◆ spring 的快速入门案例

① spring 是什么?

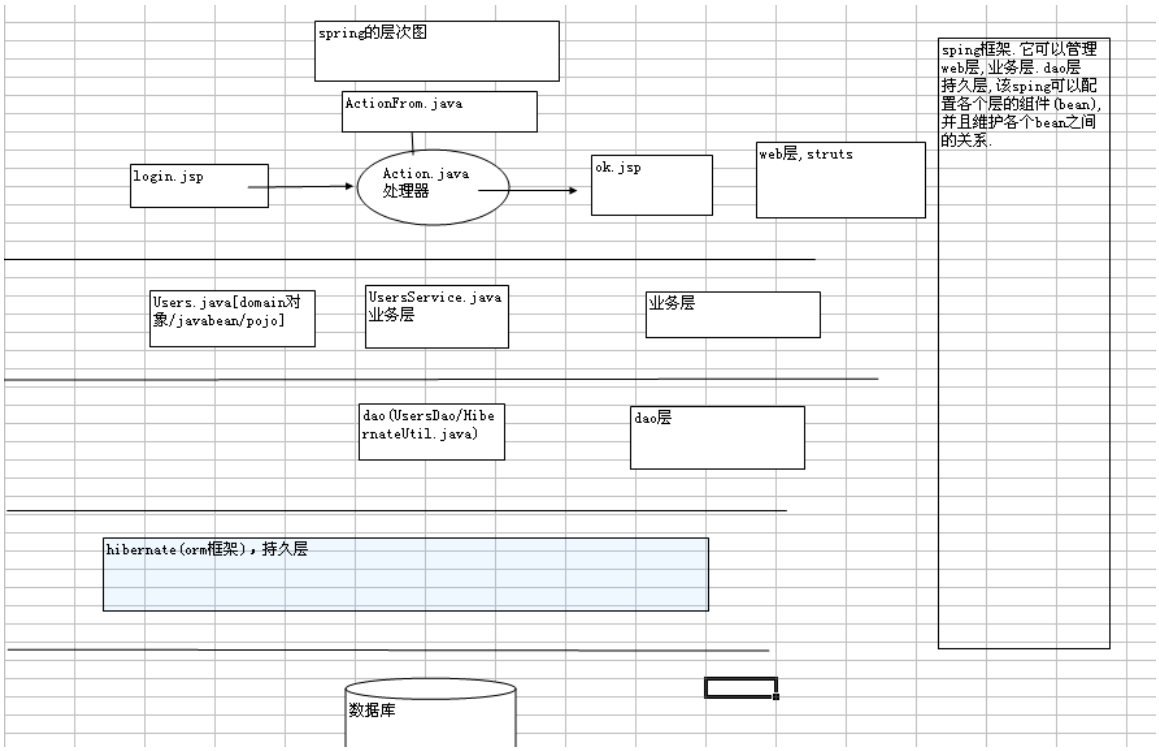
struts 是 web 框架 (jsp/action/actionfrom)

hibernate 是 orm 框架,处于持久层.

spring 是容器框架,用于配置 bean,并维护 bean 之间关系的框架

☞ spring 中有一个非常概念: bean (是 java 中的任何一种对象 javabean/service/action/数据源./dao, ioc(控制反转 inverse of control) di(dependency injection 依赖注入)

☞ 画一个框架图



◆ 快速入门

开发一个 spring 项目.

1. 引入 spring 的开发包(最小配置 spring.jar 该包把常用的 jar 都包括, 还要 写日志包 common-logging.jar
2. 创建 spring 的一个核心文件 applicationContext.xml, [hibernate 有核心 hibernate.cfg.xml struts 核心文件 struts-config.xml], 该文件一般放在 src 目录下,该文件中引入 xsd 文件 ：可以从给出的案例中拷贝一份.
3. 配置 bean

```
<!-- 在容器文件中配置 bean(service/dao/domain/action/数据源) -->
<!-- bean 元素的作用是，当我们的 spring 框架加载时候，spring 就会自动的创建一个 bean 对象，并放入内存
 UserService userSerivce=new UserService();
 userSerivce.setName("韩顺平");
 -->
<bean id="userService" class="com.service.UserService">
```

```
    <!-- 这里就体现出注入的概念. -->
    <property name="name">
        <value>韩顺平</value>
    </property>
  </bean>
```

4. 在 Test.java 中，我们怎么使用

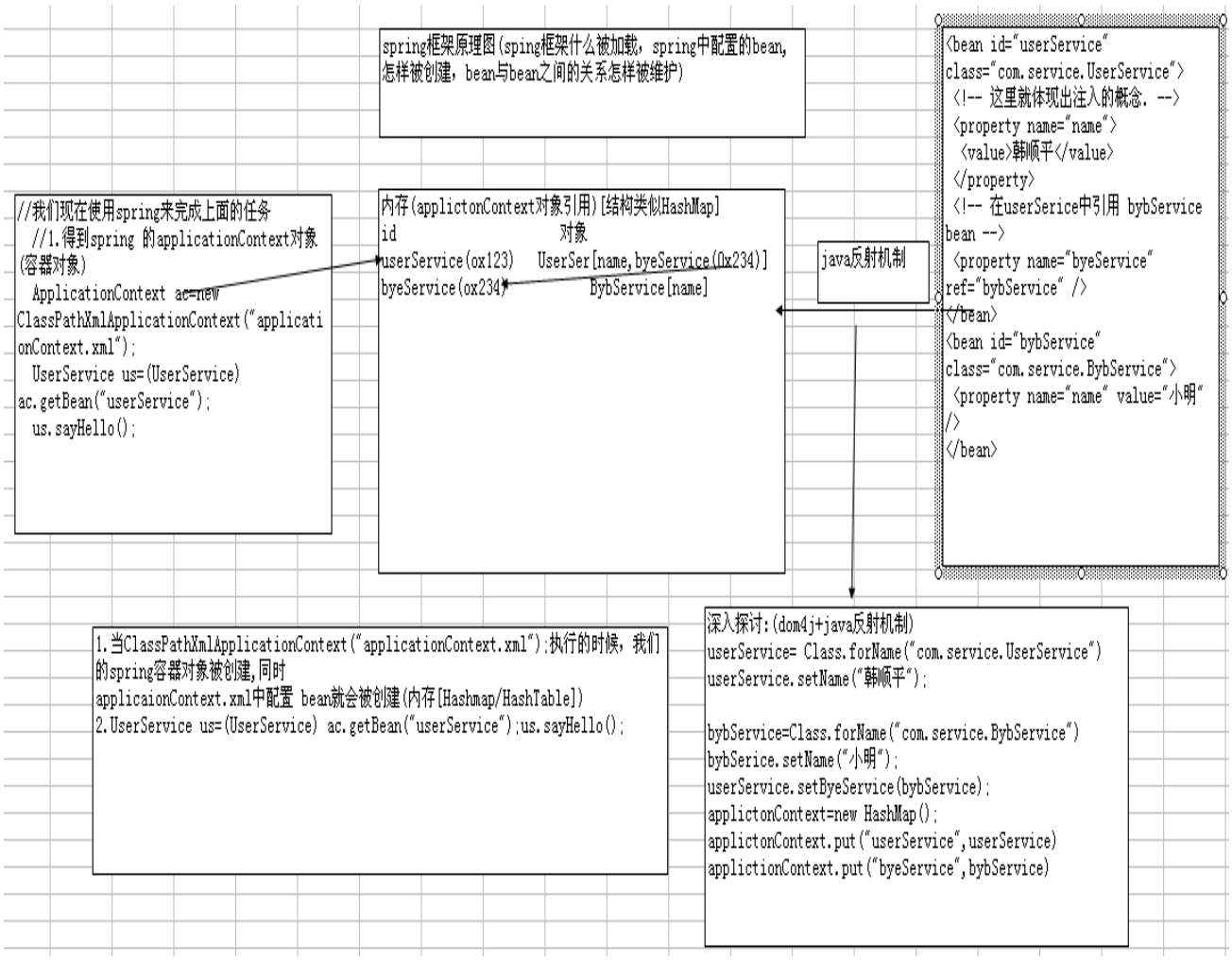
```
//我们现在使用 spring 来完成上面的任务
        //1.得到 spring 的 applicationContext 对象(容器对象)
        ApplicationContext                                          ac=new
ClassPathXmlApplicationContext("applicationContext.xml");
        UserService us=(UserService) ac.getBean("userService");
        us.sayHello();
```

5. **细节**讨论?

传统的方法和使用 spring 的方法

5.1 使用 spring ，没有 new 对象,我们把创建对象的任务交给 spring 框架

5.2 **spring 的运行原理图**:



5.3 我们再看 spring

对上面案例总结:
spring 实际上是一个容器框架，可以配置各种 bean(action/service/domain/dao),并且可以维护 bean 与 bean 的关系,当我们需要使用某个 bean 的时候，我们可以 getBean(id),使用即可.

ioc 是什么?
答 ： ioc(inverse of controll) 控制反转: 所谓控制反转就是把创建对象(bean),和维护对象(bean)的关系的权利从程序中转移到 spring 的容器(applicationContext.xml),而程序本身不再维护.

DI 是什么?
答: di(dependency injection) 依赖注入: 实际上 di 和 ioc 是同一个概念，spring 设计者认为 di 更准确表示 spring 核心技术

☞ 学习框架，最重要的就是学习各个配置.

把 Applicationcontext 做成一个单例的.

上机练习: 把我写的代码走一遍.

◆ spring 开发提倡接口编程,配合 di 技术可以层与层的解耦
举例说明:
现在我们体验一下 spring 的 di 配合接口编程的，完成一个字母大小写转换的案例:
思路:
1. 创建一个接口 ChangeLetter
2. 两个类实现接口
3. 把对象配置到 spring 容器中
4. 使用

通过上面的案例，我们可以初步体会到 di 配合接口编程，的确可以减少层(web 层) 和 业务层的耦合度.

思考题:

接口
ValidateUser
有一个方法:
check(??)
有两个类
CheckUser1 implements ValidateUser
{
    check// 安装 xml 验证
}
CheckUser2 implements VallidateUser{

check()// 到数据库去验证
}

◆ **从 ApplicationContex 应用上下文容器中获取 bean 和从 bean 工厂容器中获取 bean**

具体案例:

```
//从 ApplicationContext 中取 bean
ApplicationContext ac=new ClassPathXmlApplicationContext("com/hsp/ioc/beans.xml");
//当我们去实例化 beans.xml,该文件中配置的 bean 被实例(该 bean scope 是 singleton)
从 bean 中取出 student


//如果我们使用 beanfactory 去获取 bean，当你只是实例化该容器， 那么
//容器的 bean 不被实例化,只有当你去使用 getBean 某个 bean 时，才会实时的创建.

BeanFactory factory = new XmlBeanFactory(
        new ClassPathResource("com/hsp/ioc/beans.xml"));
factory.getBean("student");
```

结论:

1.如果使用 ApplicationContext ，则配置的 bean 如果是 singlton 不管你用不用，都被实例化.(好处就是可以预先加载,缺点就是耗内存)

2.如果是 BeanFactory ,则当你获取 beanfacotry 时候，配置的 bean 不会被马上实例化，当你使用的时候，才被实例(好处节约内存,缺点就是速度)

3.规定: 一般没有特殊要求，应当使用 ApplicatioContext 完成(90%)

◆ bean 的 scope 的细节

表 3.4. Bean作用域

作用域	描述
singleton	在每个Spring IoC容器中一个bean定义对应一个对象实例。
prototype	一个bean定义对应多个对象实例。
request	在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。
session	在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
global session	在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。

入门案例:

```
//获取两个 student
Student s1=(Student) ac.getBean("student");
```

```
        Student s2=(Student) ac.getBean("student");
        System.out.println(s1+" "+s2);
```

- request
- session
- global-session

是在 web 开发中才有意义.

◆ **三种获取 ApplicationContext 对象引用的方法**

1. ClassPathXmlApplicationContext -> 通过类路径
2. FileSystemXmlApplicationContext -> 通过文件路径

举例:

ApplicationContext ac=new FileSystemXmlApplicationContext(" 文 件 路 径 beans.xml / applicationContext.xml");

3. XmlWebApplicationContext

◆ **bean 的生命周期**

为什么总是一个生命周期当做一个重点?

Servlet -> servlet 生命周期 init() destory()

java 对象生命周期.

往往笔试，面试总喜欢问生命周期的问题

① 实例化(当我们的程序加载 beans.xml 文件)，把我们的 bean(前提是 scope=singleton)实例化到内存

② 调用 set 方法设置属性

③ 如果你实现了 bean 名字关注接口(BeanNameAware) 则，可以通过 setBeanName 获取 id 号

④ 如果你实现了 bean 工厂关注接口，(BeanFactoryAware),则可以获取 BeanFactory

⑤ 如果你实现了 ApplicationContextAware 接口，则调用方法

```
//该方法传递 ApplicationContext
    public void setApplicationContext(ApplicationContext arg0)
            throws BeansException {
        // TODO Auto-generated method stub
        System.out.println("setApplicationContext"+arg0);

    }
```

⑥ 如 果 bean 和 一 个 后 置 处 理 器 关 联 , 则 会 自 动 去 调 用 Object postProcessBeforeInitialization 方法

⑦ 如果你实现 InitializingBean 接口，则会调用 afterPropertiesSet

⑧ 如果自己在<bean init-method=”init” /> 则可以在 bean 定义自己的初始化方法.

⑨ 如果 bean 和 一个后置处理器关联,则会自动去调用 Object postProcessAfterInitialization 方法

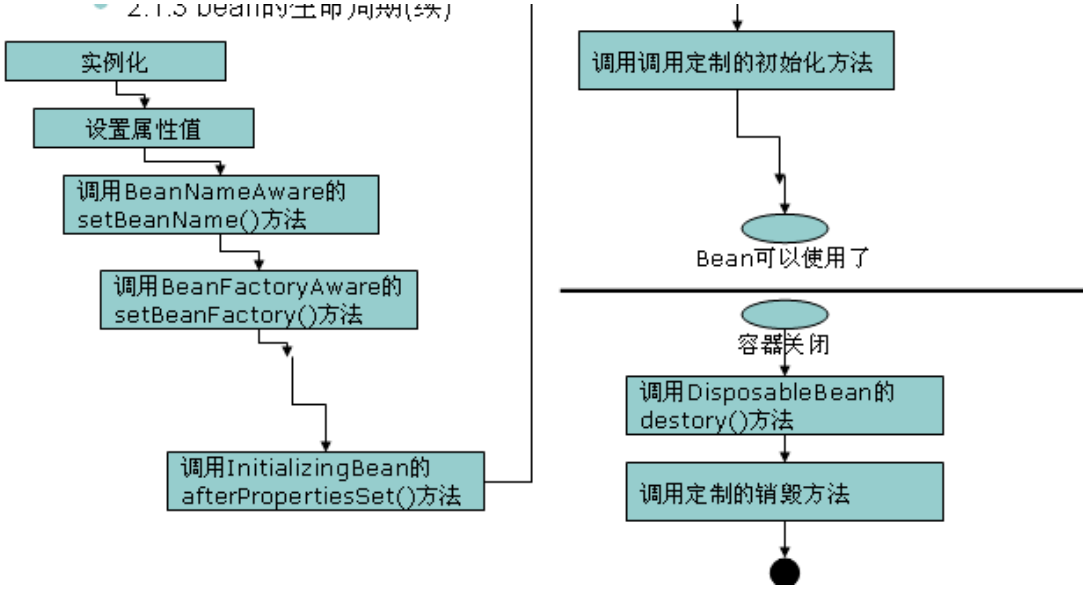
⑩ 使用我们的 bean

11. 容器关闭
12. 可以通过实现 DisposableBean 接口来调用方法 destory
13. 可以在<bean destory-method=”fun1”/> 调用定制的销毁方法

小结: 我们实际开发中往往，没有用的这么的过程,常见的是:
1->2->6->10->9->11

上机练习: 把使用每个 bean 的时间记录到一个 recoder.txt 文件 ,内容是
xxbean. 使用时间是 : 1999-11-11 11:11:11

问题:通过 BeanFactory 来获取 bean 对象，bean 的生命周期是否和 Applicationcontext 是一样吗?
不是一样的，bean 是工厂中创建的生命周期会简单一些:



◆ 配置 bean 的细节

① scope 的说明:

表 3.4. Bean作用域

作用域	描述
singleton	在每个Spring IoC容器中一个bean定义对应一个对象实例。
prototype	一个bean定义对应多个对象实例。
request	在一次HTTP请求中，一个bean定义对应一个实例；即每次HTTP请求将会有各自的bean实例， 它们依据某个bean定义创建而成。该作用域仅在基于web的Spring ApplicationContext情形下有效。
session	在一个HTTP Session中，一个bean定义对应一个实例。该作用域仅在基于web的Spring ApplicationContext情形下有效。
global session	在一个全局的HTTP Session中，一个bean定义对应一个实例。典型情况下，仅在使用portlet context的时候有效。该作用域仅在基于web的Spring ApplicationContext情形下有效。

☞ 尽量使用 scope=”singleton” ,不要使用 prototype,因为这样对我们的性能影响较大.

② 如何给集合类型注入值.

java 中主要的集合有几种: map set list / 数组

Department 类:

```
package com.hsp.collection;

import java.util.List;
import java.util.Map;
import java.util.Set;

public class Department {

    private String name;
    private String [] empName;
    private List<Employee> empList;
    private Set<Employee> empsets;
    private Map<String,Employee> empMaps;

    public Set<Employee> getEmpsets() {
        return empsets;
    }
    public void setEmpsets(Set<Employee> empsets) {
        this.empsets = empsets;
    }
    public String[] getEmpName() {
        return empName;
    }
```

```
    public void setEmpName(String[] empName) {
        this.empName = empName;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public List<Employee> getEmpList() {
        return empList;
    }
    public void setEmpList(List<Employee> empList) {
        this.empList = empList;
    }
    public Map<String, Employee> getEmpMaps() {
        return empMaps;
    }
    public void setEmpMaps(Map<String, Employee> empMaps) {
        this.empMaps = empMaps;
    }

}

//Employeel 类
package com.hsp.collection;
public class Employee {
    private String name;
    private int id;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

beans.xml 配置文件:

```
<?xml version="1.0" encoding="utf-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:context="http://www.springframework.org/schema/context"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd
                http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

<bean id="department" class="com.hsp.collection.Department">
<property name="name" value="财务部"/>
<!-- 给数组注入值 -->
<property name="empName">
    <list>
        <value>小明</value>
        <value>小明小明</value>
        <value>小明小明小明小明</value>
    </list>
</property>
<!-- 给 list 注入值 list 中可以有相当的对象 -->
<property name="empList">
    <list>
        <ref bean="emp2" />
        <ref bean="emp1"/>
        <ref bean="emp1"/>
        <ref bean="emp1"/>
        <ref bean="emp1"/>
        <ref bean="emp1"/>
        <ref bean="emp1"/>
    </list>
</property>
<!-- 给 set 注入值 set 不能有相同的对象 -->
<property name="empsets">
    <set>
        <ref bean="emp1" />
        <ref bean="emp2"/>
        <ref bean="emp2"/>
        <ref bean="emp2"/>
        <ref bean="emp2"/>
    </set>
</property>
```

```
<!-- 给 map 注入值  map 只有 key 不一样，就可以装配 value -->
<property name="empMaps">
    <map>
        <entry key="11" value-ref="emp1" />
        <entry key="22" value-ref="emp2"/>
        <entry key="33" value-ref="emp1"/>
    </map>
</property>
<!-- 给属性集合配置 -->【点 http 协议  referer 】
<property name="pp">
    <props>
        <prop key="pp1">abcd</prop>
        <prop key="pp2">hello</prop>
    </props>
</property>
</bean>
<bean id="emp1" class="com.hsp.collection.Employee">
<property name="name" value="北京"/>
<property name="id" value="1"/>
</bean>
<bean id="emp2" class="com.hsp.collection.Employee">
<property name="name" value="天津"/>
<property name="id" value="2"/>
</bean>
</beans>
```

③ 内部 bean

```
<bean id=”foo” class=”....Foo”>
    <property name=”属性”>
    <!—第一方法引用-->
    <ref bean=’bean 对象名’/>
    <!—内部 bean-->
    <bean>
    <properyt></property>
    </bean>
    </property>
</bean>
```

④ 继承配置

```
public class Student
public class Gradate extends Student

在 beans.xml 文件中体现配置
<!-- 配置一个学生对象 -->
<bean id="student" class="com.hsp.inherit.Student">
```

```
    <property name="name" value="顺平" />
    <property name="age" value="30"/>
</bean>
<!-- 配置 Grdate 对象 -->
<bean id="grdate" parent="student" class="com.hsp.inherit.Gradate">
    <!-- 如果自己配置属性 name,age,则会替换从父对象继承的数据 -->
    <property name="name" value="小明"/>
    <property name="degree" value="学士"/>
</bean>
```

思考: 目前我们都是通过 set 方式给 bean 注入值，spring 还提供其它的方式注入值，比如通过构造函数注入值!

◆ **通过构造函数注入值**

beans.xml 关键代码:

```
<!-- 配置一个雇员对象 -->
<bean id="employee" class="com.hsp.constructor.Employee">
<!-- 通过构造函数来注入属性值 -->
<constructor-arg index="0" type="java.lang.String" value="大明" />
</bean>
```

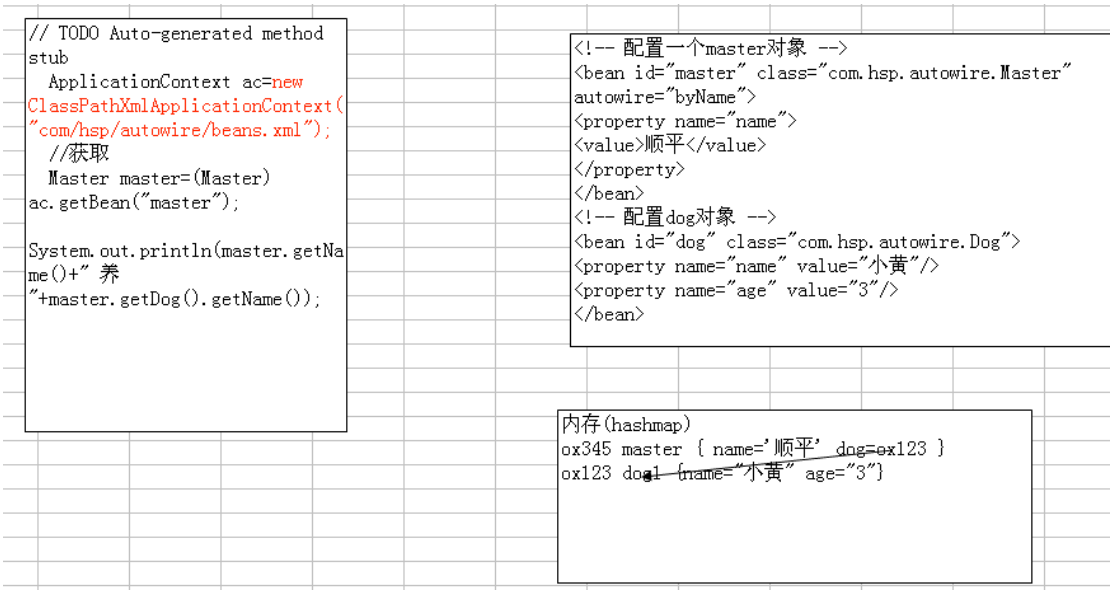
◆ 自动装配 bean 的属性值

模式	说明
no	
byName	根据属性名自动装配。此选项将检查容器并根据名字查找与属性完全一致的bean，并将其与属性自动装配。例如，在bean定义中将autowire设置为by name，而该bean包含*master*属性（同时提供*setMaster(..)*方法），Spring就会查找名为master的bean定义，并用它来装配给master属性。
byType	如果容器中存在一个与指定属性类型相同的bean，那么将与该属性自动装配。如果存在多个该类型的bean，那么将会抛出异常，并指出不能使用*byType*方式进行自动装配。若没有找到相匹配的bean，则什么事都不发生，属性也不会被设置。如果你不希望这样，那么可以通过设置dependency-check="objects"让Spring抛出异常。
constructor	与*byType*的方式类似，不同之处在于它应用于构造器参数。如果在容器中没有找到与构造器参数类型一致的bean，那么将会抛出异常。
autodetect	通过bean类的自省机制（introspection）来决定是使用*constructor*还是*byType*方式进行自动装配。如果发现默认的构造器，那么将使用*byType*方式。

(1) byName 的用法:

```
<!-- 配置一个 master 对象 -->
<bean id="master" class="com.hsp.autowire.Master" autowire="byName">
<property name="name">
<value>顺平</value>
```

</property>
</bean>
<!-- 配置 dog 对象 -->
<bean id="dog" class="com.hsp.autowire.Dog">
<property name="name" value="小黄"/>
<property name="age" value="3"/>
</bean>
原理图:



(2) byType: byType:寻找和属性类型相同的 bean,找不到,装不上,找到多个抛异常。
(3) constructor: autowire="constructor"
说明 ： 查找和 bean 的构造参数一致的一个或
多个 bean，若找不到或找到多个，抛异常。按照参数的类型装配

(4) autodetect

说明 ： autowire="autodetect"
(3)和(2)之间选一个方式。不确定
性的处理与(3)和(2)一致。
(5) defualt
这个需要在<beans defualt-autorwire="指定" />
当你在<beans >指定了 default-atuowrite 后， 所有的 bean 的 默认的 autowire 就是 指定的装配方法;
如果没有在<beans defualt-autorwire="指定" /> 没有 defualt-autorwire="指定" ，则默认是 defualt-autorwire="no"
(6) no: 不自动装配

使用 spring 的特殊 bean,完**成分散配置:**

beans.xml
说明: 当通过 context:property-placeholder 引入 属性文件的时候，有多个需要使用 , 号间隔.

```
<!-- 引入我们的 db.properties 文件 -->
<context:property-placeholder
location="classpath:com/hsp/dispatch/db.properties,classpath:com/hsp/dispatch/db2.properties"/>
<!-- 配置一 DBUtil 对象 $占位符号 -->
<bean id="dbutil" class="com.hsp.dispatch.DBUtil">
<property name="name" value="${name}" />
<property name="drivername" value="${drivername}" />
<property name="url" value="${url}" />
<property name="pwd" value="${pwd}" />
</bean>

<!-- 配置一 DBUtil 对象 -->
<bean id="dbutil2" class="com.hsp.dispatch.DBUtil">
<property name="name" value="${db2.name}" />
<property name="drivername" value="${db2.drivername}" />
<property name="url" value="${db2.url}" />
<property name="pwd" value="${db2.pwd}" />
</bean>
```

db.properties:

```
name=scott
drivername=oracle:jdbc:driver:OracleDirver
url=jdbc:oracle:thin:@127.0.0.1:1521:hsp
pwd=tiger
```

◆ aop 编程

aop(aspect oriented programming) 面向切面(方面)编程,是对所有对象或者是一类对象编程,核心是(在不增加代码的基础上， 还增加新功能)

汇编(伪机器指令 mov jump) 面向机器
c 语言(面向过程)->系统软件(操作系统，数据库, 杀毒软件，防火墙,驱动..)

语句 1;
语句 2;
...

java 语法(面向对象->类-对象)

class Dog{
    属性;->变量
    行为->函数
}

面向切面 spring(->aop) 面向 n 多对象编程

aop 特别提醒: aop 编程，实际上在开发框架本身用的多,在实际项目中，用的不是很多,但是将来会越来越多，这个一个趋势.

◆ aop 原理+案例

编程说明:
步骤:
1. 定义接口
2. 编写对象(被代理对象=目标对象)
3. 编写通知（前置通知目标方法调用前调用）
4. 在 beans.xml 文件配置
4.1 配置 被代理对象=目标对象
4.2 配置通知
4.3 配置代理对象 是 ProxyFactoryBean 的对象实例
4.3.1 <!-- 代理接口集 -->
4.3.2 织入通知
4.3.3 配置被代理对象

后面还后置通知，环绕通知，异常通知，引入通知
上机: 你把老师写的代码看看，走一遍。

提问？说 spring 的 aop 中，当你通过代理对象去实现 aop 的时候，获取的 ProxyFactoryBean 是什么类型？
答: 返回的是一个代理对象,如果目标对象实现了接口，则 spring 使用 jdk 动态代理技术,如果目标对象没有实现接口，则 spring 使用 CGLIB 技术.

提一个问题

```
class A{
        private Strinag name;
        public viod setName(String name){
        this.name=name;
        System.out.println(“name”+name);
        }
}
beans.xml
<bean id=”a” class=”...A”>
<property name=”name” value=”顺平” />
</bean>
A a=new A();
a.setName(“顺平”);
```