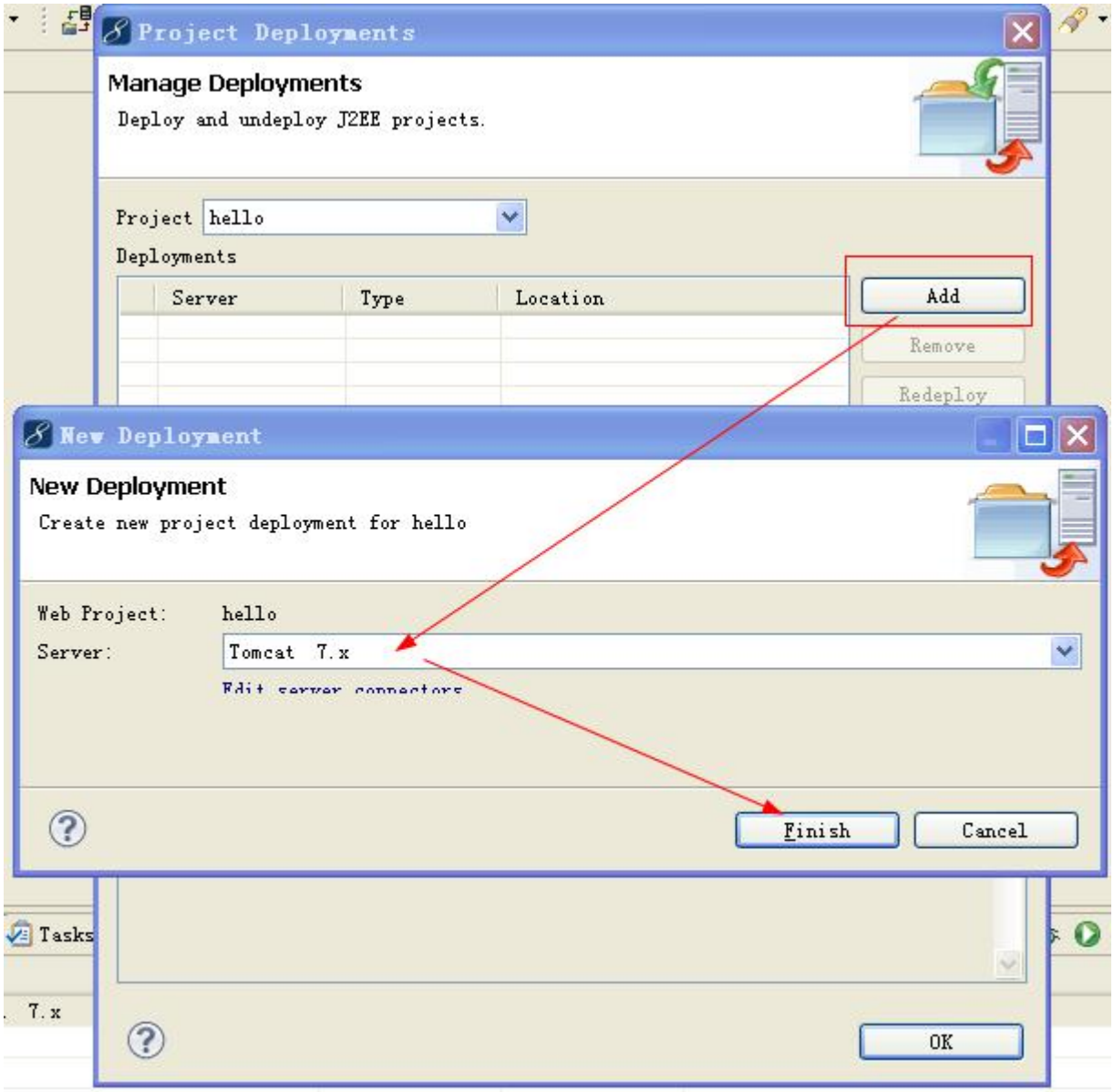
上面两种方式都可以启动 tomcat

6.4 关闭 tomcat

与启动 Tomcat 相同位置下方就是 Stop Server，即可关闭 tomcat 了。

6.5 发布项目到 tomcat 的 webapps 目录

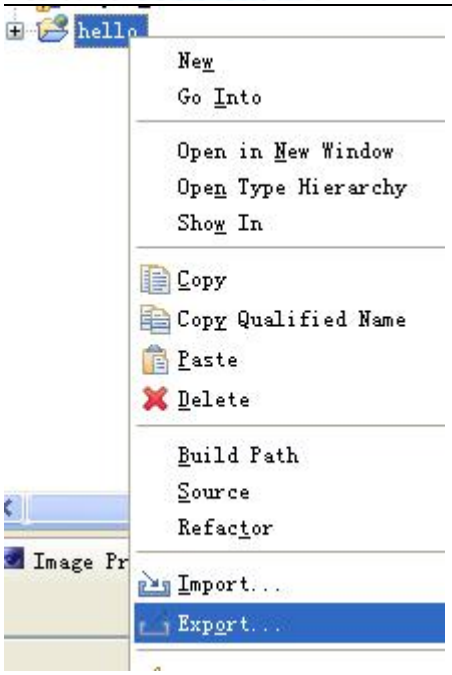


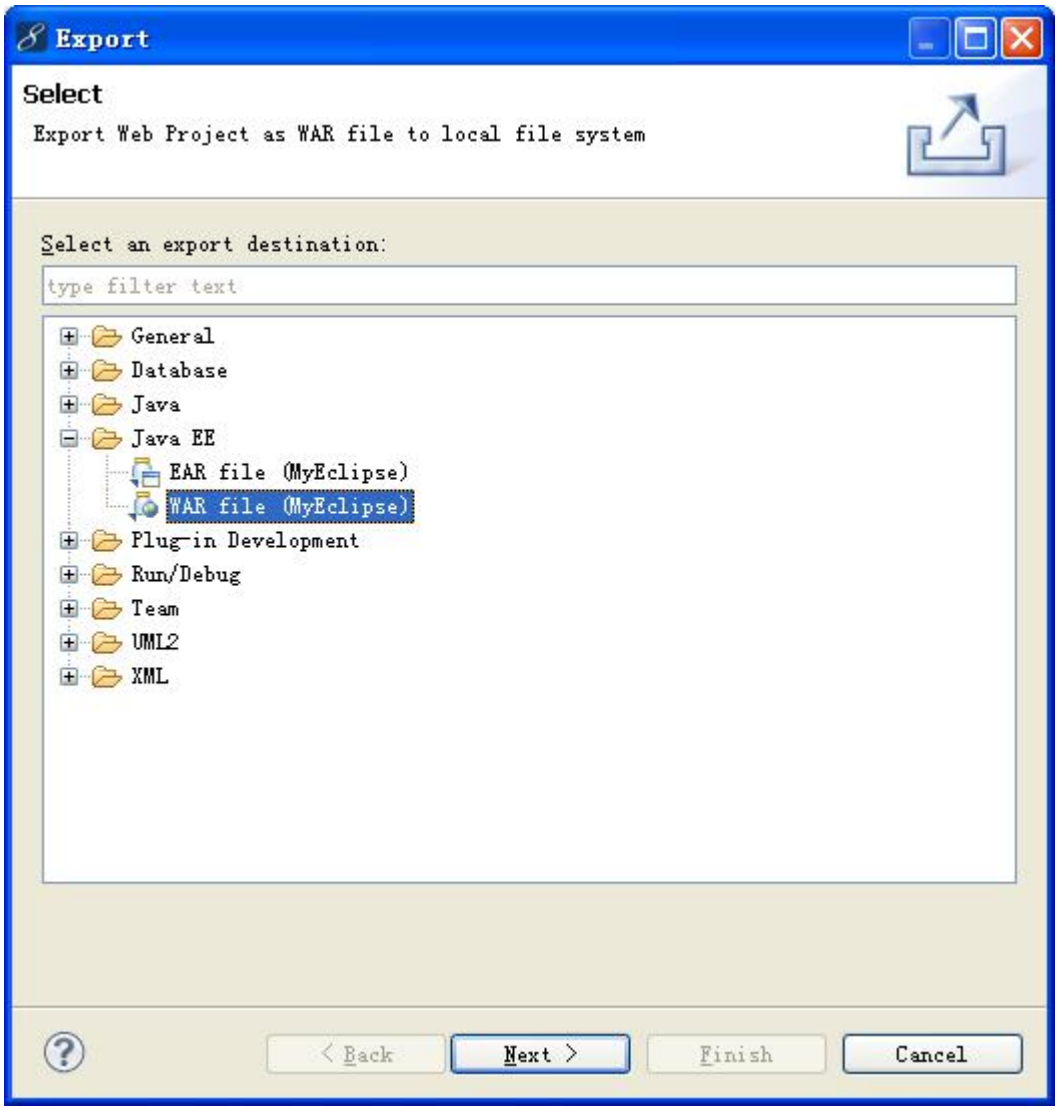


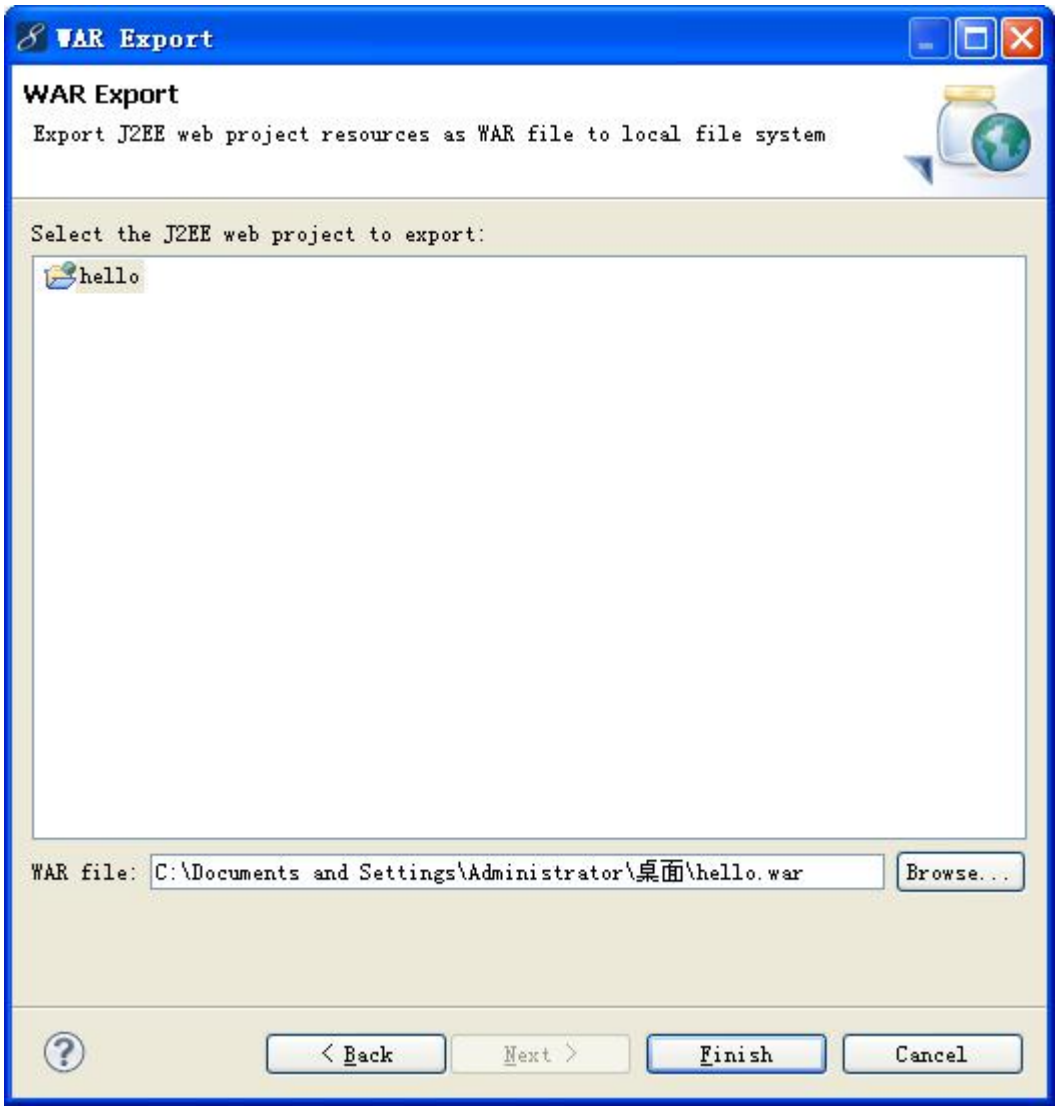
项目发布后，就是把项目的 WebRoot 目录 copy 到 Tomcat 的 webapps 目录，并把 WebRoot 重命名为项目名称，即 hello。所以在 Tomcat 的 webapps 下会多出一个文件夹 hello。

6.6 打 war 包

JavaSE 程序可以打包成 Jar 包，而 JavaWeb 程序可以打包成 war 包。然后把 war 发布到 Tomcat 的 webapps 目录下，Tomcat 会在启动时自动解压 war 包。


hello
New
Go Into
Open in New Window
Open Type Hierarchy
Show In
Copy
Copy Qualified Name
Paste
Delete
Build Path
Source
Refactor
Import...
Export...
Image Pr


Export
Select
Export Web Project as WAR file to local file system
Select an export destination:
type filter text
General
Database
Java
Java EE
EAR file (MyEclipse)
WAR file (MyEclipse)
Plug-in Development
Run/Debug
Team
UML2
XML
< Back
Next >
Finish
Cancel



HTTP 协议

1 安装 HttpWatch

HttpWatch 是专门为 IE 浏览器提供的，用来查看 HTTP 请求和响应内容的工具。而 FireFox 上需要安装 FireBug 软件。如果你使用的是 Chrome，那么就不用自行安装什么工具了，因为它自身就有查看请求和响应内容的功能！

HttpWatch 和 FireBug 这些工具对浏览器而言不是必须的，但对我们开发者是很有帮助的，通过查看 HTTP 请求响应内容，可以使我们更好的学习 HTTP 协议。

2 HTTP 概述

HTTP（hypertext transport protocol），即超文本传输协议。这个协议详细规定了浏览器和万维网服务器之间互相通信的规则。

HTTP 就是一个通信规则，通信规则规定了客户端发送给服务器的内容格式，也规定了服务器发送给客户端的内容格式。其实我们要学习的就是这个两个格式！客户端发送给服务器的格式叫“请求协议”；服务器发送给客户端的格式叫“响应协议”。

3 URL 和 URI

URL：统一资源定位符，就是一个网址，例如：http://www.itcast.cn 就是一个 URL。/hello/index.html 也是一个 URL，URL 必须是一个真实存在的网址。

URI：统一资源标识符：比 URI 包含了 URL，URI 的范围更加宽泛，URI 可以是一个不存在的网址。在网络上用来标签资源的都是 URI，例如 zhangSan@163.com 也是 URI。

4 请求协议

请求协议的格式如下：

```
请求首行；
请求头信息；
空行；
请求体。
```

浏览器发送给服务器的内容就这个格式的，如果不是这个格式服务器将无法解读！在 HTTP 协议中，请求有很多请求方法，其中最为常用的就是 GET 和 POST。不同的请求方法之间的区别，后面会一点一点的介绍。

4.1 GET 请求

打开 IE，在访问 hello 项目的 index.jsp 之间打开 HttpWatch，并点击“Record”按钮。然后访问 index.jsp 页面。查看请求内容如下：

```
GET /hello/index.jsp HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows NT 5.1; rv:5.0) Gecko/20100101 Firefox/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: zh-cn,zh;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: GB2312,utf-8;q=0.7,*;q=0.7
Connection: keep-alive
Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98
```

- GET /hello/index.jsp HTTP/1.1：GET 请求，请求服务器路径为/hello/index.jsp，协议为 1.1；
- *Host:localhost：请求的主机名为 localhost；
- *User-Agent: Mozilla/5.0 (**Windows NT 5.1**; rv:5.0) Gecko/20100101 **Firefox/5.0**：与浏览器和 OS 相关的信息。有些网站会显示用户的系统版本和浏览器版本信息，这都是通过获取 User-Agent 头信息而来的；
- Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8：告诉服务器，当前客户端可以接收的文档类型，其实这里包含了*/*，就表示什么都可以接收；
- Accept-Language: zh-cn,zh;q=0.5：当前客户端支持的语言，可以在浏览器的工具→选项中找到语言相关信息；
- Accept-Encoding: gzip, deflate：支持的压缩格式。数据在网络上传递时，可能服务器会把数据压缩后再发送；
- Accept-Charset: GB2312,utf-8;q=0.7,*;q=0.7：客户端支持的编码；
- Connection: keep-alive：客户端支持的链接方式，保持一段时间链接，默认为 3000ms；
- *Cookie: JSESSIONID=369766FDF6220F7803433C0B2DE36D98：因为不是第一次访问这个地址，所以会在请求中把上一次服务器响应中发送过来的 Cookie 在请求中一并发送去过；这个 Cookie 的名字为 JSESSIONID，然后在讲会话是讲究它！

4.2 POST 请求

为了演示 POST 请求，我们需要修改 index.jsp 页面，即添加一个表单：

```
<form action="" method="post">
  关键字：<input type="text" name="keyword"/>
  <input type="submit" value="提交"/>
</form>
```

关键字： 提交

打开 HttpWatch，输入 hello 后点击提交，查看请求内容如下：

```
POST /hello/index.jsp HTTP/1.1
Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/msword, application/vnd.ms-excel,
application/vnd.ms-powerpoint, application/x-ms-application, application/x-ms-xbap,
application/vnd.ms-xpsdocument, application/xaml+xml, */*
Referer: http://localhost:8080/hello/index.jsp
Accept-Language: zh-cn,en-US;q=0.5
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; Trident/4.0; InfoPath.2; .NET CLR
2.0.50727; .NET CLR 3.0.4506.2152; .NET CLR 3.5.30729)
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
Host: localhost:8080
Content-Length: 13
Connection: Keep-Alive
Cache-Control: no-cache
```

```
Cookie: JSESSIONID=E365D980343B9307023A1D271CC48E7D

keyword=hello
```

POST 请求是可以有体的，而 GET 请求不能有请求体。

- Referer: http://localhost:8080/hello/index.jsp：请求来自哪个页面，例如你在百度上点击链接到了这里，那么 Referer:http://www.baidu.com；如果你是在浏览器的地址栏中直接输入的地址，那么就没有 Referer 这个请求头了；
- Content-Type: application/x-www-form-urlencoded：表单的数据类型，说明会使用 url 格式编码数据；url 编码的数据都是以“%”为前缀，后面跟随两位的 16 进制，例如“传智”这两个字使用 UTF-8 的 url 编码用为“%E4%BC%A0%E6%99%BA”；
- Content-Length:13：请求体的长度，这里表示 13 个字节。
- keyword=hello：请求体内容！hello 是在表单中输入的数据，keyword 是表单字段的名字。

Referer 请求头是比较有用的一个请求头，它可以用来做统计工作，也可以用来做防盗链。

统计工作：我公司网站在百度上做了广告，但不知道在百度上做广告对我们网站的访问量是否有影响，那么可以对每个请求中的 Referer 进行分析，如果 Referer 为百度的很多，那么说明用户都是通过百度找到我们公司网站的。

防盗链：我公司网站上有一个下载链接，而其他网站盗链了这个地址，例如在我网站上的 index.html 页面中有一个链接，点击即可下载 JDK7.0，但有某个人的微博中盗链了这个资源，它也有一个链接指向我们网站的 JDK7.0，也就是说登录它的微博，点击链接就可以从我网站上下载 JDK7.0，这导致我们网站的广告没有看，但下载的却是我网站的资源。这时可以使用 Referer 进行防盗链，在资源被下载之前，我们对 Referer 进行判断，如果请求来自本网站，那么允许下载，如果非本网站，先跳转到本网站看广告，然后再允许下载。

5 响应协议

5.1 响应内容

响应协议的格式如下：

```
响应首行；
响应头信息；
空行；
响应体。
```

响应内容是由服务器发送给浏览器的内容，浏览器会根据响应内容来显示。

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=UTF-8
Content-Length: 724
Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello
Date: Wed, 25 Sep 2012 04:15:03 GMT

```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="http://localhost:8080/hello/">

    <title>My JSP 'index.jsp' starting page</title>
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="This is my page">
    <!--
    <link rel="stylesheet" type="text/css" href="styles.css">
    -->
  </head>

  <body>
<form action="" method="post">
  关键字：<input type="text" name="keyword"/>
  <input type="submit" value="提交"/>
</form>
  </body>
</html>
```

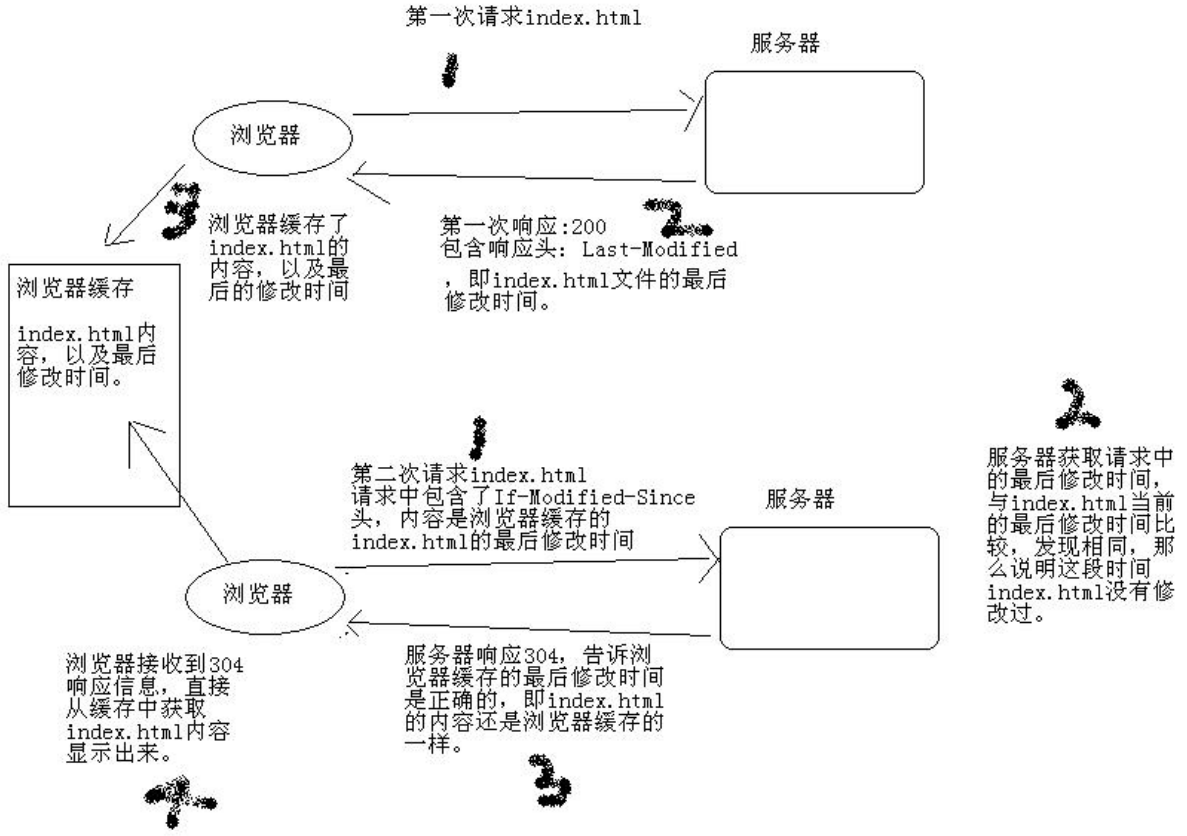
- HTTP/1.1 200 OK：响应协议为 HTTP1.1，状态码为 200，表示请求成功，OK 是对状态码的解释；
- Server: Apache-Coyote/1.1：服务器的版本信息；
- * Content-Type: text/html;charset=UTF-8：响应体使用的编码为 UTF-8；
- Content-Length: 724：响应体为 724 字节；
- Set-Cookie: JSESSIONID=C97E2B4C55553EAB46079A4F263435A4; Path=/hello：响应给客户端的 Cookie；
- Date: Wed, 25 Sep 2012 04:15:03 GMT：响应的时间，这可能会有 8 小时的时区差；

5.2 状态码

响应头对浏览器来说很重要，它说明了响应的真正含义。例如 200 表示响应成功了，302 表示重定向，这说明浏览器需要再发一个新的请求。

- 200：请求成功，浏览器会把响应体内容（通常是 html）显示在浏览器中；
- 404：请求的资源没有找到，说明客户端错误的请求了不存在的资源；
- 500：请求资源找到了，但服务器内部出现了错误；
- 302：重定向，当响应码为 302 时，表示服务器要求浏览器重新再发一个请求，服务器会发送一个响应头 Location，它指定了新请求的 URL 地址；
- 304：当用户第一次请求 index.html 时，服务器会添加一个名为 Last-Modified 响应头，这个

头说明了 index.html 的最后修改时间，浏览器会把 index.html 内容，以及最后响应时间缓存下来。当用户第二次请求 index.html 时，在请求中包含一个名为 If-Modified-Since 请求头，它的值就是第一次请求时服务器通过 Last-Modified 响应头发送给浏览器的值，即 index.html 最后的修改时间，If-Modified-Since 请求头就是在告诉服务器，我这里浏览器缓存的 index.html 最后修改时间是这个，您看看现在的 index.html 最后修改时间是不是这个，如果还是，那么您就不用再响应这个 index.html 内容了，我会把缓存的内容直接显示出来。而服务器端会获取 If-Modified-Since 值，与 index.html 的当前最后修改时间比对，如果相同，服务器会发响应码 304，表示 index.html 与浏览器上次缓存的相同，无需再次发送，浏览器可以显示自己的缓存页面，如果比对不同，那么说明 index.html 已经做了修改，服务器会响应 200。



5.3 其他响应头

告诉浏览器不要缓存的响应头：

- Expires: -1；
- Cache-Control: no-cache；
- Pragma: no-cache；

自动刷新响应头，浏览器会在 3 秒之后请求 http://www.itcast.cn：

- Refresh: 3;url=http://www.itcast.cn

5.4 HTML 中指定响应头

在 HTMl 页面中可以使用<meta http-equiv="" content="">来指定响应头，例如在 index.html 页面中给出<meta http-equiv="Refresh" content="3;url=http://www.itcast.cn">，表示浏览器只会显示 index.html 页面 3 秒，然后自动跳转到 http://www.itcast.cn。

day04

Servlet 概述

1 什么是 Servlet

Servlet 是 JavaWeb 的三大组件之一，它属于动态资源。Servlet 的作用是处理请求，服务器会把接收到的请求交给 Servlet 来处理，在 Servlet 中通常需要：

- 接收请求数据；
- 处理请求；
- 完成响应。

例如客户端发出登录请求，或者输出注册请求，这些请求都应该由 Servlet 来完成处理！Servlet 需要我们自己来编写，每个 Servlet 必须实现 javax.servlet.Servlet 接口。

2 实现 Servlet 的方式

实现 Servlet 有三种方式：

- 实现 javax.servlet.Servlet 接口；
- 继承 javax.servlet.GenericServlet 类；
- 继承 javax.servlet.http.HttpServlet 类；

通常我们会去继承 HttpServlet 类来完成我们的 Servlet，但学习 Servlet 还要从 javax.servlet.Servlet 接口开始学习。

Servlet.java

```
public interface Servlet {
    public void init(ServletConfig config) throws ServletException;
    public ServletConfig getServletConfig();
    public void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException;
    public String getServletInfo();
    public void destroy();
}
```

3 创建 helloservlet 应用

我们开始第一个 Servlet 应用吧！首先在 webapps 目录下创建 helloservlet 目录，它就是我们的应用目录了，然后在 helloservlet 目录中创建准备 JavaWeb 应用所需内容：

- 创建/helloservlet/WEB-INF 目录；

- 创建/helloservlet/WEB-INF/classes 目录；
- 创建/helloservlet/WEB-INF/lib 目录；
- 创建/helloservlet/WEB-INF/web.xml 文件；

接下来我们开始准备完成 Servlet，完成 Servlet 需要分为两步：

- 编写 Servlet 类；
- 在 web.xml 文件中配置 Servlet；

HelloServlet.java

```
public class HelloServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {}
    public ServletConfig getServletConfig() {return null;}
    public void destroy() {}
    public String getServletInfo() {return null;}

    public void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException {
        System.out.println("hello servlet!");
    }
}
```

我们暂时忽略 Servlet 中其他四个方法，只关心 service()方法，因为它是用来处理请求的方法。我们在该方法内给出一条输出语句！

web.xml（下面内容需要背下来）

```
    <servlet>
        <servlet-name>hello</servlet-name>
        <servlet-class>cn.itcast.servlet.HelloServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>hello</servlet-name>
        <url-pattern>/helloworld</url-pattern>
    </servlet-mapping>
```

在 web.xml 中配置 Servlet 的目的其实只有一个，就是把访问路径与一个 Servlet 绑定到一起，上面配置是把访问路径：“/helloworld”与“cn.itcast.servlet.HelloServlet”绑定到一起。

- <servlet>：指定 HelloServlet 这个 Servlet 的名称为 hello；
- <servlet-mapping>：指定/helloworld 访问路径所以访问的 Servlet 名为 hello。

<servlet>和<servlet-mapping>通过<servlet-name>这个元素关联在一起了！

接下来，我们编译 HelloServlet，注意，编译 HelloServlet 时需要导入 servlet-api.jar，因为 Servlet.class 等类都在 servlet-api.jar 中。

javac -classpath F:/tomcat6/lib/servlet-api.jar -d . HelloServlet.java

然后把 HelloServlet.class 放到/helloworld/WEB-INF/classes/目录下，然后启动 Tomcat，在浏览器

中访问：http://localhost:8080/helloservlet/helloworld 即可在控制台上看到输出！

- /helloservlet/WEB-INF/classes/cn/itcast/servlet/HelloServlet.class；

Servlet 接口

1 Servlet 的生命周期

所谓 xxx 的生命周期，就是说 xxx 的出生、服务，以及死亡。Servlet 生命周期也是如此！与 Servlet 的生命周期相关的方法有：

- void init(ServletConfig)；
- void service(ServletRequest,ServletResponse)；
- void destroy()；

1.1 Servlet 的出生

服务器会在 Servlet 第一次被访问时创建 Servlet，或者是在服务器启动时创建 Servlet。如果服务器启动时就创建 Servlet，那么还需要在 web.xml 文件中配置。也就是说默认情况下，Servlet 是在第一次被访问时由服务器创建的。

而且一个 Servlet 类型，服务器只创建一个实例对象，例如在我们首次访问 http://localhost:8080/helloservlet/helloworld 时，服务器通过“/helloworld”找到了绑定的 Servlet 名称为 cn.itcast.servlet.HelloServlet，然后服务器查看这个类型的 Servlet 是否已经创建过，如果没有创建过，那么服务器才会通过反射来创建 HelloServlet 的实例。当我们再次访问 http://localhost:8080/helloservlet/helloworld 时，服务器就不会再次创建 HelloServlet 实例了，而是直接使用上次创建的实例。

在 Servlet 被创建后，服务器会马上调用 Servlet 的 void init(ServletConfig)方法。请记住，Servlet 出生后马上就会调用 init()方法，而且一个 Servlet 的一生。这个方法只会被调用一次。这好比小孩子出生后马上就要去剪脐带一样，而且剪脐带一生只有一次。

我们可以把一些对 Servlet 的初始化工作放到 init 方法中！

1.2 Servlet 服务

当服务器每次接收到请求时，都会去调用 Servlet 的 service()方法来处理请求。服务器接收到一次请求，就会调用 service() 方法一次，所以 service()方法是会被调用多次的。正因为如此，所以我们才需要把处理请求的代码写到 service()方法中！

1.3 Servlet 的离去

Servlet 是不会轻易离去的，通常都是在服务器关闭时 Servlet 才会离去！在服务器被关闭时，服务器会去销毁 Servlet，在销毁 Servlet 之前服务器会先去调用 Servlet 的 destroy()方法，我们可以把

Servlet 的临终遗言放到 destroy()方法中，例如对某些资源的释放等代码放到 destroy()方法中。

1.4 测试生命周期方法

修改 HelloServlet 如下，然后再去访问 http://localhost:8080/helloservlet/helloworld

```
public class HelloServlet implements Servlet {
    public void init(ServletConfig config) throws ServletException {
        System.out.println("Servlet被创建了！");
    }
    public ServletConfig getServletConfig() {return null;}
    public void destroy() {
        System.out.println("Servlet要离去了！");
    }
    public String getServletInfo() {return null;}

    public void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException {
        System.out.println("hello servlet!");
    }
}
```

在首次访问 HelloServlet 时，init 方法会被执行，而且也会执行 service 方法。再次访问时，只会执行 service 方法，不再执行 init 方法。在关闭 Tomcat 时会调用 destroy 方法。

2 Servlet 接口相关类型

在 Servlet 接口中还存在三个我们不熟悉的类型：

- ServletRequest：service() 方法的参数，它表示请求对象，它封装了所有与请求相关的数据，它是由服务器创建的；
- ServletResponse：service()方法的参数，它表示响应对象，在 service()方法中完成对客户端的响应需要使用这个对象；
- ServletConfig：init()方法的参数，它表示 Servlet 配置对象，它对应 Servlet 的配置信息，那对应 web.xml 文件中的<servlet>元素。

2.1 ServletRequest 和 ServletResponse（第五天会详细讲解这两个对象）

ServletRequest 和 ServletResponse 是 Servlet#service() 方法的两个参数，一个是请求对象，一个是响应对象，可以从 ServletRequest 对象中获取请求数据，可以使用 ServletResponse 对象完成响应。你以后会发现，这两个对象就像是一对恩爱的夫妻，永远不分离，总是成对出现。

ServletRequest 和 ServletResponse 的实例由服务器创建，然后传递给 service()方法。如果在 service()方法中希望使用 HTTP 相关的功能，那么可以把 ServletRequest 和 ServletResponse 强转成 HttpServletRequest 和 HttpServletResponse。这也说明我们经常需要在 service()方法中对 ServletRequest 和 ServletResponse 进行强转，这是很心烦的事情。不过后面会有一个类来帮我们解决这一问题的。

HttpServletRequest 方法：

- String getParameter(String paramName)：获取指定请求参数的值；

- String getMethod()：获取请求方法，例如 GET 或 POST；
- String getHeader(String name)：获取指定请求头的值；
- void setCharacterEncoding(String encoding)：设置请求体的编码！因为 GET 请求没有请求体，所以这个方法只只对 POST 请求有效。当调用 request.setCharacterEncoding(“utf-8”)之后，再通过 getParameter()方法获取参数值时，那么参数值都已经通过了转码，即转换成了 UTF-8 编码。所以，这个方法必须在调用 getParameter()方法之前调用！

HttpServletResponse 方法：

- PrintWriter getWriter()：获取字符响应流，使用该流可以向客户端输出响应信息。例如 response.getWriter().print(“<h1>Hello JavaWeb!</h1>”)；
- ServletOutputStream getOutputStream()：获取字节响应流，当需要向客户端响应字节数据时，需要使用这个流，例如要向客户端响应图片；
- void setCharacterEncoding(String encoding)：用来设置字符响应流的编码，例如在调用 setCharacterEncoding(“utf-8”);之后，再 response.getWriter()获取字符响应流对象，这时的响应流的编码为 utf-8，使用 response.getWriter()输出的中文都会转换成 utf-8 编码后发送给客户端；
- void setHeader(String name, String value)：向客户端添加响应头信息，例如 setHeader(“Refresh”, “3;url=http://www.itcast.cn”)，表示 3 秒后自动刷新到 http://www.itcast.cn；
- void setContentType(String contentType)：该方法是 setHeader(“content-type”, “xxx”)的简便方法，即用来添加名为 content-type 响应头的方法。content-type 响应头用来设置响应数据的 MIME 类型，例如要向客户端响应 jpg 的图片，那么可以 setContentType(“image/jepg”)，如果响应数据为文本类型，那么还要台同时设置编码，例如 setContentType(“text/html;chartset=utf-8”)表示响应数据类型为文本类型中的 html 类型，并且该方法会调用 setCharacterEncoding(“utf-8”)方法；
- void sendError(int code, String errorMsg)：向客户端发送状态码，以及错误消息。例如给客户端发送 404：response(404, “您要查找的资源不存在！”)。

2.1 ServletConfig

ServletConfig 对象对应 web.xml 文件中的<servlet>元素。例如你想获取当前 Servlet 在 web.xml 文件中的配置名，那么可以使用 servletConfig.getServletName()方法获取！

```
<servlet>
  <servlet-name>One</servlet-name>
  <servlet-class>
      cn.itcast.servlet.OneServlet
  </servlet-class>
</servlet>
```

ServletConfig 对象是由服务器创建的，然后传递给 Servlet 的 init()方法，你可以在 init()方法中使用它！

- String getServletName()：获取 Servlet 在 web.xml 文件中的配置名称，即<servlet-name>指定的名称；
- ServletContext getServletContext()：用来获取 ServletContext 对象，ServletContext 会在后面讲解；
- String getInitParameter(String name)：用来获取在 web.xml 中配置的初始化参数，通过参数名

来获取参数值；

- Enumeration getInitParameterNames()：用来获取在 web.xml 中配置的所有初始化参数名称；

在<servlet>元素中还可以配置初始化参数：

```
<servlet>
  <servlet-name>One</servlet-name>
  <servlet-class>cn.itcast.servlet.OneServlet</servlet-class>
  <init-param>
      <param-name>paramName1</param-name>
      <param-value>paramValue1</param-value>
  </init-param>
  <init-param>
      <param-name>paramName2</param-name>
      <param-value>paramValue2</param-value>
  </init-param>
</servlet>
```

在 OneServlet 中，可以使用 ServletConfig 对象的 getInitParameter()方法来获取初始化参数，例如：String value1 = servletConfig.getInitParameter("paramName1");//获取到 paramValue1

GenericServlet

1 GenericServlet 概述

GenericServlet 是 Servlet 接口的实现类，我们可以通过继承 GenericServlet 来编写自己的 Servlet。下面是 GenericServlet 类的源代码：

GenericServlet.java

```
public abstract class GenericServlet implements Servlet, ServletConfig,
        java.io.Serializable {
    private static final long serialVersionUID = 1L;
    private transient ServletConfig config;
    public GenericServlet() {}
    @Override
    public void destroy() {}
    @Override
    public String getInitParameter(String name) {
        return getServletConfig().getInitParameter(name);
    }
    @Override
    public Enumeration<String> getInitParameterNames() {
        return getServletConfig().getInitParameterNames();
```

```
    }
    @Override
    public ServletConfig getServletConfig() {
        return config;
    }
    @Override
    public ServletContext getServletContext() {
        return getServletConfig().getServletContext();
    }
    @Override
    public String getServletInfo() {
        return "";
    }
    @Override
    public void init(ServletConfig config) throws ServletException {
        this.config = config;
        this.init();
    }
    public void init() throws ServletException {}
    public void log(String msg) {
        getServletContext().log(getServletName() + ": " + msg);
    }
    public void log(String message, Throwable t) {
        getServletContext().log(getServletName() + ": " + message, t);
    }
    @Override
    public abstract void service(ServletRequest req, ServletResponse res)
            throws ServletException, IOException;
    @Override
    public String getServletName() {
        return config.getServletName();
    }
}
```

2 GenericServlet 的 init()方法

在 GenericServlet 中，定义了一个 ServletConfig config 实例变量，并在 init(ServletConfig)方法中把参数 ServletConfig 赋给了实例变量。然后在该类的很多方法中使用了实例变量 config。

如果子类覆盖了 GenericServlet 的 init(StringConfig)方法，那么 this.config=config 这一条语句就会被覆盖了，也就是说 GenericServlet 的实例变量 config 的值为 null，那么所有依赖 config 的方法都不能使用了。如果真的希望完成一些初始化操作，那么去覆盖 GenericServlet 提供的 init()方法，它是没有参数的 init()方法，它会在 init(ServletConfig)方法中被调用。

3 实现了 ServletConfig 接口

GenericServlet 还实现了 ServletConfig 接口，所以可以直接调用 getInitParameter()、getServletContext()等 ServletConfig 的方法。

HttpServlet

1 HttpServlet 概述

HttpServlet 类是 GenericServlet 的子类，它提供了对 HTTP 请求的特殊支持，所以通常我们都会通过继承 HttpServlet 来完成自定义的 Servlet。

2 HttpServlet 覆盖了 service()方法

HttpServlet 类中提供了 service(HttpServletRequest,HttpServletResponse)方法，这个方法是 HttpServlet 自己的方法，不是从 Servlet 继承来的。在 HttpServlet 的 service(ServletRequest,ServletResponse)方法中会把 ServletRequest 和 ServletResponse 强转成 HttpServletRequest 和 HttpServletResponse，然后调用 service(HttpServletRequest,HttpServletResponse)方法，这说明子类可以去覆盖 service(HttpServletRequest,HttpServletResponse)方法即可，这就不用自己去强转请求和响应对象了。

其实子类也不用去覆盖 service(HttpServletRequest,HttpServletResponse)方法，因为 HttpServlet 还要做另一步简化操作，下面会介绍。

HttpServlet.java

```
public abstract class HttpServlet extends GenericServlet {
    protected void service(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
        ……
    }
    @Override
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException {

        HttpServletRequest  request;
        HttpServletResponse response;

        try {
            request = (HttpServletRequest) req;
            response = (HttpServletResponse) res;
        } catch (ClassCastException e) {
```

```
            throw new ServletException("non-HTTP request or response");
        }
        service(request, response);
    }
    ……
}
```

3 doGet()和 doPost()

在 HttpServlet 的 service(HttpServletRequest,HttpServletResponse)方法会去判断当前请求是 GET 还是 POST，如果是 GET 请求，那么会去调用本类的 doGet()方法，如果是 POST 请求会去调用 doPost()方法，这说明我们在子类中去覆盖 doGet()或 doPost()方法即可。

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("hello doGet()...");
    }
}
```

```
public class BServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("hello doPost()...");
    }
}
```

Servlet 细节

1 Servlet 与线程安全

因为一个类型的 Servlet 只有一个实例对象，那么就有可能会现时出一个 Servlet 同时处理多个请求，那么 Servlet 是否为线程安全的呢？答案是:“不是线程安全的”。这说明 Servlet 的工作效率很高，但也存在线程安全问题！

所以我们不应该在 Servlet 中便宜创建成员变量，因为可能会存在一个线程对这个成员变量进行写操作，另一个线程对这个成员变量进行读操作。

2 让服务器在启动时就创建 Servlet

默认情况下，服务器会在某个 Servlet 第一次收到请求时创建它。也可以在 web.xml 中对 Servlet 进行配置，使服务器启动时就创建 Servlet。

```
<servlet>
    <servlet-name>hello1</servlet-name>
    <servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>
    <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>hello1</servlet-name>
    <url-pattern>/hello1</url-pattern>
</servlet-mapping>
<servlet>
    <servlet-name>hello2</servlet-name>
    <servlet-class>cn.itcast.servlet.Hello2Servlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>hello2</servlet-name>
    <url-pattern>/hello2</url-pattern>
</servlet-mapping>
<servlet>
    <servlet-name>hello3</servlet-name>
    <servlet-class>cn.itcast.servlet.Hello3Servlet</servlet-class>
    <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>hello3</servlet-name>
    <url-pattern>/hello3</url-pattern>
</servlet-mapping>
```

在<servlet>元素中配置<load-on-startup>元素可以让服务器在启动时就创建该 Servlet，其中<load-on-startup>元素的值必须是大于等于的整数，它的使用是服务器启动时创建 Servlet 的顺序。上例中，根据<load-on-startup>的值可以得知服务器创建 Servlet 的顺序为 Hello1Servlet、Hello2Servlet、Hello3Servlet。

3 <url-pattern>

<url-pattern>是<servlet-mapping>的子元素，用来指定 Servlet 的访问路径，即 URL。它必须是以“/”开头！

1) 可以在<servlet-mapping>中给出多个<url-pattern>，例如：

```
<servlet-mapping>
```

```
    <servlet-name>AServlet</servlet-name>
    <url-pattern>/AServlet</url-pattern>
    <url-pattern>/BServlet</url-pattern>
  </servlet-mapping>
```

那么这说明一个 Servlet 绑定了两个 URL，无论访问/AServlet 还是/BServlet，访问的都是 AServlet。

2) 还可以在<url-pattern>中使用通配符，所谓通配符就是星号“*”，星号可以匹配任何 URL 前缀或后缀，使用通配符可以命名一个 Servlet 绑定一组 URL，例如：

- <url-pattern>/servlet/*<url-patter>：/servlet/a、/servlet/b，都匹配/servlet/*；
- <url-pattern>*.do</url-pattern>：/abc/def/ghi.do、/a.do，都匹配*.do；
- <url-pattern>/*<url-pattern>：匹配所有 URL；

请注意，通配符要么为前缀，要么为后缀，不能出现在 URL 中间位置，也不能只有通配符。例如：/*.do 就是错误的，因为星号出现在 URL 的中间位置上了。*.*也是不对的，因为一个 URL 中最多只能出现一个通配符。

注意，通配符是一种模糊匹配 URL 的方式，如果存在更具体的<url-pattern>，那么访问路径会去匹配具体的<url-pattern>。例如：

```
<servlet>
    <servlet-name>hello1</servlet-name>
    <servlet-class>cn.itcast.servlet.Hello1Servlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>hello1</servlet-name>
    <url-pattern>/servlet/hello1</url-pattern>
</servlet-mapping>
<servlet>
    <servlet-name>hello2</servlet-name>
    <servlet-class>cn.itcast.servlet.Hello2Servlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>hello2</servlet-name>
    <url-pattern>/servlet/*</url-pattern>
</servlet-mapping>
```

当访问路径为 http://localhost:8080/hello/servlet/hello1 时，因为访问路径即匹配 hello1 的<url-pattern>，又匹配 hello2 的<url-pattern>，但因为 hello1 的<url-pattern>中没有通配符，所以优先匹配，即设置 hello1。

4 web.xml 文件的继承（了解）

每个完整的 JavaWeb 应用中都需要有 web.xml，但我们不知道所有的 web.xml 文件都有一个共

同的父文件，它在 Tomcat 的 conf/web.xml 路径。

conf/web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
    <servlet>
      <servlet-name>default</servlet-name>

<servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>
      <init-param>
          <param-name>debug</param-name>
          <param-value>0</param-value>
      </init-param>
      <init-param>
          <param-name>listings</param-name>
          <param-value>false</param-value>
      </init-param>
      <load-on-startup>1</load-on-startup>
   </servlet>

   <servlet>
      <servlet-name>jsp</servlet-name>
      <servlet-class>org.apache.jasper.servlet.JspServlet</servlet-class>
      <init-param>
          <param-name>fork</param-name>
          <param-value>false</param-value>
      </init-param>
      <init-param>
          <param-name>xpoweredBy</param-name>
          <param-value>false</param-value>
      </init-param>
      <load-on-startup>3</load-on-startup>
   </servlet>

   <servlet-mapping>
      <servlet-name>default</servlet-name>
      <url-pattern>/</url-pattern>
   </servlet-mapping>
```

```
    <servlet-mapping>
        <servlet-name>jsp</servlet-name>
        <url-pattern>*.jsp</url-pattern>
        <url-pattern>*.jspx</url-pattern>
    </servlet-mapping>

    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

    <!-- 这里省略了大概4000多行的MIME类型的定义,这里只给出两种MIME类型的定义 -->
    <mime-mapping>
        <extension>bmp</extension>
        <mime-type>image/bmp</mime-type>
    </mime-mapping>
    <mime-mapping>
        <extension>htm</extension>
        <mime-type>text/html</mime-type>
    </mime-mapping>

    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.htm</welcome-file>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

ServletContext

1 ServletContext 概述

服务器会为每个应用创建一个 ServletContext 对象：

- ServletContext 对象的创建是在服务器启动时完成的；
- ServletContext 对象的销毁是在服务器关闭时完成的。

ServletContext 对象的作用是在整个 Web 应用的动态资源之间共享数据！例如在 AServlet 中向 ServletContext 对象中保存一个值，然后在 BServlet 中就可以获取这个值，这就是共享数据了。

2 获取 ServletContext

在 Servlet 中获取 ServletContext 对象：

- 在 void init(ServletConfig config) 中： ServletContext context = config.getServletContext();，ServletConfig 类的 getServletContext()方法可以用来获取 ServletContext 对象；

在 GenericeServlet 或 HttpServlet 中获取 ServletContext 对象：

- GenericServlet 类有 getServletContext()方法，所以可以直接使用 this.getServletContext()来获取；

```
public class MyServlet implements Servlet {
    public void init(ServletConfig config) {
        ServletContext context = config.getServletContext();
    }
    …
}
```

```
public class MyServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        ServletContext context = this.getServletContext();
    }
}
```

3 域对象的功能

ServletContext 是 JavaWeb 四大域对象之一：

- PageContext；
- ServletRequest；
- HttpSession；
- ServletContext；

所有域对象都有存取数据的功能，因为域对象内部有一个 Map，用来存储数据，下面是 ServletContext 对象用来操作数据的方法：

- void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute(“xxx”, “XXX”)，在 ServletContext 中保存了一个域属性，域属性名称为 xxx，域属性的值为 XXX。请注意，如果多次调用该方法，并且使用相同的 name，那么会覆盖上一次的值，这一特性与 Map 相同；
- Object getAttribute(String name)：用来获取 ServletContext 中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)servletContext.getAttribute(“xxx”);，获取名为 xxx 的域属性；
- void removeAttribute(String name)：用来移除 ServletContext 中的域属性，如果参数 name 指定的域属性不存在，那么本方法什么都不做；
- Enumeration getAttributeNames()：获取所有域属性的名称；

4 获取应用初始化参数

还可以使用 ServletContext 来获取在 web.xml 文件中配置的应用初始化参数！注意，应用初始化参数与 Servlet 初始化参数不同：

web.xml

```
<web-app ...>
  ...
  <context-param>
    <param-name>paramName1</param-name>
    <param-value>paramValue1</param-value>
  </context-param>
  <context-param>
    <param-name>paramName2</param-name>
    <param-value>paramValue2</param-value>
  </context-param>
</web-app>
```

```
        ServletContext context = this.getServletContext();
        String value1 = context.getInitParameter("paramName1");
        String value2 = context.getInitParameter("paramName2");
        System.out.println(value1 + ", " + value2);

        Enumeration names = context.getInitParameterNames();
        while(names.hasMoreElements()) {
            System.out.println(names.nextElement());
        }
```

5 获取资源相关方法

5.1 获取真实路径

还可以使用 ServletContext 对象来获取 Web 应用下的资源，例如在 hello 应用的根目录下创建 a.txt 文件，现在想在 Servlet 中获取这个资源，就可以使用 ServletContext 来获取。

- 获取 a.txt 的真实路径：String realPath = servletContext.getRealPath(“/a.txt”)，realPath 的值为 a.txt 文件的绝对路径：F:\tomcat6\webapps\hello\a.txt；
- 获取 b.txt 的真实路径：String realPath = servletContext.getRealPath(“/WEB-INF/b.txt”)；

5.2 获取资源流

不只可以获取资源的路径，还可以通过 ServletContext 获取资源流，即把资源以输入流的方式获取：

- 获取 a.txt 资源流：InputStream in = servletContext.getResourceAsStream("/a.txt");
- 获取 b.txt 资源流：InputStream in = servletContext.getResourceAsStream("/WEB-INF/b.txt");

5.3 获取指定目录下所有资源路径

还可以使用 ServletContext 获取指定目录下所有资源路径，例如获取/WEB-INF 下所有资源的路径：

```
        Set set = context.getResourcePaths("/WEB-INF");
        System.out.println(set);
```

```
[/WEB-INF/lib/, /WEB-INF/classes/, /WEB-INF/b.txt, /WEB-INF/web.xml]
```

注意，本方法必须以“/”开头!!!

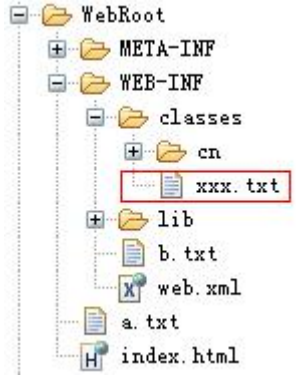
6 练习：访问量统计

相信各位一定见过很多访问量统计的网站，即“本页面被访问过 XXX 次”。因为无论是哪个用户访问指定页面，都会累计访问量，所以这个访问量统计应该是整个项目共享的！很明显，这需要使用 ServletContext 来保存访问量。

```
        ServletContext application  = this.getServletContext();
        Integer count = (Integer)application.getAttribute("count");
        if(count == null) {
            count = 1;
        } else {
            count++;
        }
        response.setContentType("text/html;charset=utf-8");
        response.getWriter().print("<h1>本页面一共被访问" + count + "次! </h1>");
        application.setAttribute("count", count);
```

获取类路径下资源

这里要讲的是获取类路径下的资源，对于 JavaWeb 应用而言，就是获取 classes 目录下的资源。



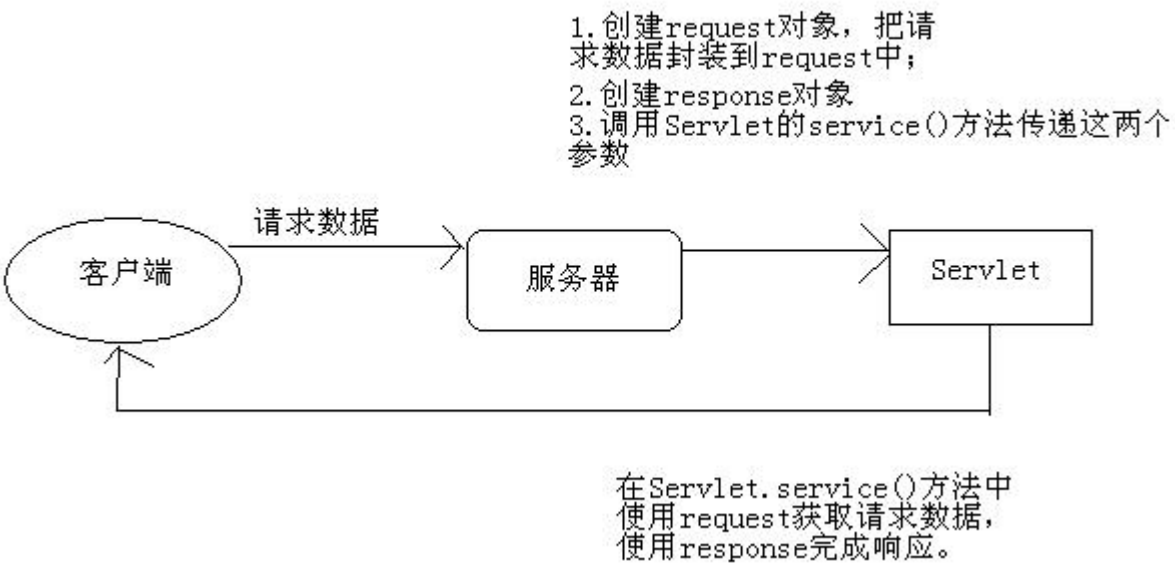

```
        InputStream in = this.getClass().getResourceAsStream("/xxx.txt");
        System.out.println(IOUtils.toString(in));
```

```
        InputStream in =
this.getClass().getClassLoader().getResourceAsStream("xxx.txt");
        System.out.println(IOUtils.toString(in));
```

- Class 类的 getResourceAsStream(String path)：
  - 路径以“/”开头，相对 classes 路径；
  - 路径不以“/”开头，相对当前 class 文件所有路径，例如在 cn.itcast.servlet.MyServlet 中执行，那么相对/classes/cn/itcast/servlet/路径；
- ClassLoader 类的 getResourceAsStream(String path)：
  - 相对 classes 路径；

day05

请求响应流程图



response

1 response 概述

response 是 Servlet.service 方法的一个参数，类型为 javax.servlet.http.HttpServletResponse。在客户端发出每个请求时，服务器都会创建一个 response 对象，并传入给 Servlet.service()方法。response 对象是用来对客户端进行响应的，这说明在 service()方法中使用 response 对象可以完成对客户端的响应工作。

response 对象的功能分为以下四种：

- 设置响应头信息；
- 发送状态码；
- 设置响应正文；
- 重定向；

2 response 响应正文

response 是响应对象，向客户端输出响应正文（响应体）可以使用 response 的响应流，repsonse 一共提供了两个响应流对象：

- PrintWriter out = response.getWriter()：获取字符流；
- ServletOutputStream out = response.getOutputStream()：获取字节流；

当然，如果响应正文内容为字符，那么使用 response.getWriter()，如果响应内容是字节，例如下载时，那么可以使用 response.getOutputStream()。

注意，在一个请求中，不能同时使用这两个流！也就是说，要么你使用 repsonse.getWriter()，要么使用 response.getOutputStream()，但不能同时使用这两个流。不然会抛出 IllegalStateException 异常。

2.1 字符响应流

- 字符编码

在使用 response.getWriter()时需要注意默认字符编码为 ISO-8859-1，如果希望设置字符流的字符编码为 utf-8，可以使用 response.setCharaceterEncoding(“utf-8”)来设置。这样可以保证输出给客户端的字符都是使用 UTF-8 编码的！

但客户端浏览器并不知道响应数据是什么编码的！如果希望通知客户端使用 UTF-8 来解读响应数据，那么还是使用 response.setContentType("text/html;charset=utf-8")方法比较好，因为这个方法不只会调用 response.setCharaceterEncoding(“utf-8”)，还会设置 content-type 响应头，客户端浏览器会使用 content-type 头来解读响应数据。

- 缓冲区

response.getWriter()是 PrintWriter 类型，所以它有缓冲区，缓冲区的默认大小为 8KB。也就是说，在响应数据没有输出 8KB 之前，数据都是存放在缓冲区中，而不会立刻发送到客户端。当 Servlet 执行结束后，服务器才会去刷新流，使缓冲区中的数据发送到客户端。

如果希望响应数据马上发送给客户端：

- 向流中写入大于 8KB 的数据；
- 调用 response.flushBuffer()方法来手动刷新缓冲区；

3 设置响应头信息

可以使用 response 对象的 setHeader()方法来设置响应头！使用该方法设置的响应头最终会发送给客户端浏览器！

- response.setHeader(“content-type”, “text/html;charset=utf-8”)：设置 content-type 响应头，该头的作用是告诉浏览器响应内容为 html 类型，编码为 utf-8。而且同时会设置 response 的字符流编码为 utf-8，即 response.setCharaceterEncoding(“utf-8”)；
- response.setHeader("Refresh","5; URL=http://www.itcast.cn")：5 秒后自动跳转到传智主页。

4 设置状态码及其他方法

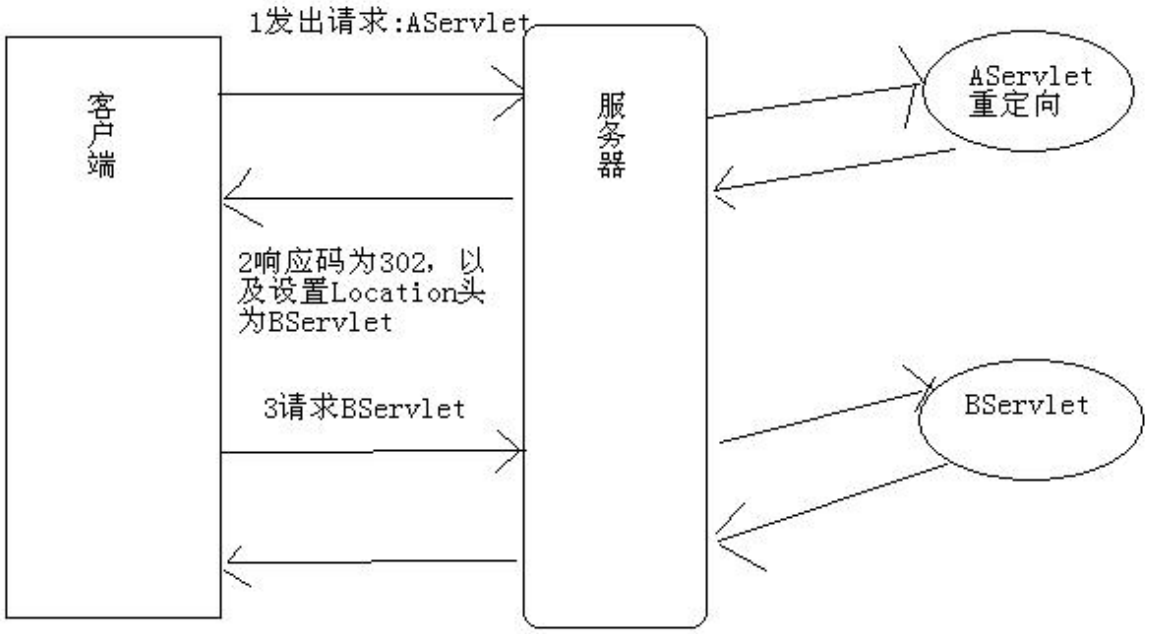
- response.setContentType("text/html;charset=utf-8") ：等同与调用 response.setHeader(“content-type”, “text/html;charset=utf-8”)；
- response.setCharacterEncoding(“utf-8”)：设置字符响应流的字符编码为 utf-8；
- response.setStatus(200)：设置状态码；
- response.sendError(404, “您要查找的资源不存在”)：当发送错误状态码时，Tomcat 会跳转到固定的错误页面去，但可以显示错误信息。

5 重定向

5.1 什么是重定向

当你访问 http://www.sun.com 时，你会发现浏览器地址栏中的 URL 会变成 http://www.oracle.com/us/sun/index.htm，这就是重定向了。

重定向是服务器通知浏览器去访问另一个地址，即再发出另一个请求。



5.2 完成重定向

响应码为 200 表示响应成功，而响应码为 302 表示重定向。所以完成重定向的第一步就是设置响应码为 302。

因为重定向是通知浏览器再第二个请求，所以浏览器需要知道第二个请求的 URL，所以完成重定向的第二步是设置 Location 头，指定第二个请求的 URL 地址。

```
public class AServlet extends HttpServlet {
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setStatus(302);
        response.setHeader("Location", "http://www.itcast.cn");
    }
}
```

上面代码的作用是：当访问 AServlet 后，会通知浏览器重定向到传智主页。客户端浏览器解析到响应码为 302 后，就知道服务器让它重定向，所以它会马上获取响应头 Location，然后发出第二个请求。

5.3 便捷的重定向方式

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.sendRedirect("http://www.itcast.cn");
    }
}
```

response.sendRedirect()方法会设置响应头为 302，以设置 Location 响应头。

如果要重定向的 URL 是在同一个服务器内，那么可以使用相对路径，例如：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.sendRedirect("/hello/BServlet");
    }
}
```

重定向的 URL 地址为：http://localhost:8080/hello/BServlet。

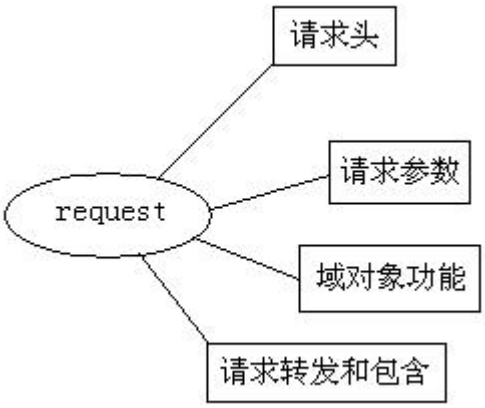
5.4 重定向小结

- 重定向是两次请求；
- 重定向的 URL 可以是其他应用，不局限于当前应用；
- 重定向的响应头为 302，并且必须要有 Location 响应头；
- 重定向就不要再使用 response.getWriter()或 response.getOutputStream()输出数据，不然可能会出现异常；

request

1 request 概述

request 是 Servlet.service()方法的一个参数，类型为 javax.servlet.http.HttpServletRequest。在客户端发出每个请求时，服务器都会创建一个 request 对象，并把请求数据封装到 request 中，然后在调用 Servlet.service()方法时传递给 service()方法，这说明在 service()方法中可以通过 request 对象来获取请求数据。



request 的功能可以分为以下几种：

- 封装了请求头数据；
- 封装了请求正文数据，如果是 GET 请求，那么就没有正文；
- request 是一个域对象，可以把它当成 Map 来添加获取数据；
- request 提供了请求转发和请求包含功能。

2 request 域方法

request 是域对象！在 JavaWeb 中一共四个域对象，其中 ServletContext 就是域对象，它在整个应用中只创建一个 ServletContext 对象。request 其中一个，request 可以在一个请求中共享数据。

一个请求会创建一个 request 对象，如果在一个请求中经历了多个 Servlet，那么多个 Servlet 就可以使用 request 来共享数据。现在我们还不知道如何在一个请求中经历之个 Servlet，后面在学习请求转发和请求包含后就知道了。

下面是 request 的域方法：

- void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：servletContext.setAttribute("xxx", "XXX")，在 request 中保存了一个域属性，域属性名称为 xxx，域属性的值为 XXX。请注意，如果多次调用该方法，并且使用相同的 name，那么会覆盖上一次的值，这一特性与 Map 相同；
- Object getAttribute(String name)：用来获取 request 中的数据，当前在获取之前需要先去存储才行，例如：String value = (String)request.getAttribute("xxx");，获取名为 xxx 的域属性；
- void removeAttribute(String name)：用来移除 request 中的域属性，如果参数 name 指定的域属性不存在，那么本方法什么都不做；
- Enumeration getAttributeNames()：获取所有域属性的名称；

3 request 获取请求头数据

request 与请求头相关的方法有：

- String getHeader(String name)：获取指定名称的请求头；
- Enumeration getHeaderNames()：获取所有请求头名称；
- int getIntHeader(String name)：获取值为 int 类型的请求头。

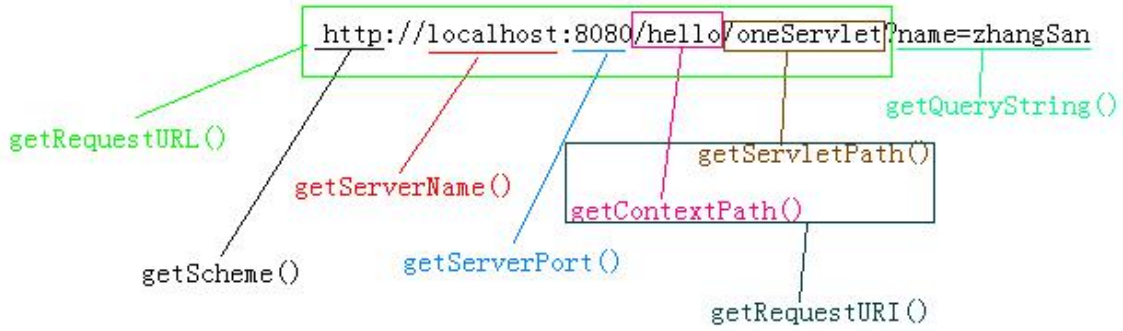
4 request 获取请求相关的其它方法

request 中还提供了与请求相关的其他方法，有些方法是为了我们更加便捷的方法请求头数据而设计，有些是与请求 URL 相关的方法。

- int getContentLength()：获取请求体的字节数，GET 请求没有请求体，没有请求体返回-1；
- String getContentType()：获取请求类型，如果请求是 GET，那么这个方法返回 null；如果是 POST 请求，那么默认为 application/x-www-form-urlencoded，表示请求体内容使用了 URL 编码；
- String getMethod()：返回请求方法，例如：GET
- Locale getLocale()：返回当前客户端浏览器的 Locale。java.util.Locale 表示国家和言语，这个东西在国际化中很有用；
- String getCharacterEncoding()：获取请求体编码，如果没有 setCharacterEncoding()，那么返回 null，表示使用 ISO-8859-1 编码；
- void setCharacterEncoding(String code)：设置请求编码，只对请求体有效！注意，对于 GET 而言，没有请求体!!! 所以此方法只能对 POST 请求中的参数有效！
- String getContextPath()：返回上下文路径，例如：/hello
- String getQueryString()：返回请求 URL 中的参数，例如：name=zhangSan
- String getRequestURI()：返回请求 URI 路径，例如：/hello/oneServlet
- StringBuffer getRequestURL()：返回请求 URL 路径，例如：http://localhost/hello/oneServlet，即返回除了参数以外的路径信息；
- String getServletPath()：返回 Servlet 路径，例如：/oneServlet
- String getRemoteAddr()：返回当前客户端的 IP 地址；
- String getRemoteHost()：返回当前客户端的主机名，但这个方法的实现还是获取 IP 地址；
- String getScheme()：返回请求协议，例如：http；
- String getServerName()：返回主机名，例如：localhost
- int getServerPort()：返回服务器端口号，例如：8080

```
http://localhost:8080/hello/oneServlet?name=zhangSan
```

```
request.getContextPath(): /hello
request.getQueryString(): name=hello
request.getRequestURI(): /hello/oneServlet
request.getRequestURL(): http://localhost/hello/oneServlet
request.getServletPath(): /oneServlet
request.getScheme(): http
request.getServerName(): localhost
request.getServerPort(): 80
```




```
System.out.println("request.getContentLength(): " + request.getContentLength());
System.out.println("request.getContentType(): " + request.getContentType());
System.out.println("request.getContextPath(): " + request.getContextPath());
System.out.println("request.getMethod(): " + request.getMethod());
System.out.println("request.getLocale(): " + request.getLocale());

System.out.println("request.getQueryString(): " + request.getQueryString());
System.out.println("request.getRequestURI(): " + request.getRequestURI());
System.out.println("request.getRequestURL(): " + request.getRequestURL());
System.out.println("request.getServletPath(): " + request.getServletPath());
System.out.println("request.getRemoteAddr(): " + request.getRemoteAddr());
System.out.println("request.getRemoteHost(): " + request.getRemoteHost());
System.out.println("request.getRemotePort(): " + request.getRemotePort());
System.out.println("request.getScheme(): " + request.getScheme());
System.out.println("request.getServerName(): " + request.getServerName());
System.out.println("request.getServerPort(): " + request.getServerPort());
```

4.1 案例：request.getRemoteAddr()：封 IP

可以使用 request.getRemoteAddr()方法获取客户端的 IP 地址，然后判断 IP 是否为禁用 IP。

```
String ip = request.getRemoteAddr();
System.out.println(ip);
if(ip.equals("127.0.0.1")) {
    response. getWriter().print("您的IP已被禁止！");
} else {
```

```
        response.getWriter().print("Hello!");
    }
```

5 request 获取请求参数

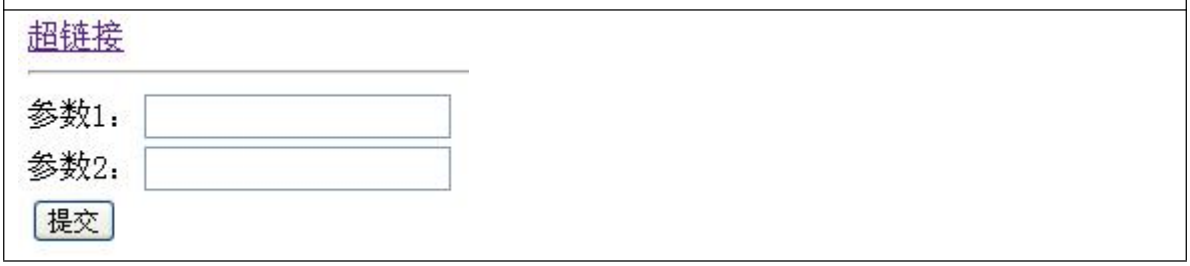
最为常见的客户端传递参数方式有两种：

- 浏览器地址栏直接输入：一定是 GET 请求；
- 超链接：一定是 GET 请求；
- 表单：可以是 GET，也可以是 POST，这取决与<form>的 method 属性值；

GET 请求和 POST 请求的区别：

- GET 请求：
  - 请求参数会在浏览器的地址栏中显示，所以不安全；
  - 请求参数长度限制长度在 1K 之内；
  - GET 请求没有请求体，无法通过 request.setCharacterEncoding()来设置参数的编码；
- POST 请求：
  - 请求参数不会显示浏览器的地址栏，相对安全；
  - 请求参数长度没有限制；

```
<a href="/hello/ParamServlet?p1=v1&p2=v2">超链接</a>
<hr/>
<form action="/hello/ParamServlet" method="post">
参数1：<input type="text" name="p1"/><br/>
参数2：<input type="text" name="p2"/><br/>
<input type="submit" value="提交"/>
</form>
```



下面是使用 request 获取请求参数的 API：

- String getParameter(String name)：通过指定名称获取参数值；

```
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String v1 = request.getParameter("p1");
```

```
        String v2 = request.getParameter("p2");
        System.out.println("p1=" + v1);
        System.out.println("p2=" + v2);
    }

    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String v1 = request.getParameter("p1");
        String v2 = request.getParameter("p2");
        System.out.println("p1=" + v1);
        System.out.println("p2=" + v2);
    }
```

- String[] getParameterValues(String name)：当多个参数名称相同时，可以使用方法来获取；

```
<a href="/hello/ParamServlet?name=zhangSan&name=liSi">超链接</a>
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String[] names = request.getParameterValues("name");
        System.out.println(Arrays.toString(names));
    }
```

- Enumeration getParameterNames()：获取所有参数的名字；

```
    <form action="/hello/ParamServlet" method="post">
    参数1：<input type="text" name="p1"/><br/>
    参数2：<input type="text" name="p2"/><br/>
    <input type="submit" value="提交"/>
    </form>
```

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        Enumeration names = request.getParameterNames();
        while(names.hasMoreElements()) {
            System.out.println(names.nextElement());
        }
    }
```

- Map getParameterMap()：获取所有参数封装到 Map 中，其中 key 为参数名，value 为参数值，因为一个参数名称可能有多个值，所以参数值是 String[]，而不是 String。

```
<a href="/day05_1/ParamServlet?p1=v1&p1=vv1&p2=v2&p2=vv2">超链接</a>
```

```
        Map<String,String[]> paramMap = request.getParameterMap();
        for(String name : paramMap.keySet()) {
```

```
            String[] values = paramMap.get(name);
            System.out.println(name + ": " + Arrays.toString(values));
        }
```

```
p2: [v2, vv2]
p1: [v1, vv1]
```

6 请求转发和请求包含

无论是请求转发还是请求包含，都表示由多个 Servlet 共同来处理一个请求。例如 Servlet1 来处理请求，然后 Servlet1 又转发给 Servlet2 来继续处理这个请求。

6.1 请求转发

在 AServlet 中，把请求转发到 BServlet：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("AServlet");
        RequestDispatcher rd = request.getRequestDispatcher("/BServlet");
        rd.forward(request, response);
    }
}
```

```
public class BServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("BServlet");
    }
}
```

```
Aservlet
BServlet
```

6.2 请求包含

在 AServlet 中，把请求包含到 BServlet：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("AServlet");
        RequestDispatcher rd = request.getRequestDispatcher("/BServlet");
```

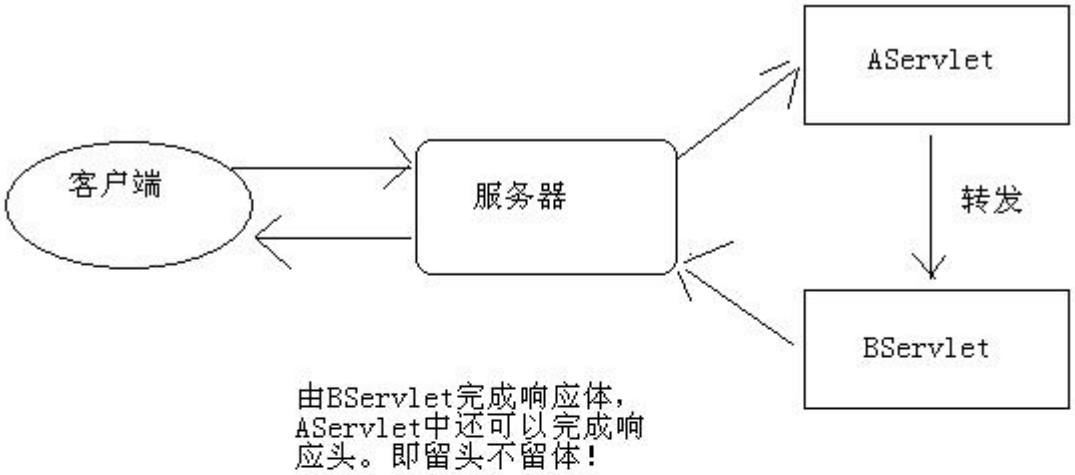
```
        rd.include(request, response);
    }
}
```

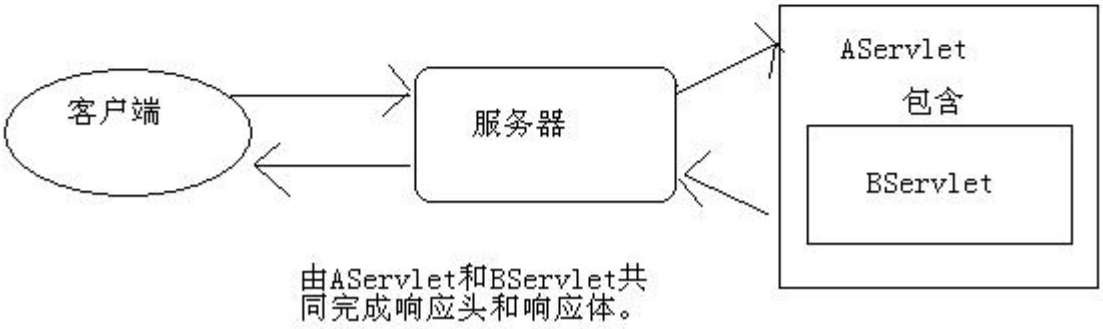
```
public class BServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        System.out.println("BServlet");
    }
}
```

```
Aservlet
BServlet
```

6.3 请求转发与请求包含比较

- 如果在 AServlet 中请求转发到 BServlet，那么在 AServlet 中就不允许再输出响应体，即不能再使用 response.getWriter()和 response.getOutputStream()向客户端输出，这一工作应该由 BServlet 来完成；如果是使用请求包含，那么没有这个限制；
- 请求转发虽然不能输出响应体，但还是可以设置响应头的，例如：response.setContentType(”text/html;charset=utf-8”);
- 请求包含大多是应用在 JSP 页面中，完成多页面的合并；
- 请求请求大多是应用在 Servlet 中，转发目标大多是 JSP 页面；





6.4 请求转发与重定向比较

- 请求转发是一个请求，而重定向是两个请求；
- 请求转发后浏览器地址栏不会有变化，而重定向会有变化，因为重定向是两个请求；
- 请求转发的目标只能是本应用中的资源，重定向的目标可以是其他应用；
- 请求转发对 AServlet 和 BServlet 的请求方法是相同的，即要么都是 GET，要么都是 POST，因为请求转发是一个请求；
- 重定向的第二个请求一定是 GET；

路径

1 与路径相关的操作

- 超链接
- 表单
- 转发

- 包含
- 重定向
- <url-pattern>
- ServletContext 获取资源
- Class 获取资源
- ClassLoader 获取资源

2 客户端路径

超链接、表单、重定向都是客户端路径，客户端路径可以分为三种方式：

- 绝对路径；
- 以“/”开头的相对路径；
- 不以“/”开头的相对路径；

例如：http://localhost:8080/hello1/pages/a.html 中的超链接和表单如下：

```
绝对路径：<a href="http://localhost:8080/hello2/index.html">链接1</a>
客户端路径：<a href="/hello3/pages/index.html">链接2</a>
相对路径：<a href="index.html">链接3</a>
<hr/>
绝对路径：
<form action="http://localhost:8080/hello2/index.html">
  <input type="submit" value="表单1"/>
</form>
客户端路径：
<form action="/hello2/index.html">
  <input type="submit" value="表单2"/>
</form>
相对路径：
<form action="index.html">
  <input type="submit" value="表单3"/>
</form>
```

- 链接 1 和表单 1：没什么可说的，它使用绝对路径；
- 链接 2 和表单 2：以“/”开头，相对主机，与当前 a.html 的主机相同，即最终访问的页面为 http://localhost:8080/hello2/index.html；
- 链接 3 和表单 3：不以“/”开头，相对当前页面的路径，即 a.html 所有路径，即最终访问的路径为：http://localhost:8080/hello1/pages/index.html；

重定向 1：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.sendRedirect("/hello/index.html");
```

```
    }
}
```

假设访问 AServlet 的路径为：http://localhost:8080/hello/servlet/AServlet
因为路径以“/”开头，所以相对当前主机，即 http://localhost:8080/hello/index.html。

重定向 2：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.sendRedirect("index.html");
    }
}
```

假设访问 AServlet 的路径为：http://localhost:8080/hello/servlet/AServlet
因为路径不以“/”开头，所以相对当前路径，即 http://localhost:8080/hello/servlet/index.html

2.1 建议使用“/”

强烈建议使用“/”开头的路径，这说明在页面中的超链接和表单都要以“/”开头，后面是当前应用的名称，再是访问路径：

```
<form action="/hello/servlet/AServlet">
</form>
<a href="/hello/b.html">链接</a>
```

其中/hello 是当前应用名称，这也说明如果将来修改了应用名称，那么页面中的所有路径也要修改，这一点确实是个问题。这一问题的处理方案会在学习了 JSP 之后讲解！

在 Servlet 中的重定向也建议使用“/”开头。同理，也要给出应用的名称！例如：

```
response.sendRedirect("/hello/BServlet");
```

其中/hello 是当前应用名，如果将来修改了应用名称，那么也要修改所有重定向的路径，这一问题的处理方案是使用 request.getContextPath()来获取应用名称。

```
response.sendRedirect(request.getContextPath() + "/BServlet");
```

3 服务器端路径

服务器端路径必须是相对路径，不能是绝对路径。但相对路径有两种形式：

- 以“/”开头；
- 不以“/”开头；

其中请求转发、请求包含都是服务器端路径，服务器端路径与客户端路径的区别是：

- 客户端路径以“/”开头：相对当前主机；
- 服务器端路径以“/”开头：相对当前应用；

转发 1：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.getRequestDispatcher("/BServlet").forward(request, response);
    }
}
```

假设访问 AServlet 的路径为：http://localhost:8080/hello/servlet/AServlet
因为路径以“/”开头，所以相对当前应用，即 http://localhost:8080/hello/BServlet。

转发 2：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.getRequestDispatcher("BServlet").forward(request, response);
    }
}
```

假设访问 AServlet 的路径为：http://localhost:8080/hello/servlet/AServlet
因为路径不以“/”开头，所以相对当前应用，即 http://localhost:8080/hello/servlet/BServlet。

4 <url-pattern>路径

<url-pattern>必须使用“/”开头，并且相对的是当前应用。

5 ServletContext 获取资源

必须是相对路径，可以“/”开头，也可以不使用“/”开头，但无论是否使用“/”开头都是相对当前应用路径。

例如在 AServlet 中获取资源，AServlet 的路径路径为：http://localhost:8080/hello/servlet/AServlet：

```
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String path1 = this.getServletContext().getRealPath("a.txt");
```

```
        String path2 = this.getServletContext().getRealPath("/a.txt");
        System.out.println(path1);
        System.out.println(path2);
    }
}
```

path1 和 path2 是相同的结果：http://localhost:8080/hello/a.txt

6 Class 获取资源

Class 获取资源也必须是相对路径，可以“/”开头，也可以不使用“/”开头。

```
package cn.itcast;

import java.io.InputStream;

public class Demo {
    public void fun1() {
        InputStream in = Demo.class.getResourceAsStream("/a.txt");
    }

    public void fun2() {
        InputStream in = Demo.class.getResourceAsStream("a.txt");
    }
}
```

其中 fun1()方法获取资源时以“/”开头，那么相对的是当前类路径，即/hello/WEB-INF/classes/a.txt 文件；

其中 fun2()方法获取资源时没有以“/”开头，那么相对当前 Demo.class 所在路径，因为 Demo 类在 cn.itcast 包下，所以资源路径为：/hello/WEB-INF/classes/cn/itcast/a.txt。

7 ClassLoader 获取资源

ClassLoader 获取资源也必须是相对路径，可以“/”开头，也可以不使用“/”开头。但无论是否以“/”开头，资源都是相对当前类路径。

```
public class Demo {
    public void fun1() {
        InputStream in =
Demo.class.getClassLoader().getResourceAsStream("/a.txt");
    }

    public void fun2() {
        InputStream in =
```

```
Demo.class.getClassLoader().getResourceAsStream("a.txt");
    }
}
```

fun1()和 fun2()方法的资源都是相对类路径，即 classes 目录，即/hello/WEB-INF/classes/a.txt

编码

1 请求编码

1.1 直接在地址栏中给出中文

请求数据是由客户端浏览器发送服务器的，请求数据的编码是由浏览器决定的。例如在浏览器地址栏中给出：http://localhost:8080/hello/AServlet?name=传智，那么其中“传智”是什么编码的呢？不同浏览器使用不同的编码，所以这是不确定的！

- IE：使用 GB2312；
- FireFox：使用 GB2312；
- Chrome：使用 UTF-8；

通常没有哪个应用要求用户在浏览器地址栏中输入请求数据的，所以大家只需了解一下即可。

1.2 在页面中发出请求

通常向服务器发送请求数据都需要先请求一个页面，然后用户在页面中输入数据。页面中有超链接和表单，通过超链接和表单就可以向服务器发送数据了。

因为页面是服务器发送到客户端浏览器的，所以这个页面本身的编码由服务器决定。而用户在页面中输入的数据也是由页面本身的编码决定的。

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>index.html</title>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  </head>

  <body>
<form action="/hello/servlet/AServlet">
  名称:<input type="text" name="name"/>
  <input type="submit" value="提交"/>
</form>
```

```
<a href="/hello/servlet/AServlet?name=传智">链接</a>
  </body>
</html>
```

当用户在 index.html 页面中输入数据时，都是 UTF-8 列表的。因为这个页面本身就是 UTF-8 编码的！

页面的编译就是页面中输入数据的编码。

1.3 GET 请求解读编码

当客户端通过 GET 请求发送数据给服务器时，使用 request.getParameter()获取的数据是被服务器误认为 ISO-8859-1 编码的，也就是说客户端发送过来的数据无论是 UTF-8 还是 GBK，服务器都认为是 ISO-8859-1，这就说明我们需要在使用 request.getParameter()获取数据后，再转发成正确的编码。

例如客户端以 UTF-8 发送的数据，使用如下转码方式：

String name = request.getParameter(“name”);

name = new String(name.getBytes(“iso-8859-1”), “utf-8”);

1.4 POST 请求解读编码

当客户端通过 POST 请求发送数据给服务器时，可以在使用 request.getParameter()获取请求参数之前先通过 request.setCharacterEncoding()来指定编码，然后再使用 reuqest.getParameter()方法来获取请求参数，那么就是用指定的编码来读取了。

也就是说，如果是 POST 请求，服务器可以指定编码！但如果没有指定编码，那么默认还是使用 ISO-8859-1 来解读。

request.setCharacterEncoding(“utf-8”);

String name = request.getParameter(“name”);

2 响应编码

响应：服务器发送给客户端数据！响应是由 response 对象来完成，如果响应的数据不是字符数据，那么就无需去考虑编码问题。当然，如果响应的数据是字符数据，那么就一定要考虑编码的问题了。

response.getWriter().print(“传智”);

上面代码因为没有设置 repsonse.getWriter()字符流的编码，所以服务器使用默认的编码（ISO-8859-1）来处理，因为 ISO-8859-1 不支持中文，所以一定会出现编码的。

所以在使用 response.getWriter()发送数据之前，一定要设置 response.getWriter()的编码，这需要使用 response.setCharacterEncoding()方法：

response.setCharacterEncoding(“utf-8”);

response.getWriter().print(“传智”);

上面代码因为在使用 response.getWriter()输出之前已经设置了编码，所以输出的数据为 utf-8 编码。但是，因为没有告诉浏览器使用什么编码来读取响应数据，所以很可能浏览器会出现错误的解读，那么还是会出现乱码的。当然，通常浏览器都支持来设置当前页面的编码，如果用户在看到编

码时，去设置浏览器的编码，如果设置的正确那么乱码就会消失。但是我们不能让用户总去自己设置编码，而且应该直接通知浏览器，服务器发送过来的数据是什么编码，这样浏览器就直接使用服务器告诉他的编码来解读！这需要使用 content-type 响应头。

response.setContentType(“text/html;charset=utf-8”);

response.getWriter().print(“传智”);

上面代码使用 setContentType()方法设置了响应头 content-type 编码为 utf-8，这不只是在响应中添加了响应头，还等于调用了一次 response.setCharacterEncoding(“utf-8”)，也就是说，通过我们只需要调用一次 response.setContentType(“text/html;charset=utf-8”) 即可，而无需再去调用 response.setCharacterEncoding(“utf-8”)了。

在静态页面中，使用<meta>来设置 content-type 响应头，例如：

<meta http-equiv="content-type" content="text/html; charset=UTF-8">

3 URL 编码

通过页面传输数据给服务器时，如果包含了一些特殊字符是无法发送的。这时就需要先把要发送的数据转换成 URL 编码格式，再发送给服务器。

其实需要我们自己动手给数据转换成 URL 编码的只有 GET 超链接，因为表单发送数据会默认使用 URL 编码，也就是说，不用我们自己来编码。

例如：“传智”这两个字通过 URL 编码后得到的是：“%E4%BC%A0%E6%99%BA”。URL 编码是先需要把“传智”转换成字节，例如我们现在使用 UTF-8 把“传智”转换成字符，得到的结果是：“[-28, -68, -96, -26, -103, -70]”，然后再把所有负数加上 256，得到[228, 188, 160, 230, 153, 186]，再把每个 int 值转换成 16 进制，得到[E4, BC, A0, E6, 99, BA]，最后再每个 16 进制的整数前面加上“%”。

通过 URL 编码，把“传智”转换成了“%E4%BC%A0%E6%99%BA”，然后发送给服务器！服务器会自动识别出数据是使用 URL 编码过的，然后会自动把数据转换回来。

当然，在页面中我们不需要自己去通过上面的过程把“传智”转换成“%E4%BC%A0%E6%99%BA”，而是使用 Javascript 来完成即可。当后面我们学习了 JSP 后，就不用再使用 Javascript 了。

```
<script type="text/javascript">
  function _go() {
      location = "/day05_2/AServlet?name=" + encodeURIComponent("传智+播客");
  }
</script>
<a href="javascript:_go();">链接</a>
```

因为 URL 默认只支持 ISO-8859-1，这说明在 URL 中出现中文和一些特殊字符可能无法发送到服务器。所以我们需要对包含中文或特殊字符的 URL 进行 URL 编码。

服务器会自动识别数据是否使用了 URL 编码，如果使用了服务器会自动把数据解码，无需我们自己动手解码。

String s = “传智”;

s = URLEncoder.encode(s, “utf-8”);// %E4%BC%A0%E6%99%BA

s = URLDecoderdecode(s, “utf-8”);//传智

day06

JSP 入门

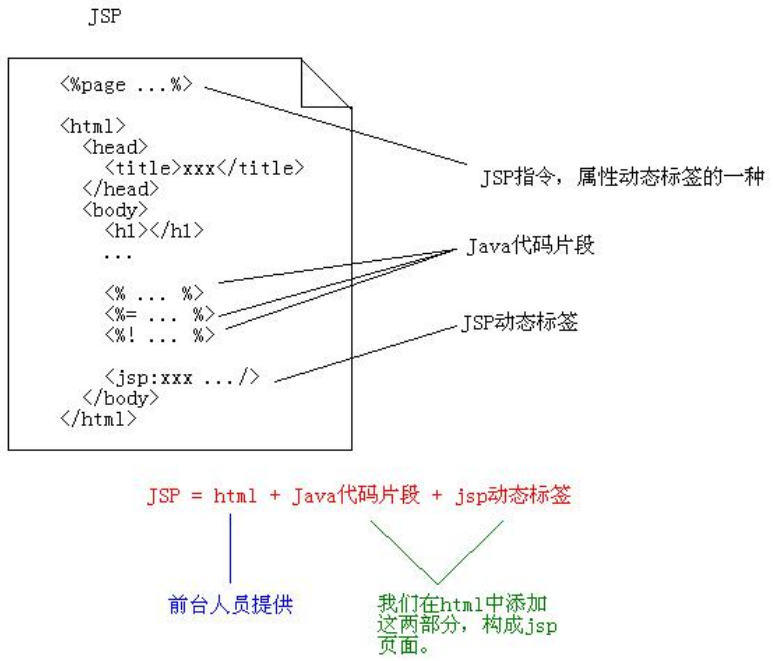
1 JSP 概述

1.1 什么是 JSP

JSP（Java Server Pages）是 JavaWeb 服务器端的**动态资源**。它与 html 页面的作用是相同的，**显示数据和获取数据**。

1.2 JSP 的组成

JSP = html + Java 脚本（代码片段） + JSP 动作标签



2 JSP 语法

2.1 JSP 脚本

JSP 脚本就是 Java 代码片段，它分为三种：

- <%...%>：Java 语句；
- <%=...%>：Java 表达式；
- <%!...%>：Java 定义类成员；

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>JSP演示</title>
  </head>

  <body>
    <h1>JSP演示</h1>
    <%
    // Java语句
    String s1 = "hello jsp";
    // 不会输出到客户端，而是在服务器端的控制台打印
    System.out.println(s1);
    %>
  <!-- 输出到客户端浏览器上 -->
    输出变量：<%=s1 %><br/>
    输出int类型常量：<%=100 %><br/>
    输出String类型常量：<%="你好" %><br/>
    <br/>
    使用表达式输出常量是很傻的一件事，因为可以直接使用html即可，下面是输出上面的常量：<br/>
    100<br/>
    你好
  </body>
</html>
```

2.2 内置对象 out

out 对象在 JSP 页面中无需创建就可以使用，它的作用是用来向客户端输出。

```
  <body>
    <h1>out.jsp</h1>
    <%
        //向客户端输出
        out.print("你好！");
    %>
  </body>
```

其中<%=...%>与 out.print()功能是相同的！它们都是向客户端输出，例如：

<%=s1%>等同于<% out.print(s1); %>

<%=”hello”%>等同于<% out.print(“hello”); %>，也等同于直接在页面中写 hello 一样。

2.3 多个<%...%>可以通用

在一个 JSP 中多个<%...%>是相通的。例如：

```
<body>
  <h1>out.jsp</h1>
  <%
      String s = "hello";
  %>
  <%
      out.print(s);
  %>
</body>
```

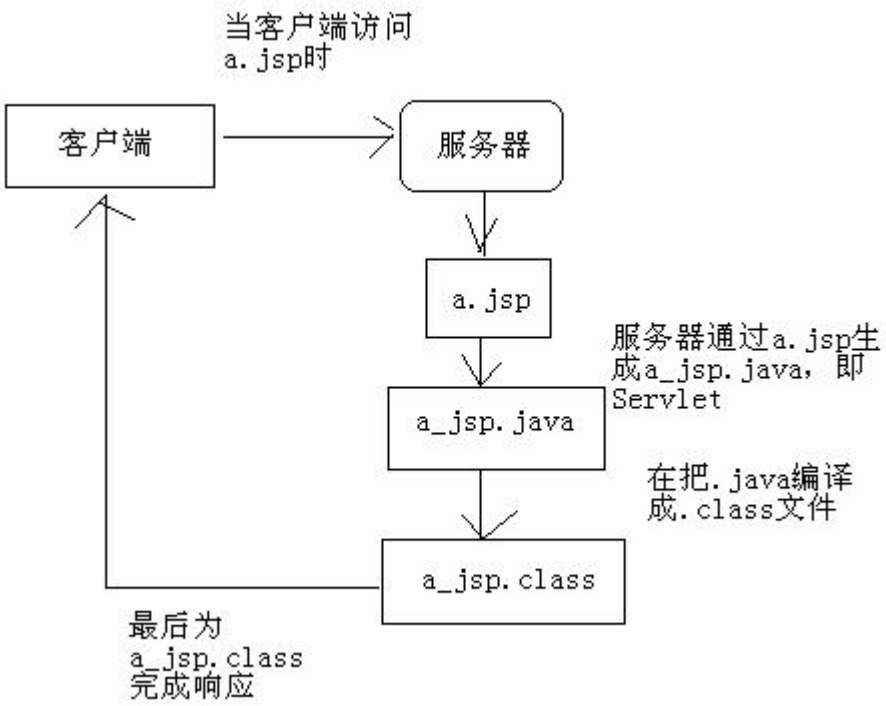
循环打印表格：

```
<body>
  <h1>表格</h1>

  <table border="1" width="50%">
      <tr>
          <th>序号</th>
          <th>用户名</th>
          <th>密码</th>
      </tr>
  <%
      for(int i = 0; i < 10; i++) {
  %>
      <tr>
          <td><%=i+1 %></td>
          <td>user<%=i %></td>
          <td><%=100 + 1 %></td>
      </tr>
  <%
      }
  %>
  </table>
</body>
```

3 JSP 的原理

3.1 JSP 是特殊的 Servlet

JSP 是一种特殊的 Servlet，当 JSP 页面首次被访问时，容器（Tomcat）会先把 JSP 编译成 Servlet，然后再去执行 Servlet。所以 JSP 其实就是一个 Servlet！



3.2 JSP 真身存放目录

JSP 生成的 Servlet 存放在${CATALANA}/work 目录下，我经常开玩笑的说，它是 JSP 的“真身”。我们打开看看其中的内容，了解一下 JSP 的“真身”。

你会发现，在 JSP 中的静态信息（例如<html>等）在“真身”中都是使用 out.write()完成打印！这些静态信息都是作为字符串输出给了客户端。

JSP 的整篇内容都会放到名为_jspService 的方法中！你可能会说<@page>不在“真身”中，<%@page>我们明天再讲。

a_jsp.java 的_jspService()方法：

```
public void _jspService(final javax.servlet.http.HttpServletRequest request,
                        final javax.servlet.http.HttpServletResponse response)
      throws java.io.IOException, javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
```

```
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html;charset=UTF-8");
      pageContext = _jspxFactory.getPageContext(this, request, response,
                  null, true, 8192, true);
      _jspx_page_context = pageContext;
      application = pageContext.getServletContext();
      config = pageContext.getServletConfig();
      session = pageContext.getSession();
      out = pageContext.getOut();
      _jspx_out = out;

    ...
}
```

4 再论 JSP 脚本

JSP 脚本一共三种形式：

- <%...%>：内容会直接放到“真身”中；
- <%=...%>：内容会放到 out.print()中，作为 out.print()的参数；
- <%!...%>：内容会放到_jspService()方法之外，被类直接包含；

前面已经讲解了<%...%>和<%=...%>，但还没有讲解<%!...%>的作用！

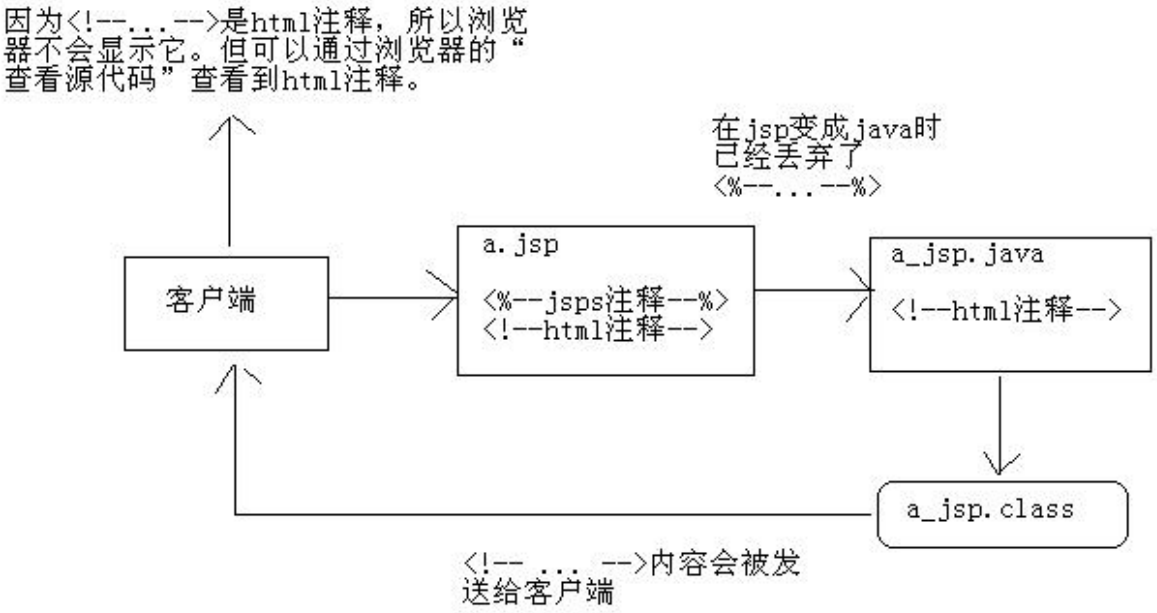
现在我们已经知道了，JSP 其实就是一个类，一个 Servlet 类。<%!...%>的作用是在类中添加方法或成员的，所以<%!...%>中的内容不会出现在_jspService()中。

```
<%!
    private String name;
    public String hello() {
        return "hello JSP!";
    }
%>
```

5 JSP 注释

我们现在已经知道 JSP 是需要先编译成.java，再编译成.class 的。其中<%-- ... --%>中的内容在 JSP 编译成.java 时会被忽略的，即 JSP 注释。

也可以在 JSP 页面中使用 html 注释：<!-- ... -->，但这个注释在 JSP 编译成的.java 中是存在的，它不会被忽略，而且会被发送到客户端浏览器。但是在浏览器显示服务器发送过来的 html 时，因为<!-- ... -->是 html 的注释，所以浏览器是不会显示它的。



会话跟踪技术

1 什么是会话跟踪技术

我们需要先了解一下什么是会话！可以把会话理解为客户端与服务器之间的一次会晤，在一次会晤中可能会包含多次请求和响应。例如你给 10086 打个电话，你就是客户端，而 10086 服务人员就是服务器了。从双方接通电话那一刻起，会话就开始了，到某一方挂断电话表示会话结束。在通话过程中，你会向 10086 发出多个请求，那么这多个请求都在一个会话中。

在 JavaWeb 中，客户向某一服务器发出第一个请求开始，会话就开始了，直到客户关闭了浏览器会话结束。

在一个会话的多个请求中共享数据，这就是会话跟踪技术。例如在一个会话中的请求如下：

- 请求银行主页；
- 请求登录（请求参数是用户名和密码）；
- 请求转账（请求参数与转账相关的数据）；
- 请求信誉卡还款（请求参数与还款相关的数据）。

在这上会话中当前用户信息必须在这个会话中共享的，因为登录的是张三，那么在转账和还款时一定是相对张三的转账和还款！这就说明我们必须在一个会话过程中有共享数据的能力。

2 会话路径技术使用 Cookie 或 session 完成

我们知道 HTTP 协议是无状态协议，也就是说每个请求都是独立的!无法记录前一次请求的状态。但 HTTP 协议中可以使用 Cookie 来完成会话跟踪!

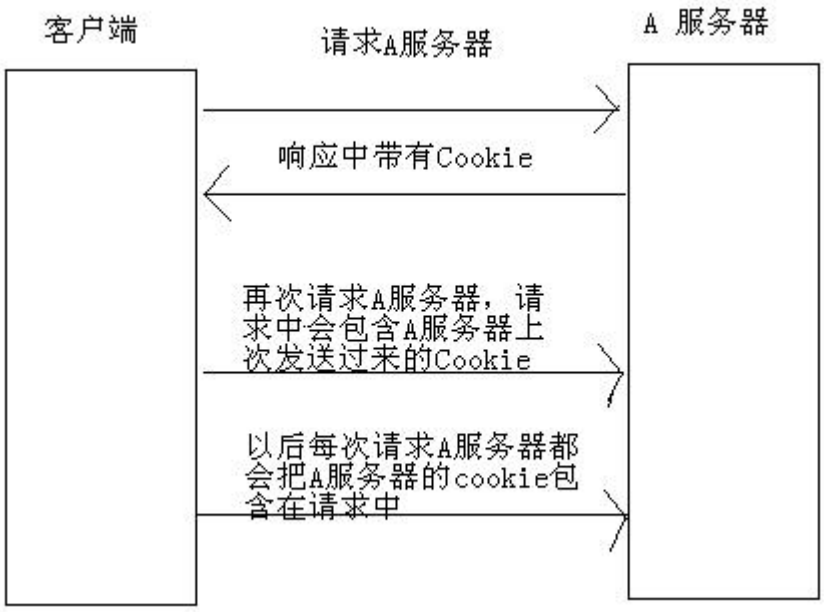
在 JavaWeb 中，使用 session 来完成会话跟踪，session 底层依赖 Cookie 技术。

Cookie

1 Cookie 概述

1.1 什么叫 Cookie

Cookie 翻译成中文是小甜点，小饼干的意思。在 HTTP 中它表示服务器送给客户端浏览器的小甜点。其实 Cookie 就是一个键和一个值构成的，随着服务器端的响应发送给客户端浏览器。然后客户端浏览器会把 Cookie 保存起来，当下一次再访问服务器时把 Cookie 再发送给服务器。



Cookie 是由服务器创建，然后通过响应发送给客户端的一个键值对。客户端会保存 Cookie，并会标注出 Cookie 的来源（哪个服务器的 Cookie）。当客户端向服务器发出请求时会把所有这个服务器 Cookie 包含在请求中发送给服务器，这样服务器就可以识别客户端了!

1.2 Cookie 规范

- Cookie 大小上限为 4KB；
- 一个服务器最多在客户端浏览器上保存 20 个 Cookie；
- 一个浏览器最多保存 300 个 Cookie；

上面的数据只是 HTTP 的 Cookie 规范，但在浏览器大战的今天，一些浏览器为了打败对手，为了展现自己的能力起见，可能对 Cookie 规范“扩展”了一些，例如每个 Cookie 的大小为 8KB，最多可保存 500 个 Cookie 等！但也不会出现把你硬盘占满的可能！

注意，不同浏览器之间是不共享 Cookie 的。也就是说在你使用 IE 访问服务器时，服务器会把 Cookie 发给 IE，然后由 IE 保存起来，当你在使用 FireFox 访问服务器时，不可能把 IE 保存的 Cookie 发送给服务器。

1.3 Cookie 与 HTTP 头

Cookie 是通过 HTTP 请求和响应头在客户端和服务器端传递的：

- Cookie：请求头，客户端发送给服务器端；
  - 格式：Cookie: a=A; b=B; c=C。即多个 Cookie 用分号离开；
- Set-Cookie：响应头，服务器端发送给客户端；
  - 一个 Cookie 对应一个 Set-Cookie：

    Set-Cookie: a=A

    Set-Cookie: b=B

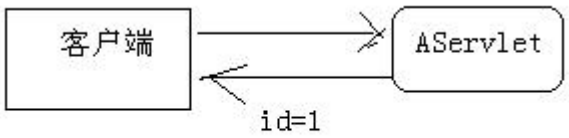
    Set-Cookie: c=C

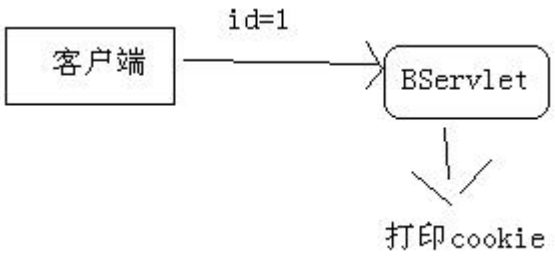
1.4 Cookie 的覆盖

如果服务器端发送重复的 Cookie 那么会覆盖原有的 Cookie，例如客户端的第一个请求服务器端发送的 Cookie 是：Set-Cookie: a=A；第二请求服务器端发送的是：Set-Cookie: a=AA，那么客户端只留下一个 Cookie，即：a=AA。

1.5 Cookie 第一例

我们这个案例是，客户端访问 AServlet，AServlet 在响应中添加 Cookie，浏览器会自动保存 Cookie。然后客户端访问 BServlet，这时浏览器会自动在请求中带上 Cookie，BServlet 获取请求中的 Cookie 打印出来。





AServlet.java

```
package cn.itcast.servlet;

import java.io.IOException;
import java.util.UUID;

import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * 给客户端发送Cookie
 * @author Administrator
 *
 */
public class AServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=utf-8");

        String id = UUID.randomUUID().toString();//生成一个随机字符串
        Cookie cookie = new Cookie("id", id);//创建Cookie对象，指定名字和值
        response.addCookie(cookie);//在响应中添加Cookie对象
        response.getWriter().print("已经给你发送了ID");
    }
}
```

BServlet.java

```
package cn.itcast.servlet;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * 获取客户端请求中的Cookie
 * @author Administrator
 *
 */
public class BServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=utf-8");

        Cookie[] cs = request.getCookies();//获取请求中的Cookie
        if(cs != null) {//如果请求中存在Cookie
            for(Cookie c : cs) {//遍历所有Cookie
                if(c.getName().equals("id")) {//获取Cookie名字，如果Cookie名字是
id
                    response.getWriter().print("您的ID是: " + c.getValue());//打
印Cookie值
                }
            }
        }
    }
}
```

2 Cookie 的生命

2.1 什么是 Cookie 的生命

Cookie 不只是有 name 和 value，Cookie 还是生命。所谓生命就是 Cookie 在客户端的有效时间，

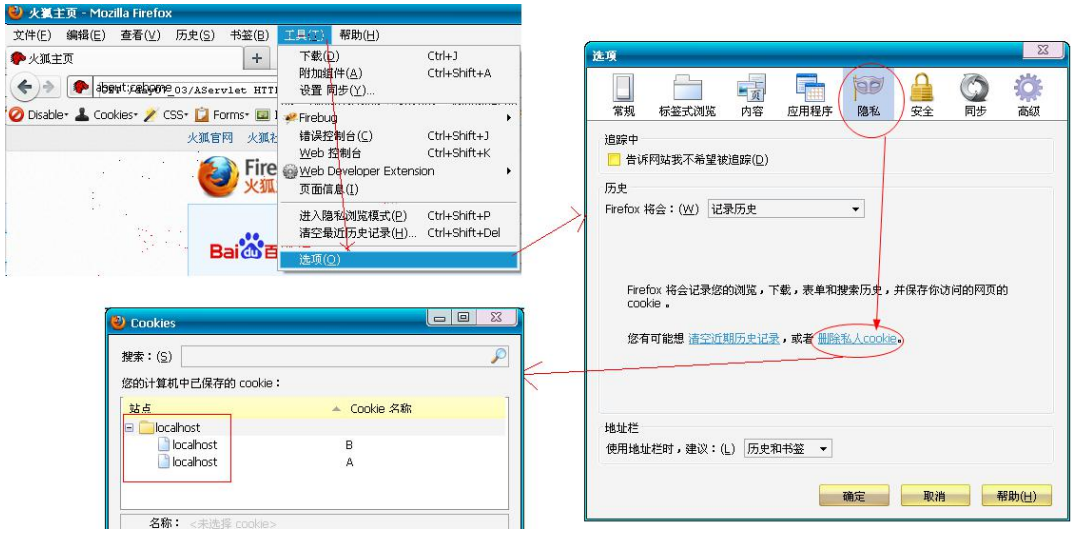
可以通过 setMaxAge(int)来设置 Cookie 的有效时间。

- cookie.setMaxAge(-1)：cookie 的 maxAge 属性的默认值就是-1，表示只在浏览器内存中存活。一旦关闭浏览器窗口，那么 cookie 就会消失。
- cookie.setMaxAge(60*60)：表示 cookie 对象可存活 1 小时。当生命大于 0 时，浏览器会把 Cookie 保存到硬盘上，就算关闭浏览器，就算重启客户端电脑，cookie 也会存活 1 小时；
- cookie.setMaxAge(0)：cookie 生命等于 0 是一个特殊的值，它表示 cookie 被作废！也就是说，如果原来浏览器已经保存了这个 Cookie，那么可以通过 Cookie 的 setMaxAge(0)来删除这个 Cookie。无论是在浏览器内存中，还是在客户端硬盘上都会删除这个 Cookie。

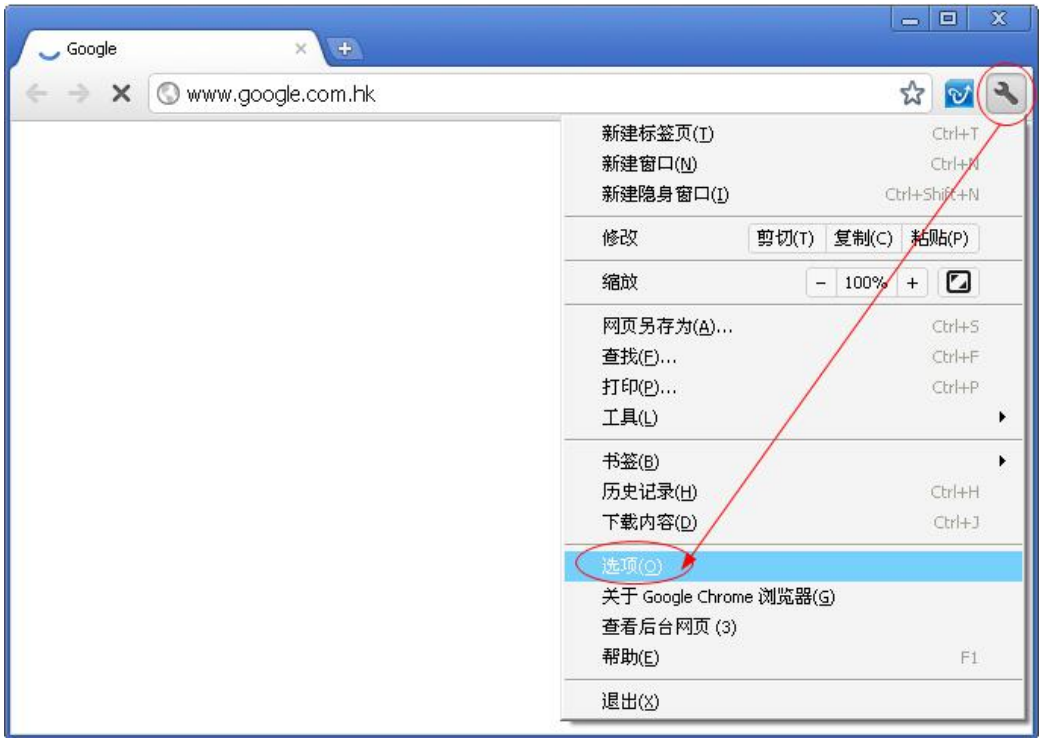
2.2 浏览器查看 Cookie

下面是浏览器查看 Cookie 的方式：

- IE 查看 Cookie 文件的路径：C:\Documents and Settings\Administrator\Cookies；
- FireFox 查看 Cooke：



- Google 查看 Cookie：


Google
www.google.com.hk
新建标签页(T) Ctrl+T
新建窗口(N) Ctrl+N
新建隐身窗口(I) Ctrl+Shift+N
修改 剪切(T) 复制(C) 粘贴(P)
缩放 - 100% +
网页另存为(A)... Ctrl+S
查找(F)... Ctrl+F
打印(P)... Ctrl+P
工具(L)
书签(B)
历史记录(H) Ctrl+H
下载内容(D) Ctrl+J
选项(O)
关于 Google Chrome 浏览器(G)
查看后台网页 (3)
帮助(E) F1
退出(X)


Google
选项 - 高级选项
chrome://settings/advanced
选项
搜索选项
基本设置
个人资料
高级选项
高级选项
隐私设置
内容设置...
清除浏览数据...
Google Chrome 浏览器可能会使用网络服务改善您的浏览
您可以视情况停用这些服务 了解详情
使用网络服务帮助解决导航错误
借助联想式建议服务，帮您在地址栏中自动填充未输完
预测网络操作，以提高网页载入速度
启用针对网上诱骗和恶意软件的防护功能
将使用情况统计信息和崩溃报告自动发送给 Google
网页内容
字体大小：
中
网页缩放：
100%

2.3 案例：显示上次访问时间

- 创建 Cookie，名为 lasttime，值为当前时间，添加到 response 中；
- 在 AServlet 中获取请求中名为 lasttime 的 Cookie；
- 如果不存在输出“您是第一次访问本站”，如果存在输出“您上一次访问本站的时间是 xxx”；

AServlet.java

```
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
```

```
        throws ServletException, IOException {
    response.setContentType("text/html;charset=utf-8");

    Cookie cookie = new Cookie("lasttime", new Date().toString());
    cookie.setMaxAge(60 * 60);
    response.addCookie(cookie);

    Cookie[] cs = request.getCookies();
    String s = "您是首次访问本站！";
    if(cs != null) {
        for(Cookie c : cs) {
            if(c.getName().equals("lasttime")) {
                s = "您上次的访问时间是：" + c.getValue();
            }
        }
    }

    response.getWriter().print(s);
}
```

3 Cookie 的 path

3.1 什么是 Cookie 的路径

现在有 WEB 应用 A，向客户端发送了 10 个 Cookie，这就说明客户端无论访问应用 A 的哪个 Servlet 都会把这 10 个 Cookie 包含在请求中！但是也许只有 AServlet 需要读取请求中的 Cookie，而其他 Servlet 根本就不会获取请求中的 Cookie。这说明客户端浏览器有时发送这些 Cookie 是多余的！

可以通过设置 Cookie 的 path 来指定浏览器，在访问什么样的路径时，包含什么样的 Cookie。

3.2 Cookie 路径与请求路径的关系

下面我们来看看 Cookie 路径的作用：

下面是客户端浏览器保存的 3 个 Cookie 的路径：

a: /cookietest；

b: /cookietest/servlet；

c: /cookietest/jsp；

下面是浏览器请求的 URL：

A: http://localhost:8080/cookietest/AServlet；

B: http://localhost:8080/cookietest/servlet/BServlet；

C: http://localhost:8080/cookietest/servlet/CServlet；

- 请求 A 时，会在请求中包含 a；

- 请求 B 时，会在请求中包含 a、b；
- 请求 C 时，会在请求中包含 a、c；

也就是说，请求路径如果包含了 Cookie 路径，那么会在请求中包含这个 Cookie，否则不会请求中不会包含这个 Cookie。

- A 请求的 URL 包含了“/cookietest”，所以会在请求中包含路径为“/cookietest”的 Cookie；
- B 请求的 URL 包含了“/cookietest”，以及“/cookietest/servlet”，所以请求中包含路径为“/cookietest”和“/cookietest/servlet”两个 Cookie；
- B 请求的 URL 包含了“/cookietest”，以及“/cookietest/jsp”，所以请求中包含路径为“/cookietest”和“/cookietest/jsp”两个 Cookie；

3.3 设置 Cookie 的路径

设置 Cookie 的路径需要使用 setPath()方法，例如：
cookie.setPath(“/cookietest/servlet”);

如果没有设置 Cookie 的路径，那么 Cookie 路径的默认值当前访问资源所在路径，例如：

- 访问 http://localhost:8080/cookietest/AServlet 时添加的 Cookie 默认路径为/cookietest；
- 访问 http://localhost:8080/cookietest/servlet/BServlet 时添加的 Cookie 默认路径为/cookietest/servlet；
- 访问 http://localhost:8080/cookietest/jsp/BServlet 时添加的 Cookie 默认路径为/cookietest/jsp；

4 Cookie 的 domain

Cookie 的 domain 属性可以让网站中二级域共享 Cookie，次要！

百度你是了解的对吧！
http://www.baidu.com
http://zhidao.baidu.com
http://news.baidu.com
http://tieba.baidu.com

现在我希望在这些主机之间共享 Cookie（例如在 www.baidu.com 中响应的 cookie，可以在 news.baidu.com 请求中包含）。很明显，现在不是路径的问题了，而是主机的问题，即域名的问题。处理这一问题其实很简单，只需要下面两步：

- 设置 Cookie 的 path 为“/”：c.setPath(“/”)；
- 设置 Cookie 的 domain 为“.baidu.com”：c.setDomain(“.baidu.com”)。

当 domain 为“.baidu.com”时，无论前缀是什么，都会共享 Cookie 的。但是现在我们需要设置两个虚拟主机：www.baidu.com 和 news.baidu.com。

第一步：设置 windows 的 DNS 路径解析
找到 C:\WINDOWS\system32\drivers\etc\hosts 文件，添加如下内容

```
127.0.0.1       localhost
```

```
127.0.0.1        www.baidu.com
127.0.0.1        news.baidu.com
```

第二步：设置 Tomcat 虚拟主机

找到 server.xml 文件，添加<Host>元素，内容如下：

```
<Host name="www.baidu.com"    appBase="F:\webapps\www"
        unpackWARs="true" autoDeploy="true"
        xmlValidation="false" xmlNamespaceAware="false"/>
<Host name="news.baidu.com"    appBase="F:\webapps\news"
        unpackWARs="true" autoDeploy="true"
        xmlValidation="false" xmlNamespaceAware="false"/>
```

第三步：创建 A 项目，创建 AServlet，设置 Cookie。

```
Cookie c = new Cookie("id", "baidu");
c.setPath("/");
c.setDomain(".baidu.com");
c.setMaxAge(60*60);
response.addCookie(c);
response.getWriter().print("OK");
```

把 A 项目的 WebRoot 目录复制到 F:\webapps\www 目录下，并把 WebRoot 目录的名字修改为 ROOT。

第四步：创建 B 项目，创建 BServlet，获取 Cookie，并打印出来。

```
Cookie[] cs = request.getCookies();
if(cs != null) {
    for(Cookie c : cs) {
        String s = c.getName() + ": " + c.getValue() + "<br/>";
        response.getWriter().print(s);
    }
}
```

把 B 项目的 WebRoot 目录复制到 F:\webapps\news 目录下，并把 WebRoot 目录的名字修改为 ROOT。

第五步：访问 www.baidu.com\AServlet，然后再访问 news.baidu.com\BServlet。

5 Cookie 中保存中文

Cookie 的 name 和 value 都不能使用中文，如果希望在 Cookie 中使用中文，那么需要先对中文进行 URL 编码，然后把编码后的字符串放到 Cookie 中。

向客户端响应中添加 Cookie

```
String name = URLEncoder.encode("姓名", "UTF-8");
```

```
        String value = URLEncoder.encode("张三", "UTF-8");
        Cookie c = new Cookie(name, value);
        c.setMaxAge(3600);
        response.addCookie(c);
```

从客户端请求中获取 Cookie

```
        response.setContentType("text/html;charset=utf-8");
        Cookie[] cs = request.getCookies();
        if(cs != null) {
            for(Cookie c : cs) {
                String name = URLDecoder.decode(c.getName(), "UTF-8");
                String value = URLDecoder.decode(c.getValue(), "UTF-8");
                String s = name + ": " + value + "<br/>";
                response.getWriter().print(s);
            }
        }
```

6 显示曾经浏览过的商品

index.jsp

```
<body>
  <h1>商品列表</h1>
  <a href="/day06_3/GoodServlet?name=ThinkPad">ThinkPad</a><br/>
  <a href="/day06_3/GoodServlet?name=Lenovo">Lenovo</a><br/>
  <a href="/day06_3/GoodServlet?name=Apple">Apple</a><br/>
  <a href="/day06_3/GoodServlet?name=HP">HP</a><br/>
  <a href="/day06_3/GoodServlet?name=SONY">SONY</a><br/>
  <a href="/day06_3/GoodServlet?name=ACER">ACER</a><br/>
  <a href="/day06_3/GoodServlet?name=DELL">DELL</a><br/>

  <hr/>
  您浏览过的商品:
  <%
  Cookie[] cs = request.getCookies();
  if(cs != null) {
      for(Cookie c : cs) {
          if(c.getName().equals("goods")) {
              out.print(c.getValue());
          }
      }
  }
  %>
```

```
</body>
```

GoodServlet

```
public class GoodServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String goodName = request.getParameter("name");
        String goods = CookieUtils.getCookValue(request, "goods");

        if(goods != null) {
            String[] arr = goods.split(", ");
            Set<String> goodSet = new LinkedHashSet(Arrays.asList(arr));
            goodSet.add(goodName);
            goods = goodSet.toString();
            goods = goods.substring(1, goods.length() - 1);
        } else {
            goods = goodName;
        }
        Cookie cookie = new Cookie("goods", goods);
        cookie.setMaxAge(1 * 60 * 60 * 24);
        response.addCookie(cookie);

        response.sendRedirect("/day06_3/index.jsp");
    }
}
```

CookieUtils

```
public class CookieUtils {
    public static String getCookValue(HttpServletRequest request, String name)
{
        Cookie[] cs = request.getCookies();
        if(cs == null) {
            return null;
        }
        for(Cookie c : cs) {
            if(c.getName().equals(name)) {
                return c.getValue();
            }
        }
        return null;
    }
}
```

HttpSession

1 HttpSession 概述

1.1 什么是 HttpSesssion

javax.servlet.http.HttpSession 接口表示一个会话，我们可以把一个会话内需要共享的数据保存到 HttSession 对象中！

1.2 获取 HttpSession 对象

- HttpSession request.getSesssion()：如果当前会话已经有了 session 对象那么直接返回，如果当前会话还不存在会话，那么创建 session 并返回；
- HttpSession request.getSession(boolean)：当参数为 true 时，与 requeset.getSession()相同。如果参数为 false，那么如果当前会话中存在 session 则返回，不存在返回 null；

1.3 HttpSession 是域对象

我们已经学习过 HttpServletRequest、ServletContext，它们都是域对象，现在我们又学习了一个 HttpSession，它也是域对象。它们三个是 Servlet 中可以使用的域对象，而 JSP 中可以多使用一个域对象，明天我们再讲解 JSP 的第四个域对象。

- HttpServletRequest：一个请求创建一个 request 对象，所以在同一个请求中可以共享 request，例如一个请求从 AServlet 转发到 BServlet，那么 AServlet 和 BServlet 可以共享 request 域中的数据；
- ServletContext：一个应用只创建一个 ServletContext 对象，所以在 ServletContext 中的数据可以在整个应用中共享，只要不启动服务器，那么 ServletContext 中的数据就可以共享；
- HttpSession：一个会话创建一个 HttpSession 对象，同一会话中的多个请求中可以共享 session 中的数据；

下载是 session 的域方法：

- void setAttribute(String name, Object value)：用来存储一个对象，也可以称之为存储一个域属性，例如：session.setAttribute(“xxx”, “XXX”)，在 session 中保存了一个域属性，域属性名称为 xxx，域属性的值为 XXX。请注意，如果多次调用该方法，并且使用相同的 name，那么会覆盖上一次的值，这一特性与 Map 相同；
- Object getAttribute(String name)：用来获取 session 中的数据，当前在获取之前需要先去存储才行，例如：String value = (String) session.getAttribute(“xxx”);，获取名为 xxx 的域属性；
- void removeAttribute(String name)：用来移除 HttpSession 中的域属性，如果参数 name 指定的域属性不存在，那么本方法什么都不做；
- Enumeration getAttributeNames()：获取所有域属性的名称；

2 登录案例

需要的页面：

- login.jsp：登录页面，提供登录表单；
- index1.jsp：主页，显示当前用户名称，如果没有登录，显示您还没登录；
- index2.jsp：主页，显示当前用户名称，如果没有登录，显示您还没登录；

Servlet：

- LoginServlet：在 login.jsp 页面提交表单时，请求本 Servlet。在本 Servlet 中获取用户名、密码进行校验，如果用户名、密码错误，显示“用户名或密码错误”，如果正确保存用户名 session 中，然后重定向到 index1.jsp；

当用户没有登录时访问 index1.jsp 或 index2.jsp，显示“您还没有登录”。如果用户在 login.jsp 登录成功后到达 index1.jsp 页面会显示当前用户名，而且不用再次登录去访问 index2.jsp 也会显示用户名。因为多次请求在一个会话范围，index1.jsp 和 index2.jsp 都会到 session 中获取用户名，session 对象在一个会话中是相同的，所以都可以获取到用户名！

login.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>login.jsp</title>
  </head>

  <body>
    <h1>login.jsp</h1>
    <hr/>
    <form action="/day06_4/LoginServlet" method="post">
    用户名：<input type="text" name="username" /><br/>
        <input type="submit" value="Submit"/>
    </form>
  </body>
</html>
```

index1.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
```

```
    <title>index1.jsp</title>
  </head>

  <body>
<h1>index1.jsp</h1>
<%
    String username = (String)session.getAttribute("username");
    if(username == null) {
        out.print("您还没有登录! ");
    } else {
        out.print("用户名: " + username);
    }
%>
<hr/>
<a href="/day06_4/index2.jsp">index2</a>
  </body>
</html>
```

index2.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>index2.jsp</title>
  </head>

  <body>
<h1>index2.jsp</h1>
<%
    String username = (String)session.getAttribute("username");
    if(username == null) {
        out.print("您还没有登录! ");
    } else {
        out.print("用户名: " + username);
    }
%>
<hr/>
<a href="/day06_4/index1.jsp">index1</a>
  </body>
</html>
```

LoginServlet

```
public class LoginServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        String username = request.getParameter("username");

        if(username.equalsIgnoreCase("itcast")) {
            response.getWriter().print("用户名或密码错误！");
        } else {
            HttpSession session = request.getSession();
            session.setAttribute("username", username);
            response.sendRedirect("/day06_4/index1.jsp");
        }
    }
}
```

3 session 的实现原理

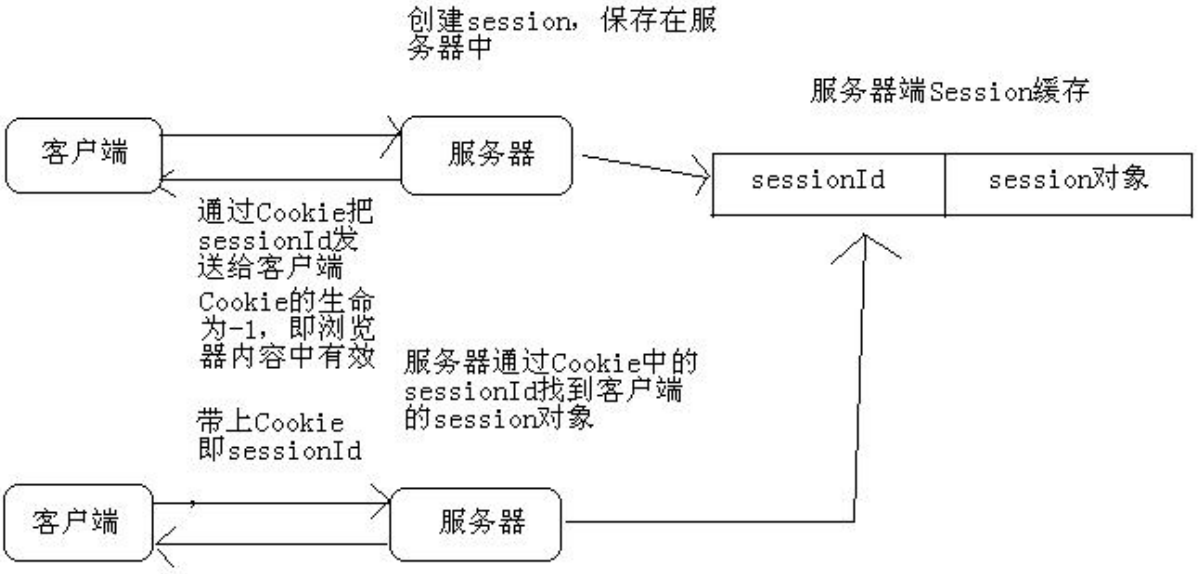
session 底层是依赖 Cookie 的！我们来理解一下 session 的原理吧！

当我首次去银行时，因为还没有账号，所以需要开一个账号，我获得的是银行卡，而银行这边的数据库中留下了我的账号，我的钱是保存在银行的账号中，而我带走的是我的卡号。

当我再次去银行时，只需要带上我的卡，而无需再次开一个账号了。只要带上我的卡，那么我在银行操作的一定是我的账号！

当首次使用 session 时，服务器端要创建 session，session 是保存在服务器端，而给客户端的 session 的 id（一个 cookie 中保存了 sessionId）。客户端带走的是 sessionId，而数据是保存在 session 中。

当客户端再次访问服务器时，在请求中会带上 sessionId，而服务器会通过 sessionId 找到对应的 session，而无需再创建新的 session。



4 session 与浏览器

session 保存在服务器，而 sessionId 通过 Cookie 发送给客户端，但这个 Cookie 的生命不-1，即只在浏览器内存中存在，也就是说如果用户关闭了浏览器，那么这个 Cookie 就丢失了。

当用户再次打开浏览器访问服务器时，就不会有 sessionId 发送给服务器，那么服务器会认为你没有 session，所以服务器会创建一个 session，并在响应中把 sessionId 中到 Cookie 中发送给客户端。

你可能会说，那原来的 session 对象会怎样？当一个 session 长时间没人使用的话，服务器会把 session 删除了！这个时长在 Tomcat 中配置是 30 分钟，可以在${CATALANA}/conf/web.xml 找到这个配置，当然你也可以在自己的 web.xml 中覆盖这个配置！

web.xml

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

session 失效时间也说明一个问题！如果你打开网站的一个页面开始长时间不动，超出了 30 分钟后，再去点击链接或提交表单时你会发现，你的 session 已经丢失了！

5 session 其他常用 API

- String getId()：获取 sessionId；
- int getMaxInactiveInterval()：获取 session 可以的最大不活动时间（秒），默认为 30 分钟。当 session 在 30 分钟内没有使用，那么 Tomcat 会在 session 池中移除这个 session；
- void setMaxInactiveInterval(int interval)：设置 session 允许的最大不活动时间（秒），如果设置为 1 秒，那么只要 session 在 1 秒内不被使用，那么 session 就会被移除；
- long getCreationTime()：返回 session 的创建时间，返回值为当前时间的毫秒值；
- long getLastAccessedTime()：返回 session 的最后活动时间，返回值为当前时间的毫秒值；
- void invalidate()：让 session 失效！调用这个方法会被 session 失效，当 session 失效后，客

户端再次请求，服务器会给客户端创建一个新的 session，并在响应中给客户端新 session 的 sessionId；

- boolean isNew()：查看 session 是否为新。当客户端第一次请求时，服务器为客户端创建 session，但这时服务器还没有响应客户端，也就是还没有把 sessionId 响应给客户端时，这时 session 的状态为新。

6 URL 重写

我们知道 session 依赖 Cookie，那么 session 为什么依赖 Cookie 呢？因为服务器需要在每次请求中获取 sessionId，然后找到客户端的 session 对象。那么如果客户端浏览器关闭了 Cookie 呢？那么 session 是不是就会不存在了呢？

其实还有一种方法让服务器收到的每个请求中都带有 sessioinId，那就是 URL 重写！在每个页面中的每个链接和表单中都添加名为 jSessionId 的参数，值为当前 sessionid。当用户点击链接或提交表单时也服务器可以通过获取 jSessionId 这个参数来得到客户端的 sessionId，找到 sessoin 对象。

index.jsp

```
  <body>
<h1>URL重写</h1>
<a href='/day06_5/index.jsp;jsessionid=<%=session.getId() %>' >主页</a>

<form action='/day06_5/index.jsp;jsessionid=<%=session.getId() %>'
method="post">
    <input type="submit" value="提交"/>
</form>
  </body>
```

也可以使用 response.encodeURL()对每个请求的 URL 处理，这个方法会自动追加 jsessionid 参数，与上面我们手动添加是一样的效果。

```
<a href='<%=response.encodeURL("/day06_5/index.jsp") %>' >主页</a>

<form action='<%=response.encodeURL("/day06_5/index.jsp") %>' method="post">
    <input type="submit" value="提交"/>
</form>
```

使用 response.encodeURL()更加“智能”，它会判断客户端浏览器是否禁用了 Cookie，如果禁用了，那么这个方法在 URL 后面追加 jsessionid，否则不会追加。

案例：一次性图片验证码

1 验证码有啥用

在我们注册时，如果没有验证码的话，我们可以使用 URLConnection 来写一段代码发出注册请求。甚至可以使用 while(true)来注册！那么服务器就废了！

验证码可以去识别发出请求的是人还是程序！当然，如果聪明的程序可以去分析验证码图片！但分析图片也不是一件容易的事，因为一般验证码图片都会带有干扰线，人都看不清，那么程序一定分析不出来。

2 VerifyCode 类

现在我们已经有了 cn.itcast.utils.VerifyCode 类，这个类可以生成验证码图片！下面来看一个小例子。

```
    public void fun1() throws IOException {
        // 创建验证码类
        VerifyCode vc = new VerifyCode();
        // 获取随机图片
        BufferedImage image = vc.getImage();
        // 获取刚刚生成的随机图片上的文本
        String text = vc.getText();
        System.out.println(text);
        // 保存图片
        FileOutputStream out = new FileOutputStream("F:/xxx.jpg");
        VerifyCode.output(image, out);
    }
```

3 在页面中显示动态图片

我们需要写一个 VerifyCodeServlet，在这个 Servlet 中我们生成动态图片，然后它图片写入到 response.getOutputStream()流中！然后让页面的<img>元素指定这个 VerifyCodServlet 即可。

VerifyCodeServlet

```
public class VerifyCodeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        VerifyCode vc = new VerifyCode();
        BufferedImage image = vc.getImage();
        String text = vc.getText();
        System.out.println("text:" + text);
        VerifyCode.output(image, response.getOutputStream());
    }
}
```

index.jsp

```
    <script type="text/javascript">
    function _change() {
        var imgEle = document.getElementById("vCode");
        imgEle.src = "/day06_6/VerifyCodeServlet?" + new Date().getTime();
    }
    </script>
...
  <body>
    <h1>验证码</h1>
    <img id="vCode" src="/day06_6/VerifyCodeServlet"/>
    <a href="javascript:_change()">看不清，换一张</a>
  </body>
```

4 在注册页面中使用验证码

```
    <form action="/day06_6/RegistServlet" method="post">
    用户名：<input type="text" name="username"/><br/>
    验证码：<input type="text" name="code" size="3"/>
    <img id="vCode" src="/day06_6/VerifyCodeServlet"/>
    <a href="javascript:_change()">看不清，换一张</a>
    <br/>
       <input type="submit" value="Submit"/>
    </form>
```

5 RegistServlet

修改 VerifyCodeServlet

```
public class VerifyCodeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        VerifyCode vc = new VerifyCode();
        BufferedImage image = vc.getImage();
        request.getSession().setAttribute("vCode", vc.getText());
        VerifyCode.output(image, response.getOutputStream());
    }
}
```

RegistServlet

```
public class RegistServlet extends HttpServlet {
```

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        String username = request.getParameter("username");
        String vCode = request.getParameter("code");

        String sessionVerifyCode =
(String)request.getSession().getAttribute("vCode");

        if(vCode.equalsIgnoreCase(sessionVerifyCode)) {
            response.getWriter().print(username + ", 恭喜！注册成功！");
        } else {
            response.getWriter().print("验证码错误！");
        }
    }
}
```

6 总结验证码案例

- VerifyCodeServlet：
  - 生成验证码：VerifyCode vc = new VerifyCode(); BufferedImage image = vc.getImage()；
  - 在 session 中保存验证码文本：request.getSession.getAttribute(“vCode”, vc.getText())；
  - 把验证码输出到页面：VerifyCode.output(image, response.getOutputStream)；
- regist.jsp：
  - 表单中包含 username 和 code 字段；
  - 在表单中给出<img>指向 VerifyCodeServlet，用来在页面中显示验证码图片；
  - 提供“看不清，换一张”链接，指向_change()函数；
  - 提交到 RegistServlet；
- RegistServlet：
  - 获取表单中的 username 和 code；
  - 获取 session 中的 vCode；
  - 比较 code 和 vCode 是否相同；
  - 相同说明用户输入的验证码正确，否则输入验证码错误。

day07

JSP 指令

1 JSP 指令概述

JSP 指令的格式：<%@指令名 attr1="" attr2="" %>，一般都会把 JSP 指令放到 JSP 文件的最上方，但这不是必须的。

JSP 中有三大指令：page、include、taglib，最为常用，也最为复杂的就是 page 指令了。

2 page 指令

page 指令是最为常用的指定，也是属性最多的属性！

page 指令没有必须属性，都是可选属性。例如<%@page %>，没有给出任何属性也是可以的！

在 JSP 页面中，任何指令都可以重复出现！

<%@ page language="java"%>

<%@ page import="java.util.*"%>

<%@ page pageEncoding="utf-8"%>

这也是可以的！

2.1 page 指令的 pageEncoding 和 contentType（重点）

pageEncoding 指定当前 JSP 页面的编码！这个编码是给服务器看的，服务器需要知道当前 JSP 使用的编码，不然服务器无法正确把 JSP 编译成 java 文件。所以这个编码只需要与真实的页面编码一致即可！在 MyEclipse 中，在 JSP 文件上点击右键，选择属性就可以看到当前 JSP 页面的编码了。

contentType 属性与 response.setContentType()方法的作用相同！它会完成两项工作，一是设置响应字符流的编码，二是设置 content-type 响应头。例如：<%@ contentType="text/html;charset=utf-8"%>，它会使“真身”中出现 response.setContentType("text/html;charset=utf-8")。

无论是 page 指令的 pageEncoding 还是 contentType，它们的默认值都是 ISO-8859-1，我们知道 ISO-8859-1 是无法显示中文的，所以 JSP 页面中存在中文的话，一定要设置这两个属性。

其实 pageEncoding 和 contentType 这两个属性的关系很“暧昧”：

- 当设置了 pageEncoding，而没设置 contentType 时： contentType 的默认值为 pageEncoding；
- 当设置了 contentType，而没设置 pageEncoding 时： pageEncoding 的默认值与 contentType；

也就是说，当 pageEncoding 和 contentType 只出现一个时，那么另一个的值与出现的值相同。如果两个都不出现，那么两个属性的值都是 ISO-8859-1。所以通过我们至少设置它们两个其中一个！

2.2 page 指令的 import 属性

import 是 page 指令中一个很特别的属性！
import 属性值对应“真身”中的 import 语句。
import 属性值可以使逗号：<%@page import=”java.net.*,java.util.*,java.sql.*”%>

import 属性是唯一可以重复出现的属性：
<%@page import=”java.util.*” import=”java.net.*” import=”java.sql.*”%>
但是，我们一般会使用多个 page 指令来导入多个包：
<%@ page import=”java.util.*”%>
<%@ page import=”java.net.*”%>
<%@ page import=”java.text.*”%>

2.3 page 指令的 errorPage 和 isErrorPage

我们知道，在一个 JSP 页面出错后，Tomcat 会响应给用户错误信息（500 页面）！如果你不希望 Tomcat 给用户输出错误信息，那么可以使用 page 指令的 errorPage 来指定错误页！也就是自定义错误页面，例如：<%@page errorPage=”xxx.jsp”%>。这时，在当前 JSP 页面出现错误时，会请求转发到 xxx.jsp 页面。

a.jsp

```
<%@ page import="java.util.*" pageEncoding="UTF-8"%>
<%@ page  errorPage="b.jsp" %>
    <%
    if(true)
        throw new Exception("哈哈~");
    %>
```

b.jsp

```
<%@ page pageEncoding="UTF-8"%>
<html>
  <body>
   <h1>出错啦！</h1>
  </body>
</html>
```

在上面代码中，a.jsp 抛出异常后，会请求转发到 b.jsp。在浏览器的地址栏中还是 a.jsp，因为是请求转发！

而且客户端浏览器收到的响应码为 200，表示请求成功！如果希望客户端得到 500，那么需要指定 b.jsp 为错误页面。

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ page  isErrorPage="true" %>
```

```
<html>
  <body>
   <h1>出错啦！</h1>
    <%=exception.getMessage() %>
  </body>
</html>
```

注意，当 isErrorPage 为 true 时，说明当前 JSP 为错误页面，即专门处理错误的页面。那么这个页面中就可以使用一个内置对象 exception 了。其他页面是不能使用这个内置对象的！

温馨提示：*IE 会在状态码为 500 时，并且响应正文的长度小于等于 512B 时不给予显示！而是显示“网站无法显示该页面”字样。这时你只需要添加一些响应内容即可，例如上例中的 b.jsp 中我给出一些内容，IE 就可以正常显示了！*

2.3.1 web.xml 中配置错误页面

不只可以通过 JSP 的 page 指令来配置错误页面，还可以在 web.xml 文件中指定错误页面。这种方式其实与 page 指令无关，但想来想去还是在这个位置来讲解比较合适！

web.xml

```
  <error-page>
    <error-code>404</error-code>
    <location>/error404.jsp</location>
  </error-page>
  <error-page>
    <error-code>500</error-code>
    <location>/error500.jsp</location>
  </error-page>
  <error-page>
    <exception-type>java.lang.RuntimeException</exception-type>
    <location>/error.jsp</location>
  </error-page>
```

<error-page>有两种使用方式：

- <error-code>和<location>子元素；
- <exception-type>和<location>子元素；

其中<error-code>是指定响应码；<location>指定转发的页面；<exception-type>是指定抛出的异常类型。

在上例中：

- 当出现 404 时，会跳转到 error404.jsp 页面；
- 当出现 RuntimeException 异常时，会跳转到 error.jsp 页面；
- 当出现非 RuntimeException 的异常时，会跳转到 error500.jsp 页面。

这种方式会在控制台看到异常信息！而使用 page 指令时不会在控制台打印异常信息。

2.4 page 指令的 autFlush 和 buffer

buffer 表示当前 JSP 的输出流（out 隐藏对象）的缓冲区大小，**默认为 8kb**。

authFlush 表示在 out 对象的缓冲区满时如何处理！当 authFlush 为 true 时，表示缓冲区满时把缓冲区数据输出到客户端；当 authFlush 为 false 时，表示缓冲区满时，抛出异常。**authFlush 的默认值为 true**。

这两个属性一般我们也不会去特意设置，都是保留默认值！

2.5 page 指令的 isELIgnored

后面我们会讲解 EL 表达式语言，page 指令的 isElIgnored 属性表示当前 JSP 页面是否忽略 EL 表达式，默认值为 false，表示不忽略（即支持）。

2.6 page 指令的其他属性

- language：**只能是 Java**，这个属性可以看出 JSP 最初设计时的野心！希望 JSP 可以转换成其他语言！但是，到现在 JSP 也只能转换成 Java 代码；
- info：JSP 说明性信息；
- isThreadSafe：默认为 false，为 true 时，JSP 生成的 Servlet 会去实现一个过时的标记接口 SingleThreadModel，这时 JSP 就只能处理单线程的访问；
- session：默认为 true，表示当前 JSP 页面可以使用 session 对象，如果为 false 表示当前 JSP 页面不能使用 session 对象；
- extends：指定当前 JSP 页面生成的 Servlet 的父类；

2.7 <jsp-config>（了解）

在 web.xml 页面中配置<jsp-config>也可以完成很多 page 指定的功能！

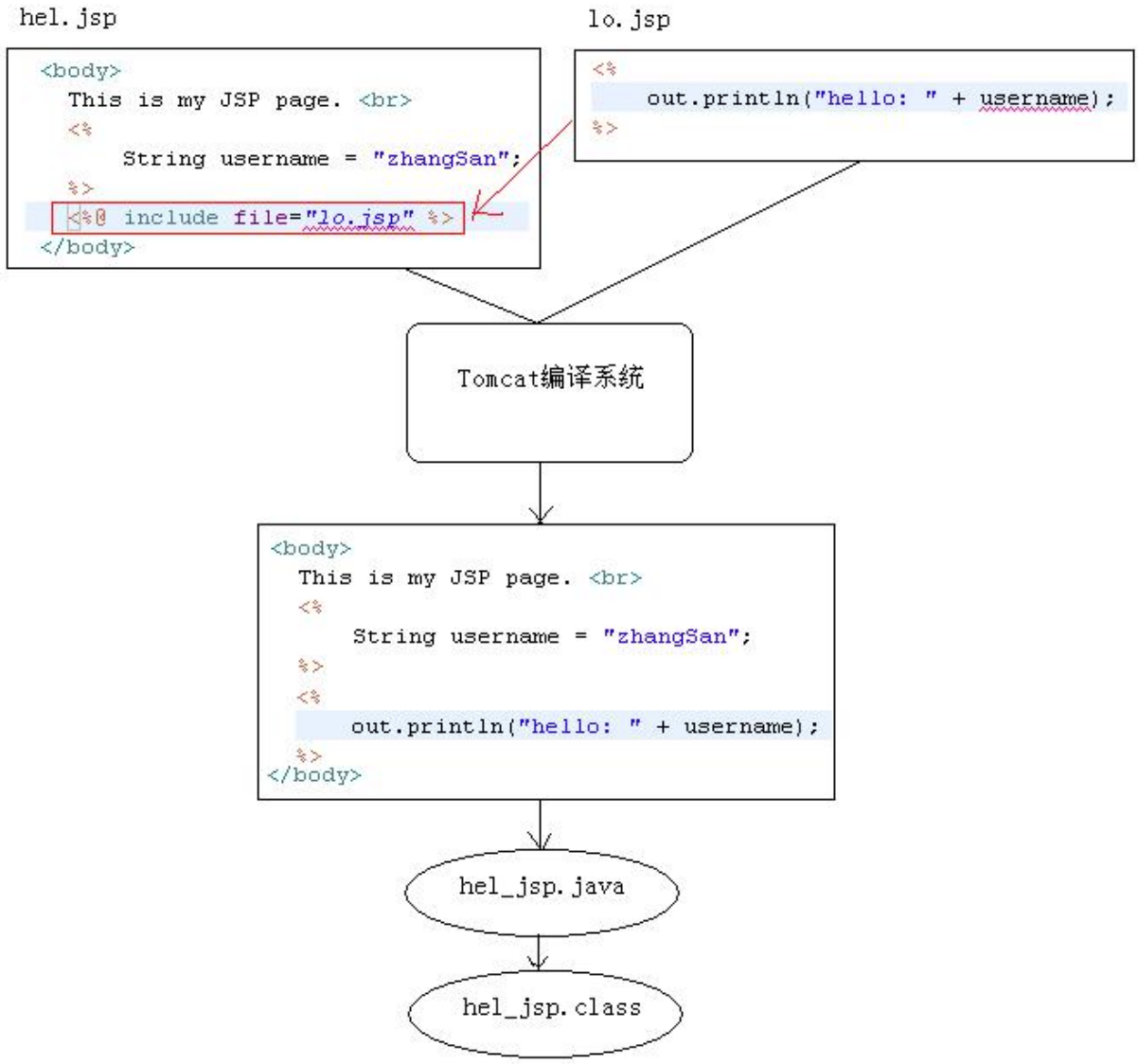
```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
        <page-encoding>UTF-8</page-encoding>
        <scripting-invalid>true</scripting-invalid>
    </jsp-property-group>
</jsp-config>
```

3 include 指令

include 指令表示静态包含！即目的是把多个 JSP 合并成一个 JSP 文件！

include 指令只有一个属性：file，指定要包含的页面，例如：<%@include file="b.jsp"%>。

静态包含：当 hel.jsp 页面包含了 lo.jsp 页面后，在编译 hel.jsp 页面时，需要把 hel.jsp 和 lo.jsp 页面合并成一个文件，然后再编译成 Servlet（Java 文件）。



很明显，在 ol.jsp 中在使用 username 变量，而这个变量在 hel.jsp 中定义的，所以只有这两个 JSP 文件合并后才能使用。通过 include 指定完成对它们的合并！

4 taglib 指令

这个指令需要在学习了自定义标签后才会使用，现在只能做了了解而已！

在 JSP 页面中使用第三方的标签库时，需要使用 taglib 指令来“导包”。例如：

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

其中 prefix 表示标签的前缀，这个名称可以随便起。uri 是由第三方标签库定义的，所以你需要知道第三方定义的 uri。

JSP 九大内置对象

1 什么是 JSP 九大内置对象

在 JSP 中无需创建就可以使用的 9 个对象，它们是：

- out（JspWriter）：等同与 response.getWriter()，用来向客户端发送文本数据；
- config（ServletConfig）：对应“真身”中的 ServletConfig；
- page（当前 JSP 的真身类型）：当前 JSP 页面的“this”，即当前对象；
- pageContext（PageContext）：页面上下文对象，它是最后一个没讲的域对象；
- exception（Throwable）：只有在错误页面中可以使用这个对象；
- request（HttpServletRequest）：即 HttpServletRequest 类的对象；
- response（HttpServletResponse）：即 HttpServletResponse 类的对象；
- application（ServletContext）：即 ServletContext 类的对象；
- session（HttpSession）：即 HttpSession 类的对象，不是每个 JSP 页面中都可以使用，如果在某个 JSP 页面中设置<%@page session=”false”%>，说明这个页面不能使用 session。

在这 9 个对象中有很多是极少会被使用的，例如：config、page、exception 基本不会使用。
在这 9 个对象中有两个对象不是每个 JSP 页面都可以使用的：exception、session。
在这 9 个对象中有很多前面已经学过的对象：out、request、response、application、session、config。

2 通过“真身”来对照 JSP

我们知道 JSP 页面的内容出现在“真身”的_jspService()方法中，而在_jspService()方法开头部分已经创建了 9 大内置对象。

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {

    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;


    try {
      response.setContentType("text/html;charset=UTF-8");
```

```
        pageContext = _jspxFactory.getPageContext(this, request, response,
                    null, true, 8192, true);
        _jspx_page_context = pageContext;
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        _jspx_out = out;

          从这里开始，才是 JSP 页面的内容
    }...
```

3 pageContext 对象

在 JavaWeb 中一共四个域对象，其中 Servlet 中可以使用的是 request、session、application 三个对象，而在 JSP 中可以使用 pageContext、request、session、application 四个域对象。

pageContext 对象是 PageContext 类型，它的主要功能有：

- 域对象功能；
- 代理其它域对象功能；
- 获取其他内置对象；

3.1 域对象功能

pageContext 也是域对象，它的范围是当前页面。它的范围也是四个域对象中最小的！

- void setAttribute(String name, Object value)；
- Object getAttrbiute(String name, Object value)；
- void removeAttribute(String name, Object value)；

3.2 代理其它域对象功能

还可以使用 pageContext 来代理其它 3 个域对象的功能，也就是说可以使用 pageContext 向 request、session、application 对象中存取数据，例如：

```
pageContext.setAttribute("x", "X");
pageContext.setAttribute("x", "XX", PageContext.REQUEST_SCOPE);
pageContext.setAttribute("x", "XXX", PageContext.SESSION_SCOPE);
pageContext.setAttribute("x", "XXXX", PageContext.APPLICATION_SCOPE);
```

- void setAttribute(String name, Object value, int scope)：在指定范围中添加数据；
- Object getAttribute(String name, int scope)：获取指定范围的数据；
- void removeAttribute(String name, int scope)：移除指定范围的数据；
- Object findAttribute(String name)：依次在 page、request、session、application 范围查找名称为 name 的数据，如果找到就停止查找。这说明在这个范围内有相同名称的数据，那么 page

范围的优先级最高！

3.3 获取其他内置对象

一个 pageContext 对象等于所有内置对象，即 1 个当 9 个。这是因为可以使用 pageContext 对象获取其它 8 个内置对象：

- JspWriter getOut()：获取 out 内置对象；
- ServletConfig getServletConfig()：获取 config 内置对象；
- Object getPage()：获取 page 内置对象；
- ServletRequest getRequest()：获取 request 内置对象；
- ServletResponse getResponse()：获取 response 内置对象；
- HttpSession getSession()：获取 session 内置对象；
- ServletContext getServletContext()：获取 application 内置对象；
- Exception getException()：获取 exception 内置对象；

JSP 动作标签

1 JSP 动作标签概述

动作标签的作用是用来简化 Java 脚本的！

JSP 动作标签是 JavaWeb 内置的动作标签，它们是已经定义好的动作标签，我们可以拿来直接使用。

如果 JSP 动作标签不够用时，还可以使用自定义标签（今天不讲）。JavaWeb 一共提供了 20 个 JSP 动作标签，但有很多基本没有用，这里只介绍一些有坐标的动作标签。

JSP 动作标签的格式：<jsp:标签名 …>

2 <jsp:include>

<jsp:include>标签的作用是用来包含其它 JSP 页面的！你可能会说，前面已经学习了 include 指令了，它们是否相同呢？虽然它们都是用来包含其它 JSP 页面的，但它们的实现的级别是不同的！

include 指令是在编译级别完成的包含，即把当前 JSP 和被包含的 JSP 合并成一个 JSP，然后再编译成一个 Servlet。

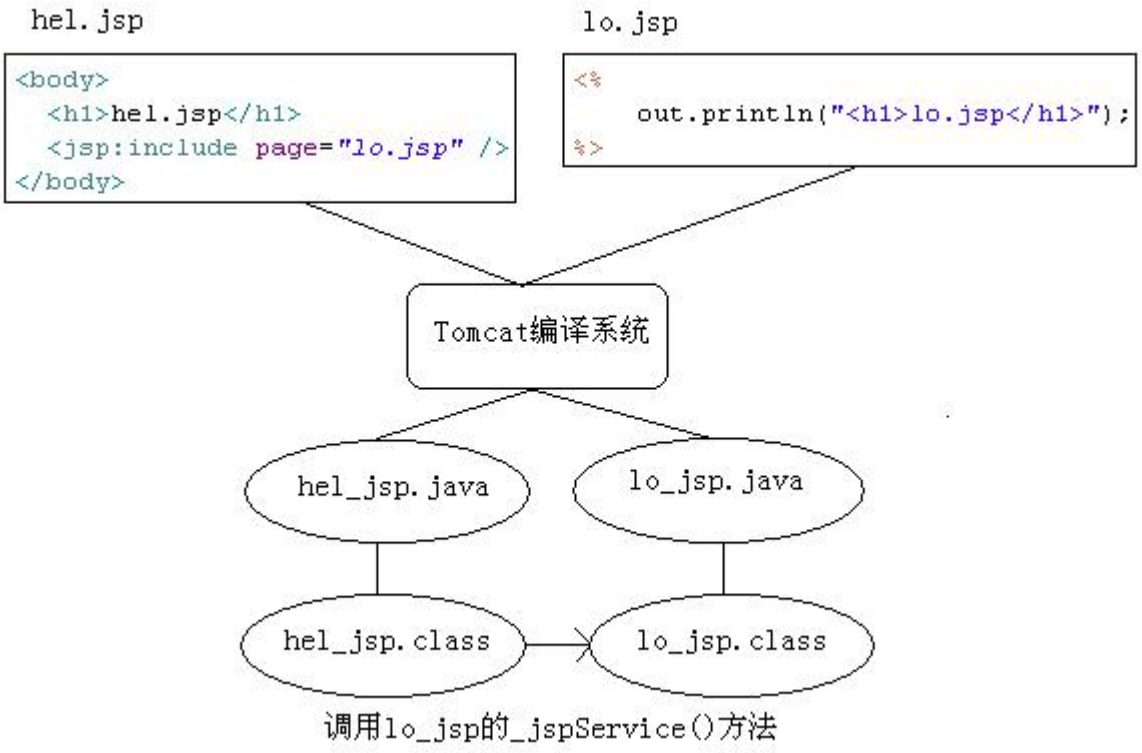
include 动作标签是在运行级别完成的包含，即当前 JSP 和被包含的 JSP 都会各自生成 Servlet，然后在执行当前 JSP 的 Servlet 时完成包含另一个 JSP 的 Servlet。它与 RequestDispatcher 的 include()方法是相同的！

hel.jsp

```
<body>
  <h1>hel.jsp</h1>
  <jsp:include page="lo.jsp" />
</body>
```

lo.jsp

```
<%
    out.println("<h1>lo.jsp</h1>");
%>
```



其实<jsp:include>在“真身”中不过是一句方法调用，即调用另一个 Servlet 而已。

3 <jsp:forward>

forward 标签的作用是请求转发！forward 标签的作用与 RequestDispatcher#forward()方法相同。

hel.jsp

```
<body>
  <h1>hel.jsp</h1>
  <jsp:forward page="lo.jsp"/>
</body>
```

lo.jsp

```
<%
    out.println("<h1>lo.jsp</h1>");
%>
```

注意，最后客户端只能看到 lo.jsp 的输出，而看不到 hel.jsp 的内容。也就是说在 hel.jsp 中的<h1>hel.jsp</h1>是不会发送到客户端的。<jsp:forward>的作用是“别在显示我，去显示它吧！”。

4 <jsp:param>

还可以在<jsp:include>和<jsp:forward>标签中使用<jsp:param>子标签，它是用来传递参数的。下面用<jsp:include>来举例说明<jsp:param>的使用。

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>a.jsp</title>
  </head>

  <body>
    <h1>a.jsp</h1>
    <hr/>
    <jsp:include page="/b.jsp">
        <jsp:param value="zhangSan" name="username"/>
    </jsp:include>
  </body>
</html>
```

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>b.jsp</title>
  </head>

  <body>
    <h1>b.jsp</h1>
    <hr/>
    <%
        String username = request.getParameter("username");
        out.print("你好: " + username);
    %>
  </body>
</html>
```

JavaBean

1 JavaBean 概述

1.1 什么是 JavaBean

JavaBean 是一种规范，也就是对类的要求。它要求 Java 类的成员变量提供 getter/setter 方法，这样的成员变量被称之为 JavaBean 属性。

JavaBean 还要求类必须提供仅有的无参构造器，例如：public User() {...}

User.java

```
package cn.itcast.domain;

public class User {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
}
```

1.2 JavaBean 属性

JavaBean 属性是具有 getter/setter 方法的成员变量。

- 也可以只提供 getter 方法，这样的属性叫只读属性；
- 也可以只提供 setter 方法，这样的属性叫只写属性；
- 如果属性类型为 boolean 类型，那么读方法的格式可以是 get 或 is。例如名为 abc 的 boolean 类型的属性，它的读方法可以是 getAbc()，也可以是 isAbc()；

JavaBean 属性名要求：前两个字母要么都大写，要么都小写：

```
public class User {
    private String iD;
```

```
    private String ID;
    private String qQ;
    private String QQ;
    …
}
```

JavaBean 可能存在属性，但不存在这个成员变量，例如：

```
public class User {
    public String getUsername() {
        return "zhangSan";
    }
}
```

上例中 User 类有一个名为 username 的只读属性！但 User 类并没有 username 这个成员变量！

还可以并变态一点：

```
public class User {
    private String hello;

    public String getUsername() {
        return hello;
    }

    public void setUsername(String username) {
        this.hello = username;
    }
}
```

上例中 User 类中有一个名为 username 的属性，它是可读可写的属性！而 Use 类的成员变量名为 hello！也就是说 JavaBean 的属性名取决于方法名称，而不是成员变量的名称。但通常没有人做这么变态的事情。

2 内省（了解）

内省的目标是得到 JavaBean 属性的读、写方法的反射对象，通过反射对 JavaBean 属性进行操作的一组 API。例如 User 类有名为 username 的 JavaBean 属性，通过两个 Method 对象（一个是 getUsenrmae()，一个是 setUsername()）来操作 User 对象。

如果你还不能理解内省是什么，那么我们通过一个问题来了解内省的作用。现在我们有一个 Map，内容如下：

```
        Map<String,String> map = new HashMap<String,String>();
        map.put("username", "admin");
        map.put("password", "admin123");
```

```
public class User {
    private String username;
```

```
    private String password;

    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    public User() {
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

现在需要把 map 的数据封装到一个 User 对象中!User 类有两个 JavaBean 属性,一个叫 username,另一个叫 password。

你可能想到的是反射，通过 map 的 key 来查找 User 类的 Field！这么做是没有问题的，但我们要知道类的成员变量是私有的，虽然也可以通过反射去访问类的私有的成员变量，但我们也要清楚反射访问私有的东西是有“危险”的,所以还是建议通过 getUsername 和 setUsername 来访问 JavaBean 属性。

2.1 内省之获取 BeanInfo

我们这里不想去对 JavaBean 规范做过多的介绍，所以也就不在多介绍 BeanInfo 的“出身”了。你只需要知道如何得到它，以及 BeanInfo 有什么。

通过 java.beans.Introspector 的 getBeanInfo()方法来获取 java.beans.BeanInfo 实例。

```
BeanInfo beanInfo = Introspector.getBeanInfo(User.class);
```

2.2 得到所有属性描述符（PropertyDescriptor）

通过 BeanInfo 可以得到这个类的所有 JavaBean 属性的 PropertyDescriptor 对象。然后就可以通过 PropertyDescriptor 对象得到这个属性的 getter/setter 方法的 Method 对象了。

```
PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();
```

每个 PropertyDescriptor 对象对应一个 JavaBean 属性：

- String getName()：获取 JavaBean 属性名称；
- Method getReadMethod：获取属性的读方法；
- Method getWriteMethod：获取属性的写方法。

2.3 完成 Map 数据封装到 User 对象中

```
public void fun1() throws Exception {
    Map<String,String> map = new HashMap<String,String>();
    map.put("username", "admin");
    map.put("password", "admin123");

    BeanInfo beanInfo = Introspector.getBeanInfo(User.class);

    PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();

    User user = new User();
    for(PropertyDescriptor pd : pds) {
        String name = pd.getName();
        String value = map.get(name);
        if(value != null) {
            Method writeMethod = pd.getWriteMethod();
            writeMethod.invoke(user, value);
        }
    }

    System.out.println(user);
}
```

3 commons-beanutils

提到内省，不能不提 commons-beanutils 这个工具。它底层使用了内省，对内省进行了大量的简化！

使用 beanutils 需要的 jar 包：

- commons-beanutils.jar；
- commons-logging.jar；

3.1 设置 JavaBean 属性

```
        User user = new User();
```

```
        BeanUtils.setProperty(user, "username", "admin");
        BeanUtils.setProperty(user, "password", "admin123");

        System.out.println(user);
```

3.2 获取 JavaBean 属性

```
        User user = new User("admin", "admin123");

        String username = BeanUtils.getProperty(user, "username");
        String password = BeanUtils.getProperty(user, "password");

        System.out.println("username=" + username + ", password=" + password);
```

3.3 封装 Map 数据到 JavaBean 对象中

```
        Map<String,String> map = new HashMap<String,String>();
        map.put("username", "admin");
        map.put("password", "admin123");

        User user = new User();

        BeanUtils.populate(user, map);

        System.out.println(user);
```

4 JSP 与 JavaBean 相关的动作标签

在 JSP 中与 JavaBean 相关的标签有：

- <jsp:useBean>：创建 JavaBean 对象；
- <jsp:setProperty>：设置 JavaBean 属性；
- <jsp:getProperty>：获取 JavaBean 属性；

我们需要先创建一个 JavaBean 类：

User.java

```
package cn.itcast.domain;

public class User {
    private String username;
    private String password;
```

```
    public User(String username, String password) {
        this.username = username;
        this.password = password;
    }
    public User() {
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String toString() {
        return "User [username=" + username + ", password=" + password + "]";
    }
}
```

4.1 <jsp:useBean>

<jsp:useBean>标签的作用是创建 JavaBean 对象：

- 在当前 JSP 页面创建 JavaBean 对象；
- 把创建的 JavaBean 对象保存到域对象中；

```
<jsp:useBean id="user1" class="cn.itcast.domain.User" />
```

上面代码表示在当前 JSP 页面中创建 User 类型的对象，并且把它保存到 page 域中了。下面我们把<jsp:useBean>标签翻译成 Java 代码：

```
<%
cn.itcast.domain.User user1 = new cn.itcast.domain.User();
pageContext.setAttribute("user1", user1);
%>
```

这说明我们可以在 JSP 页面中完成下面的操作：

```
<jsp:useBean id="user1" class="cn.itcast.domain.User" />
<%=user1 %>
<%
    out.println(pageContext.getAttribute("user1"));
```

```
%>
```

<jsp:useBean>标签默认是把 JavaBean 对象保存到 page 域，还可以通过 scope 标签属性来指定保存的范围：

```
<jsp:useBean id="user1" class="cn.itcast.domain.User" scope="page"/>
<jsp:useBean id="user2" class="cn.itcast.domain.User" scope="request"/>
<jsp:useBean id="user3" class="cn.itcast.domain.User" scope="session"/>
<jsp:useBean id="user4" class="cn.itcast.domain.User" scope="applicatioin"/>
```

<jsp:useBean>标签其实不一定会创建对象!!!其实它会先在指定范围中查找这个对象，如果对象不存在才会创建，我们需要重新对它进行翻译：

```
<jsp:useBean id="user4" class="cn.itcast.domain.User" scope="applicatioin"/>
<%
    cn.itcast.domain.User user4 =
(cn.itcast.domain.User)application.getAttribute("user4");
    if(user4 == null) {
        user4 = new cn.itcast.domain.User();
        application.setAttribute("user4", user4);
    }
%>
```

4.2 <jsp:setProperty>和<jsp:getProperty>

<jsp:setProperty>标签的作用是给 JavaBean 设置属性值，而<jsp:getProperty>是用来获取属性值。在使用它们之前需要先创建 JavaBean：

```
<jsp:useBean id="user1" class="cn.itcast.domain.User" />
<jsp:setProperty property="username" name="user1" value="admin"/>
<jsp:setProperty property="password" name="user1" value="admin123"/>

用户名：<jsp:getProperty property="username" name="user1"/><br/>
密　码：<jsp:getProperty property="password" name="user1"/><br/>
```

EL（表达式语言）

1 EL 概述

1.1 EL 的作用

JSP2.0 要把 html 和 css 分离、要把 html 和 javascript 分离、要把 Java 脚本替换成标签。标签的好处是非 Java 人员都可以使用。

JSP2.0 – 纯标签页面，即：不包含<% … %>、<%! … %>，以及<%= … %>

EL（Expression Language）是一门表达式语言，它对应<%=…%>。我们知道在 JSP 中，表达式会被输出，所以 EL 表达式也会被输出。

1.2 EL 的格式

格式：${…}

例如：${1 + 2}

1.3 关闭 EL

如果希望整个 JSP 忽略 EL 表达式，需要在 page 指令中指定 isELIgnored="true"。

如果希望忽略某个 EL 表达式，可以在 EL 表达式之前添加“\”，例如：\${1 + 2}。

1.4 EL 运算符

运算符	说明	范例	结果
+	加	${17+5}	22
-	减	${17-5}	12
*	乘	${17*5}	85
/或 div	除	${17/5}或${17 div 5}	3
%或 mod	取余	${17%5}或${17 mod 5}	2
==或 eq	等于	${5==5}或${5 eq 5}	true
!=或 ne	不等于	${5!=5}或${5 ne 5}	false
<或 lt	小于	${3<5}或${3 lt 5}	true
>或 gt	大于	${3>5}或${3 gt 5}	false
<=或 le	小于等于	${3<=5}或${3 le 5}	true
>=或 ge	大于等于	${3>=5}或${3 ge 5}	false
&&或 and	并且	${true&&false}或${true and false}	false
!或 not	非	${!true}或${not true}	false
||或 or	或者	${true\|\|false}或${true or false}	true
empty	是否为空	${empty “”}，可以判断字符串、数据、集合的长度是否为 0，为 0 返回 true。empty 还可以与 not 或!一起使用。${not empty “”}	true

1.5 EL 不显示 null

当 EL 表达式的值为 null 时，会在页面上显示空白，即什么都不显示。

2 EL 表达式格式

先来了解一下 EL 表达式的格式！现在还不能演示它，因为需要学习了 EL11 个内置对象后才方

便显示它。

- 操作 List 和数组：${list[0]}、${arr[0]}；
- 操作 bean 的属性：${person.name}、${person['name']}，对应 person.getName()方法；
- 操作 Map 的值：${map.key}、${map['key']}，对应 map.get(key)。

3 EL 内置对象

EL 一共 11 个内置对象，无需创建即可以使用。这 11 个内置对象中有 10 个是 Map 类型的，最后一个是 pageContext 对象。

- pageScope
- requestScope
- sessionScope
- applicationScope
- param；
- paramValues；
- header；
- headerValues；
- initParam；
- cookie；
- pageContext；

3.1 域相关内置对象（重点）

域内置对象一共有四个：

- pageScope：${pageScope.name}等同与 pageContext.getAttribute("name")；
- requestScope：${requestScope.name}等同与 request.getAttribute("name")；
- sessionScoep： ${sessionScope.name}等同与 session.getAttribute("name")；
- applicationScope：${applicationScope.name}等同与 application.getAttribute("name")；

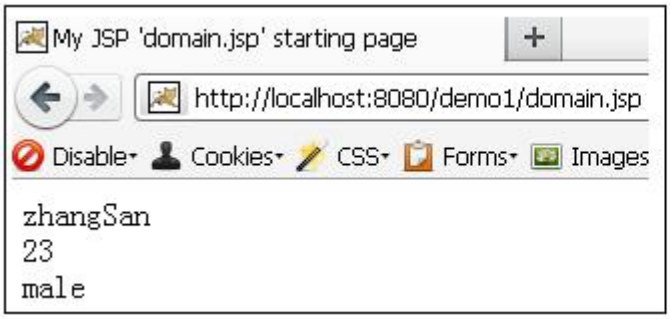
如果在域中保存的是 JavaBean 对象，那么可以使用 EL 来访问 JavaBean 属性。因为 EL 只做读取操作，所以 JavaBean 一定要提供 get 方法，而 set 方法没有要求。

Person.java

```
public class Person {
    private String name;
    private int age;
    private String sex;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getAge() {
```

```
        return age;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
}
```

```
  <body>
<%
    pageContext.setAttribute("p1", new Person("zhangSan", 23, "male"));
%>
${pageScope.p1.name }<br/>
${pageScope.p1.age }<br/>
${pageScope.p1.sex }<br/>
  </body>
```

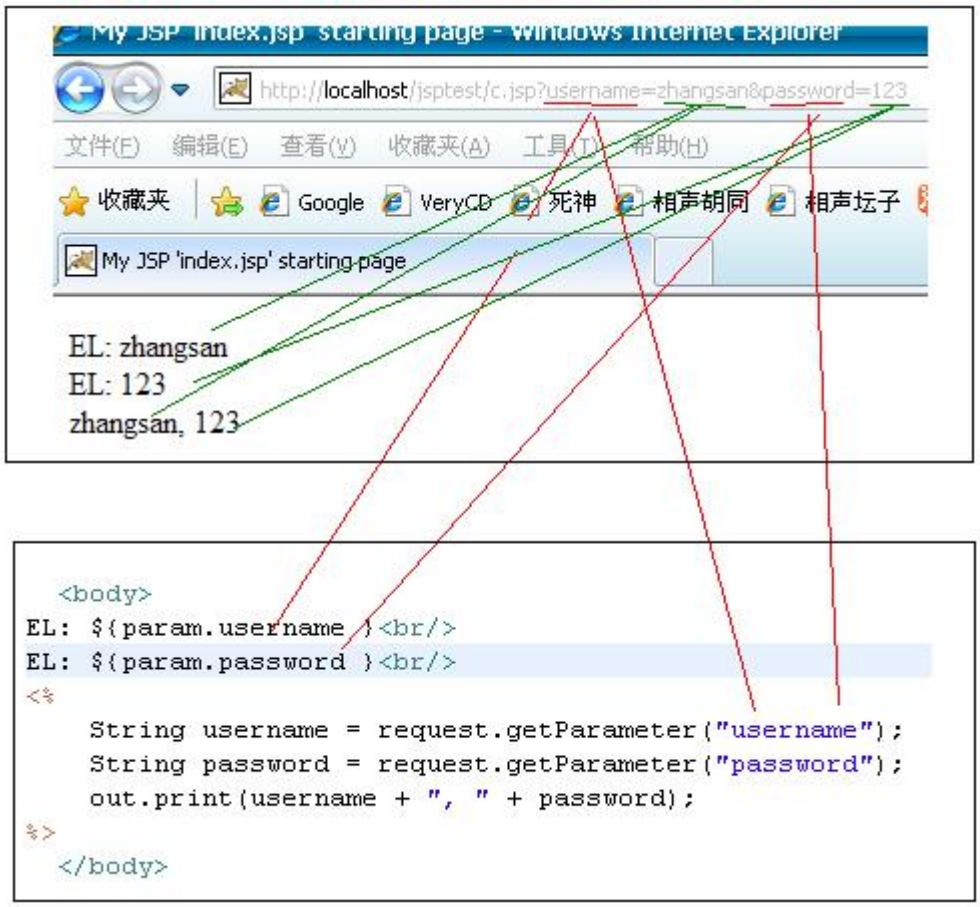


全域查找：${person}表示依次在 pageScope、requesScopet、sessionScope、appliationScope 四个域中查找名字为 person 的属性。

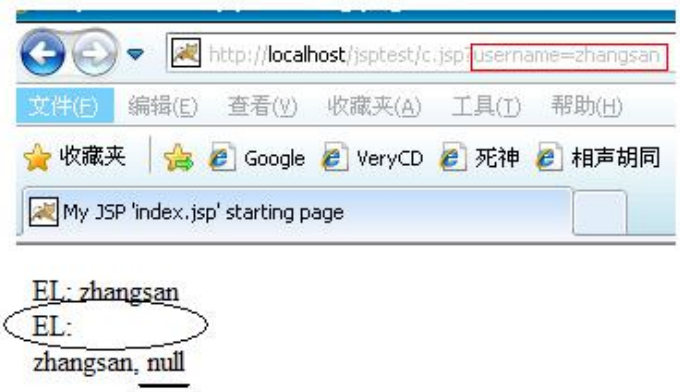
3.2 请求参数相关内置对象

param 和 paramValues 这两个内置对象是用来获取请求参数的。

- param：Map<String,String>类型，param 对象可以用来获取参数，与 request.getParameter()方法相同。

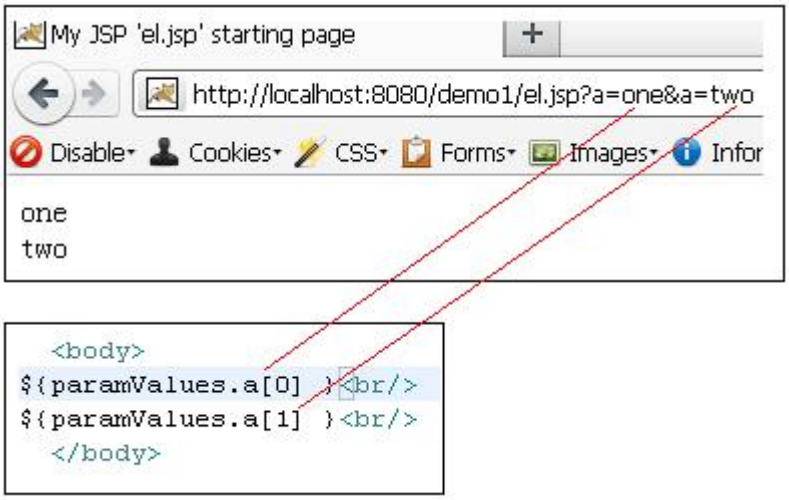


注意，在使用 EL 获取参数时，如果参数不存在，返回的是空字符串，而不是 null。这一点与使用 request.getParameter()方法是不同的。



在没有password参数时，EL显示空字符串，而request返回的是null

- paramValues：paramValues 是 Map<String, String[]>类型，当一个参数名，对应多个参数值时可以使用它。



3.3 请求头相关内置对象

header 和 headerValues 是与请求头相关的内置对象：

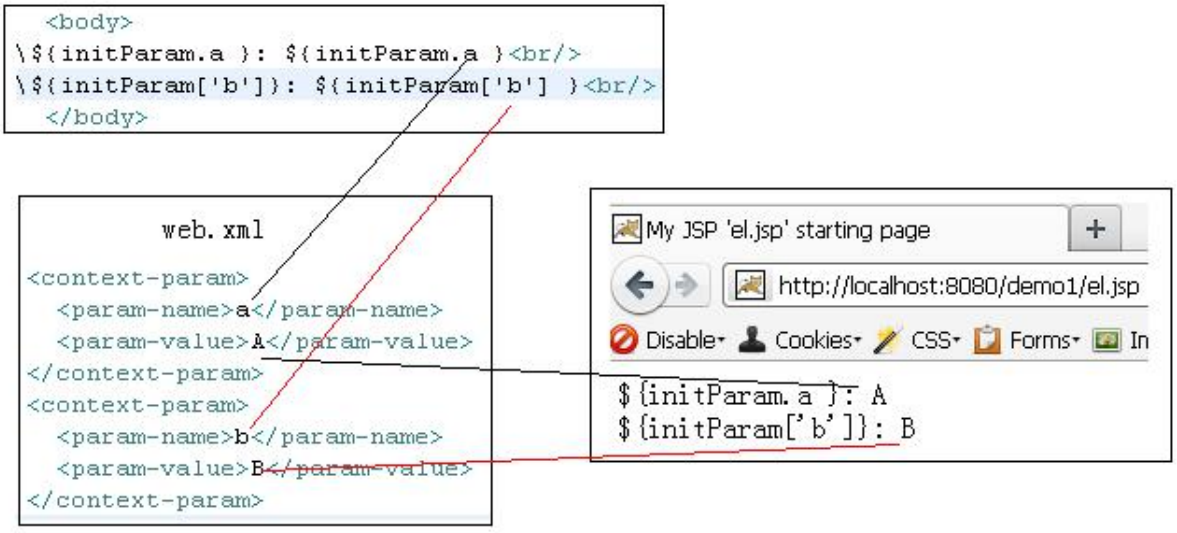
- header： Map<String,String>类型，用来获取请求头。


```
<body>
${header.Host }<br/>
</body>
```

- headerValues：headerValues 是 Map<String,String[]>类型。当一个请求头名称，对应多个值时，使用该对象，这里就不在赘述。

3.4 应用初始化参数相关内置对象

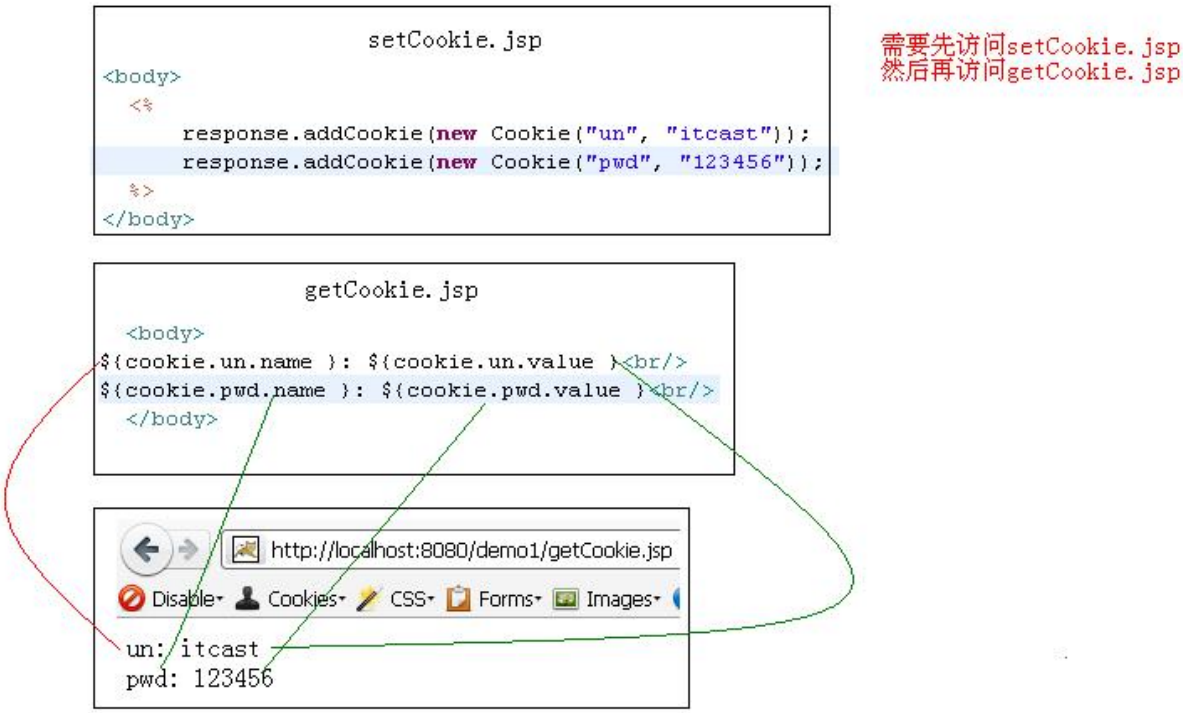
- initParam：initParam 是 Map<String,String>类型。它对应 web.xml 文件中的<context-param>参数。



EL表达式在获取Map的值或Bean的属性值时，可以使用“点”的方法，也可以使用“下标”方法。
${initParam.a}与${initParam['a']}，它们是完成相同的。但是，如果Map的键，或Bean的属性名中包含下划线时，那么就必须使用“下标”方法：${initParam['a_a']}

3.5 Cookie 相关内置对象

- cookie：cookie 是 Map<String,Cookie>类型，其中 key 是 Cookie 的名字，而值是 Cookie 对象本身。



3.6 pageContext 对象

pageContext：pageContext 是 PageContext 类型！可以使用 pageContext 对象调用 getXXX()方法，例如 pageContext.getRequest()，可以${pageContext.request}。也就是读取 JavaBean 属性！！！

EL 表达式	说明
${pageContext.request.queryString}	pageContext.getRequest().getQueryString();
${pageContext.request.requestURL}	pageContext.getRequest().getRequestURL();
${pageContext.request.contextPath}	pageContext.getRequest().getContextPath();
${pageContext.request.method}	pageContext.getRequest().getMethod();
${pageContext.request.protocol}	pageContext.getRequest().getProtocol();
${pageContext.request.remoteUser}	pageContext.getRequest().getRemoteUser();
${pageContext.request.remoteAddr}	pageContext.getRequest().getRemoteAddr();
${pageContext.session.new}	pageContext.getSession().isNew();
${pageContext.session.id}	pageContext.getSession().getId();
${pageContext.servletContext.serverInfo}	pageContext.getServletContext().getServerInfo();

EL 函数库

1 什么 EL 函数库

EL 函数库是由第三方对 EL 的扩展，我们现在学习的 EL 函数库是由 JSTL 添加的。JSTL 明天再学！

EL 函数库就是定义一些有**返回值**的**静态方法**。然后通过 EL 语言来调用它们！当然，不只是 JSTL 可以定义 EL 函数库，我们也可以自定义 EL 函数库。

EL 函数库中包含了很多对字符串的操作方法，以及对集合对象的操作。例如：${fn:length(“abc”)}会输出 3，即字符串的长度。

2 导入函数库

因为是第三方的东西，所以需要导入。导入需要使用 taglib 指令！

```
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
```

3 EL 函数库介绍

- String toUpperCase(String input)：
- String toLowerCase(String input)：
- int indexOf(String input, String substring)：
- boolean contains(String input, String substring)：
- boolean containsIgnoreCase(String input, String substring)：

- boolean startsWith(String input, String substring)：
- boolean endsWith(String input, String substring)：
- String substring(String input, int beginIndex, int endIndex)：
- String substringAfter(String input, String substring)：
- substringBefore(String input, String substring)：
- String escapeXml(String input)：
- String trim(String input)：
- String replace(String input, String substringBefore, String substringAfter)：
- String[] split(String input, String delimiters)：
- int length(Object obj)：
- String join(String array[], String separator)：

```
<%@taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
…
String[] strs = {"a", "b","c"};
List list = new ArrayList();
list.add("a");
pageContext.setAttribute("arr", strs);
pageContext.setAttribute("list", list);
%>
${fn:length(arr) }<br/><!--3-->
${fn:length(list) }<br/><!--1-->
${fn:toLowerCase("Hello") }<br/> <!-- hello -->
${fn:toUpperCase("Hello") }<br/> <!-- HELLO -->
${fn:contains("abc", "a")}<br/><!-- true -->
${fn:containsIgnoreCase("abc", "Ab")}<br/><!-- true -->
${fn:contains(arr, "a")}<br/><!-- true -->
${fn:containsIgnoreCase(list, "A")}<br/><!-- true -->
${fn:endsWith("Hello.java", ".java")}<br/><!-- true -->
${fn:startsWith("Hello.java", "Hell")}<br/><!-- true -->
${fn:indexOf("Hello-World", "-")}<br/><!-- 5 -->
${fn:join(arr, ";")}<br/><!-- a;b;c -->
${fn:replace("Hello-World", "-", "+")}<br/><!-- Hello+World -->
${fn:join(fn:split("a;b;c;", ";"), "-")}<br/><!-- a-b-c -->

${fn:substring("0123456789", 6, 9)}<br/><!-- 678 -->
${fn:substring("0123456789", 5, -1)}<br/><!-- 56789 -->
${fn:substringAfter("Hello-World", "-")}<br/><!-- World -->
${fn:substringBefore("Hello-World", "-")}<br/><!-- Hello -->
${fn:trim("    a b c    ")}<br/><!-- a b c -->
${fn:escapeXml("<html></html>")}<br/> <!-- <html></html> -->
```

4 自定义 EL 函数库

- 写一个类，写一个有返回值的静态方法；
- 编写 itcast.tld 文件，可以参数 fn.tld 文件来写，把 itcast.tld 文件放到/WEB-INF 目录下；
- 在页面中添加 taglib 指令，导入自定义标签库。

ItcastFuncations.java

```
package cn.itcast.el.funcations;

public class ItcastFuncations {
    public static String test() {
        return "传智播客自定义EL函数库测试";
    }
}
```

itcast.tld（放到 classes 下）

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                      http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.0</tlib-version>
  <short-name>itcast</short-name>
  <uri>http://www.itcast.cn/jsp/functions</uri>

  <function>
    <name>test</name>
    <function-class>cn.itcast.el.funcations.ItcastFuncations</function-class>
    <function-signature>String test()</function-signature>
  </function>
</taglib>
```

index.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib prefix="itcast" uri="/WEB-INF/itcast.tld" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h1>${itcast:test() }</h1>
```

```
  </body>
</html>
```

day08

EL 函数库

1 什么 EL 函数库

EL 函数库是由第三方对 EL 的扩展，我们现在学习的 EL 函数库是由 JSTL 添加的。JSTL 明天再学！

EL 函数库就是定义一些有**返回值**的**静态方法**。然后通过 EL 语言来调用它们！当然，不只是 JSTL 可以定义 EL 函数库，我们也可以自定义 EL 函数库。

EL 函数库中包含了很多对字符串的操作方法，以及对集合对象的操作。例如：${fn:length(“abc”)} 会输出 3，即字符串的长度。

2 导入函数库

因为是第三方的东西，所以需要导入。导入需要使用 taglib 指令！

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

3 EL 函数库介绍

- String toUpperCase(String input)：
- String toLowerCase(String input)：
- int indexOf(String input, String substring)：
- boolean contains(String input, String substring)：
- boolean containsIgnoreCase(String input, String substring)：
- boolean startsWith(String input, String substring)：
- boolean endsWith(String input, String substring)：
- String substring(String input, int beginIndex, int endIndex)：
- String substringAfter(String input, String substring)：
- substringBefore(String input, String substring)：
- String escapeXml(String input)：
- String trim(String input)：
- String replace(String input, String substringBefore, String substringAfter)：
- String[] split(String input, String delimiters)：
- int length(Object obj)：
- String join(String array[], String separator)：

```
<%@taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
…
```

```
String[] strs = {"a", "b","c"};
List list = new ArrayList();
list.add("a");
pageContext.setAttribute("arr", strs);
pageContext.setAttribute("list", list);
%>
${fn:length(arr) }<br/><!--3-->
${fn:length(list) }<br/><!--1-->
${fn:toLowerCase("Hello") }<br/> <!-- hello -->
${fn:toUpperCase("Hello") }<br/> <!-- HELLO -->
${fn:contains("abc", "a")}<br/><!-- true -->
${fn:containsIgnoreCase("abc", "Ab")}<br/><!-- true -->
${fn:contains(arr, "a")}<br/><!-- true -->
${fn:containsIgnoreCase(list, "A")}<br/><!-- true -->
${fn:endsWith("Hello.java", ".java")}<br/><!-- true -->
${fn:startsWith("Hello.java", "Hell")}<br/><!-- true -->
${fn:indexOf("Hello-World", "-")}<br/><!-- 5 -->
${fn:join(arr, ";")}<br/><!-- a;b;c -->
${fn:replace("Hello-World", "-", "+")}<br/><!-- Hello+World -->
${fn:join(fn:split("a;b;c;", ";"), "-")}<br/><!-- a-b-c -->

${fn:substring("0123456789", 6, 9)}<br/><!-- 678 -->
${fn:substring("0123456789", 5, -1)}<br/><!-- 56789 -->
${fn:substringAfter("Hello-World", "-")}<br/><!-- World -->
${fn:substringBefore("Hello-World", "-")}<br/><!-- Hello -->
${fn:trim("    a b c    ")}<br/><!-- a b c -->
${fn:escapeXml("<html></html>")}<br/> <!-- <html></html> -->
```

4 自定义 EL 函数库

- 写一个类，写一个有返回值的静态方法；
- 编写 itcast.tld 文件，可以参数 fn.tld 文件来写，把 itcast.tld 文件放到/WEB-INF 目录下；
- 在页面中添加 taglib 指令，导入自定义标签库。

ItcastFuncations.java

```
package cn.itcast.el.funcations;

public class ItcastFuncations {
    public static String test() {
        return "传智播客自定义EL函数库测试";
    }
}
```

itcast.tld（放到 classes 下）

```
<?xml version="1.0" encoding="UTF-8" ?>

<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
                      http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
  version="2.0">

  <tlib-version>1.0</tlib-version>
  <short-name>itcast</short-name>
  <uri>http://www.itcast.cn/jsp/functions</uri>

  <function>
    <name>test</name>
    <function-class>cn.itcast.el.funcations.ItcastFuncations</function-class>
    <function-signature>String test()</function-signature>
  </function>
</taglib>
```

index.jsp

```
<%@ page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<%@ taglib prefix="itcast" uri="/WEB-INF/itcast.tld" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h1>${itcast:test() }</h1>
  </body>
</html>
```

JSTL 标签库

1 什么是 JSTL

JSTL 是 apache 对 EL 表达式的扩展（也就是说 JSTL 依赖 EL），JSTL 是标签语言！JSTL 标签使用以来非常方便，它与 JSP 动作标签一定，只不过它不是 JSP 内置的标签，需要我们自己导包，以及指定标签库而已！

如果你使用 MyEclipse 开发 JavaWeb，那么在把项目发布到 Tomcat 时，你会发现，MyEclipse 会

在 lib 目录下存放 jstl 的 Jar 包！如果你没有使用 MyEclipse 开发那么需要自己来导入这个 JSTL 的 Jar 包：jstl-1.2.jar。

2 JSTL 标签库

JSTL 一共包含四大标签库：

- core：核心标签库，我们学习的重点；
- fmt：格式化标签库，只需要学习两个标签即可；
- sql：数据库标签库，不需要学习了，它过时了；
- xml：xml 标签库，不需要学习了，它过时了。

3 使用 taglib 指令导入标签库

除了 JSP 动作标签外，使用其他第三方的标签库都需要：

- 导包；
- 在使用标签的 JSP 页面中使用 taglib 指令导入标签库；

下面是导入 JSTL 的 core 标签库：

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
```

- prefix="c"：指定标签库的前缀，这个前缀可以随便给值，但大家都会在使用 core 标签库时指定前缀为 c；
- uri="http://java.sun.com/jstl/core"：指定标签库的 uri，它不一定是真实存在的网址，但它可以让 JSP 找到标签库的描述文件；

4 core 标签库常用标签

4.1 out 和 set

out

<c:out value=”aaa”/>	输出 aaa 字符串常量
<c:out value=”${aaa}”/>	与${aaa}相同
<c:out value=”${aaa}” default=”xxx”/>	当${aaa}不存在时，输出 xxx 字符串
<% request.setAttribute("a","<script>alert('hello');</script>"); %> <c:out value="${a }" default="xxx" escapeXml="false" />	当 escapeXml 为 false，不会转换“<”、“>”。这可能会受到 JavaScript 攻击。

set

<c:set var=”a” value=”hello”/>	在 pageContext 中添加 name 为 a，value 为 hello 的数据。
<c:set var=”a” value=”hello” scope=”session”/>	在 session 中添加 name 为 a，value

	为 hello 的数据。

4.2 remove

<%
pageContext.setAttribute("a", "pageContext");
request.setAttribute("a", "session");
session.setAttribute("a", "session");
application.setAttribute("a", "application");
%>
<c:remove var="a"/>
<c:out value="${a }" default="none"/>
	删除所有域中 name 为 a 的数据！

<c:remove var="a" scope=”page”/>	删除 pageContext 中 name 为 a 的数据！

4.3 url

url 标签会在需要 URL 重写时添加 sessionId。

<c:url value="/"/>	输出上下文路径：/day08_01/
<c:url value="/" var="a" scope="request"/>	把本该输出的结果赋给变量 a。范围为 request
<c:url value="/AServlet"/>	输出：/day08_01/AServlet
<c:url value="/AServlet"> <c:param name="username" value="abc"/> <c:param name="password" value="123"/> </c:url>	输出：/day08_01/AServlet?username=abc&password=123 如果参数中包含中文，那么会自动使用 URL 编码！

4.4 if

if 标签的 test 属性必须是一个 boolean 类型的值，如果 test 的值为 true，那么执行 if 标签的内容，否则不执行。

```
<c:set var="a" value="hello"/>
<c:if test="${not empty a }">
    <c:out value="${a }"/>
</c:if>
```

4.5 choose

choose 标签对应 Java 中的 if/else if/else 结构。when 标签的 test 为 true 时，会执行这个 when 的内容。当所有 when 标签的 test 都为 false 时，才会执行 otherwise 标签的内容。

```
<c:set var="score" value="${param.score }"/>
<c:choose>
    <c:when test="${score > 100 || score < 0}">错误的分数：${score }</c:when>
    <c:when test="${score >= 90 }">A级</c:when>
```

```
    <c:when test="${score >= 80 }">B级</c:when>
    <c:when test="${score >= 70 }">C级</c:when>
    <c:when test="${score >= 60 }">D级</c:when>
    <c:otherwise>E级</c:otherwise>
</c:choose>
```

4.6 forEach

forEach 当前就是循环标签了，forEach 标签有多种两种使用方式：

- 使用循环变量，指定开始和结束值，类似 for(int i = 1; i <= 10; i++) {};
- 循环遍历集合，类似 for(Object o : 集合);

循环变量方式：

```
<c:set var="sum" value="0" />
<c:forEach var="i" begin="1" end="10">
    <c:set var="sum" value="${sum + i}" />
</c:forEach>
<c:out value="sum = ${sum }"/>
```

```
<c:set var="sum" value="0" />
<c:forEach var="i" begin="1" end="10" step="2">
    <c:set var="sum" value="${sum + i}" />
</c:forEach>
<c:out value="sum = ${sum }"/>
```

遍历集合或数组方式：

```
<%
String[] names = {"zhangSan", "liSi", "wangWu", "zhaoLiu"};
pageContext.setAttribute("ns", names);
%>
<c:forEach var="item" items="${ns }">
    <c:out value="name: ${item }"/><br/>
</c:forEach>
```

遍历 List

```
<%
    List<String> names = new ArrayList<String>();
    names.add("zhangSan");
    names.add("liSi");
    names.add("wangWu");
    names.add("zhaoLiu");
    pageContext.setAttribute("ns", names);
%>
```

```
<c:forEach var="item" items="${ns }">
    <c:out value="name: ${item }"/><br/>
</c:forEach>
```

遍历 Map

```
<%
    Map<String,String> stu = new LinkedHashMap<String,String>();
    stu.put("number", "N_1001");
    stu.put("name", "zhangSan");
    stu.put("age", "23");
    stu.put("sex", "male");
    pageContext.setAttribute("stu", stu);
%>
<c:forEach var="item" items="${stu }">
    <c:out value="${item.key }: ${item.value }"/><br/>
</c:forEach>
```

forEach 标签还有一个属性：varStatus，这个属性用来指定接收“循环状态”的变量名，例如：<forEach varStatus=”vs” .../>，这时就可以使用 vs 这个变量来获取循环的状态了。

- count：int 类型，当前以遍历元素的个数；
- index：int 类型，当前元素的下标；
- first：boolean 类型，是否为第一个元素；
- last：boolean 类型，是否为最后一个元素；
- current：Object 类型，表示当前项目。

```
<c:forEach var="item" items="${ns }" varStatus="vs">
    <c:if test="${vs.first }">第一行：</c:if>
    <c:if test="${vs.last }">最后一行：</c:if>
    <c:out value="第${vs.count }行: "/>
    <c:out value="[${vs.index }]: "/>
    <c:out value="name: ${vs.current }"/><br/>
</c:forEach>
```

5 fmt 标签库常用标签

fmt 标签库是用来格式化输出的，通常需要格式化的有时间和数字。

格式化时间：

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
......
<%
    Date date = new Date();
    pageContext.setAttribute("d", date);
%>
```

```
<fmt:formatDate value="${d }" pattern="yyyy-MM-dd HH:mm:ss"/>
```

格式化数字：

```
<%
    double d1 = 3.5;
    double d2 = 4.4;
    pageContext.setAttribute("d1", d1);
    pageContext.setAttribute("d2", d2);
%>
<fmt:formatNumber value="${d1 }" pattern="0.00"/><br/>
<fmt:formatNumber value="${d2 }" pattern="#.##"/>
```

自定义标签

1 自定义标签概述

1.1 自定义标签的步骤

其实我们在 JSP 页面中使用标签就等于调用某个对象的某个方法一样，例如：<c:if test="">，这就是在调用对象的方法一样。自定义标签其实就是自定义类一样！

- 定义标签处理类：必须是 Tag 或 SimpleTag 的实现类；
- 编写标签库描述符文件（TLD）；

SimpleTag 接口是 JSP2.0 中新给出的接口，用来简化自定义标签，所以现在我们基本上都是使用 SimpleTag。

Tag 是老的，传统的自定义标签时使用的接口，现在不建议使用它了。

1.2 SimpleTag 接口介绍

SimpleTag 接口内容如下：

- void doTag()：标签执行方法；
- JspTag getParent()：获取父标签；
- void setParent(JspTag parent)：设置父标签
- void setJspContext(JspContext context)：设置 PageContext
- void setJspBody(JspFragment jspBody)：设置标签体对象；

请记住，万物皆对象！在 JSP 页面中的标签也是对象！你可以通过查看 JSP 的“真身”清楚的知道，所有标签都会变成对象的方法调用。标签对应的类我们称之为“标签处理类”！

标签的生命周期：

1. 当容器（Tomcat）第一次执行到某个标签时，会创建标签处理类的实例；
2. 然后调用 setJspContext(JspContext)方法，把当前 JSP 页面的 pageContext 对象传递给这个方法；
3. 如果当前标签有父标签，那么使用父标签的标签处理类对象调用 setParent(JspTag)方法；
4. 如果标签有标签体，那么把标签体转换成 JspFragment 对象，然后调用 setJspBody()方法；
5. 每次执行标签时，都调用 doTag()方法，它是标签处理方法。

HelloTag.java

```
public class HelloTag implements SimpleTag {
    private JspTag parent;
    private PageContext pageContext;
    private JspFragment jspBody;

    public void doTag() throws JspException, IOException {
        pageContext.getOut().print("Hello Tag!!!");
    }
    public void setParent(JspTag parent) {
        this.parent = parent;
    }
    public JspTag getParent() {
        return this.parent;
    }
    public void setJspContext(JspContext pc) {
        this.pageContext = (PageContext) pc;
    }
    public void setJspBody(JspFragment jspBody) {
        this.jspBody = jspBody;
    }
}
```

1.3 标签库描述文件（TLD）

标签库描述文件是用来描述当前标签库中的标签的！标签库描述文件的扩展名为 tld，你可以把它放到 WEB-INF 下，这样就不会被客户端直接访问到了。

hello.tld

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xml="http://www.w3.org/XML/1998/namespace"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee

    http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd ">
```

```
    <tlib-version>1.0</tlib-version>
    <short-name>itcast</short-name>
    <uri>http://www.itcast.cn/tags</uri>
    <tag>
        <name>hello</name>
        <tag-class>cn.itcast.tag.HelloTag</tag-class>
        <body-content>empty</body-content>
    </tag>
</taglib>
```

1.4 使用标签

在页面中使用标签分为两步：

- 使用 taglib 导入标签库；
- 使用标签；

```
<%@ taglib prefix="it" uri="/WEB-INF/hello.tld" %>
......
<it:hello/>
```

2 自定义标签进阶

2.1 继承 SimpleTagSupport

继承 SimpleTagSuppport 要比实现 SimpleTag 接口方便太多了，现在你只需要重写 doTag()方法，其他方法都已经被 SimpleTagSuppport 完成了。

```
public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        this.getJspContext().getOut().write("<p>Hello SimpleTag!</p>");
    }
}
```

2.2 有标签体的标签

我们先来看看标签体内容的可选值：

<body-content>元素的可选值有：

- empty：无标签体。
- JSP：传统标签支持它，**SimpleTag 已经不再支持使用<body-content>JSP</body-content>。**标签体内容可以是任何东西：EL、JSTL、<%=%>、<%%>，以及 html；
- scriptless：标签体内容不能是 Java 脚本，但可以是 EL、JSTL 等。在 SimpleTag 中，**如果需要有标签体，那么就使用该选项；**

- tagdependent：标签体内容不做运算，由标签处理类自行处理，无论标签体内容是 EL、JSP、JSTL，都不会做运算。**这个选项几乎没有人会使用！**

自定义有标签体的标签需要：

- 获取标签体对象：JspFragment jspBody = getJspBody();；
- 把标签体内容输出到页面：jspBody.invoke(null)；
- tld 中指定标签内容类型：scriptless。

```
public class HelloTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        PageContext pc = (PageContext) this.getJspContext();
        HttpServletRequest req = (HttpServletRequest) pc.getRequest();
        String s = req.getParameter("exec");
        if(s != null && s.endsWith("true")) {
            JspFragment body = this.getJspBody();
            body.invoke(null);
        }
    }
}
```

```
    <tag>
        <name>hello</name>
        <tag-class>cn.itcast.tags.HelloTag</tag-class>
        <body-content>scriptless</body-content>
    </tag>
```

```
    <itcast:hello>
        <h1>哈哈哈~</h1>
    </itcast:hello>
```

2.3 不执行标签下面的页面内容

如果希望在执行了自定义标签后，不再执行 JSP 页面下面的东西，那么就需要在 doTag()方法中使用 SkipPageException。

```
public class SkipTag extends SimpleTagSupport {
    public void doTag() throws JspException, IOException {
        this.getJspContext().getOut().print("<h1>只能看到我！</h1>");
        throw new SkipPageException();
    }
}
```

```
    <tag>
        <name>skip</name>
        <tag-class>cn.itcast.tags.SkipTag</tag-class>
        <body-content>empty</body-content>
    </tag>
```

```
<itcast:skip/>
<h1>看不见我！</h1>
```

2.4 带有属性的标签

一般标签都会带有属性，例如<c:if test="">，其中 test 就是一个 boolean 类型的属性。完成带有属性的标签需要：

- 在处理类中给出 JavaBean 属性（提供 get/set 方法）；
- 在 TLD 中部属相关属性。

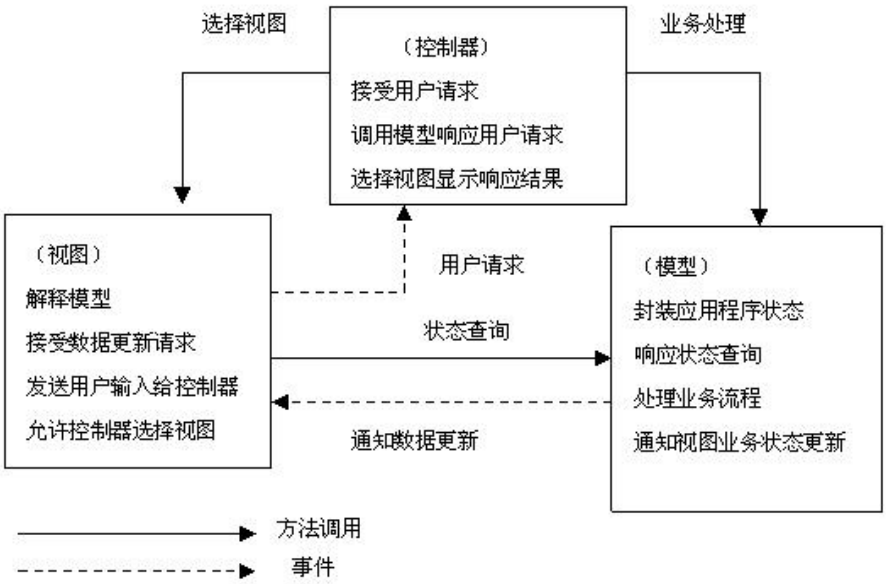
```
public class IfTag extends SimpleTagSupport {
    private boolean test;
    public boolean isTest() {
        return test;
    }
    public void setTest(boolean test) {
        this.test = test;
    }
    @Override
    public void doTag() throws JspException, IOException {
        if(test) {
            this.getJspBody().invoke(null);
        }
    }
}
```

```
    <tag>
        <name>if</name>
        <tag-class>cn.itcast.tag.IfTag</tag-class>
        <body-content>scriptless</body-content>
        <attribute>
            <name>test</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
```

```
<%
    pageContext.setAttribute("one", true);
    pageContext.setAttribute("two", false);
%>
<it:if test="${one }">xixi</it:if>
<it:if test="${two }">haha</it:if>
<it:if test="true">hehe</it:if>
```

MVC

1 MVC 设计模式



MVC 设计模式

MVC 模式（Model-View-Controller）是软件工程中的一种软件架构模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC 模式最早为 Trygve Reenskaug 提出，为施乐帕罗奥多研究中心（Xerox PARC）的 Smalltalk 语言发明的一种软件设计模式。

MVC 可对程序的后期维护和扩展提供了方便，并且使程序某些部分的重用提供了方便。而且 MVC 也使程序简化，更加直观。

- 控制器 Controller：对请求进行处理，负责请求转发；
- 视图 View：界面设计人员进行图形界面设计；
- 模型 Model：程序编写程序应用的功能（实现算法等等）、数据库管理；

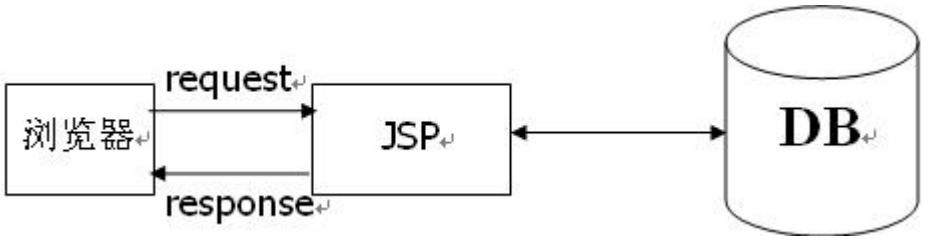
注意，MVC 不是 Java 的东西，几乎现在所有 B/S 结构的软件都采用了 MVC 设计模式。但是要注意，MVC 在 B/S 结构软件并没有完全实现，例如在我们今后的 B/S 软件中并不会有事件驱动！

2 JavaWeb 与 MVC

JavaWeb 的经历了 JSP Model1、JSP Model1 二代、JSP Model2 三个时期。

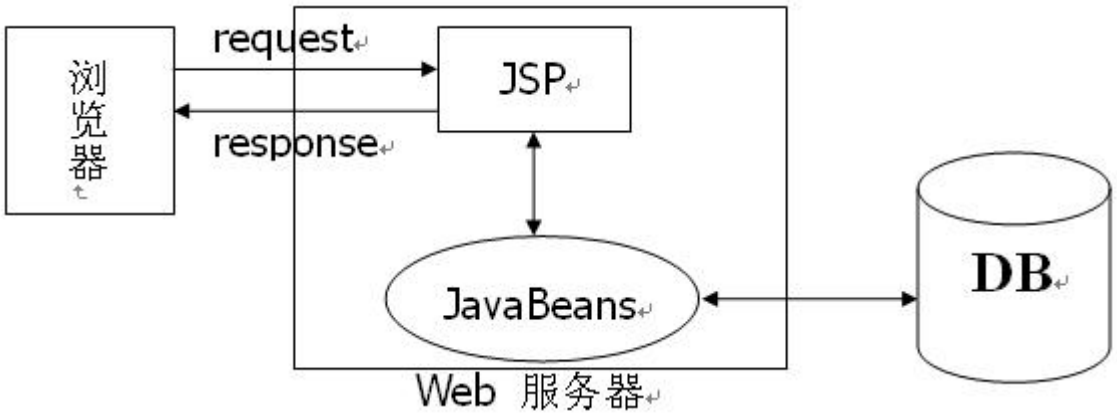
2.1 JSP Model1 第一代

JSP Model1 是 JavaWeb 早期的模型，它适合小型 Web 项目，开发成本低！Model1 第一代时期，服务器端只有 JSP 页面，所有的操作都在 JSP 页面中，连访问数据库的 API 也在 JSP 页面中完成。也就是说，所有的东西都耦合在一起，对后期的维护和扩展极为不利。



2.2 JSP Model1 第二代

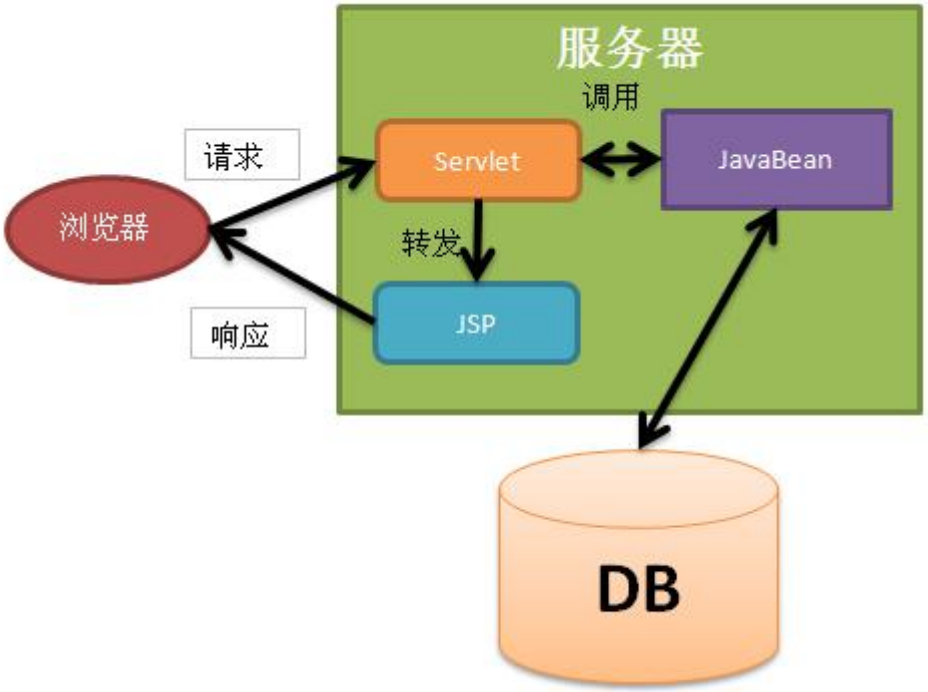
JSP Model1 第二代有所改进，把业务逻辑的内容放到了 JavaBean 中，而 JSP 页面负责显示以及请求调度的工作。虽然第二代比第一代好了些，但还让 JSP 做了过多的工作，JSP 中把视图工作和请求调度（控制器）的工作耦合在一起了。



2.3 JSP Model2

JSP Model2 模式已经可以清晰的看到 MVC 完整的结构了。

- JSP：视图层，用来与用户打交道。负责接收用来的数据，以及显示数据给用户；
- Servlet：控制层，负责找到合适的模型对象来处理业务逻辑，转发到合适的视图；
- JavaBean：模型层，完成具体的业务工作，例如：开启、转账等。



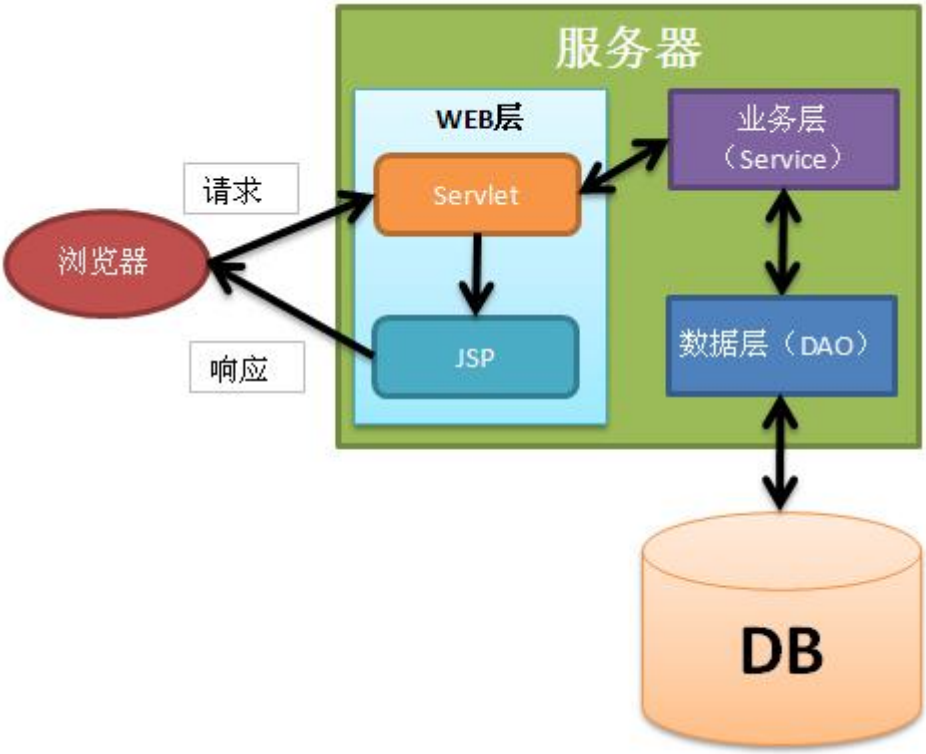
JSP Model2 适合多人合作开发大型的 Web 项目，各司其职，互不干涉，有利于开发中的分工，有利于组件的重用。但是，Web 项目的开发难度加大，同时对开发人员的技术要求也提高了。

JavaWeb 三层框架

我们常说的三层框架是由 JavaWeb 提出的，也就是说这是 JavaWeb 独有的！
所谓三层是表述层（WEB 层）、业务逻辑层（Business Logic），以及数据访问层（Data Access）。

- WEB（表述）层：包含 JSP 和 Servlet 等与 WEB 相关的内容；
- 业务逻辑层：业务层中不包含 JavaWeb API，它只关心业务逻辑，它对应功能；
- 数据访问层：封装了对数据库的访问细节；

注意，在业务层中不能出现 JavaWeb API，例如 request、response 等。也就是说，业务层代码是可重用的，甚至可以应用到非 Web 环境中。业务层的每个方法可以理解成一个万能，例如转账业务方法。业务层依赖数据层，而 Web 层依赖业务层！


服务器
WEB层
Servlet
JSP
业务层
（Service）
数据层（DAO）
请求
浏览器
响应
DB

day08

案例：用户注册登录

要求：3 层框架，使用验证码

1 功能分析

- 注册
- 登录

1.1 JSP 页面

- regist.jsp
  - 注册表单：用户输入注册信息；
  - 回显错误信息：当注册失败时，显示错误信息；
- login.jsp
  - 登录表单：用户输入登录信息；
  - 回显错误便利店：当登录失败时，显示错误信息；
- index.jsp
  - 用户已登录：显示当前用户名，以及“退出”链接；
  - 用户未登录：显示“您还没有登录”；

1.2 实体类

User：

- String username；
- String password；

1.3 Servlet

- VerifyCodeServlet
  - 生成验证码；
  - 在 session 中保存验证码文本；
  - 把图片输出到页面
- RegistServlet
  - 获取用户名、密码，封装到 User 对象中；
  - 获取验证码、获取确认密码；
  - 校验用户名、密码、验证码不能为空，校验失败，向 request 中保存错误信息，转发回

regist.jsp 显示错误信息；

- 比较两次输入的错误是否一致，如果不一致，向 request 中保存错误信息，转发回 regist.jsp 显示错误信息；
- 获取 session 中的验证码，与表单输入的验证码比较，如果不一致，向 request 中保存错误信息，转发回 regist.jsp 显示错误信息；
- 使用 UserService 的 regist()方法完成注册，如果注册失败，向 request 中保存错误信息，转发回 regist.jsp 显示错误信息，如果注册成功，转发到 login.jsp 页面，表示注册成功；

- LoginServlet
  - 获取用户名、密码、验证码；
  - 校验用户名、密码、验证码是否为空，校验失败，向 request 中保存错误信息，转发回 login.jsp 显示错误信息；
  - 获取 session 中的验证码，与表单中的验证码比较，如果不同，向 request 中保存错误信息，转发回 login.jsp 显示错误信息；
  - 删除 session 中的验证码；
  - 通过 UserService 的 login()方法完成登录，如果抛出异常，获取异常信息，保存到 request 中，转发到 login.jsp 显示错误信息；
  - 向 session 中保存当前用户对象；
  - 转发到 index.jsp 页面，表示登录成功！
- QuitServlet
  - 获取 session，销毁之；
  - 重定向到 index.jsp；

1.4 Service

UserException：为 UserService 使用的异常类；

UserService：

- void regist(User user)：
  - 使用 UserDao 的 findByUsername()方法查询名为 user.getUsername()的用户，如果用户存在，说明用户名已经被注册，抛出异常；
  - 使用 UserDao 的 add(User)方法保存用户信息；
- User login(String username, String password)：
  - 使用 UserDao 的 findByUsername()方法查询名为 user.getUsername()的用户，如果用户不存在，说明用户名错误，抛出异常；
  - 如果查询到了 User，那么比较参数 password 与 user.getPassword()是否相等，如果不等，说明密码错误，抛出异常；
  - 如果一致，表示登录成功，返回 User 对象；

1.5 DAO

UserDao：

- void add(User)：
  - 创建 SAXReader 对象，获取 Document 对象，再获取根元素；
  - 给 root 元素添加子元素；

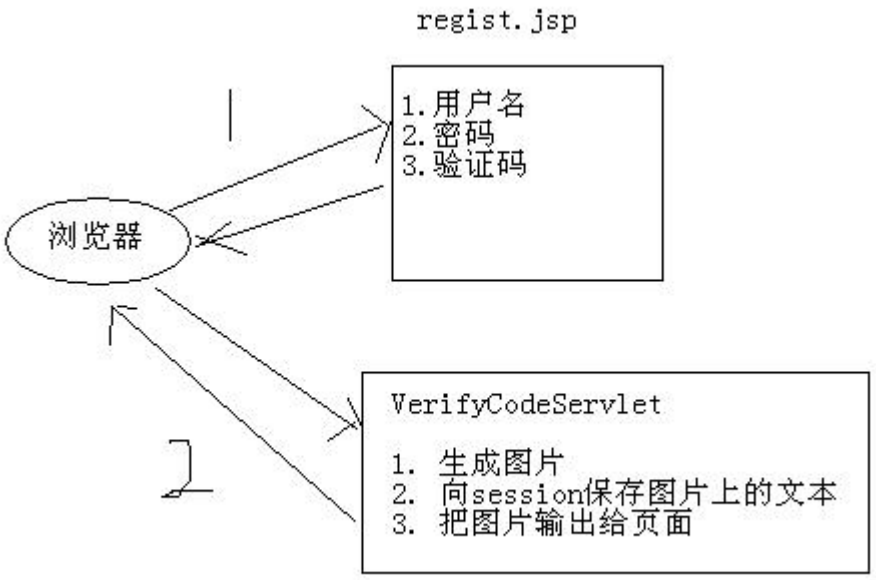
- 给子元素设置 username 属性，值为 user.getUsername()；
- 给子元素设置 password 属性，值为 user.getPassword()；
- 创建 OutputFormat 对象，指定缩进为“\t”，指定添加换行；
- 设置 OutputFormat 清空原有空白；
- 使用 FileWrtier 和 OutputFormat 创建 XMLWriter 对象；
- 使用 XMLWriter 对象的 write()保存 Document；
- 关闭 XMLWriter 对象；

- User findByUsername(String username)：
  - 创建 SAXReader 对象，获取 Document 对象；
  - 使用 Xpath（//user[username='xxx']）来查询元素；
  - 如果元素没有查询到，返回 null；
  - 如果元素查询到了，那么创建 User 对象；
  - 把元素的 username 属性赋给 User 的 username 属性；
  - 把元素的 password 属性赋给 User 的 password 属性；
  - 返回 user 对象；

2 流程图

2.1 注册

- 用户在浏览器地址栏中请求 regist.jsp；
- 服务器发送 html 给浏览器；
- 浏览器收到 html，开始解析，并显示；
- 解析到<img>时，请求 VerifyCodeServlet；
- VerifyCodeServlet 生成验证码图片，保存验证码文本，把图片响应给浏览器；
- 浏览器显示在页面中显示图片。



2.2 登录

此处省略 10000 字

3 代码

login.jsp

```
  <body>
    <h1>登录</h1>
    <hr/>
    <p style="font-weight: 900; color: red;">${msg }</p>
    <form action="<c:url value='/LoginServlet'/>" method="post">
    用户名：<input type="text" name="username" value="${user.username }" /><br/>
    密　码：<input type="password" name="password"/><br/>
    验证码：<input type="text" name="loginCode" size="2"/>
    <img id="vCode" src="<c:url value='/VerifyCodeServlet?name=loginCode'/>"
border="2"/>
    <a href="javascript:_change()" style="font-size: 12;">看不清，换一张
</a><br/>
        <input type="submit" value="登录"/>
    </form>
  </body>
```

```
    <script type="text/javascript">
    function _change() {
        var img = document.getElementById("vCode");
        img.src = "<c:url value='/VerifyCodeServlet?name=loginCode&'/>" + new
Date().getTime();
    }
    </script>
```

regist.jsp

```
  <body>
    <h1>注册</h1>
    <hr/>
    <p style="font-weight: 900; color: red;">${msg }</p>
    <form action="<c:url value='/RegistServlet'/>" method="post">
    用户名：<input type="text" name="username" value="${user.username }" /><br/>
    密　码：<input type="password" name="password"/><br/>
    确认密码：<input type="password" name="repassword"/><br/>
    验证码：<input type="text" name="registCode" size="2"/>
    <img id="vCode" src="<c:url value='/VerifyCodeServlet?name=registCode'/>"
border="2"/>
    <a href="javascript:_change()" style="font-size: 12;">看不清，换一张
</a><br/>
        <input type="submit" value="注册"/>
    </form>
  </body>
```

```
    <script type="text/javascript">
        function _change() {
            var img = document.getElementById("vCode");
            img.src = "<c:url value='/VerifyCodeServlet?name=registCode&'/>" +
new Date().getTime();
        }
    </script>
```

index.jsp

```
  <body>
    <h1>主页</h1>
    <hr/>
<c:choose>
    <c:when test="${empty sessionScope.user }">
        您还没有登录
    </c:when>
    <c:otherwise>
        用户名：${sessionScope.user.username }
        <a href="<c:url value='/QuitServlet'/>">退出</a>
    </c:otherwise>
</c:choose>
  </body>
```

VerifyCodeServlet

```
public class VerifyCodeServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String name = request.getParameter("name");

        VerifyCode vc = new VerifyCode();//创建验证码类
        BufferedImage image  = vc.getImage();//创建验证码图片
        request.getSession().setAttribute(name, vc.getText());//获取验证码文本
System.out.println(vc.getText());
        VerifyCode.output(image, response.getOutputStream());//输出图片到页面
    }
}
```

RegistServlet

```
    public class RegistServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        User user = new User();
        try {
            BeanUtils.populate(user, request.getParameterMap());
        } catch (Exception e) {
        }
        String loginCode = request.getParameter("registCode");
        String repassword = request.getParameter("repassword");

        if(user.getUsername() == null || user.getUsername().trim().isEmpty())
{
            request.setAttribute("msg", "用户名不能为空!");
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }
        if(user.getPassword() == null || user.getPassword().trim().isEmpty())
{
            request.setAttribute("msg", "密码不能为空!");
```

```
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }
        if(!user.getPassword().equals(repassword)) {
            request.setAttribute("msg", "两次输入不一致！");
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }
        if(loginCode == null || loginCode.trim().isEmpty()) {
            request.setAttribute("msg", "验证码不能为空！");
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }

        String vCode =
(String)request.getSession().getAttribute("registCode");
        request.getSession().removeAttribute("registCode");
        if(!vCode.equalsIgnoreCase(loginCode)) {
            request.setAttribute("msg", "验证码错误！");
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }

        UserService userService = new UserService();
        try {
            userService.regist(user);
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
        } catch (UserException e) {
            request.setAttribute("msg", e.getMessage());
            request.setAttribute("user", user);
            request.getRequestDispatcher("/regist.jsp").forward(request,
response);
            return;
        }
```

```
    }
}
```

LoginServlet

```
public class LoginServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        String username = request.getParameter("username");
        String password = request.getParameter("password");
        String loginCode = request.getParameter("loginCode");

        if(username == null || username.trim().isEmpty()) {
            request.setAttribute("msg", "用户名不能为空!");
            request.setAttribute("username", username);
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
            return;
        }
        if(password == null || password.trim().isEmpty()) {
            request.setAttribute("msg", "密码不能为空!");
            request.setAttribute("username", username);
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
            return;
        }
        if(loginCode == null || loginCode.trim().isEmpty()) {
            request.setAttribute("msg", "验证码不能为空! ");
            request.setAttribute("username", username);
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
            return;
        }

        String vCode = (String)request.getSession().getAttribute("loginCode");
        request.getSession().removeAttribute("loginCode");
        if(!vCode.equalsIgnoreCase(loginCode)) {
            request.setAttribute("msg", "验证码错误! ");
            request.setAttribute("username", username);
```

```
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
            return;
        }

        UserService userService = new UserService();
        User user;
        try {
            user = userService.login(username, password);
        } catch (UserException e) {
            request.setAttribute("msg", e.getMessage());
            request.setAttribute("username", username);
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
            return;
        }

        request.getSession().setAttribute("user", user);
        request.getRequestDispatcher("/index.jsp").forward(request,
response);
    }
}
```

QuitServlet

```
public class QuitServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        // 使session失效
        request.getSession().invalidate();
        response.sendRedirect(request.getContextPath() + "/index.jsp");
    }
}
```

UserException

```
public class UserException extends Exception {
    public UserException() {}
    public UserException(String message, Throwable cause) {
        super(message, cause);
    }
    public UserException(String message) {
```

```
        super(message);
    }
    public UserException(Throwable cause) {
        super(cause);
    }
}
```

UserService

```
public class UserService {
    private UserDao userDao = new UserDao();

    public User login(String username, String password) throws UserException {
        User user = userDao.findByUsername(username);
        if(user == null) {
            throw new UserException("用户名错误！");
        }
        if(!user.getPassword().equals(password)) {
            throw new UserException("密码错误！");
        }
        return user;
    }

    public void regist(User user) throws UserException {
        User _user = userDao.findByUsername(user.getUsername());
        if(_user != null) {
            throw new UserException("用户名已注册！");
        }
        userDao.add(user);
    }
}
```

UserDao

```
public class UserDao {
    private String path;

    public UserDao() {
        path = this.getClass().getResource("/users.xml").getPath();
    }

    public void add(User user) {
        try {
```

```
            SAXReader reader = new SAXReader();
            Document doc = reader.read(path);

            Element root = doc.getRootElement();
            Element userEle = root.addElement("user");
            userEle.addAttribute("username", user.getUsername());
            userEle.addAttribute("password", user.getPassword());

            // 创建格式化器，使用\t缩进，添加换行
            OutputFormat format = new OutputFormat("\t", true);
            // 清空数据中原有的换行
            format.setTrimText(true);
            // 创建XML输出流对象
            XMLWriter writer = new XMLWriter(new FileWriter(path), format);
            // 输出Document
            writer.write(doc);
            // 关闭流
            writer.close();

        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public User findByUsername(String username) {
        try {
            SAXReader reader = new SAXReader();
            Document doc = reader.read(path);
            Element ele = (Element) doc.selectSingleNode("//user[@username='" +
username + "']");
            if(ele == null) {
                return null;
            }
            User user = new User();
            user.setUsername(ele.attributeValue("username"));
            user.setPassword(ele.attributeValue("password"));
            return user;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

MySQL

数据库

1 数据库概念（了解）

1.1 什么是数据库

数据库就是用来**存储和管理**数据的仓库！
数据库存储数据的优先：

- 可存储大量数据；
- 方便检索；
- 保持数据的一致性、完整性；
- 安全，可共享；
- 通过组合分析，可产生新数据。

1.2 数据库的发展历程

- 没有数据库，使用磁盘文件存储数据；
- 层次结构模型数据库；
- 网状结构模型数据库；
- **关系结构模型数据库：使用二维表格来存储数据；**
- 关系-对象模型数据库；

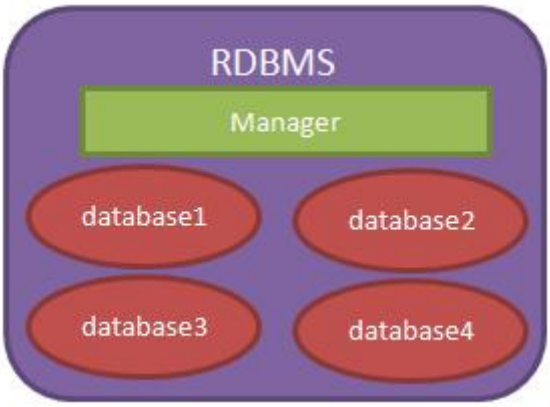
MySQL 就是关系型数据库！

1.3 常见数据库

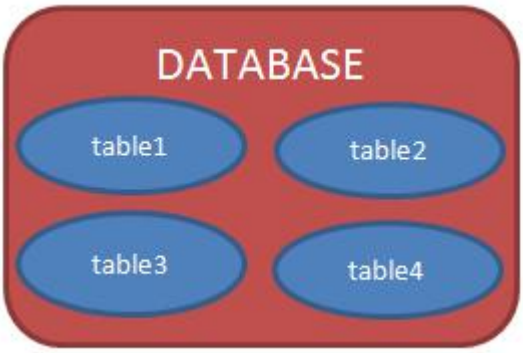
- Oracle：甲骨文；
- DB2：IBM；
- SQL Server：微软；
- Sybase：赛尔斯；
- MySQL：甲骨文；

1.4 理解数据库

我们现在所说的数据库泛指关“关系型数据库管理系统（RDBMS - Relational database management system）”，即“数据库服务器”。



当我们安装了数据库服务器后，就可以在数据库服务器中创建数据库，每个数据库中还可以包含多张表。



数据库表就是一个多行多列的表格。在创建表时，需要指定表的列数，以及列名称，列类型等信息。而不用指定表格的行数，行数是没有上限的。下面是 tab_student 表的结构：

s_id	varchar(10)
s_name	varchar(20)
s_age	int
s_sex	varchar(10)

当把表格创建好了之后，就可以向表格中添加数据了。向表格添加数据是以行为单位的！下面是 s_student 表的记录：

s_id	s_name	s_age	s_sex
S_1001	zhangSan	23	male
S_1002	liSi	32	female
S_1003	wangWu	44	male

大家要学会区分什么是表结构，什么是表记录。

1.5 应用程序与数据库

应用程序使用数据库完成对数据的存储！

2 安装 MySQL 数据库

2.1 安装 MySQL

参考：MySQL 安装图解.doc

2.2 MySQL 目录结构

MySQL 的数据存储目录为 data，data 目录通常在 C:\Documents and Settings\All Users\Application Data\MySQL\MySQL Server 5.1\data 位置。在 data 下的每个目录都代表一个数据库。

MySQL 的安装目录下：

- bin 目录中都是可执行文件；
- my.ini 文件是 MySQL 的配置文件；

3 基本命令

3.1 启动和关闭 mysql 服务器

- 启动：net start mysql；
- 关闭：net stop mysql；

在启动 mysql 服务后，打开 windows 任务管理器，会有一个名为 mysqld.exe 的进程运行，所以 mysqld.exe 才是 MySQL 服务器程序。

3.2 客户端登录退出 mysql

在启动 MySQL 服务器后，我们需要使用管理员用户登录 MySQL 服务器，然后来对服务器进行操作。登录 MySQL 需要使用 MySQL 的客户端程序：mysql.exe

- 登录：mysql -u root -p 123 -h localhost；

  - -u：后面的 root 是用户名，这里使用的是超级管理员 root；
  - -p：后面的 123 是密码，这是在安装 MySQL 时就已经指定的密码；
  - -h：后面给出的 localhost 是服务器主机名，它是可以省略的，例如：mysql -u root -p 123；
- 退出：quit 或 exit；

在登录成功后，打开 windows 任务管理器，会有一个名为 mysql.exe 的进程运行，所以 mysql.exe 是客户端程序。

SQL 语句

1 SQL 概述

1.1 什么是 SQL

SQL（Structured Query Language）是“结构化查询语言”，它是对关系型数据库的操作语言。它可以应用到所有关系型数据库中，例如：MySQL、Oracle、SQL Server 等。SQ 标准（ANSI/ISO）有：

- SQL-92：1992 年发布的 SQL 语言标准；
- SQL:1999：1999 年发布的 SQL 语言标签；
- SQL:2003：2003 年发布的 SQL 语言标签；

这些标准就与 JDK 的版本一样，在新的版本中总要有一些语法的变化。不同时期的数据库对不同标准做了实现。

虽然 SQL 可以用在所有关系型数据库中，但很多数据库还都有标准之后的一些语法，我们可以称之为“方言”。例如 MySQL 中的 LIMIT 语句就是 MySQL 独有的方言，其它数据库都不支持！当然，Oracle 或 SQL Server 都有自己的方言。

1.2 语法要求

- SQL 语句可以单行或多行书写，以分号结尾；
- 可以用空格和缩进来来增强语句的可读性；
- 关键字不区别大小写，建议使用大写；

2 分类

- DDL（Data Definition Language）：数据定义语言，用来定义数据库对象：库、表、列等；
- DML（Data Manipulation Language）：数据操作语言，用来定义数据库记录（数据）；
- DCL（Data Control Language）：数据控制语言，用来定义访问权限和安全级别；
- DQL（Data Query Language）：数据查询语言，用来查询记录（数据）。

3 DDL

3.1 基本操作

- 查看所有数据库名称：SHOW DATABASES;
- 切换数据库：USE mydb1，切换到 mydb1 数据库；

3.2 操作数据库

- 创建数据库：CREATE DATABASE [IF NOT EXISTS] mydb1；

创建数据库，例如：CREATE DATABASE mydb1，创建一个名为 mydb1 的数据库。如果这个数据已经存在，那么会报错。例如 CREATE DATABASE IF NOT EXISTS mydb1，在名为 mydb1 的数据库不存在时创建该库，这样可以避免报错。

- 删除数据库：DROP DATABASE [IF EXISTS] mydb1；

删除数据库，例如：DROP DATABASE mydb1，删除名为 mydb1 的数据库。如果这个数据库不存在，那么会报错。DROP DATABASE IF EXISTS mydb1，就算 mydb1 不存在，也不会的报错。

- 修改数据库编码：ALTER DATABASE mydb1 CHARACTER SET utf8

修改数据库 mydb1 的编码为 utf8。注意，在 MySQL 中所有的 UTF-8 编码都不能使用中间的“-”，即 UTF-8 要书写为 UTF8。

3.3 数据类型

MySQL 与 Java 一样，也有数据类型。MySQL 中数据类型主要应用在列上。

常用类型：

- int：整型
- double：浮点型，例如 double(5,2)表示最多 5 位，其中必须有 2 位小数，即最大值为 999.99；
- decimal：浮点型，在表单钱方面使用该类型，因为不会出现精度缺失问题；
- char：固定长度字符串类型；
- varchar：可变长度字符串类型；
- blob：字节类型；
- date：日期类型，格式为：yyyy-MM-dd；
- timestamp：时间戳类型；

3.4 操作表

- 创建表：

```
CREATE TABLE 表名(
    列名 列类型,
    列名 列类型,
    ......
);
```

例如：

```
CREATE TABLE stu(
    sid         CHAR(6),
    sname       VARCHAR(20),
    age         INT,
    gender      VARCHAR(10)
);
```

再例如：

```
CREATE TABLE emp(
    eid         CHAR(6),
    ename       VARCHAR(50),
    age         INT,
    gender      VARCHAR(6),
    birthday    DATE,
    hiredate    DATE,
    salary      DECIMAL(7,2),
    resume      VARCHAR(1000)
);
```

- 查看当前数据库中所有表名称：SHOW TABLES；
- 查看指定表的创建语句：SHOW CREATE TABLE emp，查看 emp 表的创建语句；
- 查看表结构：DESC emp，查看 emp 表结构；
- 删除表：DROP TABLE emp，删除 emp 表；
- 修改表：

1. 修改之添加列：给 stu 表添加 classname 列：
   ALTER TABLE stu ADD (classname varchar(100));
2. 修改之修改列类型：修改 stu 表的 gender 列类型为 CHAR(2)：
   ALTER TABLE stu MODIFY gender CHAR(2);
3. 修改之修改列名：修改 stu 表的 gender 列名为 sex：
   ALTER TABLE stu change gender sex CHAR(2);
4. 修改之删除列：删除 stu 表的 classname 列：
   ALTER TABLE stu DROP classname;

4 DML

4.1 插入数据

语法：

INSERT INTO 表名(列名 1,列名 2, ...) VALUES(值 1, 值 2)

```
INSERT INTO stu(sid, sname,age,gender) VALUES('s_1001', 'zhangSan', 23, 'male');
INSERT INTO stu(sid, sname) VALUES('s_1001', 'zhangSan');
```

语法：

INSERT INTO 表名 VALUES(值 1,值 2,…)

因为没有指定要插入的列，表示按创建表时列的顺序插入所有列的值：

```
INSERT INTO stu VALUES('s_1002', 'liSi', 32, 'female');
```

注意：所有字符串数据必须使用单引用！

4.2 修改数据

语法：

UPDATE 表名 SET 列名 1=值 1, … 列名 n=值 n [WHERE 条件]

```
UPDATE stu SET sname='zhangSanSan', age='32', gender='female' WHERE sid='s_1001';
UPDATE stu SET sname='liSi', age='20' WHERE age>50 AND gender='male';
UPDATE stu SET sname='wangWu', age='30' WHERE age>60 OR gender='female';
UPDATE stu SET gender='female' WHERE gender IS NULL
UPDATE stu SET age=age+1 WHERE sname='zhaoLiu';
```

4.3 删除数据

语法：

DELETE FROM 表名 [WHERE 条件]

```
DELETE FROM stu WHERE sid='s_1001';
DELETE FROM stu WHERE sname='chenQi' OR age > 30;
DELETE FROM stu;
```

语法：

TRUNCATE TABLE 表名

```
TRUNCATE TABLE stu;
```

虽然 TRUNCATE 和 DELETE 都可以删除表的所有记录，但有原理不同。DELETE 的效率没有 TRUNCATE 高！

TRUNCATE 其实属性 DDL 语句，因为它是先 DROP TABLE，再 CREATE TABLE。而且 TRUNCATE 删除的记录是无法回滚的，但 DELETE 删除的记录是可以回滚的（回滚是事务的知识！）。

5 DCL

5.1 创建用户

语法：

CREATE USER 用户名@地址 IDENTIFIED BY '密码';

```
CREATE USER user1@localhost IDENTIFIED BY '123';
CREATE USER user2@'%' IDENTIFIED BY '123';
```

5.2 给用户授权

语法：

GRANT 权限 1, … , 权限 n ON 数据库.* TO 用户名@IP

```
GRANT CREATE,ALTER,DROP,INSERT,UPDATE,DELETE,SELECT ON mydb1.* TO user1@localhost;
GRANT ALL ON mydb1.* TO user2@localhost;
```

5.3 撤销授权

语法：

REVOKE 权限 1, … , 权限 n ON 数据库.* FORM 用户名

```
REVOKE CREATE,ALTER,DROP ON mydb1.* FROM user1@localhost;
```

5.4 查看用户权限

语法：

SHOW GRANTS FOR 用户名

```
SHOW GRANTS FOR user1@localhost;
```

5.5 删除用户

语法：

DROP USER 用户名

```
DROP USER user1@localhost;
```

5.6 修改用户密码

语法：

UPDATE USER SET PASSWORD=PASSWORD(‘密码’) WHERE User=’用户名’;

FLUSH PRIVILEGES;

```
UPDATE USER SET PASSWORD=PASSWORD('1234') WHERE User='user2'
FLUSH PRIVILEGES;
```

数据查询语法（DQL）

DQL 就是数据查询语言，数据库执行 DQL 语句不会对数据进行改变，而是让数据库发送结果集给客户端。

语法：

SELECT selection_list /*要查询的列名称*/

```
FROM table_list /*要查询的表名称*/
WHERE condition /*行条件*/
GROUP BY grouping_columns /*对结果分组*/
HAVING condition /*分组后的行条件*/
ORDER BY sorting_columns /*对结果分组*/
LIMIT offset_start, row_count /*结果限定*/
```

创建名：

- 学生表：stu

字段名称	字段类型	说明
sid	char(6)	学生学号
sname	varchar(50)	学生姓名
age	int	学生年龄
gender	varchar(50)	学生性别

```
CREATE TABLE stu (
    sid   CHAR(6),
    sname        VARCHAR(50),
    age       INT,
    gender    VARCHAR(50)
);
```

```
INSERT INTO stu VALUES('S_1001', 'liuYi', 35, 'male');
INSERT INTO stu VALUES('S_1002', 'chenEr', 15, 'female');
INSERT INTO stu VALUES('S_1003', 'zhangSan', 95, 'male');
INSERT INTO stu VALUES('S_1004', 'liSi', 65, 'female');
INSERT INTO stu VALUES('S_1005', 'wangWu', 55, 'male');
INSERT INTO stu VALUES('S_1006', 'zhaoLiu', 75, 'female');
INSERT INTO stu VALUES('S_1007', 'sunQi', 25, 'male');
INSERT INTO stu VALUES('S_1008', 'zhouBa', 45, 'female');
INSERT INTO stu VALUES('S_1009', 'wuJiu', 85, 'male');
INSERT INTO stu VALUES('S_1010', 'zhengShi', 5, 'female');
INSERT INTO stu VALUES('S_1011', 'xxx', NULL, NULL);
```

- 雇员表：emp

字段名称	字段类型	说明
empno	int	员工编号
ename	varchar(50)	员工姓名
job	varchar(50)	员工工作
mgr	int	领导编号
hiredate	date	入职日期
sal	decimal(7,2)	月薪
comm	decimal(7,2)	奖金

deptno	int	部分编号

```
CREATE TABLE emp(
        empno       INT,
        ename       VARCHAR(50),
        job     VARCHAR(50),
        mgr     INT,
        hiredate    DATE,
        sal     DECIMAL(7,2),
        comm        decimal(7,2),
        deptno      INT
) ;
```

```
INSERT INTO emp values(7369,'SMITH','CLERK',7902,'1980-12-17',800,NULL,20);
INSERT INTO emp values(7499,'ALLEN','SALESMAN',7698,'1981-02-20',1600,300,30);
INSERT INTO emp values(7521,'WARD','SALESMAN',7698,'1981-02-22',1250,500,30);
INSERT INTO emp values(7566,'JONES','MANAGER',7839,'1981-04-02',2975,NULL,20);
INSERT INTO emp values(7654,'MARTIN','SALESMAN',7698,'1981-09-28',1250,1400,30);
INSERT INTO emp values(7698,'BLAKE','MANAGER',7839,'1981-05-01',2850,NULL,30);
INSERT INTO emp values(7782,'CLARK','MANAGER',7839,'1981-06-09',2450,NULL,10);
INSERT INTO emp values(7788,'SCOTT','ANALYST',7566,'1987-04-19',3000,NULL,20);
INSERT INTO emp values(7839,'KING','PRESIDENT',NULL,'1981-11-17',5000,NULL,10);
INSERT INTO emp values(7844,'TURNER','SALESMAN',7698,'1981-09-08',1500,0,30);
INSERT INTO emp values(7876,'ADAMS','CLERK',7788,'1987-05-23',1100,NULL,20);
INSERT INTO emp values(7900,'JAMES','CLERK',7698,'1981-12-03',950,NULL,30);
INSERT INTO emp values(7902,'FORD','ANALYST',7566,'1981-12-03',3000,NULL,20);
INSERT INTO emp values(7934,'MILLER','CLERK',7782,'1982-01-23',1300,NULL,10);
```

- 部门表：dept

字段名称	字段类型	说明
deptno	int	部分编码
dname	varchar(50)	部分名称
loc	varchar(50)	部分所在地点

```
CREATE TABLE dept(
        deptno      INT,
        dname       varchar(14),
        loc     varchar(13)
);
```

```
INSERT INTO dept values(10, 'ACCOUNTING', 'NEW YORK');
INSERT INTO dept values(20, 'RESEARCH', 'DALLAS');
INSERT INTO dept values(30, 'SALES', 'CHICAGO');
INSERT INTO dept values(40, 'OPERATIONS', 'BOSTON');
```

1 基础查询

1.1 查询所有列

```
SELECT * FROM stu;
```

1.2 查询指定列

```
SELECT sid, sname, age FROM stu;
```

2 条件查询

2.1 条件查询介绍

条件查询就是在查询时给出 WHERE 子句，在 WHERE 子句中可以使用如下运算符及关键字：

- =、!=、<>、<、<=、>、>=;
- BETWEEN…AND;
- IN(set);
- IS NULL;
- AND;
- OR;
- NOT;

2.2 查询性别为女，并且年龄小于 50 的记录

```
SELECT * FROM stu
WHERE gender='female' AND age<50;
```

2.3 查询学号为 S_1001，或者姓名为 liSi 的记录

```
SELECT * FROM stu
WHERE sid ='S_1001' OR sname='liSi';
```

2.4 查询学号为 S_1001，S_1002，S_1003 的记录

```
SELECT * FROM stu
WHERE sid IN ('S_1001','S_1002','S_1003');
```

2.5 查询学号不是 S_1001，S_1002，S_1003 的记录

```
SELECT * FROM tab_student
WHERE s_number NOT IN ('S_1001','S_1002','S_1003');
```

2.6 查询年龄为 null 的记录

```
SELECT * FROM stu
WHERE age IS NULL;
```

2.7 查询年龄在 20 到 40 之间的学生记录

```
SELECT *
FROM stu
WHERE age>=20 AND age<=40;
```

或者

```
SELECT *
FROM stu
WHERE age BETWEEN 20 AND 40;
```

2.8 查询性别非男的学生记录

```
SELECT *
FROM stu
WHERE gender!='male';
```

或者

```
SELECT *
FROM stu
WHERE gender<>'male';
```

或者

```
SELECT *
FROM stu
WHERE NOT gender='male';
```

2.9 查询姓名不为 null 的学生记录

```
SELECT *
FROM stu
WHERE NOT sname IS NULL;
```

或者

```
SELECT *
FROM stu
WHERE sname IS NOT NULL;
```

3 模糊查询

当想查询姓名中包含 a 字母的学生时就需要使用模糊查询了。模糊查询需要使用关键字 LIKE。

3.1 查询姓名由 5 个字母构成的学生记录

```
SELECT *
FROM stu
WHERE sname LIKE '_____';
```

模糊查询必须使用 LIKE 关键字。其中 “_”匹配任意一个字母，5 个“_”表示 5 个任意字母。

3.2 查询姓名由 5 个字母构成，并且第 5 个字母为“i”的学生记录

```
SELECT *
FROM stu
WHERE sname LIKE '____i';
```

3.3 查询姓名以“z”开头的学生记录

```
SELECT *
FROM stu
WHERE sname LIKE 'z%';
```

其中“%”匹配 0~n 个任何字母。

3.4 查询姓名中第 2 个字母为“i”的学生记录

```
SELECT *
FROM stu
WHERE sname LIKE '_i%';
```

3.5 查询姓名中包含“a”字母的学生记录

```
SELECT *
FROM stu
WHERE sname LIKE '%a%';
```

4 字段控制查询

4.1 去除重复记录

去除重复记录（两行或两行以上记录中系列的上的数据都相同），例如 emp 表中 sal 字段就存在

相同的记录。当只查询 emp 表的 sal 字段时，那么会出现重复记录，那么想去除重复记录，需要使用 DISTINCT：

```
SELECT DISTINCT sal FROM emp;
```

4.2 查看雇员的月薪与佣金之和

因为 sal 和 comm 两列的类型都是数值类型，所以可以做加运算。如果 sal 或 comm 中有一个字段不是数值类型，那么会出错。

```
SELECT *,sal+comm FROM emp;
```

comm 列有很多记录的值为 NULL，因为任何东西与 NULL 相加结果还是 NULL，所以结算结果可能会出现 NULL。下面使用了把 NULL 转换成数值 0 的函数 IFNULL：

```
SELECT *,sal+IFNULL(comm,0) FROM emp;
```

4.3 给列名添加别名

在上面查询中出现列名为 sal+IFNULL(comm,0)，这很不美观，现在我们给这一列给出一个别名，为 total：

```
SELECT *, sal+IFNULL(comm,0) AS total FROM emp;
```

给列起别名时，是可以省略 AS 关键字的：

```
SELECT *,sal+IFNULL(comm,0) total FROM emp;
```

5 排序

5.1 查询所有学生记录，按年龄升序排序

```
SELECT *
FROM stu
ORDER BY sage ASC;
```

或者

```
SELECT *
FROM stu
ORDER BY sage;
```

5.2 查询所有学生记录，按年龄降序排序

```
SELECT *
FROM stu
ORDER BY age DESC;
```

5.3 查询所有雇员，按月薪降序排序，如果月薪相同时，按编号升序排序

```
SELECT * FROM emp
ORDER BY sal DESC,empno ASC;
```

6 聚合函数

聚合函数是用来做纵向运算的函数：

- COUNT()：统计指定列不为 NULL 的记录行数；
- MAX()：计算指定列的最大值，如果指定列是字符串类型，那么使用字符串排序运算；
- MIN()：计算指定列的最小值，如果指定列是字符串类型，那么使用字符串排序运算；
- SUM()：计算指定列的数值和，如果指定列类型不是数值类型，那么计算结果为 0；
- AVG()：计算指定列的平均值，如果指定列类型不是数值类型，那么计算结果为 0；

6.1 COUNT

当需要纵向统计时可以使用 COUNT()。

- 查询 emp 表中记录数：

```
SELECT COUNT(*) AS cnt FROM emp;
```

- 查询 emp 表中有佣金的人数：

```
SELECT COUNT(comm) cnt FROM emp;
```

注意，因为 count()函数中给出的是 comm 列，那么只统计 comm 列非 NULL 的行数。

- 查询 emp 表中月薪大于 2500 的人数：

```
SELECT COUNT(*) FROM emp
WHERE sal > 2500;
```

- 统计月薪与佣金之和大于 2500 元的人数：

```
SELECT COUNT(*) AS cnt FROM emp WHERE sal+IFNULL(comm,0) > 2500;
```

- 查询有佣金的人数，以及有领导的人数：

```
SELECT COUNT(comm), COUNT(mgr) FROM emp;
```

6.2 SUM 和 AVG

当需要纵向求和时使用 sum()函数。

- 查询所有雇员月薪和：

```
SELECT SUM(sal) FROM emp;
```

- 查询所有雇员月薪和，以及所有雇员佣金和：

```
SELECT SUM(sal), SUM(comm) FROM emp;
```

- 查询所有雇员月薪+佣金和：

```
SELECT SUM(sal+IFNULL(comm,0)) FROM emp;
```

- 统计所有员工平均工资：

```
SELECT SUM(sal), COUNT(sal) FROM emp;
```

或者

```
SELECT AVG(sal) FROM emp;
```

6.3 MAX 和 MIN

- 查询最高工资和最低工资：

```
SELECT MAX(sal), MIN(sal) FROM emp;
```

7 分组查询

当需要分组查询时需要使用 GROUP BY 子句，例如查询每个部门的工资和，这说明要使用部分来分组。

7.1 分组查询

- 查询每个部门的部门编号和每个部门的工资和：

```
SELECT deptno, SUM(sal)
FROM emp
GROUP BY deptno;
```

- 查询每个部门的部门编号以及每个部门的人数：

```
SELECT deptno,COUNT(*)
FROM emp
GROUP BY deptno;
```

- 查询每个部门的部门编号以及每个部门工资大于 1500 的人数：

```
SELECT deptno,COUNT(*)
FROM emp
WHERE sal>1500
GROUP BY deptno;
```

7.2 HAVING 子句

- 查询工资总和大于 9000 的部门编号以及工资和：

```
SELECT deptno, SUM(sal)
FROM emp
```

```
GROUP BY deptno
HAVING SUM(sal) > 9000;
```

注意，WHERE 是对分组前记录的条件，如果某行记录没有满足 WHERE 子句的条件，那么这行记录不会参加分组；而 HAVING 是对分组后数据的约束。

8 LIMIT（MySQL 方言）

LIMIT 用来限定查询结果的起始行，以及总行数。

8.1 查询 5 行记录，起始行从 0 开始

```
SELECT * FROM emp LIMIT 0, 5;
```

注意，起始行从 0 开始，即第一行开始！

8.2 查询 10 行记录，起始行从 3 开始

```
SELECT * FROM emp LIMIT 3, 10;
```

8.3 分页查询

如果一页记录为 10 条，希望查看第 3 页记录应该怎么查呢？

- 第一页记录起始行为 0，一共查询 10 行；
- 第二页记录起始行为 10，一共查询 10 行；
- 第三页记录起始行为 20，一共查询 10 行；

完整性约束

完整性约束是为了表的数据的正确性！如果数据不正确，那么一开始就不能添加到表中。

1 主键（唯一，非空）

当某一列添加了主键约束后，那么这一列的数据就不能重复出现。这样每行记录中其主键列的值就是这一行的唯一标识。例如学生的学号可以用来做唯一标识，而学生的姓名是不能做唯一标识的，因为学习有可能同名。

主键列的值不能为 NULL，也不能重复！

指定主键约束使用 PRIMARY KEY 关键字

- 创建表：定义列时指定主键：

```
CREATE TABLE stu(
    sid     CHAR(6) PRIMARY KEY,
    sname   VARCHAR(20),
    age     INT,
    gender  VARCHAR(10)
);
```

- 创建表：定义列之后独立指定主键：

```
CREATE TABLE stu(
    sid     CHAR(6),
    sname   VARCHAR(20),
    age     INT,
    gender  VARCHAR(10),
    PRIMARY KEY(sid)
);
```

- 修改表时指定主键：

```
ALTER TABLE stu
ADD PRIMARY KEY(sid);
```

- 删除主键（只是删除主键约束，而不会删除主键列）：

```
ALTER TABLE stu DROP PRIMARY KEY;
```

2 主键自增长

MySQL 提供了主键自动增长的功能！这样用户就不用再为是否有主键是否重复而烦恼了。当主键设置为自动增长后，在没有给出主键值时，主键的值会自动生成，而且是最大主键值+1，也就不会出现重复主键的可能了。

- 创建表时设置主键自增长（主键必须是整型才可以自增长）：

```
CREATE TABLE stu(
    sid INT PRIMARY KEY AUTO_INCREMENT,
    sname   VARCHAR(20),
    age     INT,
    gender  VARCHAR(10)
);
```

- 修改表时设置主键自增长：

```
ALTER TABLE stu CHANGE sid sid INT AUTO_INCREMENT;
```

- 修改表时删除主键自增长：

```
ALTER TABLE stu CHANGE sid sid INT;
```

3 非空

指定非空约束的列不能没有值，也就是说在插入记录时，对添加了非空约束的列一定要给值；在修改记录时，不能把非空列的值设置为 NULL。

- 指定非空约束：

```
CREATE TABLE stu(
    sid INT PRIMARY KEY AUTO_INCREMENT,
    sname VARCHAR(10) NOT NULL,
    age     INT,
    gender  VARCHAR(10)
);
```

当为 sname 字段指定为非空后，在向 stu 表中插入记录时，必须给 sname 字段指定值，否则会报错：

```
INSERT INTO stu(sid) VALUES(1);
```

插入的记录中 sname 没有指定值，所以会报错！

4 唯一

还可以为字段指定唯一约束！当为字段指定唯一约束后，那么字段的值必须是唯一的。这一点与主键相似！例如给 stu 表的 sname 字段指定唯一约束：

```
CREATE TABLE tab_ab(
    sid INT PRIMARY KEY AUTO_INCREMENT,
    sname VARCHAR(10) UNIQUE
);

INSERT INTO sname(sid, sname) VALUES(1001, 'zs');
INSERT INTO sname(sid, sname) VALUES(1002, 'zs');
```

当两次插入相同的名字时，MySQL 会报错！

5 外键

主外键是构成表与表关联的唯一途径！

外键是另一张表的主键！例如员工表与部门表之间就存在关联关系，其中员工表中的部门编号字段就是外键，是相对部门表的外键。

我们再来看 BBS 系统中：用户表（t_user）、分类表（t_section）、帖子表（t_topic）三者之间的关系。

例如在 t_section 表中 sid 为 1 的记录说明有一个分类叫 java，版主是 t_user 表中 uid 为 1 的用户，即 zs！

例如在 t_topic 表中 tid 为 2 的记录是名字为“Java 是咖啡”的帖子，它是 java 版块的帖子，它的作者是 ww。

外键就是用来约束这一列的值必须是另一张表的主键值！！！

- 创建 t_user 表，指定 uid 为主键列：

```
CREATE TABLE t_user(
    uid   INT PRIMARY KEY AUTO_INCREMENT,
    uname    VARCHAR(20) UNIQUE NOT NULL
);
```

- 创建 t_section 表，指定 sid 为主键列，u_id 为相对 t_user 表的 uid 列的外键：

```
CREATE TABLE t_section(
    sid   INT PRIMARY KEY AUTO_INCREMENT,
    sname    VARCHAR(30),
    u_id INT,
    FOREIGN KEY(u_id) REFERENCES t_user(uid)
);
```

6 表与表之间的关系

- 一对一：例如 t_person 表和 t_card 表，即人和身份证。这种情况需要找出主从关系，即谁是主表，谁是从表。人可以没有身份证，但身份证必须要有人才行，所以人是主表，而身份证是从表。设计从表可以有两种方案：
  - 在 t_card 表中添加外键列（相对 t_user 表），并且给外键添加唯一约束；

- 给 t_card 表的主键添加外键约束（相对 t_user 表），即 t_card 表的主键也是外键。

- 一对多（多对一）：最为常见的就是一对多！一对多和多对一，这是从哪个角度去看得出来的。t_user 和 t_section 的关系，从 t_user 来看就是一对多，而从 t_section 的角度来看就是多对一！这种情况都是在多方创建外键！
- 多对多：例如 t_stu 和 t_teacher 表，即一个学生可以有多个老师，而一个老师也可以有多个学生。这种情况通常需要创建中间表来处理多对多关系。例如再创建一张表 t_stu_tea 表，给出两个外键，一个相对 t_stu 表的外键，另一个相对 t_teacher 表的外键。

编码

1 查看 MySQL 编码

SHOW VARIABLES LIKE 'char%';

```
mysql> show variables like 'char%';
+--------------------------+-------------
| Variable_name            | Value
+--------------------------+-------------
| character_set_client     | utf8
| character_set_connection | utf8
| character_set_database   | utf8
| character_set_filesystem | binary
| character_set_results    | utf8
| character_set_server     | utf8
| character_set_system     | utf8
| character_sets_dir       | D:\Program
+--------------------------+-------------
```

因为当初安装时指定了字符集为 UTF8，所以所有的编码都是 UTF8。

- character_set_client：你发送的数据必须与 client 指定的编码一致!!!服务器会使用该编码来解读客户端发送过来的数据；
- character_set_connection：通常该编码与 client 一致！该编码不会导致乱码！当执行的是查询语句时，客户端发送过来的数据会先转换成 connection 指定的编码。但只要客户端发送过来的数据与 client 指定的编码一致，那么转换就不会出现问题；
- character_set_database：数据库默认编码，在创建数据库时，如果没有指定编码，那么默认使用 database 编码；
- character_set_server：MySQL 服务器默认编码；
- character_set_result：响应的编码，即查询结果返回给客户端的编码。这说明客户端必须使用 result 指定的编码来解码；

2 控制台编码

修改 character_set_client、character_set_result、character_set_connection 为 GBK，就不会出现乱码了。但其实只需要修改 character_set_client 和 character_set_result。

控制台的编码只能是 GBK，而不能修改为 UTF8，这就出现一个问题。客户端发送的数据是 GBK，而 character_set_client 为 UTF8，这就说明客户端数据到了服务器端后一定会出现乱码。既然不能修改控制台的编码，那么只能修改 character_set_client 为 GBK 了。

服务器发送给客户端的数据编码为 character_set_result，它如果是 UTF8，那么控制台使用 GBK 解码也一定会出现乱码。因为无法修改控制台编码，所以只能把 character_set_result 修改为 GBK。

- 修改 character_set_client 变量：**set character_set_client=gbk;**
- 修改 character_set_results 变量：**set character_set_results=gbk;**

设置编码只对当前连接有效，这说明每次登录 MySQL 提示符后都要去修改这两个编码，但可以通过修改配置文件来处理这一问题：配置文件路径：D:\Program Files\MySQL\MySQL Server 5.1\ my.ini

```
# CLIENT SECTION
# ----------                认清楚这个，不要修改错了
#
# The following options will k
# Note that only client applic
# to read this section. If you
# honor these values, you need
# MySQL client library initial
#
[client]

port=3306

[mysql]          这个修改是一劳永逸的

default-character-set=gbk
```

3 MySQL 工具

使用 MySQL 工具是不会出现乱码的，因为它们会每次连接时都修改 character_set_client、character_set_results、character_set_connection 的编码。这样对 my.ini 上的配置覆盖了，也就不会出现乱码了。

多表查询

1 什么是多表查询

今天我们学习数据库的目的是为了学习 JDBC！所以对多表查询只是简单了解一下！

多表查询就是一次查询涉及到多张表！多表查询分为连接查询和子查询，我们这里简单介绍一下连接查询！

多表查询会产生笛卡尔积，假设集合 A={a,b}，集合 B={0,1,2}，则两个集合的笛卡尔积为

{(a,0),(a,1),(a,2),(b,0),(b,1),(b,2)}。可以扩展到多个集合的情况。

那么多表查询产生这样的结果并不是我们想要的，那么怎么去除重复的，不想让的记录呢，当然是通过条件过滤。通常要查询的多个表之间都存在关联关系，那么就通过关联关系去除笛卡尔积。

emp

empno	ename	deptno
1	zs	10
2	ls	20
3	ww	10

dept

deptno	dname
10	sales
20	research

查询 emp 和 dept 两张表的结果为：

empno	ename	deptno	deptno	dname
1	**zs**	**10**	**10**	**sales**
1	zs	10	20	research
2	ls	20	10	sales
2	**ls**	**20**	**20**	**research**
3	**ww**	**10**	**10**	**sales**
3	ww	10	20	research

我们可以通过关联关系做为条件，把多余的记录去除！这两张表的关系就是 deptno，所以我们可以给出如下条件：emp.deptno=dept.deptno！

1 内连接

简化语法：

- SELECT * FROM emp, dept；
- SELECT * FROM emp e, dept d；
- SELECT * FROM emp e, dept d WHERE e.deptno=d.deptno；

正规语法（SQL99）：

- SELECT * FROM emp e INNER JOIN dept d ON e.deptno=d.deptno；
- SELECT * FROM emp e INNER JOIN dept d USING(deptno)：只有关联字段名称相同时才可以使用这种方式，而且这么方法关联字段只会出现一次，而上面的会出现两次（emp 和 dept 表都有 deptno 字段）。

2 左右外连接

先不要去管什么是外连接，我们先给 emp 和 dept 表添加字段：

emp

empno	ename	deptno
1	zs	10
2	ls	20
3	ww	10
4	zl	NULL

dept

deptno	dname
10	sales
20	research
30	operations

在 emp 表中添加了一条记录，它没有部分；在 dept 表中添加了一条记录，表示运营部，但没有人是运营部的员工。

现在使用内连接来查询：SELECT * FROM emp e INNER JOIN dept d USING(deptno)

deptno	empno	ename	dname
10	1	zs	sales
20	2	ls	sales
10	3	ww	sales

结果中不存在 zl 这个员工，也不存在 operations 这个部分!!!

现在我想让 emp 表中的记录都可以出现，无论是否满足条件（emp.deptno=dept.deptno）都出现，这时就使用外连接。

- 左外连接：SELECT * FROM emp e LEFT JOIN dept d ON e.deptno=d.deptno：因为 emp 是左表，所以左表中的记录无论是否满足条件都会出现。左表中不满足条件的行，右表部分使用 NULL 补空；

empno	ename	deptno	deptno	dname
1	zs	10	10	sales
2	ls	20	20	research
3	ww	10	10	sales
4	**zl**	**NULL**	**NULL**	**NULL**

- 右外连接：SELECT * FROM emp e RIGHT JOIN dept d ON e.deptno=d.deptno：因为 dept 是右表，所以右表中的记录无论是否满足了条件都会陋。右表中不满足条件的行，左表部分使用 NULL 补空。

empno	ename	deptno	deptno	dname
1	zs	10	10	sales
3	ww	10	10	sales
2	ls	20	20	research
NULL	**NULL**	**NULL**	**30**	**operations**

MySQL 数据库备份与还原

备份和恢复数据

1 生成 SQL 脚本

在控制台使用 mysqldump 命令可以用来生成指定数据库的脚本文本，但要注意，脚本文本中只包含数据库的内容，而不会存在创建数据库的语句！所以在恢复数据时，还需要自己手动创建一个数据库之后再去恢复数据。

mysqldump –u 用户名 –p 密码 数据库名>生成的脚本文件路径

```
C:\>mysqldump -uroot -p123 mydb1>C:\mydb1.sql

C:\>
```

现在可以在 C 盘下找到 mydb1.sql 文件了！

注意，mysqldump 命令是在 Windows 控制台下执行，无需登录 mysql!!!

2 执行 SQL 脚本

执行 SQL 脚本需要登录 mysql，然后进入指定数据库，才可以执行 SQL 脚本!!!

执行 SQL 脚本不只是用来恢复数据库，也可以在平时编写 SQL 脚本，然后使用执行 SQL 脚本来操作数据库！大家都知道，在黑屏下编写 SQL 语句时，就算发现了错误，可能也不能修改了。所以我建议大家使用脚本文件来编写 SQL 代码，然后执行之！

SOURCE C:\mydb1.sql

```
mysql> source c:\mydb1.sql
```

注意，在执行脚本时需要先行核查当前数据库中的表是否与脚本文件中的语句有冲突！例如在脚本文件中存在 create table a 的语句，而当前数据库中已经存在了 a 表，那么就会出错！

还可以通过下面的方式来执行脚本文件：

mysql -uroot -p123 mydb1<c:\mydb1.sql

mysql –u 用户名 –p 密码 数据库<要执行脚本文件路径

```
C:\>mysql -uroot -p123 mydb1<c:\mydb1.sql
```

这种方式无需登录 mysql!

day11

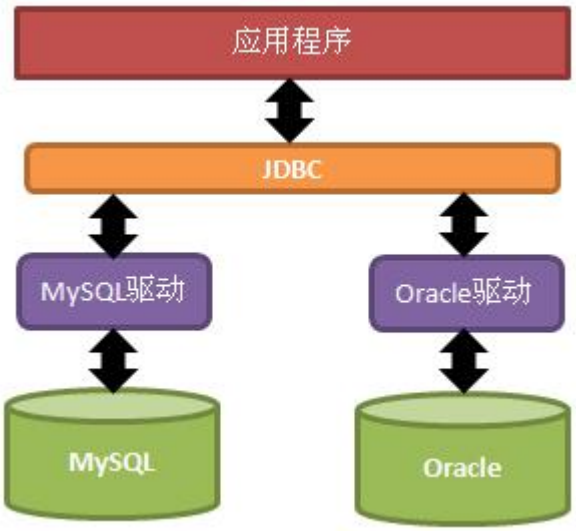
JDBC 入门

1 什么是 JDBC

JDBC（Java DataBase Connectivity）就是 Java 数据库连接，说白了就是用 Java 语言来操作数据库。原来我们操作数据库是在控制台使用 SQL 语句来操作数据库，JDBC 是用 Java 语言向数据库发送 SQL 语句。

2 JDBC 原理

早期 SUN 公司的天才们想编写一套可以连接天下所有数据库的 API，但是当他们刚刚开始时就发现这是不可完成的任务，因为各个厂商的数据库服务器差异太大了。后来 SUN 开始与数据库厂商们讨论，最终得出的结论是，由 SUN 提供一套访问数据库的规范（就是一组接口），并提供连接数据库的协议标准，然后各个数据库厂商会遵循 SUN 的规范提供一套访问自己公司的数据库服务器的 API 出现。SUN 提供的规范命名为 JDBC，而各个厂商提供的，遵循了 JDBC 规范的，可以访问自己数据库的 API 被称之为驱动！



JDBC 是接口，而 JDBC 驱动才是接口的实现，没有驱动无法完成数据库连接！每个数据库厂商都有自己的驱动，用来连接自己公司的数据库。

当然还有第三方公司专门为某一数据库提供驱动，这样的驱动往往不是开源免费的！

3 JDBC 核心类（接口）介绍

JDBC 中的核心类有：DriverManager、Connection、Statement，和 ResultSet！

DriverManger（驱动管理器）的作用有两个：

- 注册驱动：这可以让 JDBC 知道要使用的是哪个驱动；
- 获取 Connection：如果可以获取到 Connection，那么说明已经与数据库连接上了。

Connection 对象表示连接，与数据库的通讯都是通过这个对象展开的：

- Connection 最为重要的一个方法就是用来获取 Statement 对象；

Statement 是用来向数据库发送 SQL 语句的，这样数据库就会执行发送过来的 SQL 语句：

- void executeUpdate(String sql)：执行更新操作（insert、update、delete 等）；
- ResultSet executeQuery(String sql)：执行查询操作，数据库在执行查询后会把查询结果，查询结果就是 ResultSet；

ResultSet 对象表示查询结果集，只有在执行查询操作后才会有结果集的产生。结果集是一个二维的表格，有行有列。操作结果集要学习移动 ResultSet 内部的“行光标”，以及获取当前行上的每一列上的数据：

- boolean next()：使“行光标”移动到下一行，并返回移动后的行是否存在；
- XXX getXXX(int col)：获取当前行指定列上的值，参数就是列数，列数从 1 开始，而不是 0。

4 Hello JDBC

下面开始编写第一个 JDBC 程序

4.1 mysql 数据库的驱动 jar 包：mysql-connector-java-5.1.13-bin.jar；

4.2 获取连接

获取连接需要两步，一是使用 DriverManager 来注册驱动，二是使用 DriverManager 来获取 Connection 对象。

1. 注册驱动

看清楚了，注册驱动就只有一句话：Class.forName(“com.mysql.jdbc.Driver”)，下面的内容都是对这句代码的解释。今后我们的代码中，与注册驱动相关的代码只有这一句。

DriverManager 类的 registerDriver()方法的参数是 java.sql.Driver，但 java.sql.Driver 是一个接口，实现类由 mysql 驱动来提供，mysql 驱动中的 java.sql.Driver 接口的实现类为 com.mysql.jdbc.Driver！那么注册驱动的代码如下：

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

上面代码虽然可以注册驱动，但是出现硬编码（代码依赖 mysql 驱动 jar 包），如果将来想连接 Oracle 数据库，那么必须要修改代码的。并且其实这种注册驱动的方式是注册了两次驱动！

JDBC 中规定，驱动类在被加载时，需要自己“主动”把自己注册到 DriverManger 中，下面我们来看看 com.mysql.jdbc.Driver 类的源代码：

com.mysql.jdbc.Driver.java

```
public class Driver extends NonRegisteringDriver implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }
    ……
}
```

com.mysql.jdbc.Driver 类中的 static 块会创建本类对象，并注册到 DriverManager 中。这说明只要去加载 com.mysql.jdbc.Driver 类，那么就会执行这个 static 块，从而也就会把 com.mysql.jdbc.Driver 注册到 DriverManager 中，所以可以把**注册驱动类**的代码修改为**加载驱动类**。

Class.forName(“com.mysql.jdbc.Driver”);

2. 获取连接

获取连接的也只有一句代码：DriverManager.getConnection(url,username,password)，其中 username 和 password 是登录数据库的用户名和密码，如果我没说错的话，你的 mysql 数据库的用户名和密码分别是：root、123。

url 查对复杂一点，它是用来找到要连接数据库“网址”，就好比你要浏览器中查找百度时，也需要提供一个 url。下面是 mysql 的 url：

jdbc:mysql://localhost:3306/mydb1

JDBC 规定 url 的格式由三部分组成，每个部分中间使用逗号分隔。

- 第一部分是 jdbc，这是固定的；
- 第二部分是数据库名称，那么连接 mysql 数据库，第二部分当然是 mysql 了；
- 第三部分是由数据库厂商规定的，我们需要了解每个数据库厂商的要求，mysql 的第三部分分别由数据库服务器的 IP 地址（localhost）、端口号（3306），以及 DATABASE 名称(mydb1)组成。

下面是获取连接的语句：

Connection con = DriverManager.getConnection(“jdbc:mysql://localhost:3306/mydb1”,”root”,”123”);

还可以在 url 中提供参数：

jdbc:mysql://localhost:3306/mydb1**?useUnicode=true&characterEncoding=UTF8**

useUnicode 参数指定这个连接数据库的过程中，使用的字节集是 Unicode 字节集；

characherEncoding 参数指定穿上连接数据库的过程中，使用的字节集编码为 UTF-8 编码。请注意，mysql 中指定 UTF-8 编码是给出的是 UTF8，而不是 UTF-8。要小心了！

4.3 获取 Statement

在得到 Connectoin 之后，说明已经与数据库连接上了，下面是通过 Connection 获取 Statement 对象的代码：

Statement stmt = con.createStatement();

Statement 是用来向数据库发送要执行的 SQL 语句的!

4.4 发送 SQL 增、删、改语句

String sql = “insert into user value(’zhangSan’, ’123’)”;

int m = stmt.executeUpdate(sql);

其中 int 类型的返回值表示执行这条 SQL 语句所影响的行数，我们知道，对 insert 来说，最后只能影响一行，而 update 和 delete 可能会影响 0~n 行。

如果 SQL 语句执行失败，那么 executeUpdate()会抛出一个 SQLException。

4.5 发送 SQL 查询语句

String sql = “select * from user”;

ResultSet rs = stmt.executeQuery(sql);

请注册，执行查询使用的不是 executeUpdate()方法，而是 executeQuery()方法。executeQuery()方法返回的是 ResultSet，ResultSet 封装了查询结果，我们称之为结果集。

4.6 读取结果集中的数据

ResultSet 就是一张二维的表格，它内部有一个“行光标”，光标默认的位置在“第一行上方”，我们可以调用 rs 对象的 next()方法把“行光标”向下移动一行，当第一次调用 next()方法时，“行光标”就到了第一行记录的位置，这时就可以使用 ResultSet 提供的 getXXX(int col)方法来获取指定列的数据了：

rs.next();//光标移动到第一行

rs.getInt(1);//获取第一行第一列的数据

当你使用 rs.getInt(1)方法时，你必须可以肯定第 1 列的数据类型就是 int 类型，如果你不能肯定，那么最好使用 rs.getObject(1)。在 ResultSet 类中提供了一系列的 getXXX()方法，比较常用的方法有：

Object getObject(int col)

String getString(int col)

int getInt(int col)

double getDouble(int col)

4.7 关闭

与 IO 流一样，使用后的东西都需要关闭！关闭的顺序是先得到的后关闭，后得到的先关闭。

rs.close();

stmt.close();

con.close();

4.8 代码

```
public static Connection getConnection() throws Exception {
    Class.forName("com.mysql.jdbc.Driver");
    String url = "jdbc:mysql://localhost:3306/mydb1";
```

```
        return DriverManager.getConnection(url, "root", "123");
    }
```

```
    @Test
    public void insert() throws Exception {
        Connection con = getConnection();
        Statement stmt = con.createStatement();
        String sql = "insert into user values('zhangSan', '123')";
        stmt.executeUpdate(sql);
        System.out.println("插入成功! ");
    }
```

```
    @Test
    public void update() throws Exception {
        Connection con = getConnection();
        Statement stmt = con.createStatement();
        String sql = "update user set password='456' where username='zhangSan'";
        stmt.executeUpdate(sql);
        System.out.println("修改成功! ");
    }
```

```
    @Test
    public void delete() throws Exception {
        Connection con = getConnection();
        Statement stmt = con.createStatement();
        String sql = "delete from user where username='zhangSan'";
        stmt.executeUpdate(sql);
        System.out.println("删除成功! ");
    }
```

```
    @Test
    public void query() throws Exception {
        Connection con = getConnection();
        Statement stmt = con.createStatement();
        String sql = "select * from user";
        ResultSet rs = stmt.executeQuery(sql);
        while(rs.next()) {
            String username = rs.getString(1);
            String password = rs.getString(2);
            System.out.println(username + ", " + password);
        }
    }
```

4.9 规范化代码

所谓规范化代码就是无论是否出现异常，都要关闭 ResultSet、Statement，以及 Connection，如果你还记得 IO 流的规范化代码，那么下面的代码你就明白什么意思了。

```
@Test
public void query() {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        con = getConnection();
        stmt = con.createStatement();
        String sql = "select * from user";
        rs = stmt.executeQuery(sql);
        while(rs.next()) {
            String username = rs.getString(1);
            String password = rs.getString(2);
            System.out.println(username + ", " + password);
        }
    } catch(Exception e) {
        throw new RuntimeException(e);
    } finally {
        try {
            if(rs != null) rs.close();
            if(stmt != null) stmt.close();
            if(con != null) con.close();
        } catch(SQLException e) {}
    }
}
```

JDBC 对象介绍

1 JDBC 中的主要类（接口）

在 JDBC 中常用的类有：

- DriverManager；
- Connection；
- Statement；
- ResultSet。

2 DriverManager

其实我们今后只需要会用 DriverManager 的 getConnection()方法即可：

```
1. Class.forName("com.mysql.jdbc.Driver");//注册驱动
2. String url = "jdbc:mysql://localhost:3306/mydb1";
3. String username = "root";
4. String password = "123";
5. Connection con = DriverManager.getConnection(url, username, password);
```

注意，上面代码可能出现的两种异常：

1. ClassNotFoundException：这个异常是在第 1 句上出现的，出现这个异常有两个可能：
   - 你没有给出 mysql 的 jar 包；
   - 你把类名称打错了，查看类名是不是 com.mysql.jdbc.Driver。
2. SQLException：这个异常出现在第 5 句，出现这个异常就是三个参数的问题，往往 username 和 password 一般不是出错，所以需要认真查看 url 是否打错。

对于 DriverManager.registerDriver()方法了解即可，因为我们今后注册驱动只会 Class.forName()，而不会使用这个方法。

3 Connection

Connection 最为重要的方法就是获取 Statement：

- Statement stmt = con.createStatement();

后面在学习 ResultSet 方法时，还要学习一下下面的方法：

- Statement stmt = con.createStatement(int,int);

4 Statement

Statement 最为重要的方法是：

- int executeUpdate(String sql)：执行更新操作，即执行 insert、update、delete 语句，其实这个方法也可以执行 create table、alter table，以及 drop table 等语句，但我们很少会使用 JDBC 来执行这些语句；
- ResultSet executeQuery(String sql)：执行查询操作，执行查询操作会返回 ResultSet，即结果集。

boolean execute()

Statement 还有一个 boolean execute()方法，这个方法可以用来执行增、删、改、查所有 SQL 语句。该方法返回的是 boolean 类型，表示 SQL 语句是否执行成功。

如果使用 execute()方法执行的是更新语句，那么还要调用 int getUpdateCount()来获取 insert、update、delete 语句所影响的行数。

如果使用 execute()方法执行的是查询语句，那么还要调用 ResultSet getResultSet()来获取 select 语句的查询结果。

5 ResultSet 之滚动结果集（了解）

ResultSet 表示结果集，它是一个二维的表格！ResultSet 内部维护一个行光标（游标），ResultSet 提供了一系列的方法来移动游标：

- void beforeFirst()：把光标放到第一行的前面，这也是光标默认的位置；
- void afterLast()：把光标放到最后一行的后面；
- boolean first()：把光标放到第一行的位置上，返回值表示调控光标是否成功；
- boolean last()：把光标放到最后一行的位置上；
- boolean isBeforeFirst()：当前光标位置是否在第一行前面；
- boolean isAfterLast()：当前光标位置是否在最后一行的后面；
- boolean isFirst()：当前光标位置是否在第一行上；
- boolean isLast()：当前光标位置是否在最后一行上；
- boolean previous()：把光标向上挪一行；
- boolean next()：把光标向下挪一行；
- boolean relative(int row)：相对位移，当 row 为正数时，表示向下移动 row 行，为负数时表示向上移动 row 行；relative()相对位移
- boolean absolute(int row)：绝对位移，把光标移动到指定的行上；absolute(5)
- int getRow()：返回当前光标所有行。

获取结果集的行数：

```
1. rs.last();//移动到最后一行
2. int num = rs.getRow();//获取最后一行的行号，即共多少行。
3. rs.beforeFirst(); //回到默认位置。
```

上面方法分为两类，一类用来判断游标位置的，另一类是用来移动游标的。如果结果集是不可滚动的，那么只能使用 next()方法来移动游标，而 beforeFirst()、afterLast()、first()、last()、previous()、relative()方法都不能使用!!!

结果集是否支持滚动，要从 Connection 类的 createStatement()方法说起。也就是说创建的 Statement 决定了使用 Statement 创建的 ResultSet 是否支持滚动。

Statement createStatement(int resultSetType, int resultSetConcurrency)

resultSetType 的可选值：

- ResultSet.TYPE_FORWARD_ONLY：不滚动结果集；
- ResultSet.TYPE_SCROLL_INSENSITIVE：滚动结果集，但结果集数据不会再跟随数据库而变化；
- ResultSet.TYPE_SCROLL_SENSITIVE：滚动结果集，但结果集数据不会再跟随数据库而变化；

可以看出，如果想使用滚动的结果集，我们应该选择 TYPE_SCROLL_INSENSITIVE！其实很少有数据库驱动会支持 TYPE_SCROLL_SENSITIVE 的特性!通常我们也不需要查询到的结果集再受到数据库变化的影响。

resultSetConcurrency 的可选值：

- CONCUR_READ_ONLY：结果集是只读的，不能通过修改结果集而反向影响数据库；
- CONCUR_UPDATABLE：结果集是可更新的，对结果集的更新可以反向影响数据库。

通常可更新结果集这一“高级特性”我们也是不需要的！

获取滚动结果集的代码如下：

```
Connection con = …
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, CONCUR_READ_ONLY);
String sql = …//查询语句
ResultSet rs = stmt.executeQuery(sql);//这个结果集是可滚动的
```

6 ResultSet 之获取列数据

可以通过 next()方法使 ResultSet 的游标向下移动，当游标移动到你需要的行时，就需要来获取该行的数据了，ResultSet 提供了一系列的获取列数据的方法：

- String getString(int columnIndex)：获取指定列的 String 类型数据；
- int getInt(int columnIndex)：获取指定列的 int 类型数据；
- double getDouble(int columnIndex)：获取指定列的 double 类型数据；
- boolean getBoolean(int columnIndex)：获取指定列的 boolean 类型数据；
- Object getObject(int columnIndex)：获取指定列的 Object 类型的数据。

上面方法中，参数 columnIndex 表示列的索引，列索引从 1 开始，而不是 0，这第一点与数组不同。如果你清楚当前列的数据类型，那么可以使用 getInt()之类的方法来获取，如果你不清楚列的类型，那么你应该使用 getObject()方法来获取。

ResultSet 还提供了一套通过列名称来获取列数据的方法：

- String getString(String columnName)：获取名称为 columnName 的列的 String 数据；
- int getInt(String columnName)：获取名称为 columnName 的列的 int 数据；
- double getDouble(String columnName)：获取名称为 columnName 的列的 double 数据；
- boolean getBoolean(String columnName)：获取名称为 columnName 的列的 boolean 数据；
- Object getObject(String columnName)：获取名称为 columnName 的列的 Object 数据；

PreparedStatement

1 什么是 SQL 攻击

在需要用户输入的地方，用户输入的是 SQL 语句的片段，最终用户输入的 SQL 片段与我们 DAO 中写的 SQL 语句合成一个完整的 SQL 语句！例如用户在登录时输入的用户名和密码都是为 SQL 语句的片段！

2 演示 SQL 攻击

首先我们需要创建一张用户表，用来存储用户的信息。

```
CREATE TABLE user(
```

```
    uid   CHAR(32) PRIMARY KEY,
    username      VARCHAR(30) UNIQUE KEY NOT NULL,
    PASSWORD      VARCHAR(30)
);

INSERT INTO user VALUES('U_1001', 'zs', 'zs');
SELECT * FROM user;
```

现在用户表中只有一行记录，就是 zs。

下面我们写一个 login()方法！

```
public void login(String username, String password) {
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        con = JdbcUtils.getConnection();
        stmt = con.createStatement();
        String sql = "SELECT * FROM user WHERE " +
                "username='" + username +
                "' and password='" + password + "'";
        rs = stmt.executeQuery(sql);
        if(rs.next()) {
            System.out.println("欢迎" + rs.getString("username"));
        } else {
            System.out.println("用户名或密码错误！");
        }
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        JdbcUtils.close(con, stmt, rs);
    }
}
```

下面是调用这个方法的代码：

```
login("a' or 'a'='a", "a' or 'a'='a");
```

这行当前会使我们登录成功！因为是输入的用户名和密码是 SQL 语句片段，最终与我们的 login()方法中的 SQL 语句组合在一起！我们来看看组合在一起的 SQL 语句：

```
SELECT * FROM tab_user WHERE username='a' or 'a'='a' and password='a' or 'a'='a'
```

3 防止 SQL 攻击

- 过滤用户输入的数据中是否包含非法字符；
- 分步校验！先使用用户名来查询用户，如果查找到了，再比较密码；

- 使用 PreparedStatement。

4 PreparedStatement 是什么？

PreparedStatement 叫预编译声明！

PreparedStatement 是 Statement 的子接口，你可以使用 PreparedStatement 来替换 Statement。

PreparedStatement 的好处：

- 防止 SQL 攻击；
- 提高代码的可读性，以可维护性；
- 提高效率。

5 PreparedStatement 的使用

```
String sql = “select * from tab_student where s_number=?”;
PreparedStatement pstmt = con.prepareStatement(sql);
pstmt.setString(1, “S_1001”);
ResultSet rs = pstmt.executeQuery();
rs.close();
pstmt.clearParameters();
pstmt.setString(1, “S_1002”);
rs = pstmt.executeQuery();
```

在使用 Connection 创建 PreparedStatement 对象时需要给出一个 SQL 模板，所谓 SQL 模板就是有“?”的 SQL 语句，其中“?”就是参数。

在得到 PreparedStatement 对象后，调用它的 setXXX()方法为“?”赋值，这样就可以得到把模板变成一条完整的 SQL 语句，然后再调用 PreparedStatement 对象的 executeQuery()方法获取 ResultSet 对象。

注意 PreparedStatement 对象独有的 executeQuery()方法是没有参数的，而 Statement 的 executeQuery()是需要参数（SQL 语句）的。因为在创建 PreparedStatement 对象时已经让它与一条 SQL 模板绑定在一起了，所以在调用它的 executeQuery()和 executeUpdate()方法时就不再需要参数了。

PreparedStatement 最大的好处就是在于重复使用同一模板，给予其不同的参数来重复的使用它。这才是真正提高效率的原因。

所以，建议大家在今后的开发中，无论什么情况，都去需要 PreparedStatement，而不是使用 Statement。

JdbcUtils 工具类

1 JdbcUtils 的作用

你也看到了，连接数据库的四大参数是：驱动类、url、用户名，以及密码。这些参数都与特定数据库关联，如果将来想更改数据库，那么就要去修改这四大参数，那么为了不去修改代码，我们写一个 JdbcUtils 类，让它从配置文件中读取配置参数，然后创建连接对象。

2 JdbcUtils 代码

JdbcUtils.java

```
public class JdbcUtils {
    private static final String dbconfig = "dbconfig.properties";
    private static Properties prop = new Properties();
    static {
        try {
            InputStream in =
Thread.currentThread().getContextClassLoader().getResourceAsStream(dbconfig
);
            prop.load(in);
            Class.forName(prop.getProperty("driverClassName"));
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
    }

    public static Connection getConnection() {
        try {
            return DriverManager.getConnection(prop.getProperty("url"),
                    prop.getProperty("username"),
prop.getProperty("password"));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

dbconfig.properties

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mydb1?useUnicode=true&characterEncoding=UTF8
username=root
password=123
```

UserDao

MVC – 模型、视图、控制。jsp + servlet + … (WEB 结构！ php、.net、javaweb)
Java 的软件三层：表述层（WEB 层 jsp + servlet）+ 逻辑层(service) + 数据访问层(dao)

(jsp + servlet) + service + dao

User 类 java bean < -- > tab_user 表

```
class User {
   private String uid;
   private String username;
   private String password
}
```

tab_user

uid	username	password

UserDao（封装对数据的访问） 接口（面向接口编程）
UserDaoImpl 接口的实现
UserDaoFactory Dao 工厂

1 DAO 模式

DAO（Data Access Object）模式就是写一个类，把访问数据库的代码封装起来。DAO 在数据库与业务逻辑（Service）之间。

- 实体域，即操作的对象，例如我们操作的表是 user 表，那么就需要先写一个 User 类；
- DAO 模式需要先提供一个 DAO 接口；
- 然后再提供一个 DAO 接口的实现类；
- 再编写一个 DAO 工厂，Service 通过工厂来获取 DAO 实现。

2 代码

User.java

```
public class User {
   private String uid;
```

```
    private String username;
    private String password;
    …
}
```

UserDao.java

```
public interface UserDao {
    public void add(User user);
    public void mod(User user);
    public void del(String uid);
    public User load(String uid);
    public List<User> findAll();
}
```

UserDaoImpl.java

```
public class UserDaoImpl implements UserDao {
    public void add(User user) {
        Connection con = null;
        PreparedStatement pstmt = null;
        try {
            con = JdbcUtils.getConnection();
            String sql = "insert into user value(?,?,?)";
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, user.getUid());
            pstmt.setString(2, user.getUsername());
            pstmt.setString(3, user.getPassword());
            pstmt.executeUpdate();
        } catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            } catch(SQLException e) {}
        }
    }

    public void mod(User user) {
        Connection con = null;
        PreparedStatement pstmt = null;
        try {
            con = JdbcUtils.getConnection();
            String sql = "update user set username=?, password=? where uid=?";
```

```
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, user.getUsername());
            pstmt.setString(2, user.getPassword());
            pstmt.setString(3, user.getUid());
            pstmt.executeUpdate();
        } catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            } catch(SQLException e) {}
        }
    }

    public void del(String uid) {
        Connection con = null;
        PreparedStatement pstmt = null;
        try {
            con = JdbcUtils.getConnection();
            String sql = "delete from user where uid=?";
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, uid);
            pstmt.executeUpdate();
        } catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            } catch(SQLException e) {}
        }
    }

    public User load(String uid) {
        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            con = JdbcUtils.getConnection();
            String sql = "select * from user where uid=?";
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, uid);
```

```
            rs = pstmt.executeQuery();
            if(rs.next()) {
                return new User(rs.getString(1), rs.getString(2),
rs.getString(3));
            }
            return null;
        } catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            } catch(SQLException e) {}
        }
    }

    public List<User> findAll() {
        Connection con = null;
        PreparedStatement pstmt = null;
        ResultSet rs = null;
        try {
            con = JdbcUtils.getConnection();
            String sql = "select * from user";
            pstmt = con.prepareStatement(sql);
            rs = pstmt.executeQuery();
            List<User> userList = new ArrayList<User>();
            while(rs.next()) {
                userList.add(new User(rs.getString(1), rs.getString(2),
rs.getString(3)));
            }
            return userList;
        } catch(Exception e) {
            throw new RuntimeException(e);
        } finally {
            try {
                if(pstmt != null) pstmt.close();
                if(con != null) con.close();
            } catch(SQLException e) {}
        }
    }
}
```

UserDaoFactory.java

```
public class UserDaoFactory {
    private static UserDao userDao;
    static {
        try {
            InputStream in = Thread.currentThread().getContextClassLoader()
                    .getResourceAsStream("dao.properties");
            Properties prop = new Properties();
            prop.load(in);
            String className = prop.getProperty("cn.itcast.jdbc.UserDao");
            Class clazz = Class.forName(className);
            userDao = (UserDao) clazz.newInstance();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static UserDao getUserDao() {
        return userDao;
    }
}
```

dao.properties

```
cn.itcast.jdbc.UserDao=cn.itcast.jdbc.UserDaoImpl
```

时间类型

1 Java 中的时间类型

java.sql 包下给出三个与数据库相关的日期时间类型，分别是：

- Date：表示日期，只有年月日，没有时分秒。会丢失时间；
- Time：表示时间，只有时分秒，没有年月日。会丢失日期；
- Timestamp：表示时间戳，有年月日时分秒，以及毫秒。

这三个类都是 java.util.Date 的子类。

2 时间类型相互转换

把数据库的三种时间类型赋给 java.util.Date，基本不用转换，因为这是把子类对象给父类的引用，不需要转换。

java.sql.Date date = …

java.util.Date d = date;

java.sql.Time time = …
java.util.Date d = time;

java.sql.Timestamp timestamp = …
java.util.Date d = timestamp;

当需要把 java.util.Date 转换成数据库的三种时间类型时，这就不能直接赋值了，这需要使用数据库三种时间类型的构造器。java.sql 包下的 Date、Time、TimeStamp 三个类的构造器都需要一个 long 类型的参数，表示毫秒值。创建这三个类型的对象，只需要有毫秒值即可。我们知道 java.util.Date 有 getTime()方法可以获取毫秒值，那么这个转换也就不是什么问题了。

java.utl.Date d = new java.util.Date();
java.sql.Date date = new java.sql.Date(d.getTime());//会丢失时分秒
Time time = new Time(d.getTime());//会丢失年月日
Timestamp timestamp = new Timestamp(d.getTime());

3 代码

我们来创建一个 dt 表：

```
CREATE TABLE dt(
  d DATE,
  t TIME,
  ts TIMESTAMP
)
```

下面是向 dt 表中插入数据的代码：

```
@Test
public void fun1() throws SQLException {
    Connection con = JdbcUtils.getConnection();
    String sql = "insert into dt value(?,?,?)";
    PreparedStatement pstmt = con.prepareStatement(sql);

    java.util.Date d = new java.util.Date();
    pstmt.setDate(1, new java.sql.Date(d.getTime()));
    pstmt.setTime(2, new Time(d.getTime()));
    pstmt.setTimestamp(3, new Timestamp(d.getTime()));
    pstmt.executeUpdate();
}
```

下面是从 dt 表中查询数据的代码：

```
@Test
```

```
    public void fun2() throws SQLException {
        Connection con = JdbcUtils.getConnection();
        String sql = "select * from dt";
        PreparedStatement pstmt = con.prepareStatement(sql);
        ResultSet rs = pstmt.executeQuery();

        rs.next();
        java.util.Date d1 = rs.getDate(1);
        java.util.Date d2 = rs.getTime(2);
        java.util.Date d3 = rs.getTimestamp(3);

        System.out.println(d1);
        System.out.println(d2);
        System.out.println(d3);
    }
```

大数据

1 什么是大数据

所谓大数据，就是大的字节数据，或大的字符数据。标准 SQL 中提供了如下类型来保存大数据类型：

类型	长度
tinyblob	2^{8}-1B（256B）
blob	2^{16}-1B（64KB）
mediumblob	2^{24}-1B（16MB）
longblob	2^{32}-1B（4GB）
tinyclob	2^{8}-1B（256B）
clob	2^{16}-1B（64K）
mediumclob	2^{24}-1B（16M）
longclob	2^{32}-1B（4G）

但是，在 mysql 中没有提供 tinyclob、clob、mediumclob、longclob 四种类型，而是使用如下四种类型来处理文本大数据：

类型	长度
tinytext	2^{8}-1B（256B）
text	2^{16}-1B（64K）
mediumtext	2^{24}-1B（16M）
longtext	2^{32}-1B（4G）

首先我们需要创建一张表，表中要有一个 mediumblob（16M）类型的字段。

```
CREATE TABLE tab_bin(
    id    INT  PRIMARY KEY AUTO_INCREMENT,
    filename VARCHAR(100),
    data      MEDIUMBLOB
);
```

向数据库插入二进制数据需要使用 PreparedStatement 为原 setBinaryStream(int, InputSteam)方法来完成。

```
con = JdbcUtils.getConnection();
String sql = "insert into tab_bin(filename,data) values(?, ?)";
pstmt = con.prepareStatement(sql);
pstmt.setString(1, "a.jpg");
InputStream in = new FileInputStream("f:\\a.jpg");
pstmt.setBinaryStream(2, in);
pstmt.executeUpdate();
```

读取二进制数据，需要在查询后使用 ResultSet 类的 getBinaryStream()方法来获取输入流对象。也就是说，PreparedStatement 有 setXXX()，那么 ResultSet 就有 getXXX()。

```
con = JdbcUtils.getConnection();
String sql = "select filename,data from tab_bin where id=?";
pstmt = con.prepareStatement(sql);
pstmt.setInt(1, 1);
rs = pstmt.executeQuery();
rs.next();

String filename = rs.getString("filename");
OutputStream out = new FileOutputStream("F:\\" + filename);

InputStream in = rs.getBinaryStream("data");
IOUtils.copy(in, out);
out.close();
```

还有一种方法，就是把要存储的数据包装成 Blob 类型，然后调用 PreparedStatement 的 setBlob()方法来设置数据

```
con = JdbcUtils.getConnection();
String sql = "insert into tab_bin(filename,data) values(?, ?)";
pstmt = con.prepareStatement(sql);
pstmt.setString(1, "a.jpg");
File file = new File("f:\\a.jpg");
byte[] datas = FileUtils.getBytes(file);//获取文件中的数据
Blob blob = new SerialBlob(datas);//创建Blob对象
pstmt.setBlob(2, blob);//设置Blob类型的参数
```

```
        pstmt.executeUpdate();
```

```
        con = JdbcUtils.getConnection();
        String sql = "select filename,data from tab_bin where id=?";
        pstmt = con.prepareStatement(sql);
        pstmt.setInt(1, 1);
        rs = pstmt.executeQuery();
        rs.next();

        String filename = rs.getString("filename");
        File file = new File("F:\\" + filename) ;
        Blob blob = rs.getBlob("data");
        byte[] datas = blob.getBytes(0, (int)file.length());
        FileUtils.writeByteArrayToFile(file, datas);
```

批处理

1 Statement 批处理

批处理就是一批一批的处理，而不是一个一个的处理！

当你有 10 条 SQL 语句要执行时，一次向服务器发送一条 SQL 语句，这么做效率上很差！处理的方案是使用批处理，即一次向服务器发送多条 SQL 语句，然后由服务器一次性处理。

批处理只针对更新（增、删、改）语句，批处理没有查询什么事儿！

可以多次调用 Statement 类的 addBatch(String sql)方法，把需要执行的所有 SQL 语句添加到一个“批”中，然后调用 Statement 类的 executeBatch()方法来执行当前“批”中的语句。

- void addBatch(String sql)：添加一条语句到“批”中；
- int[] executeBatch()：执行“批”中所有语句。返回值表示每条语句所影响的行数据；
- void clearBatch()：清空“批”中的所有语句。

```
            for(int i = 0; i < 10; i++) {
                String number = "S_10" + i;
                String name = "stu" + i;
                int age = 20 + i;
                String gender = i % 2 == 0 ? "male" : "female";
                String sql = "insert into stu values('" + number + "', '" + name
+ "', " + age + ", '" + gender + "')";
                stmt.addBatch(sql);
            }
            stmt.executeBatch();
```

当执行了“批”之后，“批”中的 SQL 语句就会被清空！也就是说，连续两次调用 executeBatch()相当于调用一次！因为第二次调用时，“批”中已经没有 SQL 语句了。

还可以在执行“批”之前，调用 Statement 的 clearBatch()方法来清空“批”!

2 PreparedStatement 批处理

PreparedStatement 的批处理有所不同，因为每个 PreparedStatement 对象都绑定一条 SQL 模板。所以向 PreparedStatement 中添加的不是 SQL 语句，而是给“?”赋值。

```
con = JdbcUtils.getConnection();
String sql = "insert into stu values(?,?,?,?)";
pstmt = con.prepareStatement(sql);
for(int i = 0; i < 10; i++) {
    pstmt.setString(1, "S_10" + i);
    pstmt.setString(2, "stu" + i);
    pstmt.setInt(3, 20 + i);
    pstmt.setString(4, i % 2 == 0 ? "male" : "female");
    pstmt.addBatch();
}
pstmt.executeBatch();
```

day12 总结

今日内容

- 事务
- 连接池

事务

事务概述

为了方便演示事务，我们需要创建一个 account 表：

```
CREATE TABLE account(
	id INT PRIMARY KEY AUTO_INCREMENT,
	NAME VARCHAR(30),
	balance NUMERIC(10.2)
);

INSERT INTO account(NAME,balance) VALUES('zs', 100000);
INSERT INTO account(NAME,balance) VALUES('ls', 100000);
INSERT INTO account(NAME,balance) VALUES('ww', 100000);

SELECT * FROM account;
```

1 什么是事务

银行转账！张三转 10000 块到李四的账户，这其实需要两条 SQL 语句：

- 给张三的账户减去 10000 元；
- 给李四的账户加上 10000 元。

如果在第一条 SQL 语句执行成功后，在执行第二条 SQL 语句之前，程序被中断了（可能是抛出了某个异常，也可能是其他什么原因），那么李四的账户没有加上 10000 元，而张三却减去了 10000 元。这肯定是不行的！

你现在可能已经知道什么是事务了吧！事务中的多个操作，要么完全成功，要么完全失败！不可能存在成功一半的情况！也就是说给张三的账户减去 10000 元如果成功了，那么给李四的账户加上 10000 元的操作也必须是成功的；否则给张三减去 10000 元，以及给李四加上 10000 元都是失败的！

2 事务的四大特性（ACID）

面试！

事务的四大特性是：

- 原子性（Atomicity）：事务中所有操作是不可再分割的原子单位。事务中所有操作要么全部执行成功，要么全部执行失败。
- 一致性（Consistency）：事务执行后，数据库状态与其它业务规则保持一致。如转账业务，无论事务执行成功与否，参与转账的两个账号余额之和应该是不变的。
- 隔离性（Isolation）：隔离性是指在并发操作中，不同事务之间应该隔离开来，使每个并发中的事务不会相互干扰。
- 持久性（Durability）：一旦事务提交成功，事务中所有的数据操作都必须被持久化到数据库中，即使提交事务后，数据库马上崩溃，在数据库重启时，也必须能保证通过某种机制恢复数据。

3 MySQL 中的事务

在默认情况下，MySQL 每执行一条 SQL 语句，都是一个单独的事务。如果需要在一个事务中包含多条 SQL 语句，那么需要开启事务和结束事务。

- 开启事务：**start transaction**；
- 结束事务：**commit** 或 **rollback**。

在执行 SQL 语句之前，先执行 strat transaction，这就开启了一个事务（事务的起点），然后可以去执行多条 SQL 语句，最后要结束事务，commit 表示提交，即事务中的多条 SQL 语句所做出的影响会持久化到数据库中。或者 rollback，表示回滚，即回滚到事务的起点，之前做的所有操作都被撤消了！

下面演示 zs 给 li 转账 10000 元的示例：

```
START TRANSACTION;
UPDATE account SET balance=balance-10000 WHERE id=1;
UPDATE account SET balance=balance+10000 WHERE id=2;
ROLLBACK;
```

```
START TRANSACTION;
UPDATE account SET balance=balance-10000 WHERE id=1;
UPDATE account SET balance=balance+10000 WHERE id=2;
COMMIT;
```

```
START TRANSACTION;
UPDATE account SET balance=balance-10000 WHERE id=1;
UPDATE account SET balance=balance+10000 WHERE id=2;
quit;
```

JDBC 事务

1 JDBC 中的事务

Connection 的三个方法与事务相关：

- setAutoCommit(boolean)：设置是否为自动提交事务，如果 true（默认值就是 true）表示自动提交，也就是每条执行的 SQL 语句都是一个单独的事务，如果设置 false，那么就相当于开启了事务了；
- commit()：提交结束事务；
- rollback()：回滚结束事务。

```
public void transfer(boolean b) {
    Connection con = null;
    PreparedStatement pstmt = null;

    try {
        con = JdbcUtils.getConnection();
        //手动提交
        con.setAutoCommit(false);

        String sql = "update account set balance=balance+? where id=?";
        pstmt = con.prepareStatement(sql);

        //操作
        pstmt.setDouble(1, -10000);
        pstmt.setInt(2, 1);
        pstmt.executeUpdate();

        // 在两个操作中抛出异常
        if(b) {
            throw new Exception();
        }

        pstmt.setDouble(1, 10000);
        pstmt.setInt(2, 2);
        pstmt.executeUpdate();

        //提交事务
        con.commit();
    } catch(Exception e) {
        //回滚事务
```

```
            if(con != null) {
                try {
                    con.rollback();
                } catch(SQLException ex) {}
            }
            throw new RuntimeException(e);
        } finally {
            //关闭
            JdbcUtils.close(con, pstmt);
        }
    }
```

2 保存点（了解）

保存点是 JDBC3.0 的东西！当要求数据库服务器支持保存点方式的回滚。

校验数据库服务器是否支持保存点！

```
boolean b = con.getMetaData().supportsSavepoints();
```

保存点的作用是允许事务回滚到指定的保存点位置。在事务中设置好保存点，然后回滚时可以选择回滚到指定的保存点，而不是回滚整个事务！**注意，回滚到指定保存点并没有结束事务!!! 只有回滚了整个事务才算是结束事务了！**

Connection 类的设置保存点，以及回滚到指定保存点方法：

- 设置保存点：Savepoint setSavepoint();
- 回滚到指定保存点：void rollback(Savepoint)。

```
    /*
     * 李四对张三说，如果你给我转1W，我就给你转100W。
     * ========================================
     *
     * 张三给李四转1W（张三减去1W，李四加上1W）
     * 设置保存点！
     * 李四给张三转100W（李四减去100W，张三加上100W）
     * 查看李四余额为负数，那么回滚到保存点。
     * 提交事务
     */
    @Test
    public void fun() {
        Connection con = null;
        PreparedStatement pstmt = null;

        try {
            con = JdbcUtils.getConnection();
            //手动提交
```

```
            con.setAutoCommit(false);

            String sql = "update account set balance=balance+? where name=?";
            pstmt = con.prepareStatement(sql);

            //操作1（张三减去1W）
            pstmt.setDouble(1, -10000);
            pstmt.setString(2, "zs");
            pstmt.executeUpdate();

            //操作2（李四加上1W）
            pstmt.setDouble(1, 10000);
            pstmt.setString(2, "ls");
            pstmt.executeUpdate();

            // 设置保存点
            Savepoint sp = con.setSavepoint();

            //操作3（李四减去100W）
            pstmt.setDouble(1, -1000000);
            pstmt.setString(2, "ls");
            pstmt.executeUpdate();

            //操作4（张三加上100W）
            pstmt.setDouble(1, 1000000);
            pstmt.setString(2, "zs");
            pstmt.executeUpdate();

            //操作5（查看李四余额）
            sql = "select balance from account where name=?";
            pstmt = con.prepareStatement(sql);
            pstmt.setString(1, "ls");
            ResultSet rs = pstmt.executeQuery();
            rs.next();
            double balance = rs.getDouble(1);
           //如果李四余额为负数，那么回滚到指定保存点
           if(balance < 0) {
               con.rollback(sp);
               System.out.println("张三，你上当了！");
           }

            //提交事务
            con.commit();
```

```
        } catch(Exception e) {
            //回滚事务
            if(con != null) {
                try {
                    con.rollback();
                } catch(SQLException ex) {}
            }
            throw new RuntimeException(e);
        } finally {
            //关闭
            JdbcUtils.close(con, pstmt);
        }
    }
```

事务隔离级别

1 事务的并发读问题

- 脏读：读取到另一个事务未提交数据；
- 不可重复读：两次读取不一致；
- 幻读（虚读）：读到另一事务已提交数据。

2 五大并发事务问题

因为并发事务导致的问题大致有 5 类，其中两类是更新问题，三类是读问题。

- 脏读（dirty read）：**读到未提交更新数据**

时间	转账事务 A	取款事务 B
T1		**开始事务**
T2	**开始事务**	
T3		查询账户余额为 1000 元
T4		取出 500 元把余额改为 500 元
T5	查看账户余额为 500 元（脏读）	
T6		撤销事务，余额恢复为 1000 元
T7	汇入 100 元把余额改为 600 元	
T8	**提交事务**	

A 事务查询到了 B 事务未提交的更新数据，A 事务依据这个查询结果继续执行相关操作。但是接着 B 事务撤销了所做的更新，这会导致 A 事务操作的是脏数据。（这是绝对不允许出现的事情）

- 虚读（phantom read）：**读到已提交插入数据**

时间	统计金额事务 A	转账事务 B
T1		**开始事务**
T2	**开始事务**	
T3	统计总存款数为 10000 元	
T4		新增一个存款账户，存款为 100 元
T5		**提交事务**
T6	再次统计总存款数为 10100 元	

A 事务第一次查询时，没有问题，第二次查询时查到了 B 事务已提交的新插入数据，这导致两次查询结果不同。（在实际开发中，很少会对相同数据进行两次查询，所以可以考虑是否允许虚读）

- 不可重复读（unrepeatable read）：**读到已提交更新数据**

时间	取款事务 A	转账事务 B
T1		**开始事务**
T2	**开始事务**	
T3		查询账户余额为 1000 元
T4	查询账户余额为 1000 元	
T5		取出 100 元，把余额改为 900 元
T6		**提交事务**
T7	查询账户余额为 900 元（与 T4 读取的一不一致）	

不可重复读与虚读有些相似，都是两次查询的结果不同。后者是查询到了另一个事务已提交的新插入数据，而前者是查询到了另一个事务已提交的更新数据。

3 四大隔离级别

隔离级别	脏读	不可重复读	虚读	第一类丢失更新	第二类丢失更新
READ UNCOMMITTED	允许	允许	允许	不允许	允许
READ COMMITTED	不允许	允许	允许	不允许	允许
REPEATABLE READ	不允许	不允许	允许	不允许	不允许
SERIALIZABLE	不允许	不允许	不允许	不允许	不允许

4 哪种隔离级别最好

4 个等级的事务隔离级别，在相同数据环境下，使用相同的输入，执行相同的工作，根据不同的隔离级别，可以导致不同的结果。不同事务隔离级别能够解决的数据并发问题的能力是不同的。

1 SERIALIZABLE（串行化）

当数据库系统使用 SERIALIZABLE 隔离级别时，一个事务在执行过程中完全看不到其他事务对数据库所做的更新。当两个事务同时操作数据库中相同数据时，如果第一个事务已经在访问该数据，第二个事务只能停下来等待，必须等到第一个事务结束后才能恢复运行。因此这两个事务实际上是串行化方式运行。

2 REPEATABLE READ（可重复读）

当数据库系统使用 REPEATABLE READ 隔离级别时，一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，但是不能看到其他事务对已有记录的更新。

3 READ COMMITTED（读已提交数据）

当数据库系统使用 READ COMMITTED 隔离级别时，一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，而且还能看到其他事务已经提交的对已有记录的更新。

4 READ UNCOMMITTED（读未提交数据）

当数据库系统使用 READ UNCOMMITTED 隔离级别时，一个事务在执行过程中可以看到其他事务没有提交的新插入的记录，而且还能看到其他事务没有提交的对已有记录的更新。

你可能会说，选择 SERIALIZABLE，因为它最安全！没错，它是最安全，但它也是最慢的！而且也最容易产生死锁。四种隔离级别的安全性与性能成反比！最安全的性能最差，最不安全的性能最好！

MySQL 的默认隔离级别为 REPEATABLE READ，这是一个很不错的选择吧！

5 MySQL 隔离级别

MySQL 的默认隔离级别为 Repeatable read，可以通过下面语句查看：

```
select @@session.tx_isolation
select @@global.tx_isolation
```

也可以通过下面语句来设置当前连接的隔离级别：

```
set global transaction isolationlevel [4 先 1]
set session transaction isolationlevel [4 先 1]
```

6 JDBC 设置隔离级别

con. setTransactionIsolation(int level)

参数可选值如下：

- Connection.TRANSACTION_READ_UNCOMMITTED；
- Connection.TRANSACTION_READ_COMMITTED；
- Connection.TRANSACTION_REPEATABLE_READ；
- Connection.TRANSACTION_SERIALIZABLE。

事务总结：

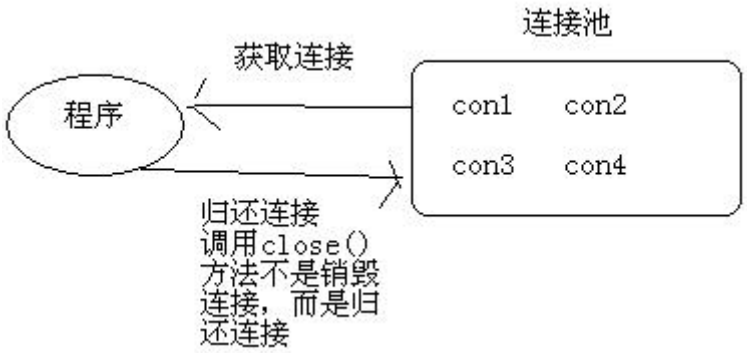
- 事务的特性：ACID；
- 事务开始边界与结束边界：开始边界（con.setAutoCommit(false)），结束边界（con.commit()或 con.rollback()）；
- 事务的隔离级别： READ_UNCOMMITTED 、 READ_COMMITTED 、 REPEATABLE_READ 、 SERIALIZABLE。多个事务并发执行时才需要考虑并发事务。

数据库连接池

数据库连接池

1 数据库连接池的概念

用池来管理 Connection，这可以重复使用 Connection。有了池，所以我们就不用自己来创建 Connection，而是通过池来获取 Connection 对象。当使用完 Connection 后，调用 Connection 的 close()方法也不会真的关闭 Connection，而是把 Connection“归还”给池。池就可以再利用这个 Connection 对象了。



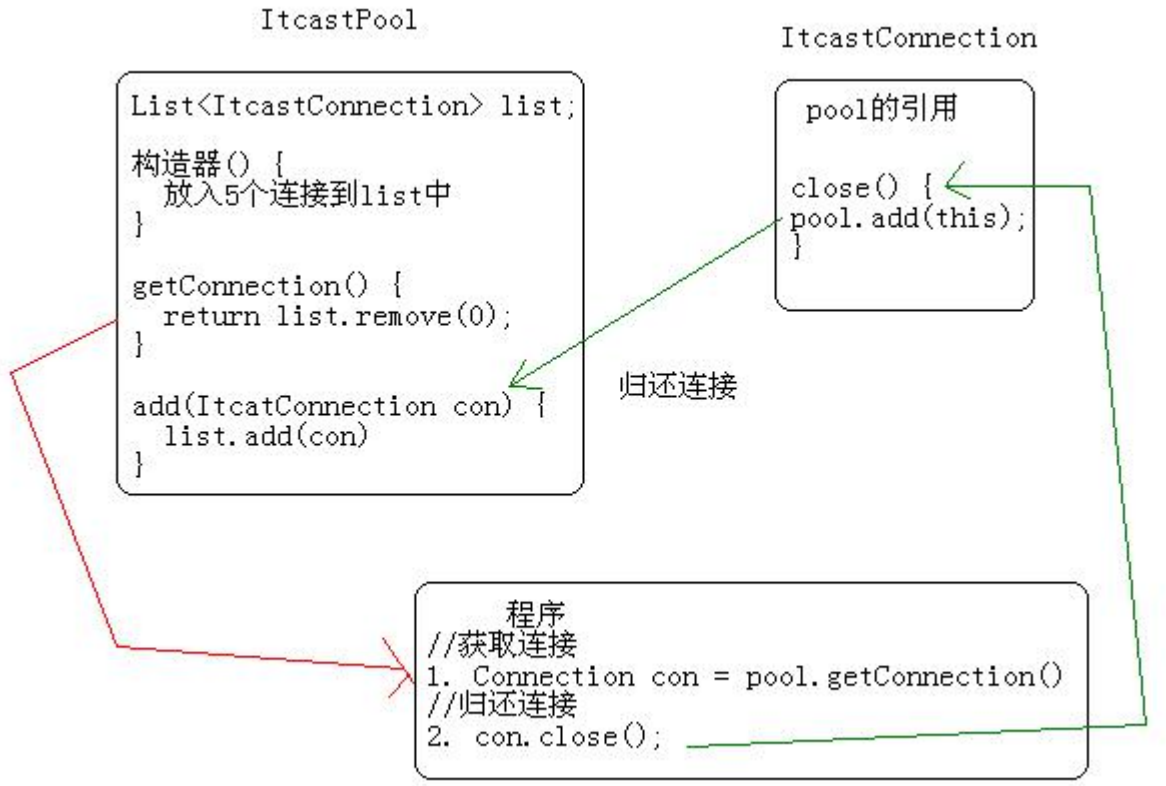
2 JDBC 数据库连接池接口（DataSource）

Java 为数据库连接池提供了公共的接口：javax.sql.DataSource，各个厂商可以让自己的连接池实现这个接口。这样应用程序可以方便的切换不同厂商的连接池！

3 自定义连接池（ItcastPool）

分析：ItcastPool 需要有一个 List，用来保存连接对象。在 ItcastPool 的构造器中创建 5 个连接对象放到 List 中！当用人调用了 ItcastPool 的 getConnection()时，那么就从 List 拿出一个返回。当 List 中没有连接可用时，抛出异常。

我们需要对 Connection 的 close()方法进行增强，所以我们需要自定义 ItcastConnection 类，对 Connection 进行装饰！即对 close()方法进行增强。因为需要在调用 close()方法时把连接“归还”给池，所以 ItcastConnection 类需要拥有池对象的引用，并且池类还要提供“归还”的方法。



ItcastPool.java

```
public class ItcastPool implements DataSource {
    private static Properties props = new Properties();
    private List<Connection> list = new ArrayList<Connection>();
    static {
        InputStream in = ItcastPool.class.getClassLoader()
                .getResourceAsStream("dbconfig.properties");
        try {
            props.load(in);
            Class.forName(props.getProperty("driverClassName"));
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public ItcastPool() throws SQLException {
        for (int i = 0; i < 5; i++) {
            Connection con = DriverManager.getConnection(
                    props.getProperty("url"), props.getProperty("username"),
```

```
                props.getProperty("password"));
            ItcastConnection conWapper = new ItcastConnection(con, this);
            list.add(conWapper);
        }
    }

    public void add(Connection con) {
        list.add(con);
    }

    public Connection getConnection() throws SQLException {
        if(list.size() > 0) {
            return list.remove(0);
        }
        throw new SQLException("没连接了");
    }
    ......
}
```

ItcastConnection.java

```
public class ItcastConnection extends ConnectionWrapper {
    private ItcastPool pool;

    public ItcastConnection(Connection con, ItcastPool pool) {
        super(con);
        this.pool = pool;
    }

    @Override
    public void close() throws SQLException {
        pool.add(this);
    }
}
```

DBCP

1 什么是 DBCP?

DBCP 是 Apache 提供的一款开源免费的数据库连接池!

Hibernate3.0 之后不再对 DBCP 提供支持！因为 Hibernate 声明 DBCP 有致命的缺欠！DBCP 因为 Hibernate 的这一毁谤很是生气，并且说自己没有缺欠。

2 DBCP 的使用

```
    public void fun1() throws SQLException {
        BasicDataSource ds = new BasicDataSource();
        ds.setUsername("root");
        ds.setPassword("123");
        ds.setUrl("jdbc:mysql://localhost:3306/mydb1");
        ds.setDriverClassName("com.mysql.jdbc.Driver");

        ds.setMaxActive(20);
        ds.setMaxIdle(10);
        ds.setInitialSize(10);
        ds.setMinIdle(2);
        ds.setMaxWait(1000);

        Connection con = ds.getConnection();
        System.out.println(con.getClass().getName());
        con.close();
    }
```

3 DBCP 的配置信息

下面是对 DBCP 的配置介绍：

```
#基本配置
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/mydb1
username=root
password=123

#初始化池大小，即一开始池中就会有10个连接对象
默认值为0
initialSize=0

#最大连接数，如果设置maxActive=50时，池中最多可以有50个连接，当然这50个连接中包含被使用的和没被使用的（空闲）
#你是一个包工头，你一共有50个工人，但这50个工人有的当前正在工作，有的正在空闲
#默认值为8，如果设置为非正数，表示没有限制！即无限大
maxActive=8
```

```
#最大空闲连接
#当设置maxIdle=30时，你是包工头，你允许最多有20个工人空闲，如果现在有30个空闲工人，那么要开除10个
#默认值为8，如果设置为负数，表示没有限制！即无限大
maxIdle=8

#最小空闲连接
#如果设置minIdel=5时，如果你的工人只有3个空闲，那么你需要再去招2个回来，保证有5个空闲工人
#默认值为0
minIdle=0

#最大等待时间
#当设置maxWait=5000时，现在你的工作都出去工作了，又来了一个工作，需要一个工人。
#这时就要等待有工人回来，如果等待5000毫秒还没回来，那就抛出异常
#没有工人的原因：最多工人数为50，已经有50个工人了，不能再招了，但50人都出去工作了。
#默认值为-1，表示无限期等待，不会抛出异常。
maxWait=-1

#连接属性
#就是原来放在url后面的参数，可以使用connectionProperties来指定
#如果已经在url后面指定了，那么就不用在这里指定了。
#useServerPrepStmts=true，MySQL开启预编译功能
#cachePrepStmts=true，MySQL开启缓存PreparedStatement功能，
#prepStmtCacheSize=50，缓存PreparedStatement的上限
#prepStmtCacheSqlLimit=300，当SQL模板长度大于300时，就不再缓存它
connectionProperties=useUnicode=true;characterEncoding=UTF8;useServerPrepStmts=true;cachePrepStmts=true;prepStmtCacheSize=50;prepStmtCacheSqlLimit=300

#连接的默认提交方式
#默认值为true
defaultAutoCommit=true

#连接是否为只读连接
#Connection有一对方法：setReadOnly(boolean)和isReadOnly()
#如果是只读连接，那么你只能用这个连接来做查询
#指定连接为只读是为了优化！这个优化与并发事务相关！
#如果两个并发事务，对同一行记录做增、删、改操作，是不是一定要隔离它们啊？
#如果两个并发事务，对同一行记录只做查询操作，那么是不是就不用隔离它们了？
#如果没有指定这个属性值，那么是否为只读连接，这就由驱动自己来决定了。即Connection的实现类自己来决定！
defaultReadOnly=false

#指定事务的事务隔离级别
```

```
#可选值：NONE,READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
#如果没有指定，那么由驱动中的Connection实现类自己来决定
defaultTransactionIsolation=REPEATABLE_READ
```

C3P0

1 C3P0 简介

C3P0 也是开源免费的连接池！C3P0 被很多人看好！

2 C3P0 的使用

C3P0 中池类是：ComboPooledDataSource。

```
    public void fun1() throws PropertyVetoException, SQLException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        ds.setJdbcUrl("jdbc:mysql://localhost:3306/mydb1");
        ds.setUser("root");
        ds.setPassword("123");
        ds.setDriverClass("com.mysql.jdbc.Driver");

        ds.setAcquireIncrement(5);
        ds.setInitialPoolSize(20);
        ds.setMinPoolSize(2);
        ds.setMaxPoolSize(50);

        Connection con = ds.getConnection();
        System.out.println(con);
        con.close();
    }
```

c3p0 也可以指定配置文件，而且配置文件可以是 properties，也可骒 xml 的。当然 xml 的高级一些了。但是 c3p0 的配置文件名必须为 c3p0-config.xml，并且必须放在类路径下。

```
<?xml version="1.0" encoding="UTF-8"?>
<c3p0-config>
    <default-config>
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/mydb1</property>
        <property name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="user">root</property>
        <property name="password">123</property>
        <property name="acquireIncrement">3</property>
```

```
        <property name="initialPoolSize">10</property>
        <property name="minPoolSize">2</property>
        <property name="maxPoolSize">10</property>
    </default-config>
    <named-config name="oracle-config">
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/mydb1</property>
        <property name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="user">root</property>
        <property name="password">123</property>
        <property name="acquireIncrement">3</property>
        <property name="initialPoolSize">10</property>
        <property name="minPoolSize">2</property>
        <property name="maxPoolSize">10</property>
    </named-config>
</c3p0-config>
```

c3p0 的配置文件中可以配置多个连接信息，可以给每个配置起个名字，这样可以方便的通过配置名称来切换配置信息。上面文件中默认配置为 mysql 的配置，名为 oracle-config 的配置也是 mysql 的配置，呵呵。

```
    public void fun2() throws PropertyVetoException, SQLException {
        ComboPooledDataSource ds = new ComboPooledDataSource();
        Connection con = ds.getConnection();
        System.out.println(con);
        con.close();
    }
```

```
    public void fun2() throws PropertyVetoException, SQLException {
        ComboPooledDataSource ds = new ComboPooledDataSource("orcale-config");
        Connection con = ds.getConnection();
        System.out.println(con);
        con.close();
    }
```

Tomcat 配置连接池

1 Tomcat 配置 JNDI 资源

JNDI（Java Naming and Directory Interface），Java 命名和目录接口。JNDI 的作用就是：在服务器上配置资源，然后通过统一的方式来获取配置的资源。

我们这里要配置的资源当然是连接池了，这样项目中就可以通过统一的方式来获取连接池对象了。

下图是 Tomcat 文档提供的：

```
<Context ...>
  ...
  <Resource name="bean/MyBeanFactory" auth="Container"
            type="com.mycompany.MyBean"
            factory="org.apache.naming.factory.BeanFactory"
            bar="23"/>
  ...
</Context>
```

配置 JNDI 资源需要到<Context>元素中配置<Resource>子元素：

- name：指定资源的名称，这个名称可以随便给，在获取资源时需要这个名称；
- factory：用来创建资源的工厂，这个值基本上是固定的，不用修改；
- type：资源的类型，我们要给出的类型当然是我们连接池的类型了；
- bar：表示资源的属性，如果资源存在名为 bar 的属性，那么就配置 bar 的值。对于 DBCP 连接池而言，你需要配置的不是 bar，因为它没有 bar 这个属性，而是应该去配置 url、username 等属性。

```
<Context>
  <Resource name="mydbcp"
            type="org.apache.tomcat.dbcp.dbcp.BasicDataSource"
            factory="org.apache.naming.factory.BeanFactory"
            username="root"
            password="123"
            driverClassName="com.mysql.jdbc.Driver"
            url="jdbc:mysql://127.0.0.1/mydb1"
            maxIdle="3"
            maxWait="5000"
            maxActive="5"
            initialSize="3"/>
</Context>
```

```
<Context>
  <Resource name="myc3p0"
            type="com.mchange.v2.c3p0.ComboPooledDataSource"
            factory="org.apache.naming.factory.BeanFactory"
            user="root"
            password="123"
            classDriver="com.mysql.jdbc.Driver"
            jdbcUrl="jdbc:mysql://127.0.0.1/mydb1"
            maxPoolSize="20"
            minPoolSize ="5"
            initialPoolSize="10"
```

```
            acquireIncrement="2"/>
</Context>
```

2 获取资源

配置资源的目的当然是为了获取资源了。只要你启动了 Tomcat，那么就可以在项目中任何类中通过 JNDI 获取资源的方式来获取资源了。

下图是 Tomcat 文档提供的，与上面 Tomcat 文档提供的配置资源是对应的。

```
Context initCtx = new InitialContext();
Context envCtx = (Context) initCtx.lookup("java:comp/env");
MyBean bean = (MyBean) envCtx.lookup("bean/MyBeanFactory");

writer.println("foo = " + bean.getFoo() + ", bar = " +
               bean.getBar());
```

获取资源：

- Context：javax.naming.Context；
- InitialContext：javax.naming.InitialContext；
- lookup(String)：获取资源的方法，其中”java:comp/env”是资源的入口（这是固定的名称），获取过来的还是一个 Context，这说明需要在获取到的 Context 上进一步进行获取。”bean/MyBeanFactory”对应<Resource>中配置的 name 值，这回获取的就是资源对象了。

```
Context cxt = new InitialContext();
DataSource ds = (DataSource)cxt.lookup("java:/comp/env/mydbcp");
Connection con = ds.getConnection();
System.out.println(con);
con.close();
```

```
Context cxt = new InitialContext();
Context envCxt = (Context)cxt.lookup("java:/comp/env");
DataSource ds = (DataSource)env.lookup("mydbcp");
Connection con = ds.getConnection();
System.out.println(con);
con.close();
```

上面两种方式是相同的效果。

修改 JdbcUtils

因为已经学习了连接池，那么 JdbcUtils 的获取连接对象的方法也要修改一下了。

JdbcUtils.java

```
public class JdbcUtils {
    private static DataSource dataSource = new ComboPooledDataSource();

    public static DataSource getDataSource() {
        return dataSource;
    }

    public static Connection getConnection() {
        try {
            return dataSource.getConnection();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

ThreadLocal

1 ThreadLocal API

ThreadLocal 类只有三个方法：

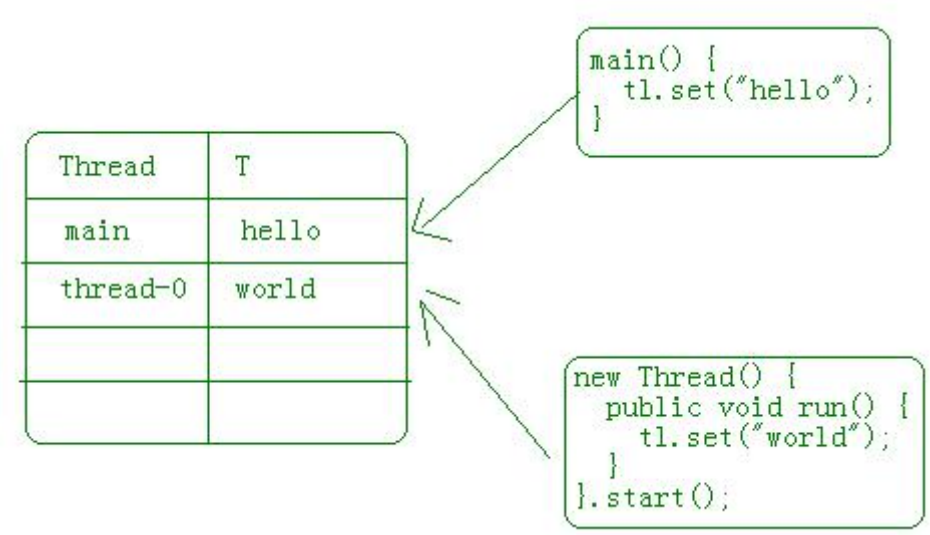
- void set(T value)：保存值；
- T get()：获取值；
- void remove()：移除值。

2 ThreadLocal 的内部是 Map

ThreadLocal 内部其实是个 Map 来保存数据。虽然在使用 ThreadLocal 时只给出了值，没有给出键，其实它内部使用了当前线程做为键。

```
class MyThreadLocal<T> {
    private Map<Thread,T> map = new HashMap<Thread,T>();
    public void set(T value) {
        map.put(Thread.currentThread(), value);
    }

    public void remove() {
        map.remove(Thread.currentThread());
    }

    public T get() {
```

```
        return map.get(Thread.currentThread());
    }
}
```



BaseServlet

1 BaseServlet 的作用

在开始客户管理系统之前，我们先写一个工具类：BaseServlet。

我们知道，写一个项目可能会出现 N 多个 Servlet，而且一般一个 Servlet 只有一个方法（doGet 或 doPost），如果项目大一些，那么 Servlet 的数量就会很惊人。

为了避免 Servlet 的“膨胀”，我们写一个 BaseServlet。它的作用是让一个 Servlet 可以处理多种不同的请求。不同的请求调用 Servlet 的不同方法。我们写好了 BaseServlet 后，让其他 Servlet 继承 BaseServlet，例如 CustomerServlet 继承 BaseServlet，然后在 CustomerServlet 中提供 add()、update()、delete()等方法，每个方法对应不同的请求。

2 BaseServlet 分析

我们知道，Servlet 中处理请求的方法是 service()方法，这说明我们需要让 service()方法去调用其他方法。例如调用 add()、mod()、del()、all()等方法！具体调用哪个方法需要在请求中给出方法名称！然后 service()方法通过方法名称来调用指定的方法。

无论是点击超链接，还是提交表单，请求中必须要有 method 参数，这个参数的值就是要请求的方法名称，这样 BaseServlet 的 service()才能通过方法名称来调用目标方法。例如某个链接如下：

添加客户

3 BaseServlet 代码

```
public class BaseServlet extends HttpServlet {
    /*
     * 它会根据请求中的m，来决定调用本类的哪个方法
     */
    protected void service(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {
        req.setCharacterEncoding("UTF-8");
        res.setContentType("text/html;charset=utf-8");

        // 例如：http://localhost:8080/demo1/xxx?m=add
        String methodName = req.getParameter("method");// 它是一个方法名称
```

```
        // 当没用指定要调用的方法时，那么默认请求的是execute()方法。
        if(methodName == null || methodName.isEmpty()) {
            methodName = "execute";
        }
        Class c = this.getClass();
        try {
            // 通过方法名称获取方法的反射对象
            Method m = c.getMethod(methodName, HttpServletRequest.class,
                    HttpServletResponse.class);
            // 反射方法目标方法，也就是说，如果methodName为add，那么就调用add方法。
            String result = (String) m.invoke(this, req, res);
            // 通过返回值完成请求转发
            if(result != null && !result.isEmpty()) {
                req.getRequestDispatcher(result).forward(req, res);
            }
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }
}
```

DBUtils

在大家学习 Hibernate 之前，你会一起使用它！
这个东西虽然很小，但还是有公司在使用它！

1 DBUtils 简介

DBUtils 是 Apache Commons 组件中的一员，开源免费！
DBUtils 是对 JDBC 的简单封装，但是它还是被很多公司使用！
DBUtils 的 Jar 包：dbutils.jar

2 DBUtils 主要类

- DbUtils：都是静态方法，一系列的 close()方法；
- QueryRunner：
  - update()：执行 insert、update、delete；
  - query()：执行 select 语句；
  - batch()：执行批处理。

3 QueryRunner 之更新

QueryRunner 的 update()方法可以用来执行 insert、update、delete 语句。

1. 创建 QueryRunner
   构造器：QueryRunner();

2. update()方法
   int update(Connection con, String sql, Object… params)

```
@Test
public void fun1() throws SQLException {
    QueryRunner qr = new QueryRunner();
    String sql = "insert into user values(?,?,?)";
    qr.update(JdbcUtils.getConnection(), sql, "u1", "zhangSan", "123");
}
```

还有另一种方式来使用 QueryRunner

1. 创建 QueryRunner
   构造器：QueryRunner(DataSource)

2. update()方法
   int update(String sql, Object… params)

这种方式在创建 QueryRunner 时传递了连接池对象，那么在调用 update()方法时就不用再传递 Connection 了。

```
@Test
public void fun2() throws SQLException {
    QueryRunner qr = new QueryRunner(JdbcUtils.getDataSource());
    String sql = "insert into user values(?,?,?)";
    qr.update(sql, "u1", "zhangSan", "123");
}
```

4 ResultSetHandler

我们知道在执行 select 语句之后得到的是 ResultSet，然后我们还需要对 ResultSet 进行转换，得到最终我们想要的数据。你可以希望把 ResultSet 的数据放到一个 List 中，也可能想把数据放到一个 Map 中，或是一个 Bean 中。

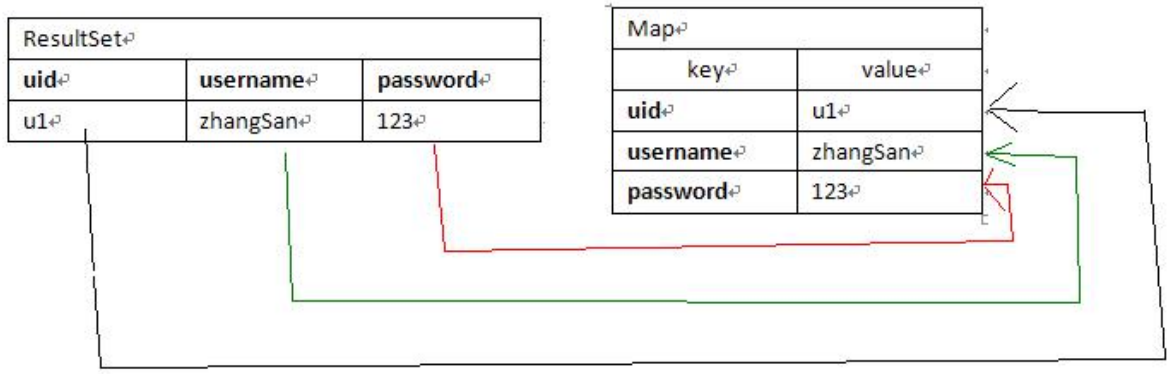
DBUtils 提供了一个接口 ResultSetHandler，它就是用来 ResultSet 转换成目标类型的工具。你可以自己去实现这个接口，把 ResultSet 转换成你想要的类型。

DBUtils 提供了很多个 ResultSetHandler 接口的实现，这些实现已经基本够用了，我们通常不用自己去实现 ResultSet 接口了。

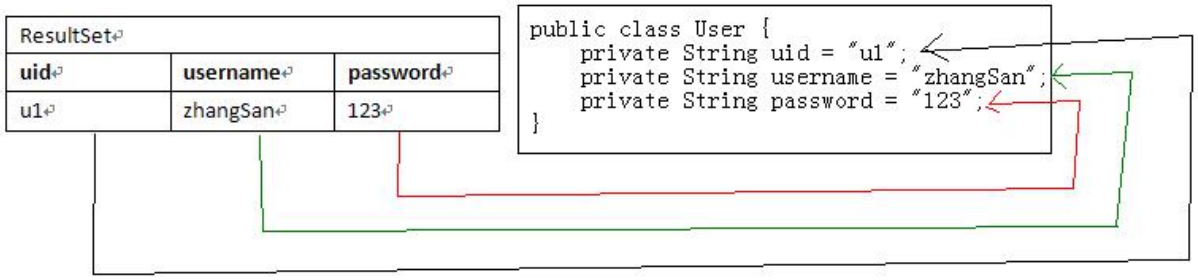
- MapHandler：单行处理器！把结果集转换成 Map<String,Object>，其中列名为键！
- MapListHandler：多行处理器！把结果集转换成 List<Map<String,Object>>；

- BeanHandler：单行处理器！把结果集转换成 Bean，该处理器需要 Class 参数，即 Bean 的类型；
- BeanListHandler：多行处理器！把结果集转换成 List<Bean>；
- ColumnListHandler：多行单列处理器！把结果集转换成 List<Object>，使用 ColumnListHandler 时需要指定某一列的名称或编号，例如：new ColumListHandler(“name”)表示把 name 列的数据放到 List 中。
- ScalarHandler：单行单列处理器！把结果集转换成 Object。一般用于聚集查询，例如 select count(*) from tab_student。

Map 处理器

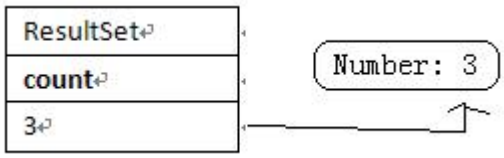


Bean 处理器



Column 处理器

Scalar 处理器



5 QueryRunner 之查询

QueryRunner 的查询方法是：

public <T> T query(String sql, ResultSetHandler<T> rh, Object… params)

public <T> T query(Connection con, String sql, ResultSetHandler<T> rh, Object… params)

query()方法会通过 sql 语句和 params 查询出 ResultSet，然后通过 rh 把 ResultSet 转换成对应的类型再返回。

```
    @Test
    public void fun1() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select * from tab_student where number=?";
        Map<String,Object> map = qr.query(sql, new MapHandler(), "S_2000");
        System.out.println(map);
    }

    @Test
    public void fun2() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select * from tab_student";
        List<Map<String,Object>> list = qr.query(sql, new MapListHandler());
        for(Map<String,Object> map : list) {
            System.out.println(map);
        }
    }

    @Test
    public void fun3() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select * from tab_student where number=?";
        Student stu = qr.query(sql, new BeanHandler<Student>(Student.class),
"S_2000");
        System.out.println(stu);
    }
```

```
    @Test
    public void fun4() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select * from tab_student";
        List<Student> list = qr.query(sql, new
BeanListHandler<Student>(Student.class));
        for(Student stu : list) {
            System.out.println(stu);
        }
    }

    @Test
    public void fun5() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select * from tab_student";
        List<Object> list = qr.query(sql, new ColumnListHandler("name"));
        for(Object s : list) {
            System.out.println(s);
        }
    }

    @Test
    public void fun6() throws SQLException {
        DataSource ds = JdbcUtils.getDataSource();
        QueryRunner qr = new QueryRunner(ds);
        String sql = "select count(*) from tab_student";
        Number number = (Number)qr.query(sql, new ScalarHandler());
        int cnt = number.intValue();
        System.out.println(cnt);
    }
```

5 QueryRunner 之批处理

QueryRunner 还提供了批处理方法：batch()。

我们更新一行记录时需要指定一个 Object[]为参数，如果是批处理，那么就要指定 Object[][]为参数了。即多个 Object[]就是 Object[][]了，其中每个 Object[]对应一行记录：

```
@Test
public void fun10() throws SQLException {
    DataSource ds = JdbcUtils.getDataSource();
    QueryRunner qr = new QueryRunner(ds);
```

```
    String sql = "insert into tab_student values(?,?,?,?)";
    Object[][] params = new Object[10][];//表示 要插入10行记录
    for(int i = 0; i < params.length; i++) {
        params[i] = new Object[]{"S_300" + i, "name" + i, 30 + i, i%2==0?"男
":"女"};
    }
    qr.batch(sql, params);
}
```

day13

今日内容

- Service 事务
- 客户关系管理系统

Service 事务

在 Service 中使用 ThreadLocal 来完成事务，为将来学习 Spring 事务打基础！

1 DAO 中的事务

在 DAO 中处理事务真是“小菜一碟”。

```
public void xxx() {
    Connection con = null;
    try {
        con = JdbcUtils.getConnection();
        con.setAutoCommitted(false);
        QueryRunner qr = new QueryRunner();
        String sql = …;
        Object[] params = …;
        qr.update(con, sql, params);

        sql = …;
        Object[] params = …;
        qr.update(con, sql, params);
        con.commit();
    } catch(Exception e) {
          try {
              if(con != null) {con.rollback();}
          } catch(Exception e) {}
    } finally {
          try {
              con.close();
          } catch(Exception e) {}
    }
}
```

2 Service 才是处理事务的地方

我们要清楚一件事，DAO 中不是处理事务的地方，因为 DAO 中的每个方法都是对数据库的一次操作，而 Service 中的方法才是对应一个业务逻辑。也就是说我们需要在 Service 中的一方法中调用 DAO 的多个方法，而这些方法应该在一起事务中。

怎么才能让 DAO 的多个方法使用相同的 Connection 呢？方法不能再自己来获得 Connection，而是由外界传递进去。

```
public void daoMethod1(Connection con, …) {
}
public void daoMethod2(Connection con, …) {
}
```

在 Service 中调用 DAO 的多个方法时，传递相同的 Connection 就可以了。

```
public class XXXService() {
    private XXXDao dao = new XXXDao();
    public void serviceMethod() {
        Connection con = null;
        try {
            con = JdbcUtils.getConnection();
            con.setAutoCommitted(false);
            dao.daoMethod1(con, …);
            dao.doaMethod2(con, …);
            com.commint();
        } catch(Exception e) {
            try {
                con.rollback();
            } catch(Exception e) {}
        } finally {
            try {
               con.close();
            } catch(Exception e) {}
        }
    }
}
```

但是，在 Service 中不应该出现 Connection，它应该只在 DAO 中出现，因为它是 JDBC 的东西，JDBC 的东西是用来连接数据库的，连接数据库是 DAO 的事儿!!! 但是，事务是 Service 的事儿，不能放到 DAO 中!!!

3 修改 JdbcUtils

我们把对事务的开启和关闭放到 JdbcUtils 中，在 Service 中调用 JdbcUtils 的方法来完成事务的处理，但在 Service 中就不会再出现 Connection 这一“禁忌”了。

DAO 中的方法不用再让 Service 来传递 Connection 了。DAO 会主动从 JdbcUtils 中获取 Connection 对象，这样，JdbcUtils 成为了 DAO 和 Service 的中介！

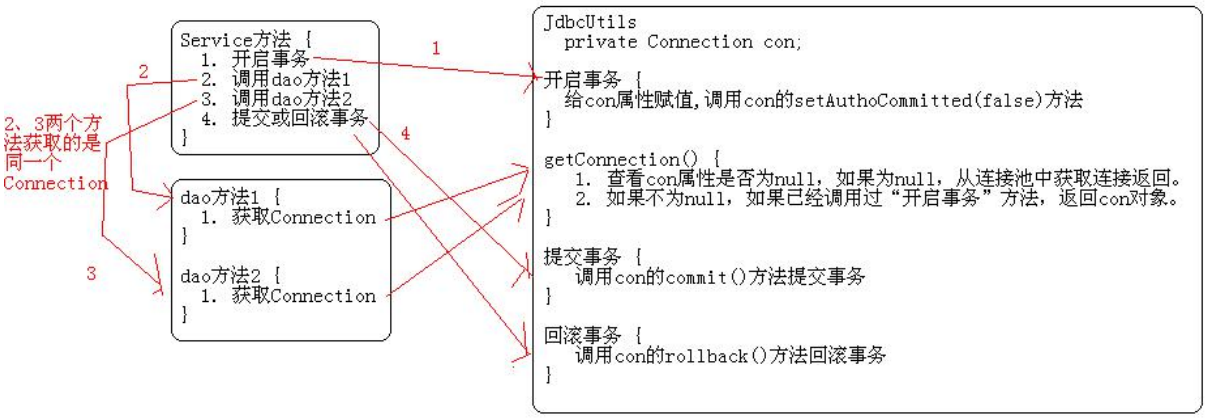
我们在 JdbcUtils 中添加 beginTransaction()和 rollbackTransaction()，以及 commitTransaction()方法。这样在 Service 中的代码如下：

```
public class XXXService() {
    private XXXDao dao = new XXXDao();
    public void serviceMethod() {
        try {
            JdbcUtils.beginTransaction();
            dao.daoMethod1(…);
            dao.daoMethod2(…);
            JdbcUtils.commitTransaction();
        } catch(Exception e) {
            JdbcUtils.rollbackTransaction();
        }
    }
}
```

DAO

```
public void daoMethod1(…) {
   Connection con = JdbcUtils.getConnection();
}
public void daoMethod2(…) {
   Connection con = JdbcUtils.getConnection();
}
```

在 Service 中调用了 JdbcUtils.beginTransaction()方法时，JdbcUtils 要做准备好一个已经调用了 setAuthCommitted(false)方法的 Connection 对象，因为在 Service 中调用 JdbcUtils.beginTransaction()之后，马上就会调用 DAO 的方法，而在 DAO 方法中会调用 JdbcUtils.getConnection()方法。这说明 JdbcUtils 要在 getConnection()方法中返回刚刚准备好的，已经设置了手动提交的 Connection 对象。



在 JdbcUtils 中创建一个 Connection con 属性，当它为 null 时，说明没有事务！当它不为 null 时，

表示开启了事务。

- 在没有开启事务时，可以调用“开启事务”方法；
- 在开启事务后，可以调用“提交事务”和“回滚事务”方法；
- getConnection()方法会在 con 不为 null 时返回 con，再 con 为 null 时，从连接池中返回连接。

beginTransaction()

判断 con 是否为 null，如果不为 null，就抛出异常！

如果 con 为 null，那么从连接池中获取一个 Connection 对象，赋值给 con！然后设置它为“手动提交”。

getConnection()

判断 con 是否为 null，如果为 null 说明没有事务，那么从连接池获取一个连接返回；

如果不为 null，说明已经开始了事务，那么返回 con 属性返回。这说明在 con 不为 null 时，无论调用多少次 getConnection()方法，返回的都是同个 Connection 对象。

commitTransaction()

判断 con 是否为 null，如果为 null，说明没有开启事务就提交事务，那么抛出异常；

如果 con 不为 null，那么调用 con 的 commit()方法来提交事务；

调用 con.close()方法关闭连接；

con = null，这表示事务已经结束！

rollbackTransaction()

判断 con 是否为 null，如果为 null，说明没有开启事务就回滚事务，那么抛出异常；

如果 con 不为 null，那么调用 con 的 rollback()方法来回滚事务；

调用 con.close()方法关闭连接；

con = null，这表示事务已经结束！

JdbcUtils.java

```
public class JdbcUtils {
    private static DataSource dataSource = new ComboPooledDataSource();
    private static Connection con = null;

    public static DataSource getDataSource() {
        return dataSource;
    }

    public static Connection getConnection() throws SQLException {
        if(con == null) {
            return dataSource.getConnection();
        }
        return con;
    }
```

```
    public static void beginTranscation() throws SQLException {
        if(con != null) {
            throw new SQLException("事务已经开启，在没有结束当前事务时，不能再开启事务！
");
        }
        con = dataSource.getConnection();
        con.setAutoCommit(false);
    }

    public static void commitTransaction() throws SQLException {
        if(con == null) {
            throw new SQLException("当前没有事务，所以不能提交事务！");
        }
        con.commit();
        con.close();
        con = null;
    }

    public static void rollbackTransaction() throws SQLException {
        if(con == null) {
            throw new SQLException("当前没有事务，所以不能回滚事务！");
        }
        con.rollback();
        con.close();
        con = null;
    }
}
```

4 再次修改 JdbcUtils

现在 JdbcUtils 有个问题，如果有两个线程！第一个线程调用了 beginTransaction()方法，另一个线程再调用 beginTransaction()方法时，因为 con 已经不再为 null，所以就会抛出异常了。

我们希望 JdbcUtils 可以多线程环境下被使用！这说明最好的方法是为每个线程提供一个 Connection，这样每个线程都可以开启自己的事务了。

还记得 ThreadLocal 类么？

```
public class JdbcUtils {
    private static DataSource dataSource = new ComboPooledDataSource();
    private static ThreadLocal<Connection> tl = new ThreadLocal<Connection>();

    public static DataSource getDataSource() {
        return dataSource;
    }
```

```
    public static Connection getConnection() throws SQLException {
        Connection con = tl.get();
        if(con == null) {
            return dataSource.getConnection();
        }
        return con;
    }

    public static void beginTranscation() throws SQLException {
        Connection con = tl.get();
        if(con != null) {
            throw new SQLException("事务已经开启，在没有结束当前事务时，不能再开启事务!
");
        }
        con = dataSource.getConnection();
        con.setAutoCommit(false);
        tl.set(con);
    }

    public static void commitTransaction() throws SQLException {
        Connection con = tl.get();
        if(con == null) {
            throw new SQLException("当前没有事务，所以不能提交事务！");
        }
        con.commit();
        con.close();
        tl.remove();
    }

    public static void rollbackTransaction() throws SQLException {
        Connection con = tl.get();
        if(con == null) {
            throw new SQLException("当前没有事务，所以不能回滚事务！");
        }
        con.rollback();
        con.close();
        tl.remove();
    }
}
```

5 转账示例

```
public class AccountDao {
```

```
    public void updateBalance(String name, double balance) throws SQLException
{
        String sql = "update account set balance=balance+? where name=?";
        Connection con = JdbcUtils.getConnection();
        QueryRunner qr = new QueryRunner();
        qr.update(con, sql, balance, name);
    }
}
```

```
public class AccountService {
    private AccountDao dao = new AccountDao();

    public void transfer(String from, String to, double balance) {
        try {
            JdbcUtils.beginTranscation();
            dao.updateBalance(from, -balance);
            dao.updateBalance(to, balance);
            JdbcUtils.commitTransaction();
        } catch(Exception e) {
            try {
                JdbcUtils.rollbackTransaction();
            } catch (SQLException e1) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

```
        AccountService as = new AccountService();
        as.transfer("zs", "ls", 100);
```

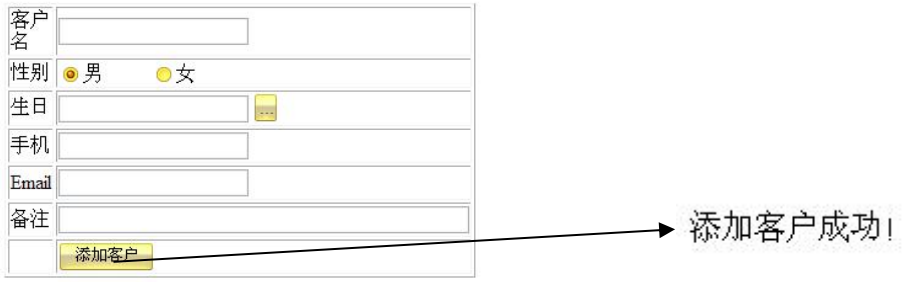
客户关系管理系统

1 功能内容

- 添加客户
- 修改客户
- 删除客户
- 查看客户（分页）

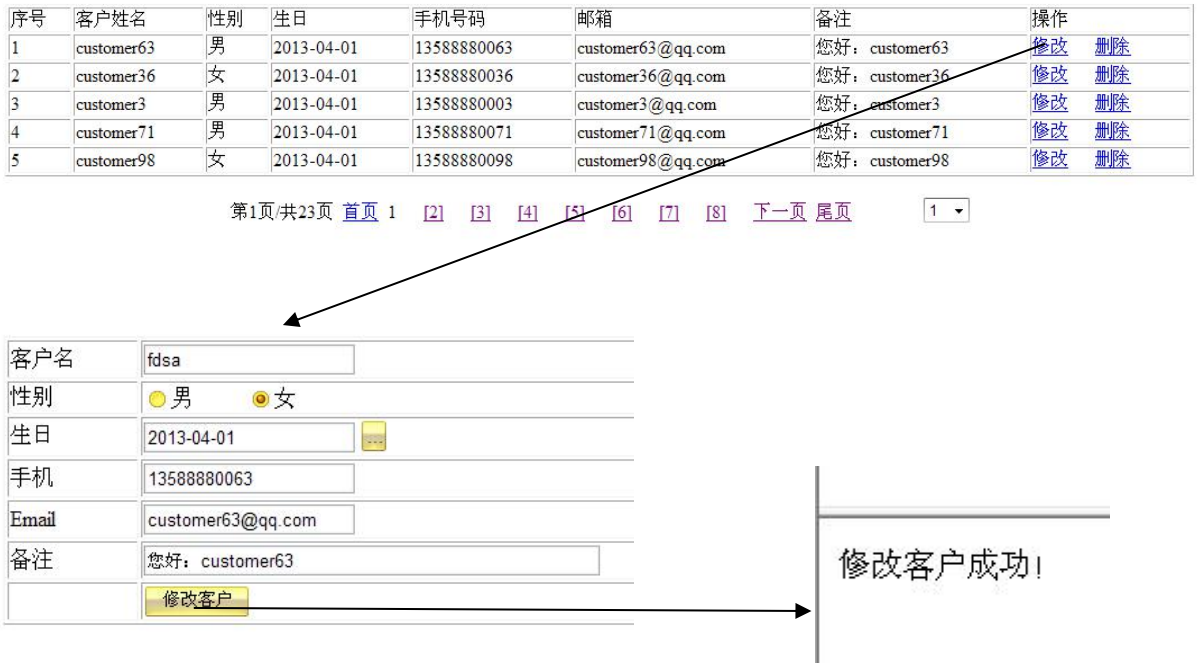
2 演示

添加客户

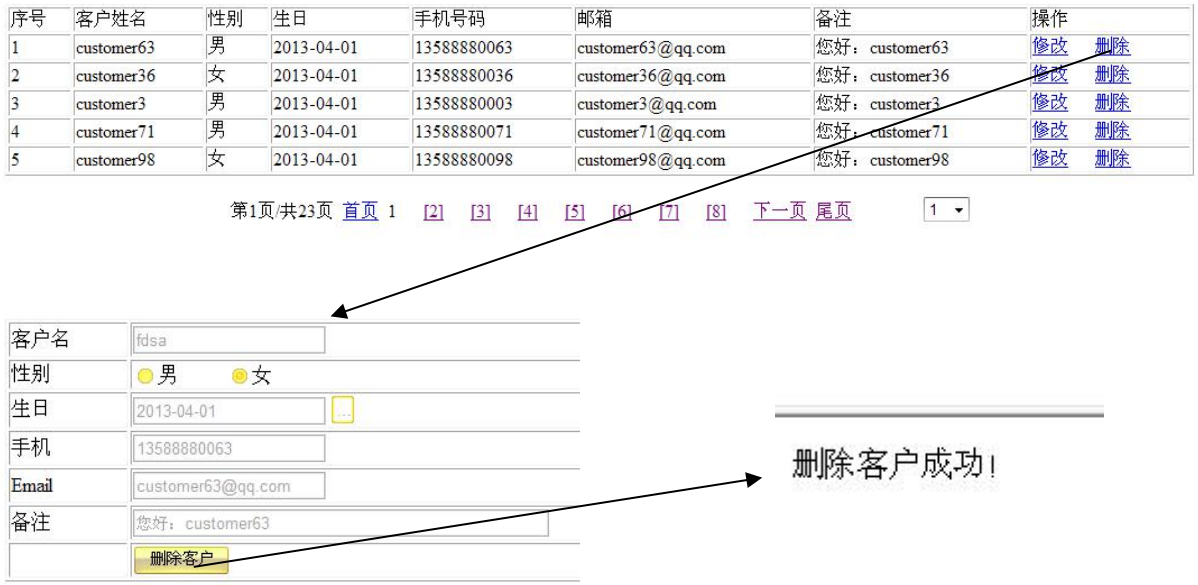


查看客户

修改客户



删除客户



3 搭建环境

1. 创建一个空项目，例如为 customer
2. 导入 jar 包：

- itcast.jar
- mysql.jar
- dbutil.jar、logging.jar
- beanutils.jar

- c3p0.jar,…

3. 页面搭建

- index.jsp（forward 到 main.jsp）
- main.jsp（框架页，两帧，对应 top.jsp 和 body.jsp）
- top.jsp（logo 和两个链接：“添加客户”和“查看客户”）
- body.jsp（只有欢迎信息）
- add.jsp（添加客户表单）
- mod.jsp（修改客户表单）
- del.jsp（删除客户表）

4. 处理页面跳转问题

编写 CustomerServlet，处理页面跳转问题！

4 创建表和类

customer

customer		
字段	类型	说明
cid	char(32)	主键
cname	varchar(30)	客户姓名
gender	varchar(6)	客户性别
birthday	date	客户生日
cellphone	varchar(20)	客户手机
email	varchar(30)	客户邮箱
description	varchar(200)	客户描述

```
CREATE TABLE customer(
    cid CHAR(32) PRIMARY KEY,
    cname VARCHAR(30) NOT NULL,
    gender VARCHAR(6) NOT NULL,
    birthday DATE,
    cellphone VARCHAR(20) NOT NULL,
    email VARCHAR(30),
    description VARCHAR(200)
);
```

Customer 类这里就省略了！

4 添加客户分析

1. 当用户点击“添加客户”链接时，通过 CustomerServlet 的 addPre 转发到 add.jsp
2. 在 add.jsp 中提交表单时，由 CustomerServlet 来处理请求：

- 获取表单数据，封装到 Customer 对象中；
- 调用 CustomerService 代码，把 Customer 添加到数据库；
- 向页面输出“添加成功”。

5 查看客户分析

1. 当用户点击“查看客户”链接时，通过 CustomerServlet 的 list 方法来处理：

- 通过 CustomerService 获取所有客户信息；
- 保存到 request 中；
- 转发到 list.jsp
- list.jsp 使用<c:forEach>查看信息

6 修改客户

1. 当用户在 list.jsp 页面中点击“修改”时，通过 CustomerServlet 的 modPre 方法来处理：

- 获取 cid，即要修改的客户的 cid；
- 通过 cid 来获取 Customer 对象
- 把 Customer 对象保存到 request 中
- 转发到 mod.jsp

2. 当用户在 mod.jsp 提交表单时，通过 Customer 的 mod 方法来处理：

- 获取表单数据，封装到 Customer 对象中；
- 调用 CustomerService 的方法完成修改；
- 向页面输出“修改成功”。

7 删除客户

1. 当用户在 list.jsp 页面中点击“删除”时，通过 CustomerServlet 的 delPre 方法来处理：

- 获取 cid；
- 通过 cid 获取 Customer 对象；
- 把 Customer 对象保存到 request 中
- 转发到 del.jsp

2. 当用户在 del.jsp 页面点击删除时，通过 Customer 的 del 广场来处理：

1. 获取 cid
2. 通过 CustomerService 来完成删除
3. 向页面输出“删除成功”

分页

1 分页数据分析

页面需要什么数据：

- 当前页页码（currPageCode）：Servlet 提供；
- 共几页（totalPage）：Servlet 提供；
- 当前页数据（datas）：Servlet 提供；

Servlet 需要什么数据：

- 当前页页码（currPageCode）：页面提供，如果页面没有提供，那么默认为 1；
- 总记录数（totalRecord）：通过数据库来查询；
- 每页记录数（pagesize）：系统数据；
- 共几页（totalPage）：通过 totalRecord 和 pagesize 来计算；
- 当前页第一行记录位置（currPageBeginIndex）：通过 currPageCode 和 pagesize 计算；
- 当前页数据（datas）：通过 currPageBginIndex 和 pagesize 查询数据库；

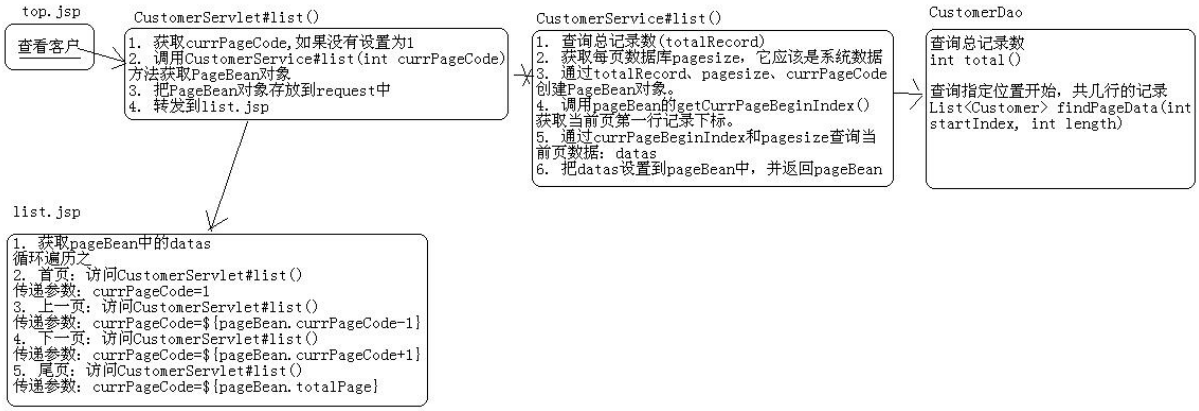
2 PageBean

把分布数据封装成 PageBean 类对象

```
public class PageBean<T> {
    private List<T> datas;// 当前页记录数, 需要传递
    private int totalRecord;// 总记录数, 需要传递
    private int currPageCode;// 当前页码, 需要传递
    private int pagesize;// 每页记录数, 需要传递
    private int totalPage;// 总页数, 计算
    private int currPageBeginIndex; //需要计算
    public PageBean(int currPageCode, int totalRecord, int pagesize) {
        this.currPageCode = currPageCode;
        this.totalRecord = totalRecord;
        this.pagesize = pagesize;

        init();
    }

    private void init() {
        this.totalPage = totalRecord / pagesize;
        if(totalRecord % pagesize != 0) {
            this.totalPage++;
        }
        this.currPageBeginIndex = (this.currPageCode-1) * this.pagesize;
```

```
    }
    ...
}
```

3 分页分析



4 页码列表

首页 1 [2] [3] [4] [5] [6] [7] [8] 下一页 尾页

其中红框中的就是页码列表！

4.1 页面需要的数据：

- 列表的开始页码（beginIndex）；
- 列表的结束页码（endIndex）。

例如开始页码为 11，结束页码为 18，那么就显示：

首页 上一页 [11] [12] [13] 14 [15] [16] [17] [18] 下一页 尾页

对于页面，它只需要 beginIndex 和 endIndex，然后使用<c:forEach>就可以循环显示了！然后再判断一下遍历的数字如果与 pageBean.currPageCode 相等，那么就不要显示为链接即可。

4.2 PageBean

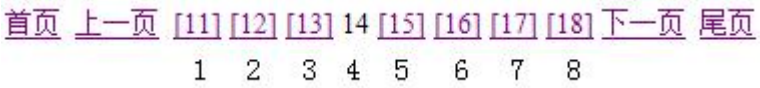
页面需要的数据由 PageBean 来提供，即为 pageBean 添加两个方法：

- int getBeginIndex()
- int getEndIndex()

但是，PageBean 想计算这两个值，也要需要两个系统数据：

- pageCodeListSize：页码列表长度，下图中的列表长度为 8，即最多显示 8 个页码；
- currPageCodeListIndex：当前页码在列表中的位置，下图中当前页码的位置为 4，位置是从 1

开始计算。



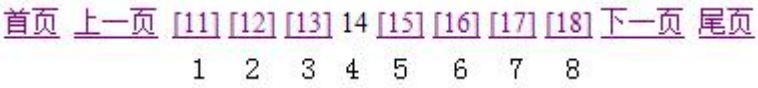
4.3 计算 beginIndex

计算 beginIndex 分为 4 步：

1. 如果总页数（totalPage）小于列表长度（pageCodeListSize），那么 beginIndex 为 1;

例如，当前数据一共 5 页，那么列表的开始页码一定是 1。

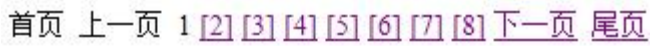
2. 当上面条件不成立时，使用当前页码在列表中的位置（currPageCodeListIndex），以及当前页码（currPageCode）来推算出 beginIndex。

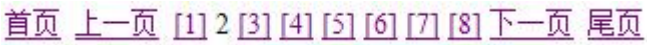


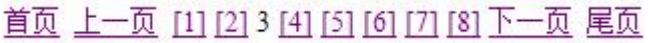
上图中当前页码（currPageCode）为 14，当前页码位置为 4（currPageCodeListIndex），推算出 beginIndex 为 11，即 beginIndex = currPageCode- currPageCodeListIndex+1;

3. 第 2 步计算出的 beginIndex 如果小于 1，即让 beginIndex=1。

第 2 步的计算可能会出现问题，例如在当前页码（currPageCode）为 1 时，那么上面的计算就会出现 beginIndex 小于 1 的情况！这种 beginIndex 小于 1 的情况，只有在 currPageCode 为 1、2、3 时会出现。你可以去套用一下第 2 步的公式，会发现这个问题的！







也就是说，beginIndex 的最小值就是 1

4. 第 2 步计算出的 beginIndex 可能会导致列表长度不正确

当 cuarrPageCode 为 30 时，那么通过第 2 步计算出的 beginIndex 为 27，但是如果 totalPage 为 30 呢？因为一共就 30 页，总不能显示 31 出来吧。那么显示：27、28、29、30，这就只有 4 个页码，但列表长度应该为 8，所以出错。

我们为了验证第 2 步是否出现这个错误，需要通过得到的 beginIndex 来推算 endIndex。例如 currPageCode 为 30，计算出 beginIndex 为 27，再通过 beginIndex 计算出 endIndex 为 34。然后查看 endIndex 是否大于 totalPage，如果大于了 totalPage，那么应该使用 totalPage 减去 pageCodeListSize 再加 1，得到正确的 beginIndex。

int endIndex = beginIndex+pageCodeListSize-1;

if(endIndex > totalPage) beginIndex = totalPage-pageCodeListSize + 1;

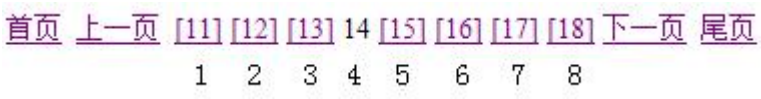
```
public int getBeginIndex() {
```

```
        if(totalPage <= pageCodeListSize) return 1;
        int begin = currPageCode - currPageCodeListIndex+1;
        if(begin < 1) begin = 1;
        int end = begin + pageCodeListSize-1;
        if(end > totalPage) begin = totalPage - pageCodeListSize+1;
        return begin;
    }
```

4.4 计算 endIndex

计算 endIndex 也是分为 4 步

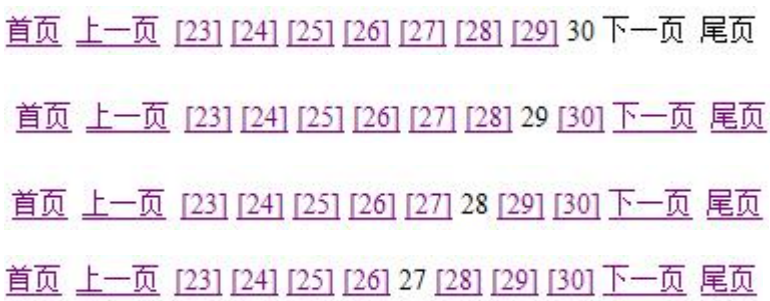
1. 如果 totalPage 小于 pageCodeListSize，那么 endIndex 为 totalPage
2. 通过 pageCodeListSize、currPageCode、currPageCodeListIndex 来推出 endIndex。



currPageCode=14，pageCodeListSize=8，currPageCodeListIndex=4，所以 endIndex = currPageCode+(pageCodeListSize=currPagecodeListIndex)=18。

3. 第 2 步的计算结果可能会大于 totalPage，那么就重置 endIndex 为 totalPage

如果 currPageCode 为 30，那么通过第 2 步计算出的 endIndex 为 34，但是如果 totalPage 为 30 呢？那么就让 endIndex=totalPage。



当 currPageCode 为 27、28、29、30 时，如果 totalPage=30，那么第 2 步都会计算出错。这时就把 endIndex=totalPage

4. 第 2 步的计算结果可能会导致错误的列表长度

当 currPageCode 为 1 时，那么通过第 2 步计算的 endIndex 为 5，即列表为 1、2、3、4、5，但列表长度应该为 8，所以当第 2 步计算出的 endIndex< pageCodeListSize，那么让 endIndex 等于列表长度。

```
    public int getEndIndex() {
        if(totalPage <= pageCodeListSize) return totalPage;
        int end = currPageCode + (pageCodeListSize-currPageCodeListIndex);
        if(end > totalPage) end = totalPage;
        if(end < pageCodeListSize) end = pageCodeListSize;
        return end;
    }
```

day14

JavaWeb 监听器

1 JavaWeb 监听器概述

在 JavaWeb 被监听的事件源为：ServletContext、HttpSession、ServletRequest，即三大域对象。

- 监听域对象“创建”与“销毁”的监听器；
- 监听域对象“操作域属性”的监听器；
- 监听 HttpSession 的监听器。

2 创建与销毁监听器

创建与销毁监听器一共有三个：

- ServletContextListener：Tomcat 启动和关闭时调用下面两个方法
  - public void contextInitialized(ServletContextEvent evt)：ServletContext 对象被创建后调用；
  - public void contextDestroyed(ServletContextEvent evt)：ServletContext 对象被销毁前调用；
- HttpSessionListener：开始会话和结束会话时调用下面两个方法
  - public void sessionCreated(HttpSessionEvent evt)：HttpSession 对象被创建后调用；
  - public void sessionDestroyed(HttpSessionEvent evt)：HttpSession 对象被销毁前调用；
- ServletRequestListener：开始请求和结束请求时调用下面两个方法
  - public void requestInitiallized(ServletRequestEvent evt)：ServletRequest 对象被创建后调用；
  - public void requestDestroyed(ServletRequestEvent evt)：ServletRequest 对象被销毁前调用。

编写测试例子：

- 编写 MyServletContextListener 类，实现 ServletContextListener 接口；
- 在 web.xml 文件中部署监听器；
- 为了看到 session 销毁的效果，在 web.xml 文件中设置 session 失效时间为 1 分钟；

```
/*
 * ServletContextListener实现类
 * contextDestroyed() -- 在ServletContext对象被销毁前调用
 * contextInitialized() --  -- 在ServletContext对象被创建后调用
 * ServletContextEvent -- 事件类对象
 *     该类有getServletContext()，用来获取ServletContext对象，即获取事件源对象
```

```
 */
public class MyServletContextListener implements ServletContextListener {
    public void contextDestroyed(ServletContextEvent evt) {
        System.out.println("销毁ServletContext对象");
    }

    public void contextInitialized(ServletContextEvent evt) {
        System.out.println("创建ServletContext对象");
    }
}
```

```
/*
 * HttpSessionListener实现类
 * sessionCreated() -- 在HttpSession对象被创建后被调用
 * sessionDestroyed() --  -- 在HttpSession对象被销毁前调用
 * HttpSessionEvent -- 事件类对象
 *     该类有getSession()，用来获取当前HttpSession对象，即获取事件源对象
 */
public class MyHttpSessionListener implements HttpSessionListener {
    public void sessionCreated(HttpSessionEvent evt) {
        System.out.println("创建session对象");
    }

    public void sessionDestroyed(HttpSessionEvent evt) {
        System.out.println("销毁session对象");
    }
}
```

```
/*
 * ServletRequestListener实现类
 * requestDestroyed() -- 在ServletRequest对象被销毁前调用
 * requestInitialized() -- 在ServletRequest对象被创建后调用
 * ServletRequestEvent -- 事件类对象
 *     该类有getServletContext()，用来获取ServletContext对象
 *     该类有getServletRequest()，用来获取当前ServletRequest对象，即事件源对象
 */
public class MyServletRequestListener implements ServletRequestListener {
    public void requestDestroyed(ServletRequestEvent evt) {
        System.out.println("销毁request对象");
    }

    public void requestInitialized(ServletRequestEvent evt) {
        System.out.println("创建request对象");
    }
```

```
}
```

```
<listener>
  <listener-class>cn.itcast.listener.MyServletContextListener</listener-class>
</listener>
<listener>
  <listener-class>cn.itcast.listener.MyHttpSessionListener</listener-class>
</listener>
<listener>
  <listener-class>cn.itcast.listener.MyServletRequestListener</listener-class>
</listener>
<session-config>
 <session-timeout>1</session-timeout>
</session-config>
```

3 操作域属性的监听器

当对域属性进行增、删、改时，执行的监听器一共有三个：

- ServletContextAttributeListener：在 ServletContext 域进行增、删、改属性时调用下面方法。
  - public void attributeAdded(ServletContextAttributeEvent evt)
  - public void attributeRemoved(ServletContextAttributeEvent evt)
  - public void attributeReplaced(ServletContextAttributeEvent evt)
- HttpSessionAttributeListener：在 HttpSession 域进行增、删、改属性时调用下面方法
  - public void attributeAdded(HttpSessionBindingEvent evt)
  - public void attributeRemoved (HttpSessionBindingEvent evt)
  - public void attributeReplaced (HttpSessionBindingEvent evt)
- ServletRequestAttributeListener：在 ServletRequest 域进行增、删、改属性时调用下面方法
  - public void attributeAdded(ServletRequestAttributeEvent evt)
  - public void attributeRemoved (ServletRequestAttributeEvent evt)
  - public void attributeReplaced (ServletRequestAttributeEvent evt)

下面对这三个监听器的事件对象功能进行介绍：

- ServletContextAttributeEvent
  - String getName()：获取当前操作的属性名；
  - Object getValue()：获取当前操作的属性值；
  - ServletContext getServletContext()：获取 ServletContext 对象。
- HttpSessionBindingEvent
  - String getName()：获取当前操作的属性名；
  - Object getValue()：获取当前操作的属性值；
  - HttpSession getSession()：获取当前操作的 session 对象。
- ServletRequestAttributeEvent
  - String getName()：获取当前操作的属性名；
  - Object getValue()：获取当前操作的属性值；

- ServletContext getServletContext()：获取 ServletContext 对象；
- ServletRequest getServletRequest()：获取当前操作的 ServletRequest 对象。

```
public class MyListener implements ServletContextAttributeListener,
        ServletRequestAttributeListener, HttpSessionAttributeListener {
    public void attributeAdded(HttpSessionBindingEvent evt) {
        System.out.println("向session中添加属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeRemoved(HttpSessionBindingEvent evt) {
        System.out.println("从session中移除属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeReplaced(HttpSessionBindingEvent evt) {
        System.out.println("修改session中的属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeAdded(ServletRequestAttributeEvent evt) {
        System.out.println("向request中添加属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeRemoved(ServletRequestAttributeEvent evt) {
        System.out.println("从request中移除属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeReplaced(ServletRequestAttributeEvent evt) {
        System.out.println("修改request中的属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeAdded(ServletContextAttributeEvent evt) {
        System.out.println("向context中添加属性：" + evt.getName() + "=" +
evt.getValue());
    }

    public void attributeRemoved(ServletContextAttributeEvent evt) {
        System.out.println("从context中移除属性：" + evt.getName() + "=" +
evt.getValue());
```

```
    }

    public void attributeReplaced(ServletContextAttributeEvent evt) {
        System.out.println("修改context中的属性: " + evt.getName() + "=" +
evt.getValue());
    }
}
```

```
public class ListenerServlet extends BaseServlet {
    public String contextOperation(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        ServletContext context = this.getServletContext();
        context.setAttribute("a", "a");
        context.setAttribute("a", "A");
        context.removeAttribute("a");
        return "/index.jsp";
    }

    ////////////////////////////////

    public String sessionOperation(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        HttpSession session = request.getSession();
        session.setAttribute("a", "a");
        session.setAttribute("a", "A");
        session.removeAttribute("a");
        return "/index.jsp";
    }

    ////////////////////////////////

    public String requestOperation(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        request.setAttribute("a", "a");
        request.setAttribute("a", "A");
        request.removeAttribute("a");
        return "/index.jsp";
    }
}
```

```
  <body>
    <a href="<c:url
```

```
value='/ListenerServlet?method=contextOperation'/>">SevletContext操作属性
</a>
    <br/>
    <a href="<c:url
value='/ListenerServlet?method=sessionOperation'/>">HttpSession操作属性</a>
    <br/>
    <a href="<c:url
value='/ListenerServlet?method=requestOperation'/>">ServletRequest操作属性
</a> |
  </body>
```

4 HttpSession 的监听器

还有两个与 HttpSession 相关的特殊的监听器，这两个监听器的特点如下：

- 不用在 web.xml 文件中部署；
- 这两个监听器不是给 session 添加，而是给 Bean 添加。即让 Bean 类实现监听器接口，然后再把 Bean 对象添加到 session 域中。

下面对这两个监听器介绍一下：

- HttpSessionBindingListener：当某个类实现了该接口后，可以感知本类对象添加到 session 中，以及感知从 session 中移除。例如让 Person 类实现 HttpSessionBindingListener 接口，那么当把 Person 对象添加到 session 中，或者把 Person 对象从 session 中移除时会调用下面两个方法：
  - public void valueBound(HttpSessionBindingEvent event)：当把监听器对象添加到 session 中会调用监听器对象的本方法；
  - public void valueUnbound(HttpSessionBindingEvent event)：当把监听器对象从 session 中移除时会调用监听器对象的本方法；

这里要注意，HttpSessionBindingListener 监听器的使用与前面介绍的都不相同，当该监听器对象添加到 session 中，或把该监听器对象从 session 移除时会调用监听器中的方法。并且无需在 web.xml 文件中部署这个监听器。

示例步骤：

- 编写 Person 类，让其实现 HttpSessionBindingListener 监听器接口；
- 编写 Servlet 类，一个方法向 session 中添加 Person 对象，另一个从 session 中移除 Person 对象；
- 在 index.jsp 中给出两个超链接，分别访问 Servlet 中的两个方法。

Pseron.java

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

    public Person(String name, int age, String sex) {
        super();
```

```
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    public Person() {
        super();
    }

    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";
    }

    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public String getSex() {
        return sex;
    }

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

    public void valueBound(HttpSessionBindingEvent evt) {
        System.out.println("把Person对象存放到session中: " + evt.getValue());
    }

    public void valueUnbound(HttpSessionBindingEvent evt) {
        System.out.println("从session中移除Pseron对象: " + evt.getValue());
```

```
    }
}
```

ListenerServlet.java

```
public class ListenerServlet extends BaseServlet {
    public String addPerson(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        Person p = new Person("zhangSan", 23, "male");
        request.getSession().setAttribute("person", p);
        return "/index.jsp";
    }

    public String removePerson(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.getSession().removeAttribute("person");
        return "/index.jsp";
    }
```

index.jsp

```
  <body>
    <a href="<c:url
value='/ListenerServlet?method=addPerson'/>">addPerson</a>
    <br/>
    <a href="<c:url
value='/ListenerServlet?method=removePerson'/>">removePerson</a>
    <br/>
  </body>
```

- HttpSessionActivationListener：Tomcat 会在 session 从时间不被使用时钝化 session 对象，所谓钝化 session，就是把 session 通过序列化的方式保存到硬盘文件中。当用户再使用 session 时，Tomcat 还会把钝化的对象再活化 session，所谓活化就是把硬盘文件中的 session 在反序列化回内存。当 session 被 Tomcat 钝化时，session 中存储的对象也被纯化，当 session 被活化时，也会把 session 中存储的对象活化。如果某个类实现了 HttpSessionActiveationListener 接口后，当对象随着 session 被钝化和活化时，下面两个方法就会被调用：
  - public void sessionWillPassivate(HttpSessionEvent se)：当对象感知被活化时调用本方法；
  - public void sessionDidActivate(HttpSessionEvent se)：当对象感知被钝化时调用本方法；

HttpSessionActivationListener 监听器与 HttpSessionBindingListener 监听器相似，都是感知型的监听器，例如让 Person 类实现了 HttpSessionActivationListener 监听器接口，并把 Person 对象添加到了 session 中后，当 Tomcat 钝化 session 时，同时也会钝化 session 中的 Person 对象，这时 Person 对象就会感知到自己被钝化了，其实就是调用 Person 对象的 sessionWillPassivate()方法。当用户再次使用

session 时，Tomcat 会活化 session，这时 Person 会感知到自己被活化，其实就是调用 Person 对象的 sessionDidActivate()方法。

注意，因为钝化和活化 session，其实就是使用序列化和反序列化技术把 session 从内存保存到硬盘，和把 session 从硬盘加载到内存。这说明如果 Person 类没有实现 Serializable 接口，那么当 session 钝化时就不会钝化 Person，而是把 Person 从 session 中移除再钝化！这也说明 session 活化后，session 中就不在有 Person 对象了。

示例步骤：

- 先不管 HttpSessionActivationListener 监听器接口，先来配置 Tomcat 钝化 session 的参数，把下面配置文件放到 tomcat\conf\catalina\localhost 目录下！文件名称为项目名称。

```
<Context>
    <Manager className="org.apache.catalina.session.PersistentManager"
maxIdleSwap="1">
        <Store className="org.apache.catalina.session.FileStore"
directory="mysession"/>
    </Manager>
</Context>
```

访问项目的 index.jsp 页面，这会使 Tomcat 创建 Session 对象，然后等待一分钟后，查看 Tomcat\work\Catalina\localhost\listener\mysession 目录下是否会产生文件，如果产生了，说明钝化 session 的配置成功了，可以开始下一步了。

- 创建 Person 类，让 Person 类实现 HttpSessionActivationListener 和 Serializable 接口：

Person.java

```
public class Person implements HttpSessionActivationListener, Serializable
{
    private String name;
    private int age;
    private String sex;

    public Person(String name, int age, String sex) {
        super();
        this.name = name;
        this.age = age;
        this.sex = sex;
    }

    public Person() {
        super();
    }

    public String toString() {
        return "Person [name=" + name + ", age=" + age + ", sex=" + sex + "]";
    }
```

```
    public String getName() {
        return name;
    }

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

    public int getAge() {
        return age;
    }

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

    public String getSex() {
        return sex;
    }

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

    public void sessionDidActivate(HttpSessionEvent evt) {
        System.out.println("session已经活化");
    }

    public void sessionWillPassivate(HttpSessionEvent evt) {
        System.out.println("session被钝化了！");
    }
}
```

- 与上例一样，编写 Servlet，提供两个方法：一个向 session 中添加 Person 对象，另一个从 session 中移除 Person 对象：

Person.java

```
public class ListenerServlet extends BaseServlet {
    public String addPerson(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        Person p = new Person("zhangSan", 23, "male");
        request.getSession().setAttribute("person", p);
        return "/index.jsp";
```

```
    }

    public String removePerson(HttpServletRequest request,
HttpServletResponse response)
            throws ServletException, IOException {
        request.getSession().removeAttribute("person");
        return "/index.jsp";
    }
}
```

- 在 index.jsp 页面中给出访问 addPerson()和 removePerson()的方法：

```
  <body>
    <a href="<c:url
value='/ListenerServlet?method=addPerson'/>">addPerson</a>
    <br/>
    <a href="<c:url
value='/ListenerServlet?method=removePerson'/>">removePerson</a>
    <br/>
  </body>
```

- 打开 index.jsp 页面，这时 Tomcat 会创建 session，必须在 1 分钟之前点击 addPerson 链接，这能保证在 session 被钝化之前把 Person 对象添加到 session 中；
- 等待一分钟，这时 session 会被钝化，也就会调用 Person 的 sessionWillPassivate()；
- 刷新一下 index.jsp 页面，这会使 session 活化，会调用 Person 的 sessionDidActivate()方法。

踢人

1 需求分析

当用户登录后，可以进入在线用户列表查看到当前在线的所有用户。用户可以点击“踢人”链接完成踢人操作。如果当前用户的 IP 不是 127.0.0.1，那么显示“您没有这个权限”，如果是那么返回到在线用户列表！

用户名	IP地址	登录时间	操作
ccc	127.0.0.1	2013-06-21 13:49:05	踢他
bbb	127.0.0.1	2013-06-21 13:49:15	踢他

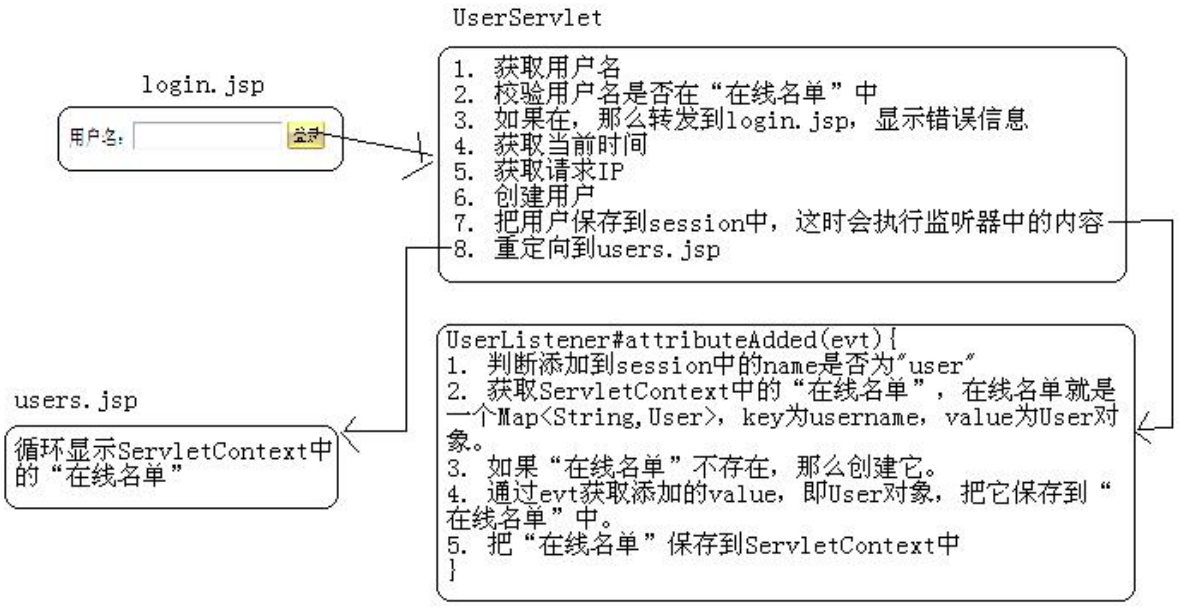
2 登录

因为要在页面中显示所有在线用户，那么就需要把用户保存起来。为了所有用户都可以看到在线用户名单，所以把在线用户名单保存到 ServletContext 中。在线用户名单保存到 Map 中，而 Map 保存到 ServletContext 中。

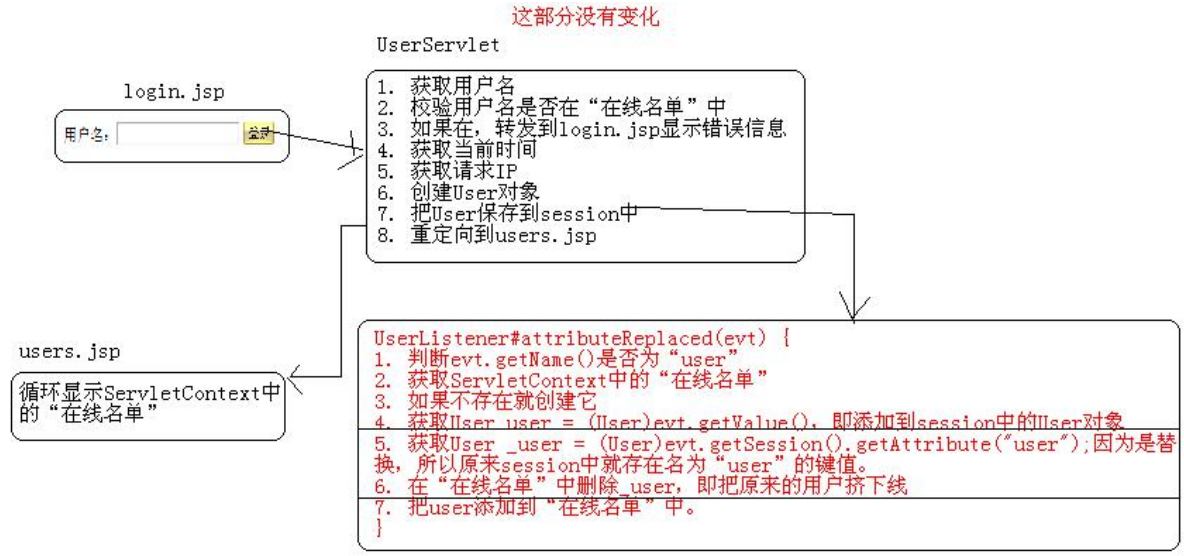
因为刚刚学生了监听器，我们现在可以尝试使用 HttpSessionAttributeListener。当用户登录成功后，会把 User 对象保存到 session 中，这时监听器的 attributeAdded()方法就会被调用。我们可以在这个方法中把登录的用户保存到在线用户名单中。

注意，如果用户在登录之后，又使用另一个用户名再次登录，这会把原来 session 中的 User 对象覆盖掉，但这会使在线用户名单中出现两个用户！所以我们还需要在 attributeReplaced()方法中把在线用户名单中的老用户删除，再添加新的用户。

用户第一次登录



session 中已经存在前一次登录的用户，再次登录



3 登录代码

index.jsp

```
<body>
  <p style="font-weight: 900; color:red;">${msg }</p>
<form action="<c:url value='/UserServlet'/>">
<input type="hidden" name="method" value="login"/>
    用户名：<input type="text" name="username"/>
    <input type="submit" value="登录"/>
</form>
  </body>
```

UserServlet#login()

```
    public String login(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        // 获取登录的用户名
        String username = request.getParameter("username");
        // 在ServletContext中获取"在线名单"
        Map<String,User> onlineUsers = (Map<String, User>)
this.getServletContext().getAttribute("onlineUsers");
        // 如果"在线名单"存在
        if(onlineUsers != null) {
            // 如果"在线名单"中已经存在这个用户名
            if(onlineUsers.containsKey(username)) {
                // 转发到登录页面，显示错误信息
                request.setAttribute("msg", "用户" + username + "已经登录！");
                return "/index.jsp";
            }
        }
        // 获取IP地址
        String ip = request.getRemoteAddr();
        // 获取登录时间
        String logintime = String.format("%tF %<tT", new java.util.Date());
        // 创建User对象
        User user = new User(username, ip, logintime);
        // 把User对象保存到session中，这会执行监听器的attributeAdded()或
attributeReplaced()方法
        request.getSession().setAttribute("user", user);

        // 重定向到users.jsp显示在线用户名单
        response.sendRedirect(request.getContextPath() + "/users.jsp");
        return null;
    }
```

UserListener

```
public class UserListener implements HttpSessionAttributeListener {
    public void attributeAdded(HttpSessionBindingEvent evt) {
        // 获取添加到session中的name是否为"user"
        if (evt.getName().equals("user")) {
            // 获取在线名单
            Map<String, User> onlineUsers = (Map<String, User>) evt.getSession()
                    .getServletContext().getAttribute("onlineUsers");
            // 如果当前登录的用户是第一个用户，那么这时还没有"在线名单"，所以要创建
            if(onlineUsers == null) {
                onlineUsers = new LinkedHashMap<String,User>();
            }
            // 获取添加到session中的value，即登录的User对象
            User user = (User) evt.getValue();
            // 把用户保存到在线名单中
            onlineUsers.put(user.getUsername(), user);
            // 把在线名单保存到ServletContext中
            evt.getSession().getServletContext().setAttribute("onlineUsers",
onlineUsers);
        }
    }

    public void attributeReplaced(HttpSessionBindingEvent evt) {
        // 获取添加到session中的name是否为"user"
        if (evt.getName().equals("user")) {
            // 获取在线名单
            Map<String, User> onlineUsers = (Map<String, User>) evt.getSession()
                    .getServletContext().getAttribute("onlineUsers");
            // 如果当前登录的用户是第一个用户，那么这时还没有"在线名单"，所以要创建
            if(onlineUsers == null) {
                onlineUsers = new LinkedHashMap<String,User>();
            }
            // 因为是替换session中的值时才会调用本方法，所以替换就会有老值和新值。
            // evt.getValue()获取的是老值，即被替换的值
            // 我们要把它从在线名单中移除
            User user = (User) evt.getValue();
            // 从在线名单中移除老值，即把上一用户从session中挤下线
            onlineUsers.remove(user.getUsername());

            // 获取session中保存的user，这是新值
            // 我们要把它保存到在线名单中
            user = (User) evt.getSession().getAttribute("user");
            onlineUsers.put(user.getUsername(), user);
```

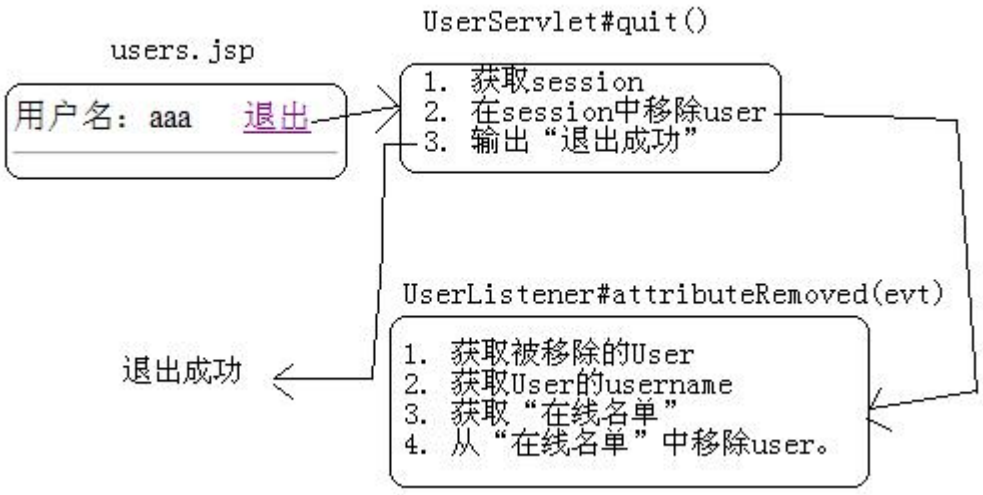
```
            // 把在线名单保存到ServletContext中
            evt.getSession().getServletContext().setAttribute("onlineUsers",
onlineUsers);
        }
    }

    public void attributeRemoved(HttpSessionBindingEvent evt) {}
}
```

users.jsp

```
  <body>
<c:choose>
    <c:when test="${empty sessionScope.user }">
        您还没有登录
    </c:when>
    <c:otherwise>

  用户名: ${sessionScope.user.username }
 <a href="<c:url value='/UserServlet?method=quit'/>">退出</a>
 <hr/>
    <table border="1" width="50%" align="center">
    <tr>
    <th>用户名</th>
    <th>IP地址</th>
    <th>登录时间</th>
    <th>操作</th>
    </tr>
<c:forEach items="${onlineUsers }" var="entry">
    <tr>
    <td>${entry.value.username }</td>
    <td>${entry.value.ip }</td>
    <td>${entry.value.logintime }</td>
    <td>
        <a href="<c:url
value='/UserServlet?method=kick&username=${entry.value.username }'/>">踢他
</a>
    </td>
    </tr>
</c:forEach>
    </table>
    </c:otherwise>
</c:choose>
  </body>
```

4 退出

当用户点击退出链接时，Servlet 中获取 session，然后从 session 中移除 User。这时监听器的 attributeRemoved()方法会被调用，在这个方法中，我们获取“在线名单”，从在线名单中移除当前用户。



UserServlet#quit()

```
    public String quit(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.getSession().removeAttribute("user");
        response.getWriter().print("退出成功！");
        return null;
    }
```

UserListener#attributeReplaced()

```
    public void attributeRemoved(HttpSessionBindingEvent evt) {
        User user = (User)evt.getValue();
        // 获取在线名单
        Map<String, User> onlineUsers = (Map<String, User>) evt.getSession()
                .getServletContext().getAttribute("onlineUsers");
        // 从在线名单中移除User
        if(onlineUsers != null) {
            onlineUsers.remove(user.getUsername());
        }
    }
```

5 踢人

在用户点击“踢人”链接时，Servlet 中可以获取到被踢用户的 username，我们需要使用 username

查找两个对象：

- 获取“在线名单”，然后从“在线名单”中删除当前被踢用户。这个没有问题；
- 获取被踢用户的 session，然后把 session 销毁。这就有问题了，因为我们一开始就没有保存过在线用户的 session！所以我们现在要修改一些代码，可以通过 username 得到当前用户的 session 对象。

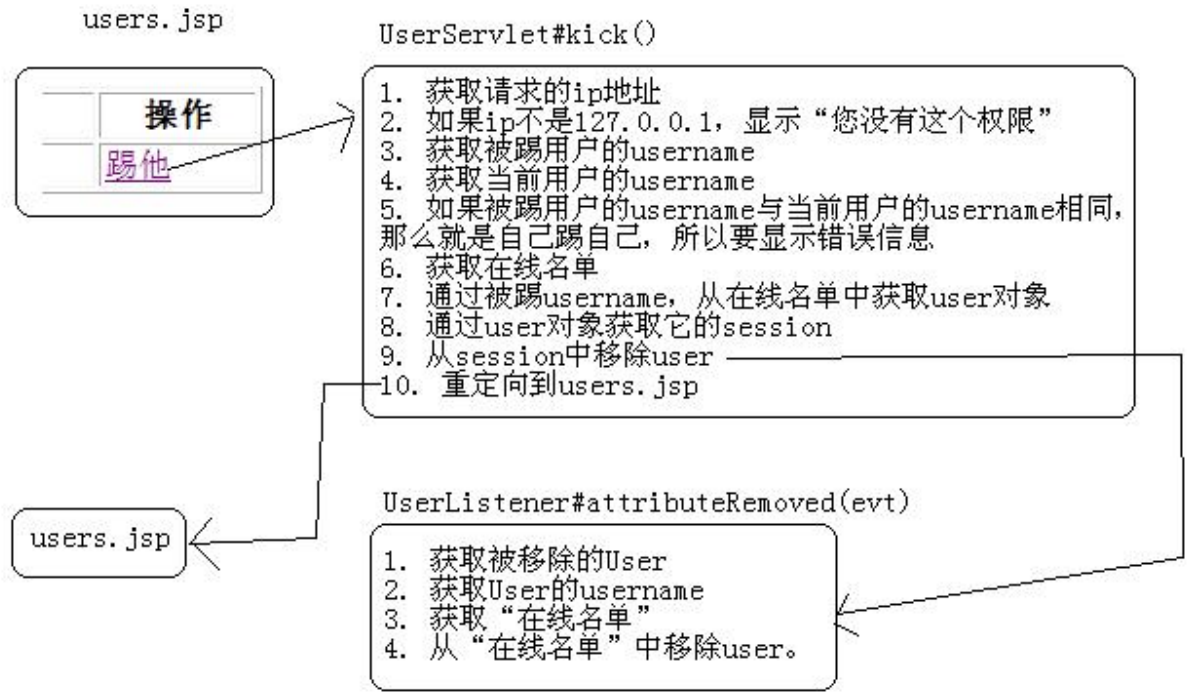
我们需要修改 User 对象，在 User 对象中设置 HttpSession 属性：

```
public class User {
    private String username;
    private String ip;
    private String logintime;
    private HttpSession session;

    public HttpSession getSession() {
        return session;
    }
    public void setSession(HttpSession session) {
        this.session = session;
    }
    ...
}
```

修改 UserServlet#login()方法，在用户登录成功后，需要把当前 session 保存到当前用户中。

```
    public String login(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        // 获取登录的用户名
        String username = request.getParameter("username");
        // 在ServletContext中获取“在线名单”
        Map<String, User> onlineUsers = (Map<String, User>) this
                .getServletContext().getAttribute("onlineUsers");
        // 如果“在线名单”存在
        if (onlineUsers != null) {
            // 如果“在线名单”中已经存在这个用户名
            if (onlineUsers.containsKey(username)) {
                // 转发到登录页面，显示错误信息
                request.setAttribute("msg", "用户" + username + "已经登录！");
                return "/index.jsp";
            }
        }
        // 获取IP地址
        String ip = request.getRemoteAddr();
        // 获取登录时间
        String logintime = String.format("%tF %<tT", new java.util.Date());
```

```
        // 创建User对象
        User user = new User(username, ip, logintime);
        // 把User对象保存到session中，这会执行监听器的attributeAdded()或
attributeReplaced()方法
        request.getSession().setAttribute("user", user);
        // 设置当前用户的session
        user.setSession(request.getSession());

        // 重定向到users.jsp显示在线用户名单
        response.sendRedirect(request.getContextPath() + "/users.jsp");
        return null;
    }
```



UserServlet.java

```
    public String kick(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String ip = request.getRemoteAddr();
        if (!ip.equals("127.0.0.1")) {
            response.getWriter().print("您没有踢人的权限！");
            return null;
        }
        String username = request.getParameter("username");
        User currUser = (User) request.getSession().getAttribute("user");
        // 如果当前用户就是被踢用户，那么就是自己踢自己
```

```
        if(currUser.getUsername().equals(username)) {
            response.getWriter().print("您是不是有病，为什么要踢自己呢！");
            return null;
        }

        // 从在线名单中获取被踢的用户
        Map<String, User> onlineUsers = (Map<String, User>) this
                .getServletContext().getAttribute("onlineUsers");
        User user = onlineUsers.get(username);
        // 获取当前用户的session，再从session中移除当前用户
        user.getSession().removeAttribute("user");
        // 重定向到users.jsp显示在线用户名单
        response.sendRedirect(request.getContextPath() + "/users.jsp");
        return null;
    }
```

国际化

1 什么是国际化

国际化就是可以把页面中的中文变成英文。例如在页面中的登录表单：

2 理解国际化

想把页面中的文字修改，那么就不能再使用硬编码，例如下面的页面中都是硬编码：

```
<body>
  <h1>登录</h1>
  <form action="" method="post">
      用户名: <input type="text" name="username"/><br/>
      密 码: <input type="password" name="password"/><br/>
      <input type="submit" value='登录'/>
  </form>
</body>
```

上图中的中文想转换成英文，那么就需要把它们都变成活编码：

```
<body>
  <h1><%=MessageUtils.getText("msg.login") %></h1>
  <form action="" method="post">
      <%=MessageUtils.getText("msg.username")%>: <input type="text" name="username"/><br/>
      <%=MessageUtils.getText("msg.password")%>: <input type="password" name="password"/><br/>
      <input type="submit" value='<%=MessageUtils.getText("msg.login")%>'/>
  </form>
</body>
```

3 Locale 类

创建 Locale 类对象：

- new Locale(“zh”, “CN”);
- new Locale(“en”, “US”);

你一定对 zh、CN 或是 en、US 并不陌生，zh、en 表示语言，而 CN、US 表示国家。一个 Locale 对象表示的就是语言和国家。

4 ResourceBundle 类

ReourceBundle 类用来获取配置文件中的内容。

下面是两个配置文件内容：

res_zh_CN.properties

name	value
msg.password	密码
msg.username	用户名

res_en_US.properties

name	value
msg.username	Username
msg.password	Password

```
public class Demo1 {
    @Test
    public void fun1() {
        ResourceBundle rb = ResourceBundle.getBundle("res", new Locale("zh",
"CN" ));
        String username = rb.getString("msg.username");
        String password = rb.getString("msg.password");
        System.out.println(username);
        System.out.println(password);
    }

    @Test
```

```
    public void fun2() {
        ResourceBundle rb = ResourceBundle.getBundle("res", new Locale("en",
"US" ));
        String username = rb.getString("msg.username");
        String password = rb.getString("msg.password");
        System.out.println(username);
        System.out.println(password);
    }
}
```

ResourceBundle 的 getBundle()方法需要两个参数：

- 第一个参数：配置文件的基本名称
- 第二个参数：Locale

getBundle()方法会通过两个参数来锁定配置文件！

5 页面国际化

其实页面国际化也是同一道理，只需要通过切换 Locale 来切换配置文件。我们写一个 MessageUtils 类，内部需要 ResourceBundle 的实例。

我们再写一个过滤器 MessageFilter，它会通过参数创建 Locale 对象，传递给 MessageUtils，然后在页面中使用 MessageUtils 来获取文本信息。

MessageUtils.java

```
public class MessageUtils {
    private static String baseName = "res";
    private static Locale locale;
    public static String getText(String key) {
        return ResourceBundle.getBundle(baseName, locale).getString(key);
    }
    public static Locale getLocale() {
        return locale;
    }
```

```
    public static void setLocale(Locale locale) {
        MessageUtils.locale = locale;
    }
}
```

MessageFilter.java

```
public class MessageFilter implements Filter {
    public void destroy() {
    }

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        String l = req.getParameter("request_locale");
        Locale locale = null;
        if(l != null && !l.isEmpty()) {
            String[] strs = l.split("_");
            locale = new Locale(strs[0], strs[1]);
            req.getSession().setAttribute("WW_TRANS_I18N_LOCALE", locale);
        } else {
            locale =
(Locale)req.getSession().getAttribute("WW_TRANS_I18N_LOCALE");
        }
        if(locale == null) {
            locale = req.getLocale();
        }
        MessageUtils.setLocale(locale);
        chain.doFilter(request, response);
    }

    public void init(FilterConfig fConfig) throws ServletException {
    }
}
```

login.jsp

```
  <body>
    <h1><%=MessageUtils.getText("msg.login") %></h1>
    <form action="<c:url value='/index.jsp'/>" method="post">
    <%=MessageUtils.getText("msg.username")%>: <input type="text"
name="username"/><br/>
    <%=MessageUtils.getText("msg.password")%>: <input type="password"
name="password"/><br/>
    <input type="submit" value='<%=MessageUtils.getText("msg.login")%>'/>
```

```
    </form>
  </body>
```

index.jsp

```
  <body>
    <p><%=MessageUtils.getText("hello") + ":"
+request.getParameter("username") %>: </p>
  </body>
```

6 NumberFormat

NumberFormat 类用来对数字进行格式化，我们只需要使用 String format(double)一个方法即可。下面是获取 NumberFormat 实例的方法：

- NumberFormat format = NumberFormat.getNumberFormat()
- NumberFormat format = NumberFormat.getNumberFormat(Locale)
- NumberFormat format = NumberFormat.getCurrentcyFormat()
- NumberFormat format = NumberFormat.getCurrentcyFormat(Locale)
- NumberFormat format = NumberFormat.getPercentFormat()
- NumberFormat format = NumberFormat.getPercentFormat(Locale)

7 DateFormat

DateFormat 类用来对日期进行格式化，我们只需要使用 String format(Date)一个方法即可。下面是获取 DateFormat 实例的方法：

- DateFormat format = DateFormat.getDateFormat();
- DateFormat format = DateFormat.getTimeFormat();
- DateFormat format = DateFormat.getDateTimeFormat();
- DateFormat format = DateFormat.getDateFormat(int style, Locale locale);
- DateFormat format = DateFormat.getTimeFormat(int style, Locale locale);
- DateFormat format = DateFormat.getDateTimeFormat(int style, Locale locale);

其中 style 是对日期的长、中、短，以及完整样式。

- SHORT；
- MEDIUM；
- LONG；
- FULL

8 MessageFormat

MessageFormat 可以把模式中的{N}使用参数来替换。我们把{N}称之为点位符。其中点位符中的 N 是从 0 开始的整数。

MessageFormat.format(String pattern, Object… params)，其中 pattern 中可以包含 0~n 个点位符，

而 params 表示对点位符的替换文本。注意，点位符需要从 0 开始。

```
String p = “{0}或{1}错误”;
String text = MessageFormat.format(p, “用户名”, “密码”);
System.out.println(text);//用户名或密码错误
```

day15

今日内容

- 过滤器

过滤器（Filter）

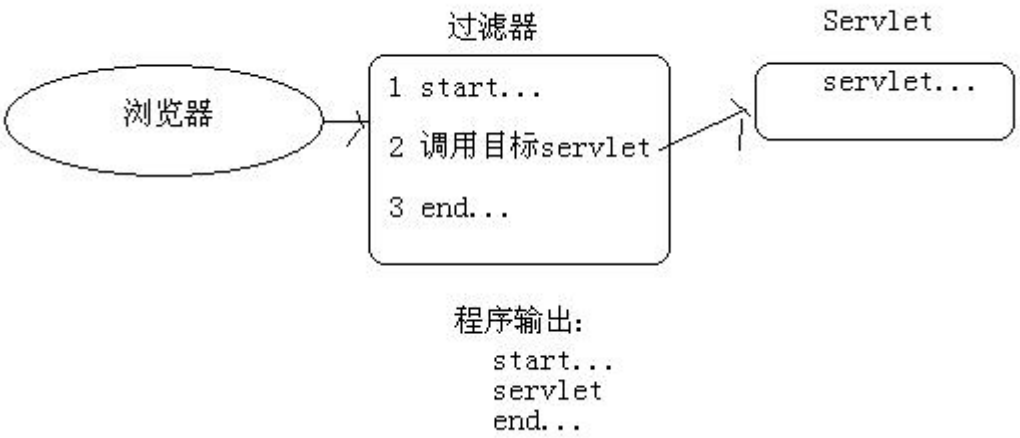
过滤器概述

1 什么是过滤器

过滤器 JavaWeb 三大组件之一，它与 Servlet 很相似！不它过滤器是用来拦截请求的，而不是处理请求的。

当用户请求某个 Servlet 时，会先执行部署在这个请求上的 Filter，如果 Filter“放行”，那么会继承执行用户请求的 Servlet；如果 Filter 不“放行”，那么就不会执行用户请求的 Servlet。

其实可以这样理解，当用户请求某个 Servlet 时，Tomcat 会去执行注册在这个请求上的 Filter，然后是否“放行”由 Filter 来决定。可以理解为，Filter 来决定是否调用 Servlet！当执行完成 Servlet 的代码后，还会执行 Filter 后面的代码。



2 过滤器之 hello world

其实过滤器与 Servlet 很相似，我们回忆一下如果写的第一个 Servlet 应用！写一个类，实现 Servlet 接口！没错，写过滤器就是写一个类，实现 Filter 接口。

```
public class HelloFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {}
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        System.out.println("Hello Filter");
    }
    public void destroy() {}
}
```

第二步也与 Servlet 一样，在 web.xml 文件中部署 Filter：

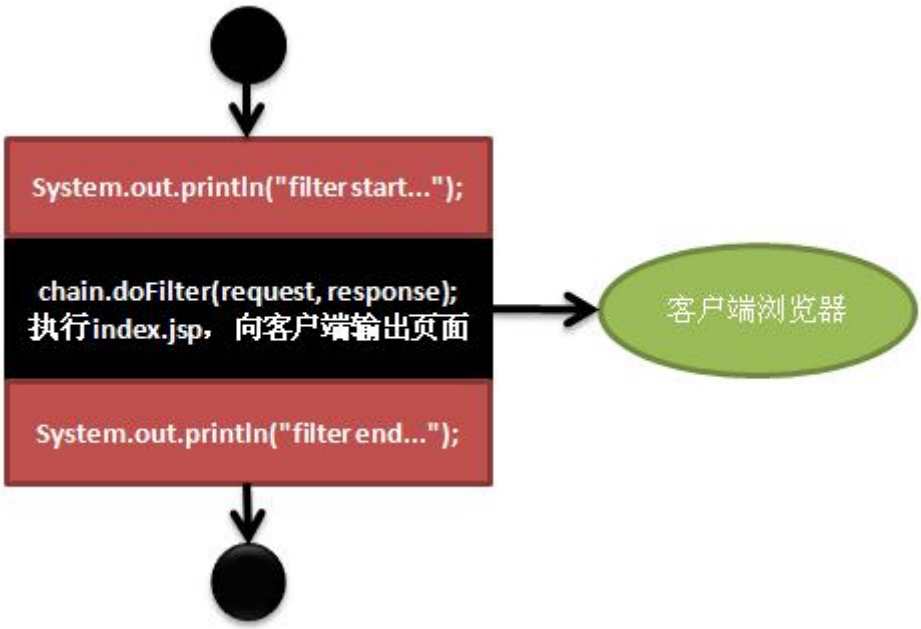
```
  <filter>
    <filter-name>helloFilter</filter-name>
    <filter-class>cn.itcast.filter.HelloFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>helloFilter</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>
```

应该没有问题吧，都可以看懂吧！

OK 了，现在可以尝试去访问 index.jsp 页面了，看看是什么效果！

当用户访问 index.jsp 页面时，会执行 HelloFilter 的 doFilter()方法！在我们的示例中，index.jsp 页面是不会被执行的，如果想执行 index.jsp 页面，那么我们需要放行！

```
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        System.out.println("filter start...");
        chain.doFilter(request, response);
        System.out.println("filter end...");
    }
```



有很多同学总是错误的认为，一个请求在给客户端输出之后就算是结束了，这是不对的！其实很多事情都需要在给客户端响应之后才能完成！

过滤器详细

1 过滤器的生命周期

我们已经学习过 Servlet 的生命周期，那么 Filter 的生命周期也就没有什么难度了！

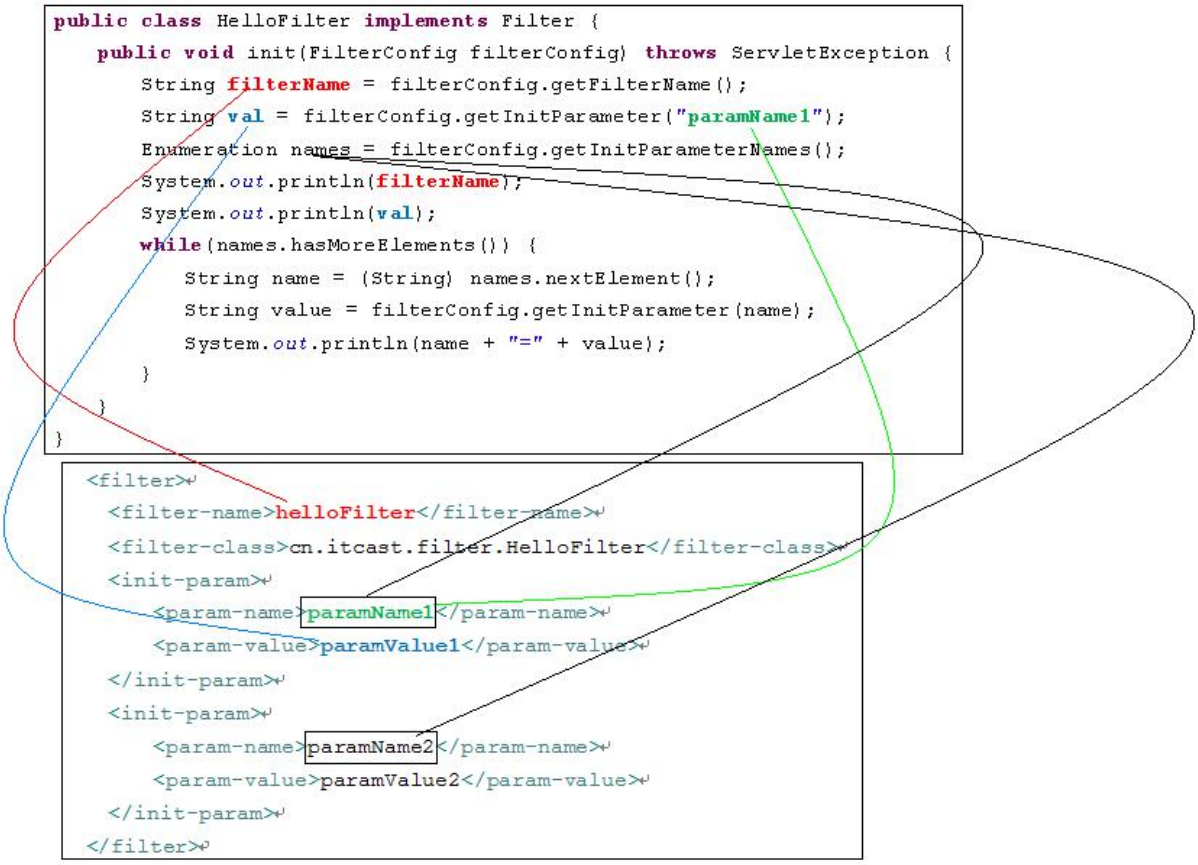
- init(FilterConfig)：在服务器启动时会创建 Filter 实例，并且每个类型的 Filter 只创建一个实例，从此不再创建！在创建完 Filter 实例后，会马上调用 init()方法完成初始化工作，这个方法只会被执行一次；
- doFilter(ServletRequest req,ServletResponse res,FilterChain chain)：这个方法会在用户每次访问“目标资源（<url->pattern>index.jsp</url-pattern>）”时执行，如果需要“放行”，那么需要调用 FilterChain 的 doFilter(ServletRequest,ServletResponse)方法，如果不调用 FilterChain 的 doFilter()方法，那么目标资源将无法执行；
- destroy()：服务器会在创建 Filter 对象之后，把 Filter 放到缓存中一直使用，通常不会销毁它。一般会在服务器关闭时销毁 Filter 对象，在销毁 Filter 对象之前，服务器会调用 Filter 对象的 destory()方法。

2 FilterConfig

你已经看到了吧，Filter 接口中的 init()方法的参数类型为 FilterConfig 类型。它的功能与 ServletConfig 相似，与 web.xml 文件中的配置信息对应。下面是 FilterConfig 的功能介绍：

- ServletContext getServletContext()：获取 ServletContext 的方法；

- String getFilterName()：获取 Filter 的配置名称；与<filter-name>元素对应；
- String getInitParameter(String name)：获取 Filter 的初始化配置，与<init-param>元素对应；
- Enumeration getInitParameterNames()：获取所有初始化参数的名称。



```
public class HelloFilter implements Filter {
    public void init(FilterConfig filterConfig) throws ServletException {
        String filterName = filterConfig.getFilterName();
        String val = filterConfig.getInitParameter("paramName1");
        Enumeration names = filterConfig.getInitParameterNames();
        System.out.println(filterName);
        System.out.println(val);
        while(names.hasMoreElements()) {
            String name = (String) names.nextElement();
            String value = filterConfig.getInitParameter(name);
            System.out.println(name + "=" + value);
        }
    }
}
```

```
<filter>
  <filter-name>helloFilter</filter-name>
  <filter-class>cn.itcast.filter.HelloFilter</filter-class>
  <init-param>
      <param-name>paramName1</param-name>
      <param-value>paramValue1</param-value>
  </init-param>
  <init-param>
      <param-name>paramName2</param-name>
      <param-value>paramValue2</param-value>
  </init-param>
</filter>
```

3 FilterChain

doFilter()方法的参数中有一个类型为 FilterChain 的参数，它只有一个方法：doFilter(ServletRequest,ServletResponse)。

前面我们说 doFilter()方法的放行，让请求流访问目标资源！但这么说不严密，其实调用该方法的意思是，“我（当前 Filter）”放行了，但不代表其他人（其他过滤器）也放行。

也就是说，一个目标资源上，可能部署了多个过滤器，就好比在你去北京的路上有多个打劫的匪人（过滤器），而其中第一伙匪人放行了，但不代表第二伙匪人也放行了，所以调用 FilterChain 类的 doFilter()方法表示的是执行下一个过滤器的 doFilter()方法，或者是执行目标资源！

如果当前过滤器是最后一个过滤器，那么调用 chain.doFilter()方法表示执行目标资源，而不是最后一个过滤器，那么 chain.doFilter()表示执行下一个过滤器的 doFilter()方法。

4 多个过滤器执行顺序

一个目标资源可以指定多个过滤器，过滤器的执行顺序是在 web.xml 文件中的部署顺序：

```
<filter>
  <filter-name>myFilter1</filter-name>
```

```
    <filter-class>cn.itcast.filter.MyFilter1</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>myFilter1</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>
  <filter>
    <filter-name>myFilter2</filter-name>
    <filter-class>cn.itcast.filter.MyFilter2</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>myFilter2</filter-name>
    <url-pattern>/index.jsp</url-pattern>
  </filter-mapping>
```

```
public class MyFilter1 extends HttpFilter {
    public void doFilter(HttpServletRequest request, HttpServletResponse
response,
            FilterChain chain) throws IOException, ServletException {
        System.out.println("filter1 start...");
        chain.doFilter(request, response);//放行，执行MyFilter2的doFilter()方法
        System.out.println("filter1 end...");
    }
}
```

```
public class MyFilter2 extends HttpFilter {
    public void doFilter(HttpServletRequest request, HttpServletResponse
response,
            FilterChain chain) throws IOException, ServletException {
        System.out.println("filter2 start...");
        chain.doFilter(request, response);//放行，执行目标资源
        System.out.println("filter2 end...");
    }
}
```

```
  <body>
    This is my JSP page. <br>
    <h1>index.jsp</h1>
    <%System.out.println("index.jsp"); %>
  </body>
```

当有用户访问 index.jsp 页面时，输出结果如下：

```
filter1 start...
filter2 start...
index.jsp
filter2 end...
```

```
filter1 end...
```

5 四种拦截方式

我们来做个测试，写一个过滤器，指定过滤的资源为 b.jsp，然后我们在浏览器中直接访问 b.jsp，你会发现过滤器执行了！

但是，当我们在 a.jsp 中 request.getRequestDispathcer("/b.jsp").forward(request,response)时，就不会再执行过滤器了！也就是说，默认情况下，只能直接访问目标资源才会执行过滤器，而 forward 执行目标资源，不会执行过滤器！

```
public class MyFilter extends HttpFilter {
    public void doFilter(HttpServletRequest request,
            HttpServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("myfilter...");
        chain.doFilter(request, response);
    }
}
```

```
    <filter>
        <filter-name>myfilter</filter-name>
        <filter-class>cn.itcast.filter.MyFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>myfilter</filter-name>
        <url-pattern>/b.jsp</url-pattern>
    </filter-mapping>
```

```
  <body>
   <h1>b.jsp</h1>
  </body>
```

```
  <h1>a.jsp</h1>
    <%
    request.getRequestDispatcher("/b.jsp").forward(request, response);
    %>
  </body>
```

http://localhost:8080/filtertest/b.jsp -->直接访问 b.jsp 时，会执行过滤器内容；

http://localhost:8080/filtertest/a.jsp --> 访问 a.jsp，但 a.jsp 会 forward 到 b.jsp，这时就不会执行过滤器！

其实过滤器有四种拦截方式！分别是：REQUEST、FORWARD、INCLUDE、ERROR。

- REQUEST：直接访问目标资源时执行过滤器。包括：在地址栏中直接访问、表单提交、超链接、重定向，只要在地址栏中可以看到目标资源的路径，就是 REQUEST；
- FORWARD：转发访问执行过滤器。包括 RequestDispatcher#forward()方法、<jsp:forward>标签都是转发访问；

- INCLUDE：包含访问执行过滤器。包括 RequestDispatcher#include()方法、<jsp:include>标签都是包含访问；
- ERROR：当目标资源在 web.xml 中配置为<error-page>中时，并且真的出现了异常，转发到目标资源时，会执行过滤器。

可以在<filter-mapping>中添加 0~n 个<dispatcher>子元素，来说明当前访问的拦截方式。

```
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <url-pattern>/b.jsp</url-pattern>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

```
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <url-pattern>/b.jsp</url-pattern>
</filter-mapping>
```

```
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <url-pattern>/b.jsp</url-pattern>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

其实最为常用的就是 REQUEST 和 FORWARD 两种拦截方式，而 INCLUDE 和 ERROR 都比较少用！其中 INCLUDE 比较好理解，我们这里不再给出代码，学员可以通过 FORWARD 方式修改，来自己测试。而 ERROR 方式不易理解，下面给出 ERROR 拦截方式的例子：

```
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <url-pattern>/b.jsp</url-pattern>
    <dispatcher>ERROR</dispatcher>
</filter-mapping>
<error-page>
    <error-code>500</error-code>
    <location>/b.jsp</location>
</error-page>
```

```
<body>
<h1>a.jsp</h1>
 <%
 if(true)
  throw new RuntimeException("嘻嘻~");
 %>
</body>
```

6 过滤器的应用场景

过滤器的应用场景：

- 执行目标资源之前做预处理工作，例如设置编码，这种试通常都会放行，只是在目标资源执行之前做一些准备工作；
- 通过条件判断是否放行，例如校验当前用户是否已经登录，或者用户 IP 是否已经被禁用；
- 在目标资源执行后，做一些后续的特殊处理工作，例如把目标资源输出的数据进行处理；

7 设置目标资源

在 web.xml 文件中部署 Filter 时，可以通过“*”来执行目标资源：

```
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

这一特性与 Servlet 完全相同！通过这一特性，我们可以在用户访问敏感资源时，执行过滤器，例如：<url-pattern>/admin/*<url-pattern>，可以把所有管理员才能访问的资源放到/admin 路径下，这时可以通过过滤器来校验用户身份。

还可以为<filter-mapping>指定目标资源为某个 Servlet，例如：

```
<servlet>
    <servlet-name>myservlet</servlet-name>
    <servlet-class>cn.itcast.servlet.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>myservlet</servlet-name>
    <url-pattern>/abc</url-pattern>
</servlet-mapping>
<filter>
    <filter-name>myfilter</filter-name>
    <filter-class>cn.itcast.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>myfilter</filter-name>
    <servlet-name>myservlet</servlet-name>
</filter-mapping>
```

当用户访问 http://localhost:8080/filtertest/abc 时，会执行名字为 myservlet 的 Servlet，这时会执行过滤器。

8 Filter 小结

Filter 的三个方法：

- void init(FilterConfig)：在 Tomcat 启动时被调用；
- void destroy()：在 Tomcat 关闭时被调用；
- void doFilter(ServletRequest,ServletResponse,FilterChain)：每次有请求时都调用该方法；

FilterConfig 类：与 ServletConfig 相似，用来获取 Filter 的初始化参数

- ServletContext getServletContext()：获取 ServletContext 的方法；
- String getFilterName()：获取 Filter 的配置名称；
- String getInitParameter(String name)：获取 Filter 的初始化配置，与<init-param>元素对应；
- Enumeration getInitParameterNames()：获取所有初始化参数的名称。

FilterChain 类：

- void doFilter(ServletRequest,ServletResponse)：放行！表示执行下一个过滤器，或者执行目标资源。可以在调用 FilterChain 的 doFilter()方法的前后添加语句，在 FilterChain 的 doFilter()方法之前的语句会在目标资源执行之前执行，在 FilterChain 的 doFilter()方法之后的语句会在目标资源执行之后执行。

四各拦截方式：REQUEST、FORWARD、INCLUDE、ERROR，默认是 REQUEST 方式。

- REQUEST：拦截直接请求方式；
- FORWARD：拦截请求转发方式；
- INCLUDE：拦截请求包含方式；
- ERROR：拦截错误转发方式。

过滤器应用案例

分 ip 统计网站的访问次数

1 说明

网站统计每个 IP 地址访问本网站的次数。

2 分析

因为一个网站可能有多个页面，无论哪个页面被访问，都要统计访问次数，所以使用过滤器最为方便。

因为需要分 IP 统计，所以可以在过滤器中创建一个 Map，使用 IP 为 key，访问次数为 value。当有用户访问时，获取请求的 IP，如果 IP 在 Map 中存在，说明以前访问过，那么在访问次数上加 1，即可；IP 在 Map 中不存在，那么设置次数为 1。

把这个 Map 存放到 ServletContext 中！

3 代码

index.jsp

```
  <body>
<h1>分IP统计访问次数</h1>
<table align="center" width="50%" border="1">
    <tr>
        <th>IP地址</th>
        <th>次数</th>
    </tr>
<c:forEach items="${applicationScope.ipCountMap }" var="entry">
    <tr>
        <td>${entry.key }</td>
        <td>${entry.value }</td>
    </tr>
</c:forEach>
</table>
  </body>
```

IPFilter

```
public class IPFilter implements Filter {
    private ServletContext context;

    public void init(FilterConfig fConfig) throws ServletException {
        context = fConfig.getServletContext();
        Map<String, Integer> ipCountMap = Collections
                .synchronizedMap(new LinkedHashMap<String, Integer>());
        context.setAttribute("ipCountMap", ipCountMap);
    }

    @SuppressWarnings("unchecked")
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        String ip = req.getRemoteAddr();

        Map<String, Integer> ipCountMap = (Map<String, Integer>) context
                .getAttribute("ipCountMap");

        Integer count = ipCountMap.get(ip);
        if (count == null) {
            count = 1;
```

```
        } else {
            count += 1;
        }
        ipCountMap.put(ip, count);

        context.setAttribute("ipCountMap", ipCountMap);
        chain.doFilter(request, response);
    }

    public void destroy() {}
}
```

```
  <filter>
    <display-name>IPFilter</display-name>
    <filter-name>IPFilter</filter-name>
    <filter-class>cn.itcast.filter.ip.IPFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>IPFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

粗粒度权限控制（拦截是否登录、拦截用户名 admin 权限）

rbac(基于角色的权限管理)

1 说明

我们给出三个页面：index.jsp、user.jsp、admin.jsp。

- index.jsp：谁都可以访问，没有限制；
- user.jsp：只有登录用户才能访问；
- admin.jsp：只有管理员才能访问。

2 分析

设计 User 类：username、password、grade，其中 grade 表示用户等级，1 表示普通用户，2 表示管理员用户。

当用户登录成功后，把 user 保存到 session 中。

创建 LoginFilter，它有两种过滤方式：

- 如果访问的是 user.jsp，查看 session 中是否存在 user；
- 如果访问的是 admin.jsp，查看 session 中是否存在 user，并且 user 的 grade 等于 2。

3 代码

User.java

```
public class User {
    private String username;
    private String password;
    private int grade;
    …
}
```

为了方便，这里就不使用数据库了，所以我们需要在 UserService 中创建一个 Map，用来保存所有用户。Map 中的 key 中用户名，value 为 User 对象。

UserService.java

```
public class UserService {
    private static Map<String,User> users = new HashMap<String,User>();
    static {
        users.put("zhangSan", new User("zhangSan", "123", 1));
        users.put("liSi", new User("liSi", "123", 2));
    }

    public User login(String username, String password) {
        User user = users.get(username);
        if(user == null) return null;
        return user.getPassword().equals(password) ? user : null;
    }
}
```

login.jsp

```
  <body>
  <h1>登录</h1>
    <p style="font-weight: 900; color: red">${msg }</p>
    <form action="<c:url value='/LoginServlet'/>" method="post">
    用户名：<input type="text" name="username"/><br/>
    密　码：<input type="password" name="password"/><br/>
    <input type="submit" value="登录"/>
    </form>
  </body>
```

index.jsp

```
  <body>
    <h1>主页</h1>
    <h3>${user.username }</h3>
    <hr/>
```

```
    <a href="<c:url value='/login.jsp'/>">登录</a><br/>
    <a href="<c:url value='/user/user.jsp'/>">用户页面</a><br/>
    <a href="<c:url value='/admin/admin.jsp'/>">管理员页面</a>
  </body>
```

/user/user.jsp

```
<body>
  <h1>用户页面</h1>
  <h3>${user.username }</h3>
  <hr/>
</body>
```

/admin/admin.jsp

```
<body>
  <h1>管理员页面</h1>
  <h3>${user.username }</h3>
  <hr/>
</body>
```

LoginServlet

```
public class LoginServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        String username = request.getParameter("username");
        String password = request.getParameter("password");
        UserService userService = new UserService();
        User user = userService.login(username, password);
        if(user == null) {
            request.setAttribute("msg", "用户名或密码错误");
            request.getRequestDispatcher("/login.jsp").forward(request,
response);
        } else {
            request.getSession().setAttribute("user", user);
            request.getRequestDispatcher("/index.jsp").forward(request,
response);
        }
    }
}
```

LoginUserFilter.java

```
  <filter>
    <display-name>LoginUserFilter</display-name>
    <filter-name>LoginUserFilter</filter-name>
    <filter-class>cn.itcast.filter.LoginUserFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>LoginUserFilter</filter-name>
    <url-pattern>/user/*</url-pattern>
  </filter-mapping>
```

```
public class LoginUserFilter implements Filter {
    public void destroy() {}
    public void init(FilterConfig fConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        response.setContentType("text/html;charset=utf-8");
        HttpServletRequest req = (HttpServletRequest) request;
        User user = (User) req.getSession().getAttribute("user");
        if(user == null) {
            response.getWriter().print("您还没有登录");
            return;
        }
        chain.doFilter(request, response);
    }
}
```

LoginAdminFilter.java

```
  <filter>
    <display-name>LoginAdminFilter</display-name>
    <filter-name>LoginAdminFilter</filter-name>
    <filter-class>cn.itcast.filter.LoginAdminFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>LoginAdminFilter</filter-name>
    <url-pattern>/admin/*</url-pattern>
  </filter-mapping>
```

```
public class LoginAdminFilter implements Filter {
    public void destroy() {}
    public void init(FilterConfig fConfig) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
```

```
        response.setContentType("text/html;charset=utf-8");
        HttpServletRequest req = (HttpServletRequest) request;
        User user = (User) req.getSession().getAttribute("user");
        if(user == null) {
            response.getWriter().print("您还没有登录!");
            return;
        }
        if(user.getGrade() < 2) {
            response.getWriter().print("您的等级不够! ");
            return;
        }
        chain.doFilter(request, response);
    }
}
```

禁用资源缓存

浏览器只是要缓存页面，这对我们在开发时测试很不方便，所以我们可以过滤所有资源，然后添加去除所有缓存!

```
public class NoCacheFilter extends HttpFilter {
    public void doFilter(HttpServletRequest request,
            HttpServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        response.setHeader("cache-control", "no-cache");
        response.setHeader("pragma", "no-cache");
        response.setHeader("expires", "0");
        chain.doFilter(request, response);
    }
}
```

但是要注意，有的浏览器可能不会理会你的设置，还是会缓存的！这时就要在页面中使用时间戳来处理了。

解决全站字符乱码（POST 和 GET 中文编码问题）

1 说明

乱码问题：

- 获取请求参数中的乱码问题：
  - POST 请求：request.setCharacterEncoding(“utf-8”)；
  - GET 请求：new String(request.getParameter(“xxx”).getBytes(“iso-8859-1”), “utf-8”)；
- 响应的乱码问题：response.setContextType(“text/html;charset=utf-8”)。

基本上在每个 Servlet 中都要处理乱码问题，所以应该把这个工作放到过滤器中来完成。

2 分析

其实全站乱码问题的难点就是处理 GET 请求参数的问题。

如果只是处理 POST 请求的编码问题，以及响应编码问题，那么这个过滤器就太！太！太简单的。

```
public class EncodingFilter extends HttpFilter {
    public void doFilter(HttpServletRequest request,
            HttpServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        String charset = this.getInitParameter("charset");
        if(charset == null || charset.isEmpty()) {
            charset = "UTF-8";
        }
        request.setCharacterEncoding(charset);
        response.setContentType("text/html;charset=" + charset);
        chain.doFilter(request, response);
    }
}
```

如果是 POST 请求，当执行目标 Servlet 时，Servlet 中调用 request.getParameter()方法时，就会根据 request.setCharacterEncoding() 设置的编码来转码！这说明在过滤器中调用 request.setCharacterEncoding()方法会影响在目标 Servlet 中的 request.getParameter()方法的行为！

但是如果是 GET 请求，我们又如何能影响 request.getParameter()方法的行为呢？这是不好做到的！我们不可能先调用 request.getParameter()方法获取参数，然后手动转码后，再施加在到 request 中！因为 request 只有 getParameter()，而没有 setParameter()方法。

处理 GET 请求参数编码问题，需要在 Filter 中放行时，把 request 对象给“调包”了，也就是让目标 Servlet 使用我们“调包”之后的 request 对象。这说明我们需要保证“调包”之后的 request 对象中所有方法都要与“调包”之前一样可以使用，并且 getParameter()方法还要有能力返回转码之后的参数。

这可能让你想起了“继承”，但是这里不能用继承，而是“装饰者模式（Decorator Pattern）”！

下面是三种对 a 对象进行增强的手段：

- 继承：AA 类继承 a 对象的类型：A 类，然后重写 fun1()方法，其中重写的 fun1()方法就是被增强的方法。但是，继承必须要知道 a 对象的真实类型，然后才能去继承。如果我们不知道 a 对象的确切类型，而只知道 a 对象是 IA 接口的实现类对象，那么就无法使用继承来增强 a 对象了；

- 装饰者模式：AA 类去实现 a 对象相同的接口：IA 接口，还需要给 AA 类传递 a 对象，然后在 AA 类中所有的方法实现都是通过代理 a 对象的相同方法完成的，只有 fun1()方法在代理 a 对象相同方法的前后添加了一些内容，这就是对 fun1()方法进行了增强；
- 动态代理：动态代理与装饰者模式比较相似，而且是通过反射来完成的。动态代理会在最后一天的基础加强中讲解，这里就不再废话了。

对 request 对象进行增强的条件，刚好符合装饰者模式的特点！因为我们不知道 request 对象的具体类型，但我们知道 request 是 HttpServletRequest 接口的实现类。这说明我们写一个类 EncodingRequest，去实现 HttpServletRequest 接口，然后再把原来的 request 传递给 EncodingRequest 类!在 EncodingRequest 中对 HttpServletRequest 接口中的所有方法的实现都是通过代理原来的 request 对象来完成的，只有对 getParameter()方法添加了增强代码！

JavaEE 已经给我们提供了一个 HttpServletRequestWrapper 类，它就是 HttpServletRequest 的包装类，但它做任何的增强！你可能会说，写一个装饰类，但不做增强，其目的是什么呢？使用这个装饰类的对象，和使用原有的 request 有什么分别呢？

HttpServletRequestWrapper 类虽然是 HttpServletRequest 的装饰类，但它不是用来直接使用的，而是用来让我们去继承的！当我们想写一个装饰类时，还要对所有不需要增强的方法做一次实现是很心烦的事情，但如果你去继承 HttpServletRequestWrapper 类，那么就只需要重写需要增强的方法即可了。

3 代码

EncodingRequest

```
public class EncodingRequest extends HttpServletRequestWrapper {
    private String charset;
    public EncodingRequest(HttpServletRequest request, String charset) {
        super(request);
        this.charset = charset;
    }

    public String getParameter(String name) {
        HttpServletRequest request = (HttpServletRequest) getRequest();

        String method = request.getMethod();
        if(method.equalsIgnoreCase("post")) {
            try {
                request.setCharacterEncoding(charset);
            } catch (UnsupportedEncodingException e) {}
        } else if(method.equalsIgnoreCase("get")) {
            String value = request.getParameter(name);
            try {
                value = new String(name.getBytes("ISO-8859-1"), charset);
            } catch (UnsupportedEncodingException e) {
```

```
            }
            return value;
        }
        return request.getParameter(name);
    }
}
```

EncodingFilter

```
public class EncodingFilter extends HttpFilter {
    public void doFilter(HttpServletRequest request,
            HttpServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        String charset = this.getInitParameter("charset");
        if(charset == null || charset.isEmpty()) {
            charset = "UTF-8";
        }
        response.setCharacterEncoding(charset);
        response.setContentType("text/html;charset=" + charset);
        EncodingRequest res = new EncodingRequest(request, charset);
        chain.doFilter(res, response);
    }
}
```

web.xml

```
  <filter>
    <filter-name>EncodingFilter</filter-name>
    <filter-class>cn.itcast.filter.EncodingFilter</filter-class>
    <init-param>
        <param-name>charset</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
  </filter>
  <filter-mapping>
    <filter-name>EncodingFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
```

页面静态化

1 说明

你到“当当”搜索最多的是什么分类，没错，就是 Java 分类！你猜猜，你去搜索 Java 分类时，“当当”会不会去查询数据库呢？当然会了，不查询数据库怎么获取 Java 分类下的图书呢！其实每天都有很多人去搜索“Java 分类”的图书，每次都去访问数据库，这会有性能上的缺失！如果是在访问静态页面（html）那么就会快的多了！静态页面本身就比动态页面快很多倍，而且动态页面总是要去数据库查询，这会更加降低速度！

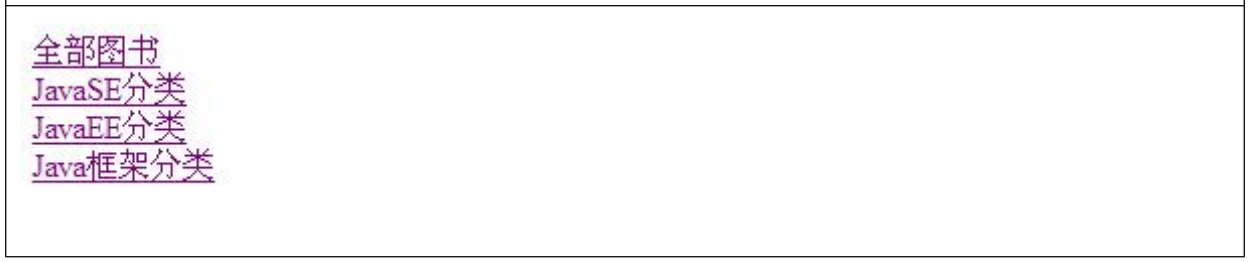
页面静态化是把动态页面生成的 html 保存到服务器的文件上，然后再有相同请求时，不再去执行动态页面，而是直接给用户响应上次已经生成的静态页面。而且静态页面还有助与搜索引擎找到你！

2 查看图书分类

我们先来写一个小例子，用来查看不同分类的图书。然后我们再去思考如何让动态页面静态化的问题。

index.jsp

```
  <body>
<a href="<c:url value='/BookServlet'/>">全部图书</a><br/>
<a href="<c:url value='/BookServlet?category=1'/>">JavaSE分类</a><br/>
<a href="<c:url value='/BookServlet?category=2'/>">JavaEE分类</a><br/>
<a href="<c:url value='/BookServlet?category=3'/>">Java框架分类</a><br/>
  </body>
```



BookServlet.java

```
public class BookServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        BookService bookService = new BookService();
        List<Book> bookList = null;
        String param = request.getParameter("category");
        if(param == null || param.isEmpty()) {
            bookList = bookService.findAll();
        } else {
            int category = Integer.parseInt(param);
            bookList = bookService.findByCategory(category);
        }
```

```
        request.setAttribute("bookList", bookList);
        request.getRequestDispatcher("/show.jsp").forward(request, response);
    }
}
```

show.jsp

```
<table border="1" align="center" width="50%">
    <tr>
        <th>图书名称</th>
        <th>图书单价</th>
        <th>图书分类</th>
    </tr>

  <c:forEach items="${bookList }" var="book">
    <tr>
        <td>${book.bname }</td>
        <td>${book.price }</td>
        <td>
            <c:choose>
                <c:when test="${book.category eq 1}"><p
style="color:red;">JavaSE分类</p></c:when>
                <c:when test="${book.category eq 2}"><p
style="color:blue;">JavaEE分类</p></c:when>
                <c:when test="${book.category eq 3}"><p
style="color:green;">Java框架分类</p></c:when>
            </c:choose>
        </td>
    </tr>
  </c:forEach>
</table>
```

图书名称	图书单价	图书分类
JavaSE_15	25.0	JavaSE分类
JavaSE_5	15.0	JavaSE分类
JavaSE_10	20.0	JavaSE分类
JavaSE_0	10.0	JavaSE分类
JavaSE_13	23.0	JavaSE分类
JavaSE_18	28.0	JavaSE分类
JavaSE_8	18.0	JavaSE分类
JavaEE_1	101.0	JavaEE分类
JavaEE_8	108.0	JavaEE分类
JavaSE_7	17.0	JavaSE分类
JavaSE_19	29.0	JavaSE分类
JavaSE_3	13.0	JavaSE分类
Java框架0	1000.0	Java框架分类

3 分析

我们的目标是在用户第一次请求动态页面（包括 Servlet）时生成静态页面，然后下次用户再访问相同的动态页面时，会重定向到上次生成的静态页面上。

这说明过滤器中需要做如下工作：

1. 获取请求的 URI；
2. 判断 URI 是否为静态页面，如果是那么直接放行；
3. 如果是动态页面还要获取请求参数，例如：/filterstatic/BookServlet?category=1；
4. 把“?”使用“_”替换，再把项目名称去除，再把“/”用“_”替换，再添加“.html”后缀，然后再添加/static/前缀：/static/_BookServlet_category=1.html。然后获取其真实路径，判断这个 html 是否存在；
5. 如果 html 存在，那么直接重定向到这个 html；
6. 如果不存在，把 response“调包”，然后放行；“调包”response 的目的是让动态页面的所有响应数据无法发送到客户端浏览器，而是写到一个字节数组中；
7. 把字节数组中的数据写入到静态页面中；
8. 再重定向到静态页面。

4 代码

想看懂代码需要：

- 理解动态页面向客户端响应文本数据，需要 response.getWriter()方法返回的流来完成；
- 理解装饰者模式；
- 对 ByteArrayOutputStream 类了解。

StaticFilter.java

```
public class StaticFilter implements Filter {
    private ServletContext sc;

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        response.setContentType("text/html;charset=utf-8");
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        String uri = req.getRequestURI().replace(req.getContextPath(), "");

        if(uri.endsWith(".html") || uri.endsWith(".htm")) {
            chain.doFilter(request, response);
            return;
        }

        String queryString = req.getQueryString();
        if(queryString != null) {
            uri += "_" + queryString + ".html";
        } else {
            uri += ".html";
        }
        uri = uri.replace("/", "_");
        String htmlDir = sc.getRealPath("/static");
        File file = new File(htmlDir, uri);
        if(file.exists()) {
            res.sendRedirect(req.getContextPath() + "/static/" + uri);
            return;
        }

        StaticResponse sr = new StaticResponse(res);
        chain.doFilter(request, sr);

        byte[] bytes = sr.toByteArray();
        FileUtils.writeByteArrayToFile(file, bytes);

        res.sendRedirect(req.getContextPath() + "/static/" + uri);
    }

    public void destroy() {
    }

    public void init(FilterConfig fConfig) throws ServletException {
```

```
        sc = fConfig.getServletContext();
    }
}
```

无论是 JSP，还是 Servlet，若想向客户端浏览器发送数据，都必须使用 response.getWriter()或 response.getOutputStream()这两个流，其中发送文本数据必须使用 repsonse.getWriter()流，而发送字节数据就必须使用 response.getOutputStream()流了。

我们“调包”repsonse 时，要把 getWriter()方法返回的流换成我们自己的 PrintWriter 流，这个 PrintWriter 流的底层流是一个 ByteArrayOutputStream，这就可以保存数据最终是写入到字节数组中了。然后我们的 response 还提供一个 toByteArray()方法用来获取用户响应的数据。

StaticResponse.java

```
public class StaticResponse extends HttpServletResponseWrapper {
    private ByteArrayOutputStream out = new ByteArrayOutputStream();
    private PrintWriter pout = null;

    public StaticResponse(HttpServletResponse response)
            throws UnsupportedEncodingException {
        super(response);
        pout = new PrintWriter(new OutputStreamWriter(out, "utf-8"));
    }

    public PrintWriter getWriter() {
        return pout;
    }

    public byte[] toByteArray() {
        pout.flush();
        return out.toByteArray();
    }
}
```

day16 总结

文件上传概述

1 文件上传的作用

例如网络硬盘！就是用来上传下载文件的。
在智联招聘上填写一个完整的简历还需要上传照片呢。

2 文件上传对页面的要求

上传文件的要求比较多，需要记一下：

1. 必须使用表单，而不能是超链接；
2. 表单的 method 必须是 POST，而不能是 GET；
3. 表单的 enctype 必须是 multipart/form-data；
4. 在表单中添加 file 表单字段，即<input type=”file”…/>

```
    <form action="${pageContext.request.contextPath }/FileUploadServlet"
method="post" enctype="multipart/form-data">
    用户名：<input type="text" name="username"/><br/>
    文件1：<input type="file" name="file1"/><br/>
    文件2：<input type="file" name="file2"/><br/>
    <input type="submit" value="提交"/>
    </form>
```

3 比对文件上传表单和普通文本表单的区别

通过 httpWatch 查看“文件上传表单”和“普通文本表单”的区别。

- 文件上传表单的 enctype=”multipart/form-data”，表示多部件表单数据；
- 普通文本表单可以不设置 enctype 属性：
  - 当 method=”post”时，enctype 的默认值为 application/x-www-form-urlencoded，表示使用 url 编码正文；
  - 当 method=”get”时，enctype 的默认值为 null，没有正文，所以就不需要 enctype 了。

对普通文本表单的测试：

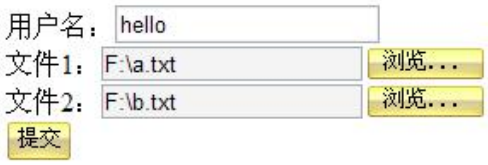
```
<form     action="${pageContext.request.contextPath     }/FileUploadServlet"
method="post">
    用户名：<input type="text" name="username"/><br/>
```

```
    文件1：<input type="file" name="file1"/><br/>
    文件2：<input type="file" name="file2"/><br/>
    <input type="submit" value="提交"/>
</form>
```

通过 httpWatch 测试，查看表单的请求数据正文，我们发现请求中只有文件名称，而没有文件内容。也就是说，当表单的 enctype 不是 multipart/form-data 时，请求中不包含文件内容，而只有文件的名称，这说明普通文本表单中 input:file 与 input:text 没什么区别了。

对文件上传表单的测试：

```
    <form action="${pageContext.request.contextPath }/FileUploadServlet"
method="post" enctype="multipart/form-data">
    用户名：<input type="text" name="username"/><br/>
    文件1：<input type="file" name="file1"/><br/>
    文件2：<input type="file" name="file2"/><br/>
    <input type="submit" value="提交"/>
    </form>
```



通过 httpWatch 测试，查看表单的请求数据正文部分，发现正文部分是由多个部件组成，每个部件对应一个表单字段，每个部件都有自己的头信息。头信息下面是空行，空行下面是字段的正文部分。多个部件之间使用随机生成的分隔线隔开。

文本字段的头信息中只包含一条头信息，即 Content-Disposition，这个头信息的值有两个部分，第一部分是固定的，即 form-data，第二部分为字段的名称。在空行后面就是正文部分了，正文部分就是在文本框中填写的内容。

文件字段的头信息中包含两条头信息，Content-Disposition 和 Content-Type。Content-Disposition 中多出一个 filename，它指定的是上传的文件名称。而 Content-Type 指定的是上传文件的类型。文件字段的正文部分就是文件的内容。

请注意，因为我们上传的文件都是普通文本文件，即 txt 文件，所以在 httpWatch 中是可以正常显示的，如果上传的是 exe、mp3 等文件，那么在 httpWatch 看到的就是乱码了。

4 文件上传对 Servlet 的要求

当提交的表单是文件上传表单时，那么对 Servlet 也是有要求的。

首先我们要肯定一点，文件上传表单的数据也是被封装到 request 对象中的。

request.getParameter(String)方法获取指定的表单字段字符内容，但文件上传表单已经不在是字符内容，而是字节内容，所以失效。

这时可以使用 request 的 getInputStream()方法获取 ServletInputStream 对象，它是 InputStream

的子类，这个 ServletInputStream 对象对应整个表单的正文部分（从第一个分隔线开始，到最后），这说明我们需要的解析流中的数据。当然解析它是很麻烦的一件事情，而 Apache 已经帮我们提供了解析它的工具：commons-fileupload。

可以尝试把 request.getInputStream()这个流中的内容打印出来，再对比 httpWatch 中的请求数据。

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        InputStream in = request.getInputStream();
        String s = IOUtils.toString(in);
        System.out.println(s);
    }
```

```
-----------------------------7ddd3370ab2
Content-Disposition: form-data; name="username"

hello
-----------------------------7ddd3370ab2
Content-Disposition: form-data; name="file1"; filename="a.txt"
Content-Type: text/plain

aaa
-----------------------------7ddd3370ab2
Content-Disposition: form-data; name="file2"; filename="b.txt"
Content-Type: text/plain

bbb
-----------------------------7ddd3370ab2--
```

commons-fileupload

为什么使用 fileupload：

上传文件的要求比较多，需要记一下：

- 必须是 POST 表单；
- 表单的 enctype 必须是 multipart/form-data；
- 在表单中添加 file 表单字段，即<input type=”file”…/>

Servlet 的要求：

- 不能再使用 request.getParameter()来获取表单数据；
- 可以使用 request.getInputStream()得到所有的表单数据，而不是一个表单项的数据；
- 这说明不使用 fileupload，我们需要自己来对 request.getInputStream()的内容进行解析!!!

1 fileupload 概述

fileupload 是由 apache 的 commons 组件提供的上传组件。它最主要的工作就是帮我们解析 request.getInputStream()。

fileupload 组件需要的 JAR 包有：

- commons-fileupload.jar，核心包；
- commons-io.jar，依赖包。

2 fileupload 简单应用

fileupload 的核心类有：DiskFileItemFactory、ServletFileUpload、FileItem。

使用 fileupload 组件的步骤如下：

1. 创建工厂类 DiskFileItemFactory 对象：DiskFileItemFactory factory = new DiskFileItemFactory()
2. 使用工厂创建解析器对象：ServletFileUpload fileUpload = new ServletFileUpload(factory)
3. 使用解析器来解析 request 对象：List<FileItem> list = fileUpload.parseRequest(request)

隆重介绍 FileItem 类，它才是我们最终要的结果。一个 FileItem 对象对应一个表单项（表单字段）。一个表单中存在文件字段和普通字段，可以使用 FileItem 类的 isFormField()方法来判断表单字段是否为普通字段，如果不是普通字段，那么就是文件字段了。

- String getName()：获取文件字段的文件名称；
- String getString()：获取字段的内容，如果是文件字段，那么获取的是文件内容，当然上传的文件必须是文本文件；
- String getFieldName()：获取字段名称，例如：<input type=”text” name=”username”/>，返回的是 username；
- String getContentType()：获取上传的文件的类型，例如：text/plain。
- int getSize()：获取上传文件的大小；
- boolean isFormField()：判断当前表单字段是否为普通文本字段，如果返回 false，说明是文件字段；
- InputStream getInputStream()：获取上传文件对应的输入流；
- void write(File)：把上传的文件保存到指定文件中。

3 简单上传示例

写一个简单的上传示例：

- 表单包含一个用户名字段，以及一个文件字段；
- Servlet 保存上传的文件到 uploads 目录，显示用户名，文件名，文件大小，文件类型。

第一步：

完成 index.jsp，只需要一个表单。注意表单必须是 post 的，而且 enctype 必须是 mulitpart/form-data 的。

```
<form action="${pageContext.request.contextPath }/FileUploadServlet"
```

```
method="post" enctype="multipart/form-data">
    用户名：<input type="text" name="username"/><br/>
    文件1：<input type="file" name="file1"/><br/>
    <input type="submit" value="提交"/>
    </form>
```

第二步：

完成 FileUploadServlet

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        // 因为要使用response打印，所以设置其编码
        response.setContentType("text/html;charset=utf-8");

        // 创建工厂
        DiskFileItemFactory dfif = new DiskFileItemFactory();
        // 使用工厂创建解析器对象
        ServletFileUpload fileUpload = new ServletFileUpload(dfif);
        try {
            // 使用解析器对象解析request，得到FileItem列表
            List<FileItem> list = fileUpload.parseRequest(request);
            // 遍历所有表单项
            for(FileItem fileItem : list) {
                // 如果当前表单项为普通表单项
                if(fileItem.isFormField()) {
                    // 获取当前表单项的字段名称
                    String fieldName = fileItem.getFieldName();
                    // 如果当前表单项的字段名为username
                    if(fieldName.equals("username")) {
                        // 打印当前表单项的内容，即用户在username表单项中输入的内容
                        response.getWriter().print("用户名: " +
fileItem.getString() + "<br/>");
                    }
                } else {//如果当前表单项不是普通表单项，说明就是文件字段
                    String name = fileItem.getName();//获取上传文件的名称
                    // 如果上传的文件名称为空，即没有指定上传文件
                    if(name == null || name.isEmpty()) {
                        continue;
                    }
                    // 获取真实路径，对应${项目目录}/uploads，当然，这个目录必须存在
                    String savepath =
this.getServletContext().getRealPath("/uploads");
                    // 通过uploads目录和文件名称来创建File对象
```

```
                    File file = new File(savepath, name);
                    // 把上传文件保存到指定位置
                    fileItem.write(file);
                    // 打印上传文件的名称
                    response.getWriter().print("上传文件名：" + name + "<br/>");
                    // 打印上传文件的大小
                    response.getWriter().print("上传文件大小：" +
fileItem.getSize() + "<br/>");
                    // 打印上传文件的类型
                    response.getWriter().print("上传文件类型：" +
fileItem.getContentType() + "<br/>");
                }
            }
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }
```

文件上传之细节

1 把上传的文件放到 WEB-INF 目录下

如果没有把用户上传的文件存放到 WEB-INF 目录下，那么用户就可以通过浏览器直接访问上传的文件，这是非常危险的。

假如说用户上传了一个 a.jsp 文件，然后用户在通过浏览器去访问这个 a.jsp 文件，那么就会执行 a.jsp 中的内容，如果在 a.jsp 中有如下语句：Runtime.getRuntime().exec(“shutdown –s –t 1”);，那么你就会…

通常我们会在 WEB-INF 目录下创建一个 uploads 目录来存放上传的文件，而在 Servlet 中找到这个目录需要使用 ServletContext 的 getRealPath(String)方法，例如在我的 upload1 项目中有如下语句：

ServletContext servletContext = this.getServletContext();

String savepath = servletContext.getRealPath(“/WEB-INF/uploads”);

其中 savepath 为：F:\tomcat6_1\webapps\upload1\WEB-INF\uploads。

2 文件名称（完整路径、文件名称）

上传文件名称可能是完整路径：

IE6 获取的上传文件名称是完整路径，而其他浏览器获取的上传文件名称只是文件名称而已。浏览器差异的问题我们还是需要处理一下的。

```
            String name = file1FileItem.getName();
            response.getWriter().print(name);
```

使用不同浏览器测试，其中 IE6 就会返回上传文件的完整路径，不知道 IE6 在搞什么，这给我们带来了很大的麻烦，就是需要处理这一问题。

处理这一问题也很简单，无论是否为完整路径，我们都去截取最后一个“\\”后面的内容就可以了。

```
            String name = file1FileItem.getName();
            int lastIndex = name.lastIndexOf("\\");//获取最后一个"\"的位置
            if(lastIndex != -1) {//注意，如果不是完整路径，那么就不会有"\"的存在。
                name = name.substring(lastIndex + 1);//获取文件名称
            }
            response.getWriter().print(name);
```

3 中文乱码问题

上传文件名称中包含中文：

当上传的谁的名称中包含中文时，需要设置编码，commons-fileupload 组件为我们提供了两种设置编码的方式：

- request.setCharacterEncoding(String)：这种方式是我们最为熟悉的方式了；
- fileUpload.setHeaderEncdoing(String)：这种方式的优先级高与前一种。

上传文件的文件内容包含中文：

通常我们不需关心上传文件的内容，因为我们会把上传文件保存到硬盘上！也就是说，文件原来是什么样子，到服务器这边还是什么样子！

但是如果你有这样的需求，非要在控制台显示上传的文件内容，那么你可以使用 fileItem.getString(“utf-8”)来处理编码。

文本文件内容和普通表单项内容使用 FileItem 类的 getString(“utf-8”)来处理编码。

4 上传文件同名问题（文件重命名）

通常我们会把用户上传的文件保存到 uploads 目录下，但如果用户上传了同名文件呢？这会出现覆盖的现象。处理这一问题的手段是使用 UUID 生成唯一名称，然后再使用“_”连接文件上传的原始名称。

例如用户上传的文件是“我的一寸照片.jpg”，在通过处理后，文件名称为：“891b3881395f4175b969256a3f7b6e10_我的一寸照片.jpg”，这种手段不会使文件丢失扩展名，并且因为 UUID 的唯一性，上传的文件同名，但在服务器端是不会出现同名问题的。

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
```

```
        DiskFileItemFactory dfif = new DiskFileItemFactory();
        ServletFileUpload fileUpload = new ServletFileUpload(dfif);
        try {
            List<FileItem> list = fileUpload.parseRequest(request);
            //获取第二个表单项，因为第一个表单项是username，第二个才是file表单项
            FileItem fileItem = list.get(1);
            String name = fileItem.getName();//获取文件名称

            // 如果客户端使用的是IE6，那么需要从完整路径中获取文件名称
            int lastIndex = name.lastIndexOf("\\");
            if(lastIndex != -1) {
                name = name.substring(lastIndex + 1);
            }

            // 获取上传文件的保存目录
            String savepath =
this.getServletContext().getRealPath("/WEB-INF/uploads");
            String uuid = CommonUtils.uuid();//生成uuid
            String filename = uuid + "_" + name;//新的文件名称为uuid + 下划线 + 原
始名称

            //创建file对象，下面会把上传文件保存到这个file指定的路径
            //savepath，即上传文件的保存目录
            //filename，文件名称
            File file = new File(savepath, filename);

            // 保存文件
            fileItem.write(file);
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }
```

5 一个目录不能存放过多的文件（存放目录打散）

一个目录下不应该存放过多的文件，一般一个目录存放 1000 个文件就是上限了，如果在多，那么打开目录时就会很“卡”。你可以尝试打印 C:\WINDOWS\system32 目录，你会感觉到的。

也就是说，我们需要把上传的文件放到不同的目录中。但是也不能为每个上传的文件一个目录，这种方式会导致目录过多。所以我们应该采用某种算法来“打散”!

打散的方法有很多，例如使用日期来打散，每天生成一个目录。也可以使用文件名的首字母来生成目录，相同首字母的文件放到同一目录下。

日期打散算法：如果某一天上传的文件过多，那么也会出现一个目录文件过多的情况；

首字母打散算法：如果文件名是中文的，因为中文过多，所以会导致目录过多的现象。

我们这里使用 hash 算法来打散：

1. 获取文件名称的 hashCode：int hCode = name.hashCode();；
2. 获取 hCode 的低 4 位，然后转换成 16 进制字符；
3. 获取 hCode 的 5~8 位，然后转换成 16 进制字符；
4. 使用这两个 16 进制的字符生成目录链。例如低 4 位字符为“5”

这种算法的好处是，在 uploads 目录下最多生成 16 个目录，而每个目录下最多再生成 16 个目录，即 256 个目录，所有上传的文件都放到这 256 个目录下。如果每个目录上限为 1000 个文件，那么一共可以保存 256000 个文件。

例如上传文件名称为：新建 文本文档.txt，那么把“新建 文本文档.txt”的哈希码获取到，再获取哈希码的低 4 位，和 5~8 位。假如低 4 位为：9，5~8 位为 1，那么文件的保存路径为 uploads/9/1/。

```
    int hCode = name.hashCode();//获取文件名的hashCode
    //获取hCode的低4位，并转换成16进制字符串
    String dir1 = Integer.toHexString(hCode & 0xF);
    //获取hCode的低5~8位，并转换成16进制字符串
    String dir2 = Integer.toHexString(hCode >>> 4 & 0xF);
    //与文件保存目录连接成完整路径
    savepath = savepath + "/" + dir1 + "/" + dir2;
    //因为这个路径可能不存在，所以创建成File对象，再创建目录链，确保目录在保存文件之前已经存在
    new File(savepath).mkdirs();
```

6 上传的单个文件的大小限制

限制上传文件的大小很简单，ServletFileUpload 类的 setFileSizeMax(long)就可以了。参数就是上传文件的上限字节数，例如 servletFileUpload.setFileSizeMax(1024*10)表示上限为 10KB。

一旦上传的文件超出了上限，那么就会抛出 FileUploadBase.FileSizeLimitExceededException 异常。我们可以在 Servlet 中获取这个异常，然后向页面输出“上传的文件超出限制”。

```
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        DiskFileItemFactory dfif = new DiskFileItemFactory();
        ServletFileUpload fileUpload = new ServletFileUpload(dfif);
        // 设置上传的单个文件的上限为10KB
        fileUpload.setFileSizeMax(1024 * 10);
        try {
            List<FileItem> list = fileUpload.parseRequest(request);
            //获取第二个表单项，因为第一个表单项是username，第二个才是file表单项
            FileItem fileItem = list.get(1);
```

```
            String name = fileItem.getName();//获取文件名称

            // 如果客户端使用的是IE6，那么需要从完整路径中获取文件名称
            int lastIndex = name.lastIndexOf("\\");
            if(lastIndex != -1) {
                name = name.substring(lastIndex + 1);
            }

            // 获取上传文件的保存目录
            String savepath =
this.getServletContext().getRealPath("/WEB-INF/uploads");
            String uuid = CommonUtils.uuid();//生成uuid
            String filename = uuid + "_" + name;//新的文件名称为uuid + 下划线 + 原
始名称

            int hCode = name.hashCode();//获取文件名的hashCode
            //获取hCode的低4位，并转换成16进制字符串
            String dir1 = Integer.toHexString(hCode & 0xF);
            //获取hCode的低5~8位，并转换成16进制字符串
            String dir2 = Integer.toHexString(hCode >>> 4 & 0xF);
            //与文件保存目录连接成完整路径
            savepath = savepath + "/" + dir1 + "/" + dir2;
            //因为这个路径可能不存在，所以创建成File对象，再创建目录链，确保目录在保存文
件之前已经存在
            new File(savepath).mkdirs();

            //创建file对象，下面会把上传文件保存到这个file指定的路径
            //savepath，即上传文件的保存目录
            //filename，文件名称
            File file = new File(savepath, filename);

            // 保存文件
            fileItem.write(file);
        } catch (Exception e) {
            // 判断抛出的异常的类型是否为
FileUploadBase.FileSizeLimitExceededException
            // 如果是，说明上传文件时超出了限制。
            if(e instanceof FileUploadBase.FileSizeLimitExceededException) {
                // 在request中保存错误信息
                request.setAttribute("msg", "上传失败！上传的文件超出了10KB！");
                // 转发到index.jsp页面中！在index.jsp页面中需要使用${msg}来显示错误信
息
                request.getRequestDispatcher("/index.jsp").forward(request,
```

```
response);
                return;
            }
            throw new ServletException(e);
        }
    }
```

7 上传文件的总大小限制

上传文件的表单中可能允许上传多个文件，例如：

有时我们需要限制一个请求的大小。也就是说这个请求的最大字节数（所有表单项之和）！实现这一功能也很简单，只需要调用 ServletFileUpload 类的 setSizeMax(long)方法即可。

例如 fileUpload.setSizeMax(1024 * 10);，显示整个请求的上限为 10KB。当请求大小超出 10KB 时，ServletFileUpload 类的 parseRequest()方法会抛出 FileUploadBase.SizeLimitExceededException 异常。

8 缓存大小与临时目录

大家想一想，如果我上传一个蓝光电影，先把电影保存到内存中，然后再通过内存 copy 到服务器硬盘上，那么你的内存能吃的消么？

所以 fileupload 组件不可能把文件都保存在内存中，fileupload 会判断文件大小是否超出 10KB，如果是那么就把文件保存到硬盘上，如果没有超出，那么就保存在内存中。

10KB 是 fileupload 默认的值，我们可以来设置它。

当文件保存到硬盘时，fileupload 是把文件保存到系统临时目录，当然你也可以去设置临时目录。

- Size threshold is 10KB.
- Repository is the system default temp directory, as returned by `System.getProperty("java.io.tmpdir")`.

```
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        DiskFileItemFactory dfif = new DiskFileItemFactory(1024*20, new
File("F:\\temp"));
        ServletFileUpload fileUpload = new ServletFileUpload(dfif);
```

```
        try {
            List<FileItem> list = fileUpload.parseRequest(request);
            FileItem fileItem = list.get(1);
            String name = fileItem.getName();
            String savepath =
this.getServletContext().getRealPath("/WEB-INF/uploads");

            // 保存文件
            fileItem.write(path(savepath, name));
        } catch (Exception e) {
            throw new ServletException(e);
        }
    }

    private File path(String savepath, String filename) {
        // 从完整路径中获取文件名称
        int lastIndex = filename.lastIndexOf("\\");
        if(lastIndex != -1) {
            filename = filename.substring(lastIndex + 1);
        }

        // 通过文件名称生成一级、二级目录
        int hCode = filename.hashCode();
        String dir1 = Integer.toHexString(hCode & 0xF);
        String dir2 = Integer.toHexString(hCode >>> 4 & 0xF);
        savepath = savepath + "/" + dir1 + "/" + dir2;
        // 创建目录
        new File(savepath).mkdirs();

        // 给文件名称添加uuid前缀
        String uuid = CommonUtils.uuid();
        filename = uuid + "_" + filename;

        // 创建文件完成路径
        return new File(savepath, filename);
    }
```

文件下载

2 通过 Servlet 下载 1

被下载的资源必须放到 WEB-INF 目录下（只要用户不能通过浏览器直接访问就 OK），然后通过 Servlet 完成下载。

在 jsp 页面中给出超链接，链接到 DownloadServlet，并提供要下载的文件名称。然后 DownloadServlet 获取文件的真实路径，然后把文件写入到 response.getOutputStream()流中。

download.jsp

```
<body>
  This is my JSP page. <br>
  <a href="<c:url value='/DownloadServlet?path=a.avi'/>">a.avi</a><br/>
  <a href="<c:url value='/DownloadServlet?path=a.jpg'/>">a.jpg</a><br/>
  <a href="<c:url value='/DownloadServlet?path=a.txt'/>">a.txt</a><br/>
</body>
```

DownloadServlet.java

```
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        String filename = request.getParameter("path");
        String filepath =
this.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);
        File file = new File(filepath);
        if(!file.exists()) {
            response.getWriter().print("您要下载的文件不存在！");
            return;
        }
        IOUtils.copy(new FileInputStream(file), response.getOutputStream());
    }
```

上面代码有如下问题：

- 可以下载 a.avi，但在下载框中的文件名称是 DownloadServlet；
- 不能下载 a.jpg 和 a.txt，而是在页面中显示它们。

3 通过 Servlet 下载 2

下面来处理上一例中的问题，让下载框中可以显示正确的文件名称，以及可以下载 a.jpg 和 a.txt 文件。

通过添加 content-disposition 头来处理上面问题。当设置了 content-disposition 头后，浏览器就会弹出下载框。

而且还可以通过 content-disposition 头来指定下载文件的名称!

```
        String filename = request.getParameter("path");
        String filepath =
this.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);
        File file = new File(filepath);
        if(!file.exists()) {
            response.getWriter().print("您要下载的文件不存在！");
            return;
        }
        response.addHeader("content-disposition", "attachment;filename=" +
filename);
        IOUtils.copy(new FileInputStream(file), response.getOutputStream());
```

虽然上面的代码已经可以处理 txt 和 jpg 等文件的下载问题，并且也处理了在下载框中显示文件名称的问题，但是如果下载的文件名称是中文的，那么还是不行的。

3 通过 Servlet 下载 3

下面是处理在下载框中显示中文的问题!

其实这一问题很简单，只需要通过 URL 来编码中文即可!

download.jsp

```
    <a href="<c:url value='/DownloadServlet?path=这个杀手不太冷.avi'/>">这个杀手不太冷.avi</a><br/>
    <a href="<c:url value='/DownloadServlet?path=白冰.jpg'/>">白冰.jpg</a><br/>
    <a  href="<c:url  value='/DownloadServlet?path= 说 明 文 档 .txt'/>"> 说 明 文 档.txt</a><br/>
```

DownloadServlet.java

```
        String filename = request.getParameter("path");
        // GET请求中，参数中包含中文需要自己动手来转换。
        // 当然如果你使用了“全局编码过滤器”，那么这里就不用处理了
        filename = new String(filename.getBytes("ISO-8859-1"), "UTF-8");

        String filepath =
this.getServletContext().getRealPath("/WEB-INF/uploads/" + filename);
        File file = new File(filepath);
        if(!file.exists()) {
            response.getWriter().print("您要下载的文件不存在！");
            return;
        }
        // 所有浏览器都会使用本地编码，即中文操作系统使用GBK
        // 浏览器收到这个文件名后，会使用iso-8859-1来解码
        filename = new String(filename.getBytes("GBK"), "ISO-8859-1");
        response.addHeader("content-disposition", "attachment;filename=" +
filename);
        IOUtils.copy(new FileInputStream(file), response.getOutputStream());
```

JavaMail

今日内容

- 邮件协议
- telnet 访问邮件服务器
- JavaMail

邮件协议

1 收发邮件

发邮件大家都会吧！发邮件是从客户端把邮件发送到邮件服务器，收邮件是把邮件服务器的邮件下载到客户端。

我们在 163、126、QQ、sohu、sina 等网站注册的 Email 账户，其实就是在邮件服务器中注册的。这些网站都有自己的邮件服务器。

2 邮件协议概述

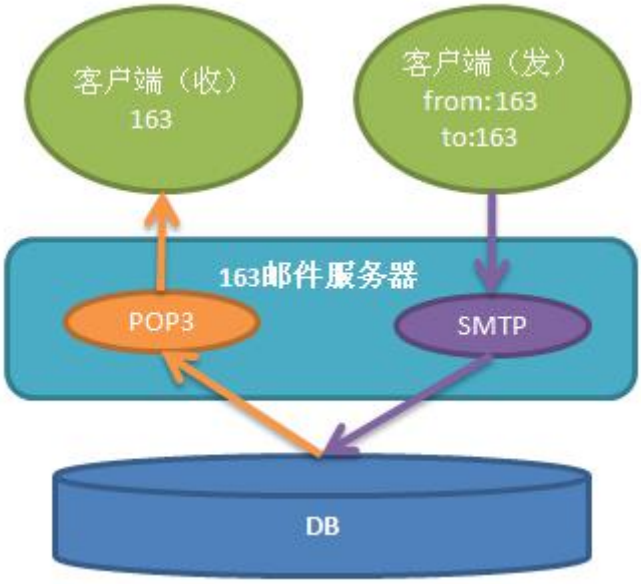
与 HTTP 协议相同，收发邮件也是需要有传输协议的。

- SMTP：（Simple Mail Transfer Protocol，简单邮件传输协议）发邮件协议；
- POP3：（Post Office Protocol Version 3，邮局协议第 3 版）收邮件协议；
- IMAP：（Internet Message Access Protocol，因特网消息访问协议）收发邮件协议，我们的课程不涉及该协议。

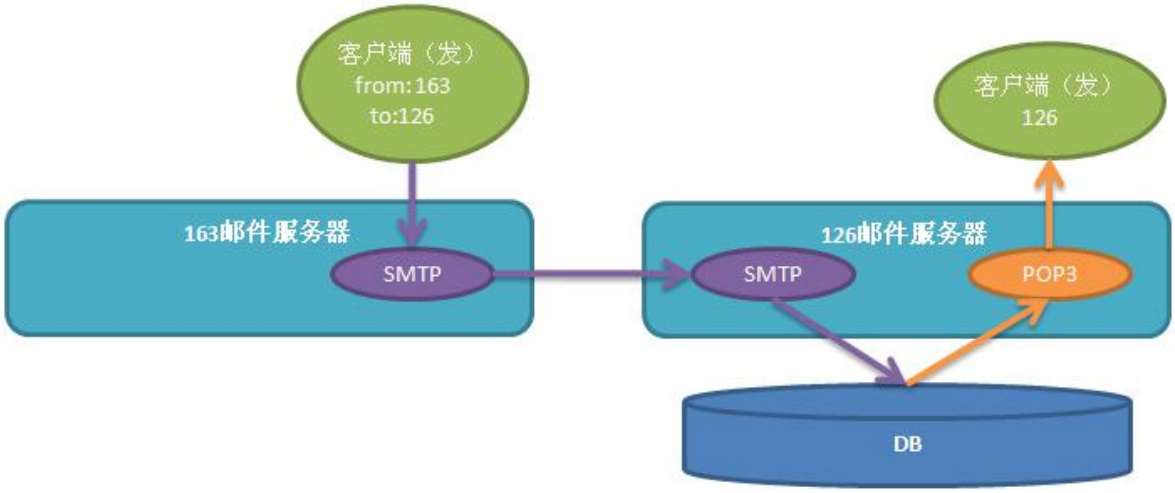
3 理解邮件收发过程

其实你可以把邮件服务器理解为邮局！如果你需要给朋友寄一封信，那么你需要把信放到邮筒中，这样你的信会“自动”到达邮局，邮局会把信邮到另一个省市的邮局中。然后这封信会被送到收信人的邮箱中。最终收信人需要自己经常查看邮箱是否有新的信件。

其实每个邮件服务器都由 SMTP 服务器和 POP3 服务器构成，其中 SMTP 服务器负责发邮件的请求，而 POP3 负责收邮件的请求。



当然，有时我们也会使用 163 的账号，向 126 的账号发送邮件。这时邮件是发送到 126 的邮件服务器，而对于 163 的邮件服务器是不会存储这封邮件的。



4 邮件服务器名称

smtp 服务器的端口号为 25，服务器名称为 smtp.xxx.xxx。
pop3 服务器的端口号为 110，服务器名称为 pop3.xxx.xxx。
例如：

- 163：smtp.163.com 和 pop3.163.com；
- 126：smtp.126.com 和 pop3.126.com；
- qq：smtp.qq.com 和 pop3.qq.com；
- sohu：smtp.sohu.com 和 pop3.sohu.com；
- sina：smtp.sina.com 和 pop3.sina.com。

telnet 收发邮件（玩玩）

1 BASE64 加密

BASE64 是一种加密算法，这种加密方式是可逆的！它的作用是使加密后的文本无法用肉眼识别。Java 提供了 sun.misc.BASE64Encoder 这个类，用来对做 Base64 的加密和解密，但我们知道，使用 sun 包下的东西会有警告！甚至在 eclipse 中根本使用不了这个类（需要设置），所以我们还是听 sun 公司的话，不要去使用它内部使用的类，我们去使用 apache commons 组件中的 codec 包下的 Base64 这个类来完成 BASE64 加密和解密。

```
package cn.itcast;

import org.apache.commons.codec.binary.Base64;

public class Base64Utils {
    public static String encode(String s) {
        return encode(s, "utf-8");
    }

    public static String decode(String s) {
        return decode(s, "utf-8");
    }

    public static String encode(String s, String charset) {
        try {
            byte[] bytes = s.getBytes(charset);
            bytes = Base64.encodeBase64(bytes);
            return new String(bytes, charset);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public static String decode(String s, String charset) {
        try {
            byte[] bytes = s.getBytes(charset);
            bytes = Base64.decodeBase64(bytes);
            return new String(bytes, charset);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

```
}
```

2 telnet 发邮件

连接 163 的 smtp 服务器：`C:\>telnet smtp.163.com 25`；

连接成功后需要如下步骤才能发送邮件：

1 与服务器打招呼：**ehlo 你的名字**

```
ehlo xxx
250-mail
250-PIPELINING
250-AUTH LOGIN PLAIN
250-AUTH=LOGIN PLAIN
250-coremail 1Uxr2xKj7kG0xkI17xGrU7I0s8FY2U3Uj8Cz28x1UUUUU7Ic2I0Y2UFlqftUUCa0xDrUUUUj
250-STARTTLS
250 8BITMIME
```

2 发出登录请求：**auth login**

```
auth login
334 dXNlcm5hbWU6
```

3 输入加密后的邮箱名：(itcast_cxf@163.com)**aXRjYXN0X2N4ZkAxNjMuY29t**

4 输入加密后的邮箱密码：(itcast)**aXRjYXN0**

```
aXRjYXN0X2N4ZkAxNjMuY29t
334 UGFzc3dvcmQ6
aXRjYXN0
235 Authentication successful
```

5 输入谁来发送邮件，即 from：**mail from:<itcast_cxf@163.com>**

```
mail from:<itcast_cxf@163.com>
250 Mail OK
```

6 输入把邮件发给谁，即 to：**rcpt to:<itcast_cxf@126.com>**

```
rcpt to:<itcast_cxf@126.com>
250 Mail OK
```

7 发送填写数据请求：**data**

```
data
354 End data with <CR><LF>.<CR><LF>
```

8 开始输入数据，数据包含：from、to、subject，以及邮件内容，如果输入结束后，以一个“.”为一行，表示输入结束：

from:<zhangBoZhi@163.com>

to:<itcast_cxf@sina.com>

subject: 我爱上你了

我已经深深的爱上你了，我是张柏芝。

.

注意，在标题和邮件正文之间要有一个空行！当要退出时，一定要以一个“.”为单行，表示输入结束。

9 最后一步：**quit**

```
from:<zhangBoZhi@163.com>
to:<itcast_cxf@sina.com>
subject: 我爱上你了

 我已经深深的爱上你了，我是张柏芝。
.
250 Mail OK queued as smtp11,D8CowEB5uVPE3ndRzfc6AA--.539S2 1366810447
quit
221 Bye
```

telnet 收邮件

1 telnet 收邮件的步骤

pop3 无需使用 Base64 加密!!!

收邮件连接的服务器是 pop3.xxx.com，pop3 协议的默认端口号是 110。请注意！这与发邮件完全不同。如果你在 163 有邮箱账户，那么你想使用 telnet 收邮件，需要连接的服务器是 pop3.163.com。

- 连接 pop3 服务器：telnet pop3.163.com 110
- user 命令：user 用户名，例如：user itcast_cxf@163.com；
- pass 命令：pass 密码，例如：pass itcast；
- stat 命令：stat 命令用来查看邮箱中邮件的个数，所有邮件所占的空间；
- list 命令：list 命令用来查看所有邮件，或指定邮件的状态，例如：list 1 是查看第一封邮件的大小，list 是查看邮件列表，即列出所有邮件的编号，及大小；
- retr 命令：查看指定邮件的内容，例如：retr 1#是查看第一封邮件的内容；
- dele 命令：标记某邮件为删除，但不是马上删除，而是在退出时才会真正删除；
- quit 命令：退出！如果在退出之前已经使用 dele 命令标记了某些邮件，那么会在退出是删除它们。

```
user itcast_cxf@126.com
+OK core mail
pass itcast
+OK 6 message(s) [9469 byte(s)]
list
+OK 6 9469
1 1474
2 1397
3 1467
4 1702
5 1709
6 1720
.
```

```
list 1#
+OK 1 1474
```

```
retr 1#
+OK 1474 octets
Received: from m50-132.163.com (unknown [123.125.50.132])
        by mx19 (Coremail) with SMTP id x8mowED58EIklYBRwqdBAA--.14398S2;
        Wed, 01 May 2013 12:08:04 +0800 (CST)
DKIM-Signature: v=1; a=rsa-sha256; c=relaxed/relaxed; d=163.com;
        s=s110527; h=Received:From:To:Message-ID:Subject:MIME-Version:
        Content-Type:Date; bh=ANVmsYRngy6b/OFIJRGz4jwzKJUO2feMkM3BgnhhOA
        4=; b=K4MHSbxOrUnFOCxd+GFLeaHtiNd+86jFeZlz5u4V/THdDIsywsanlPtyim
        oszBiLxv5ldvBhAvl9NRglOpJNR4T5yCBo0kEoU3EsquwWCJwzWftDtMFdbCIxLY
        M5inPdXcOM5io7dnc8y/Hr7u3ST8bCTjlX7m6JLc0C8Pjk1P0=
Received: from cxf (unknown [118.186.203.161])
        by smtp2 (Coremail) with SMTP id DNGowED5kUgjlYBRF5UdAA--.4052S2;
        Wed, 01 May 2013 12:08:04 +0800 (CST)
From: itcast_cxf@163.com
To: itcast_cxf@126.com
Message-ID: <33347540.1.1367381283359.JavaMail.Administrator@smtp.163.com>
Subject: =?GBK?B?taS2qw==?=
MIME-Version: 1.0
Content-Type: multipart/mixed;
        boundary="------=_Part_0_29534197.1367381283312"
X-CM-TRANSID:DNGowED5kUgjlYBRF5UdAA--.4052S2
Date: Wed, 1 May 2013 12:08:04 +0800 (CST)
X-CM-SenderInfo: 5lwft2hwbf5wi6rwjhhfrp/1tbiMRcQAlEAEFyA3AAAs1
X-Coremail-Antispam: 1Uf129KBjDUn29KB7ZKAUJUUUUU529EdanIXcx71UUUUU7v73
        VFW2AGmfu7jjvjm3AaLaJ3UbIYCTnIWIevJa73UjIFyTuYvjxUwtx6DUUUU

------=_Part_0_29534197.1367381283312
Content-Type: text/html;charset=utf-8
Content-Transfer-Encoding: base64

6L+Z5piv5LiA5bCB5rWL6K+V6YKu5Lu277yB5LiN6KaB5aSq5Zyo5oSP77yB
------=_Part_0_29534197.1367381283312--

.
```

JavaMail

1 JavaMail 概述

Java Mail 是由 SUN 公司提供的专门针对邮件的 API，主要 Jar 包：mail.jar、activation.jar。

在使用 MyEclipse 创建 web 项目时，需要小心！如果只是在 web 项目中使用 java mail 是没有什么问题的，发布到 Tomcat 上运行一点问题都没有！

但是如果是在 web 项目中写测试那就出问题了。

在 MyEclipse 中，会自动给 web 项目导入 javax.mail 包中的类，但是不全（其实是只有接口，而没有接口的实现类），所以只靠 MyEclipse 中的类是不能运行 java mail 项目的，但是如果这时你再去自行导入 mail.jar 时，就会出现冲突。

处理方案：到下面路径中找到 javaee.jar 文件，把 javax.mail 删除!!!

D:\Program Files\MyEclipse\Common\plugins\com.genuitec.eclipse.j2eedt.core_10.0.0.me201110301321\data\libraryset\EE_5

2 JavaMail 中主要类

java mail 中主要类：javax.mail.Session、javax.mail.internet.MimeMessage、javax.mail.Transport。

Session：表示会话，即客户端与邮件服务器之间的会话！想获得会话需要给出账户和密码，当然还要给出服务器名称。在邮件服务中的 Session 对象，就相当于连接数据库时的 Connection 对象。

MimeMessage：表示邮件类，它是 Message 的子类。它包含邮件的主题（标题）、内容，收件人地址、发件人地址，还可以设置抄送和暗送，甚至还可以设置附件。

Transport：用来发送邮件。它是发送器！

3 JavaMail 之 Hello World

在使用 telnet 发邮件时，还需要自己来处理 Base64 编码的问题，但使用 JavaMail 就不必理会这些问题了，都由 JavaMail 来处理。

第一步：获得 Session

```
Session session = Session.getInstance(Properties prop, Authenticator auth);
```

其中 prop 需要指定两个键值，一个是指定服务器主机名，另一个是指定是否需要认证！我们当然需要认证！

```
Properties prop = new Properties();
prop.setProperty("mail.host", "smtp.163.com");//设置服务器主机名
prop.setProperty("mail.smtp.auth", "true");//设置需要认证
```

其中 Authenticator 是一个接口表示认证器，即校验客户端的身份。我们需要自己来实现这个接口，实现这个接口需要使用账户和密码。

```
Authenticator auth = new Authenticator() {
    public PasswordAuthenticator getPasswordAuthenticator () {
        new PasswordAuthenticator("itcast_cxf", "itcast");//用户名和密码
    }
};
```

通过上面的准备，现在可以获取得 Session 对象了：

```
Session session = Session.getInstance(prop, auth);
```

第二步：创建 MimeMessage 对象

创建 MimeMessage 需要使用 Session 对象来创建：

```
MimeMessage msg = new MimeMessage(session);
```

然后需要设置发信人地址、收信人地址、主题，以及邮件正文。

msg.setFrom(new InternetAddress(“itcast_cxf@163.com”));//设置发信人
msg.addRecipients(RecipientType.TO, “itcast_cxf@qq.com,itcast_cxf@sina.com”);//设置多个收信人
msg.addRecipients(RecipientType.CC, “itcast_cxf@sohu.com,itcast_cxf@126.com”);//设置多个抄送
msg.addRecipients(RecipientType.BCC, ”itcast_cxf@hotmail.com”);//设置暗送
msg.setSubject(“这是一封测试邮件”);//设置主题（标题）
msg.setContent(“当然是 hello world!”, “text/plain;charset=utf-8”);//设置正文

第三步：发送邮件

Transport.send(msg);//发送邮件

4 JavaMail 发送带有附件的邮件（了解）

一封邮件可以包含正文、附件 N 个，所以正文与 N 个附件都是邮件的一个部份。

上面的 hello world 案例中，只是发送了带有正文的邮件！所以在调用 setContent()方法时直接设置了正文，如果想发送带有附件邮件，那么需要设置邮件的内容为 MimeMultiPart。

MimeMulitpart parts = new MimeMulitpart();//多部件对象，可以理解为是部件的集合
msg.setContent(parts);//设置邮件的内容为多部件内容。

然后我们需要把正文、N 个附件创建为“主体部件”对象（MimeBodyPart），添加到 MimeMuiltPart 中即可。

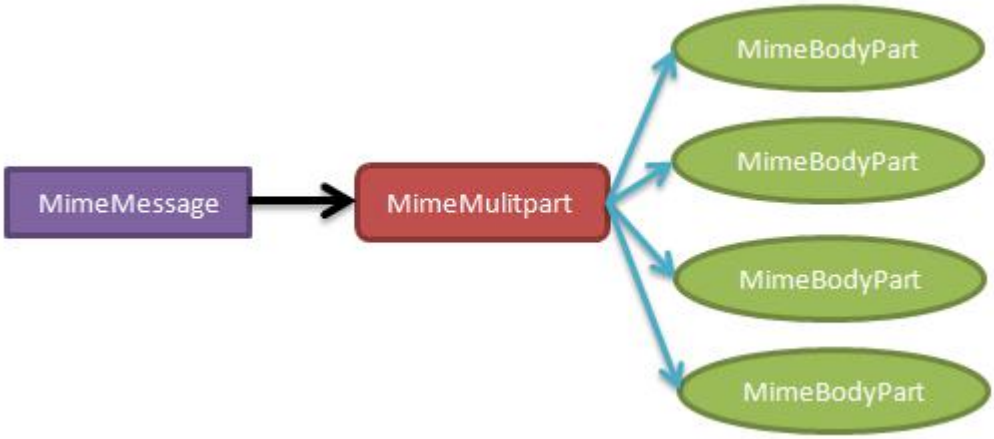
MimeBodyPart part1 = new MimeBodyPart();//创建一个部件
part1.setCotnent(“这是正文部分”, “text/html;charset=utf-8”);//给部件设置内容
parts.addBodyPart(part1);//把部件添加到部件集中。

下面我们创建一个附件：

MimeBodyPart part2 = new MimeBodyPart();//创建一个部件
part2.attachFile(“F:\\a.jpg”);//设置附件
part2.setFileName(“hello.jpg”);//设置附件名称
parts.addBodyPart(part2);//把附件添加到部件集中

注意，如果在设置文件名称时，文件名称中包含了中文的话，那么需要使用 MimeUitlity 类来给中文编码：

part2.setFileName(MimeUitlity.encodeText(“美女.jpg”));


MimeMessage
MimeMulitpart
MimeBodyPart
MimeBodyPart
MimeBodyPart
MimeBodyPart

day17

在线支付

1 在线支付概述

什么是在线支付呢？没错，就是在网上花钱！大家一定有过这样的经历。但是你可能不太了解在线支付的“内情”，下面我们来了解一下！

如果你现在开始经营一个电子商务网站，用户买了东西一定要支付，你的网站一定要可以连接各大银行了，然后在各大银行支付完成后，再返回到你的网站上显示“支付成功”！

这就是今天我们要做的事情，连接银行的网银系统完成支付。说专业一点，我们称之为**“开发在线支付的网关”**。

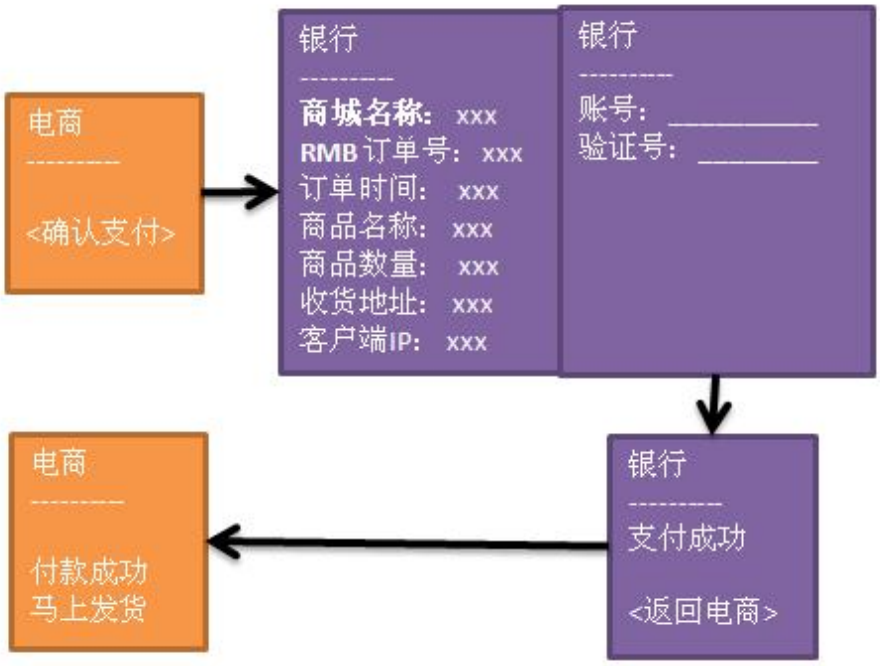
2 两种在线支付的方式

在线支付一共有两种方式：

- 电商直接与银行对接；
- 电商通过第三方支付平台与银行对接；

电商直接与银行对接，这也要银行同意才行，但可惜的是，银行很“牛”，不是谁想与它对接都可以的。如果你的电商每日的资金流量够大，那么银行会和你对接，因为客户支付给电商的钱都存到了银行的帐户中！但是如果资金流量小，银行不会理你的！

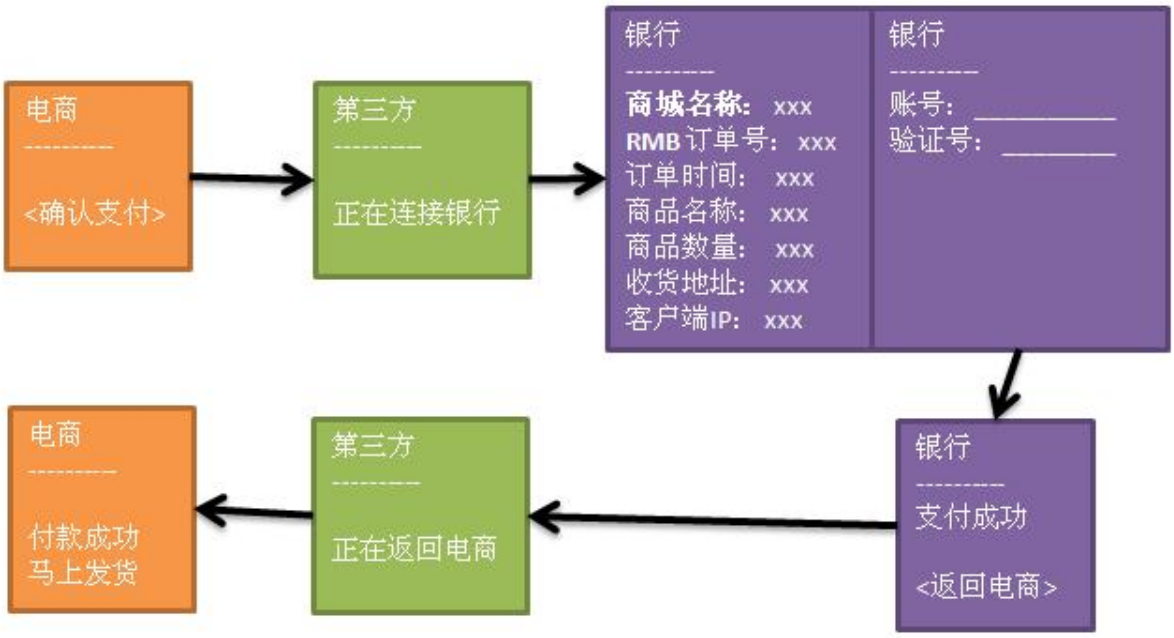
当小网站资金量不足时，不能与银行对接，那么它们会选择与第三方支付公司合作。大家也都明白这是些什么公司，例如：支付宝、易宝、财富通、快钱等公司是国内比较有名的。它们这些公司可以与银行对接（因为资金够多），然后小电商与它们对接！但是第三方是要求收费的！第三方一般会收取电商 1%的费用，不过不会收客户的钱。



通过上图大家可以了解到，在银行的页面上会显示出商城名称、RMB 订单号、订单时间。。。，这些东西银行是怎么知道的，当然是电商传递给银行的。当电商与银行对接后，电商要给银行的页面传递银行页面需要的参数，所以银行的页面才能显示这些数据！

但是，我们的商城不能只可以对接一家银行吧！怎么也要对接 BOC、CCB、ABC、ICBC 四家吧！不同的银行需要的对接参数是不相同的，这说明我们在开发时要为不同的银行写不同的对接代码。这也是直接与银行对接的缺点！当然与银行直接对接也有好处，就是安全，没有手续费！

- 为不同的银行开发不同的代码（缺点）；
- 安全（优点）；
- 没有手续费（优点）；
- 小电商银行不让对接（缺点）。



上图中已经说明，客户在电商的网站上点击确认支付后，会定向到第三方的网站，然后再由第三方与银行对接。这说明电商要传递给第三方参数！再由第三方把参数传递给银行。这种方式的好处是：只需要针对第三方开发即可，而不用再为每家银行提供参数。为每家银行提供参数的工作是第三方的任务了。但是，第三方不老可靠的，如果第三方倒闭了，人跑了，那你的钱就没了。因为客户支付的钱没有到你的银行帐户中，而是支付到了第三方的银行帐户中，而你是在第三方有一个帐户。而且第三方还要收手续费，一般是 1%，这可不是小数字啊（真黑）。

3 通过第三方在线支付规则

电商想在第三方注册商户，需要向第三方提供 ICP 认证。ICP 经营许可证是根据国家《互联网管理办法规定》，经营性网站必须办理的网站经营许可证，没有就属于非法经营。

我们不可能因为练习就去办理 ICP！所以我们无法在第三方注册商户。不过我们已经有现成的在易宝注册的商户，所以这一步就可以忽略了。

当你在易宝注册成功后，易宝会给你如下几样东西：

- 在易宝的开户账号（即商户编码）：10001126856；
- 易宝接入规范：一个 chm 文件；
- 对称加密算法类：PaymentUtil.java；
- 密钥：69cl522AV6q613Ii4W6u8K6XuW8vM1N6bFgyv769220IuYe9u37N4y7rI4Pl。

在易宝接入规范中，我们可以查找到易宝的支付网关，其实就是一个 URL，用来与易宝对接的一个网址：https://www.yeepay.com/app-merchant-proxy/node

在易宝接入规范中，还可以查找到易宝要求的参数，在电商与易宝对接时需要给支付网关传递这些参数：

正式请求地址：https://www.yeepay.com/app-merchant-proxy/node					
参数名称	参数含义	是否必填	参数长度	参数说明	签名顺序

p0_Cmd	业务类型	是	Max(20)	固定值“Buy” .	1
p1_MerId	商户编号	是	Max(11)	商户在易宝支付系统的唯一身份标识.获取方式见“如何获得商户编号”	2
p2_Order	商户订单号	否	Max(50)	若不为””，提交的订单号必须在自身账户交易中唯一;为 ”” 时，易宝支付会自动生成随机的商户订单号.易宝支付系统中对于已付或者撤销的订单，商户端不能重复提交。	3
p3_Amt	支付金额	否	Max(20)	单位:元，精确到分.此参数为空则无法直连(如直连会报错：抱歉，交易金额太小。),必须到易宝网关让消费者输入金额	4
p4_Cur	交易币种	是	Max(10)	固定值 ”CNY”.	5
p5_Pid	商品名称	否	Max(20)	用于支付时显示在易宝支付网关左侧的订单产品信息.此参数如用到中文，请注意转码.	6
p6_Pcat	商品种类	否	Max(20)	商品种类. 此参数如用到中文，请注意转码.	7
p7_Pdesc	商品描述	否	Max(20)	商品描述. 此参数如用到中文，请注意转码.	8
p8_Url	商户接收支付成功数据的地址	否	Max(200)	支付成功后易宝支付会向该地址发送两次成功通知，该地址可以带参数，如: “ www.yeepay.com/callback.action?test=test”. 注意：如不填 p8_Url 的参数值支付成功后您将得不到支付成功的通知。	9
p9_SAF	送货地址	否	Max(1)	为“1”: 需要用户将送货地址留在易宝支付系统;为“0”: 不需要，默认为 ”0”.	10
pa_MP	商户扩展信息	否	Max(200)	返回时原样返回，此参数如用到中文，请注意转码.	11

pd_FrpId	支付通道编码	否	Max(50)	默认为 ”” ，到易宝支付网关. 若不需显示易宝支付的页面，直接跳转到各银行、 神州行支付、骏网一卡通等支付页面， 该字段可依照附录:支付通道编码列表 设置参数值.如果此值设置错误则会报"error.noAvaliableFrp"错误	12
pr_NeedResponse	应答机制	否	Max(1)	**固定值为“1”**： 需要应答机制; 收到易宝支付服务器点对点支付成功通知，必须回写以”success”（无关大小写）开头的字符串，即使您收到成功通知时发现该订单已经处理过，也要正确回 写”success”，否则易宝支付将认为您的系统没有收到通知，启动重发机制，直到收到”success”为止。	13
hmac	签名数据		Max(32)	产生 hmac 需要两个参数，并调用相关 API. 参数 1: STR，列表中的参数值按照签名顺序拼接所产生的字符串，注意 null 要转换为 ””，并确保无乱码. 参数 2: 商户密钥.见"如何获得商户密钥" 各语言范例已经提供封装好了的方法用于生成此参数。 如果以上两个参数有错误，则该参数必然错误，见"抱歉,交易签名无效."	

这些参数需要追加到 URL 后面。

但是要注意，这些参数的值需要加密。加密的密钥和加密算法易宝都会提供！

其中 p8_Url 表示当支付成功后，返回到电商的哪个页面。这说明我们需要写一个显示结果的页面。第三方在支付成功后，会重定向到我们指定的返回页面，而且还会带给我们一些参数，我们的页面需要获取这些参数，显示在页面中。下面是第三方返回的参数：

订 单 查 询 返 回 参 数				
参数名称	参数含义	参数长度	参数说明	签 名 顺 序
r0_Cmd	业务类型	Max(20)	订单查询请求，固定值 ”QueryOrdDetail”	1
r1_Code	查询结果		为“1”: 查询正常; 为“50”: 订单不存在.	2

r2_TrxId	易宝支付交易流水号	Max(50)		3
r3_Amt	支付金额	Max(20)	单位:元，精确到分.	4
r4_Cur	交易币种	Max(10)	固定值 "RMB".	5
r5_Pid	商品名称	Max(20)	易宝支付返回商户设置的商品名称. 此参数如用到中文，请注意转码.	6
r6_Order	商户订单号	Max(50)	易宝支付返回商户订单号.	7
r8_MP	商户扩展信息	Max(1000)	商户可以任意填写 1K 的字符串,支付成功时将原样返回. 此参数如用到中文，请注意转码.	8
rb_PayStatus	支付状态		“INIT” 未支付; “CANCELED” 已取消; “SUCCESS“ 已支付.	9
rc_RefundCount	已退款次数			10
rd_RefundAmt	已退款金额			11
hmac	签名数据	Max(32)	产生 hmac 需要两个参数，并调用相关 API. 参数 1: STR，列表中的参数值按照签名顺序拼接所产生的字符串，注意 null 要转换为 ””. 参数 2: 商户密钥.见"如何获得商户密钥" 各语言范例已经提供封装好了的方法用于生成此参数。	

4 开发第三方在线支付系统

步骤：

- index.jsp 页面：一个表单，提交到 BuyServlet，表单项有：订单编号、付款金额、选择银行；
- BuyServlet：获取表单数据，准备连接第三方网关。因为在 index.jsp 页面中只给出 3 个参数，而第三方需要的参数有 N 多，页面没有给出的参数由 BuyServlet 补充。而且参数还需要加密，这也需要在 BuyServlet 中完成；
- BackServlet：当用户支付成功后，第三方会重定向到我们指定的返回页面，我们使用 BackServlet 作为返回页面，它用来接收第三方传递的参数，显示在页面中。

因为已经有了在易宝的注册商号，所以我们就不用自己去注册商号了。所以这里使用易宝做为第三方支付平台来测试。因为我本人没有电商（必须通过 ICP 认证的电商），所以也不能在第三方注册商号。

我们现在使用的易宝商号是由传智播客提供的，巴巴运动网在易宝注册的商号。所以在测试时支付的钱都给了巴巴运动网在易宝注册的商号了。

第一步：index.jsp

```
<form action="" method="post">
    订单号：<input type="text" name="p2_Order"/><br/>
    金　额：<input type="text" name="p3_Amt"/><br/>
    选择银行：<br/>
    <input type="radio" name="pd_FrpId" value="ICBC-NET-B2C"/>工商银行
    <img src="bank_img/icbc.bmp" align="middle"/>
    <input type="radio" name="pd_FrpId" value="BOC-NET-B2C"/>中国银行
    <img src="bank_img/bc.bmp" align="middle"/><br/><br/>
    <input type="radio" name="pd_FrpId" value="ABC-NET-B2C"/>农业银行
    <img src="bank_img/abc.bmp" align="middle"/>
    <input type="radio" name="pd_FrpId" value="CCB-NET-B2C"/>建设银行
    <img src="bank_img/ccb.bmp" align="middle"/><br/><br/>
    <input type="radio" name="pd_FrpId" value="BOCO-NET-B2C"/>交通银行
    <img src="bank_img/bcc.bmp" align="middle"/><br/>
    <input type="submit" value="确认支付"/>
</form>
```

每个银行对应的值：

可直连银行	
pd_FrpId 参数值　对应支付通道名称	LOGO 图片

1000000-NET	易宝会员支付	
ICBC-NET-B2C	工商银行	
CMBCHINA-NET-B2C	招商银行	
ABC-NET-B2C	中国农业银行	
CCB-NET-B2C	建设银行	
BCCB-NET-B2C	北京银行	
BOCO-NET-B2C CIB-NET-B2C	交通银行 兴业银行	
NJCB-NET-B2C	南京银行	
CMBC-NET-B2C	中国民生银行	
CEB-NET-B2C BOC-NET-B2C	光大银行 中国银行	
PINGANBANK-NET	平安银行	
CBHB-NET-B2C	渤海银行	
HKBEA-NET-B2C NBCB-NET-B2C	东亚银行 宁波银行	
ECITIC-NET-B2C	中信银行(需要证书才能连接到银行)	

SDB-NET-B2C	深圳发展银行	
GDB-NET-B2C	广东发展银行	
SHB-NET-B2C	上海银行	
SPDB-NET-B2C	上海浦东发展银行	
POST-NET-B2C	中国邮政	
BJRCB-NET-B2C	北京农村商业银行	
HXB-NET-B2C	华夏银行（此功能默认不开通，如需开通请与易宝支付销售人员联系）	
CZ-NET-B2C	浙商银行	

第二步：BuyServlet.java

```
public class BuyServlet extends HttpServlet {
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        request.setCharacterEncoding("utf-8");
        response.setContentType("text/html;charset=utf-8");

        String p0_Cmd = "Buy";// 业务类型，固定值为buy，即“买”
        String p1_MerId = "10001126856";// 在易宝注册的商号
        String p2_Order = request.getParameter("p2_Order");// 订单编号
        String p3_Amt = request.getParameter("p3_Amt");// 支付的金额
        String p4_Cur = "CNY";// 交易种币，固定值为CNY，表示人民币
        String p5_Pid = "";// 商品名称
        String p6_Pcat = "";// 商品各类
        String p7_Pdesc = "";// 商品描述
        String p8_Url = "http://localhost:8080/buy/BackServlet";// 电商的返回页
面，当支付成功后，易宝会重定向到这个页面
        String p9_SAF = "";// 送货地址
        String pa_MP = "";// 商品扩展信息
        String pd_FrpId = request.getParameter("pd_FrpId");// 支付通道，即选择银
行
```

```
        String pr_NeedResponse = "1";// 应答机制，固定值为1

        // 密钥，由易宝提供，只有商户和易宝知道这个密钥。
        String keyValue =
"69cl522AV6q613Ii4W6u8K6XuW8vM1N6bFgyv769220IuYe9u37N4y7rI4Pl";

        // 通过上面的参数、密钥、加密算法，生成hmac值
        // 参数的顺序是必须的，如果没有值也不能给出null，而应该给出空字符串。
        String hmac = PaymentUtil.buildHmac(p0_Cmd, p1_MerId, p2_Order, p3_Amt,
                p4_Cur, p5_Pid, p6_Pcat, p7_Pdesc, p8_Url, p9_SAF, pa_MP,
                pd_FrpId, pr_NeedResponse, keyValue);

        // 把所有参数连接到网关地址后面
        String url = "https://www.yeepay.com/app-merchant-proxy/node";
        url += "?p0_Cmd=" + p0_Cmd +
                "&p1_MerId=" + p1_MerId +
                "&p2_Order=" + p2_Order +
                "&p3_Amt=" + p3_Amt +
                "&p4_Cur=" + p4_Cur +
                "&p5_Pid=" + p5_Pid +
                "&p6_Pcat=" + p6_Pcat +
                "&p7_Pdesc=" + p7_Pdesc +
                "&p8_Url=" + p8_Url +
                "&p9_SAF=" + p9_SAF +
                "&pa_MP=" + pa_MP +
                "&pd_FrpId=" + pd_FrpId +
                "&pr_NeedResponse=" + pr_NeedResponse +
                "&hmac=" + hmac;
        System.out.println(url);
        // 重定向到网关
        response.sendRedirect(url);
    }
}
```

第三步：BackServlet

```
public class BackServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html;charset=utf-8");
        /*
         * 易宝会提供一系列的结果参数，我们获取其中需要的即可
         * 获取支付结果：r1_Code，1表示支付成功。
```

```
         * 获取支付金额：r3_Amt
         * 获取电商的订单号：r6_Order
         * 获取结果返回类型：r9_BType，1表示重定向返回，2表示点对点返回，
         *     但点对点我们收不到，因为我们的ip都是局域网ip。
         */
        String r1_Code = request.getParameter("r1_Code");
        String r3_Amt = request.getParameter("r3_Amt");
        String r6_Order = request.getParameter("r6_Order");
        String r9_BType = request.getParameter("r9_BType");

        if(r1_Code.equals("1")) {
            if(r9_BType.equals("1")) {
                response.getWriter().print("<h1>支付成功!</h1>");//其实支付不成功
时根本易宝根本就不会返回到本Servlet
                response.getWriter().print("支付金额为：" + r3_Amt + "<br/>");
                response.getWriter().print("订单号为：" + r6_Order + "<br/>");
            }
        }
    }
}
```