# Spring基础

## 认识Spring

### Spring概述

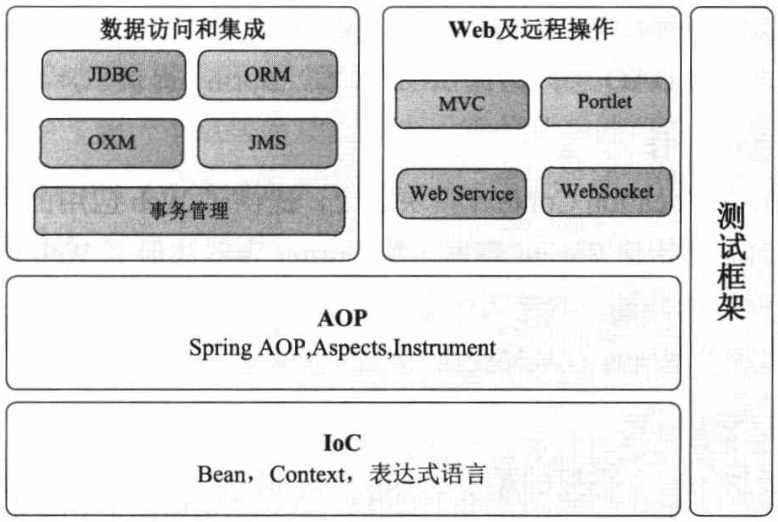
Spring是众多Java开源项目中的一员，它已经成为Java应用首选的full-stack开发框架，它本着“从实践中来，到实践中去”的原则，对传统EJB重量型框架的思想进行了颠覆性的革新。Spring是分层的Java SE/EE应用一站式的轻量级开源框架，以IoC（Inverse of Control，控制反转）和AOP（Aspect Oriented Programming，切面编程）为核心，提供了展现层Spring MVC、持久层Spring JDBC以及业务层事务管理等一站式的企业级应用技术。此外Spring以海纳百川的胸怀整合了开源世界里众多著名的第三方框架和类库，逐渐成为了使用最多的轻量级Java EE企业应用开源框架。

### Spring的优点

1. 方便解耦，简化开发。通过Spring提供的IoC容器，用户可以将对象之间的依赖关系交由Spring进行控制，避免硬编码所造成的过度程序耦合。有了Spring，用户不必再为单实例模式类、属性文件解析等这些底层的需求编写代码，可以更专注于上层的应用。
2. AOP编程的支持。通过Spring提供的AOP功能，方便进行面向切面的编程，很多不容易用传统OOP实现的功能可以通过AOP轻松应对。
3. 声明是事务的支持。在Spring中，用户可以从单调烦闷的事务管理代码中解脱出来，通过声明的方式灵活地进行事务管理，提高开发效率和质量。
4. 方便程序的测试。可以用非容器依赖的编程方式进行几乎所有的测试工作。在Spring里，测试不再是昂贵的操作，而是随手可做的事情。
5. 方便集成各种优秀框架。Spring不排斥各种优秀的开源框架，相反，Spring可以降低各种框架的使用难度。Spring提供了对各种优秀框架（如Struts、Hibernate、Hessian、Quartz等）的直接支持。
6. 降低Java EE API的使用难度。Spring对很多难用的Java EE API（如JDBC、JavaMail、远程调用等）提供了一个薄层封装，通过Spring的简易封装，这些Java EE API的使用难度大大降低。
7. Java源码是经典的学习范例。Spring的源码设计精妙、结构清晰、匠心独运，处处体现着大师对Java设计模式的灵活运用及对Java技术的高深造诣。Spring框架源码无疑是Java技术的最佳实践范例。如果想在短时间内迅速提高自己的Java技术水平和应用开发水平，学习和研究Spring源码将会收到意想不到的效果。

### Spring体系结构

Spring核心框架由4000个类组成，整个框架按其所属功能可以划分为5个主要模块。从整体来看，这5个主要模块几乎为企业应用提供了所需要的一切，从持久层、业务层到展现层都拥有相应的支持。



1. **IoC**

Spring核心模块是实现了IoC的功能，它将类与类之间的依赖从代码中脱离出来，用配置的方式进行依赖关系描述，由IoC容器负责依赖类之间的创建、拼接、管理、获取等工作。BeanFactory接口是Spring框架的核心接口，它实现了容器许多核心的功能。

Context模块构建于核心模块之上，扩展了BeanFactory的功能，添加了i18n国际化，Bean生命周期控制、框架事件体系、资源加载透明化等多项功能。此外，该模块还提供了许多企业级服务的支持，如邮件服务、任务调度、JNDI获取、EJB集成、远程访问等。ApplicationContext是Context模块的核心接口。

表达式语言模块是统一表达式语言（Unified EL）的一个扩展，该表达式语言用于查询和管理运行期的对象，支持设置/获取对象属性，调用对象方法，操作数组、集合等。此外，该模块还提供了逻辑表达式运算、变量定义等功能，可以方便地通过表达式串和Spring IoC容器进行交互。

1. **AOP**

AOP是继OOP之后，对编程设计思想影响极大的技术之一。AOP是进行横切逻辑编程的思想，它开拓了考虑问题的思路。在AOP模块里，Spring提供了满足AOP Alliance规范的实现，还整合了AspectJ这种AOP语言级的框架。在Spring里实现AOP编程有众多选择。Java5.0引入java.lang.instrument，允许在JVM启动时启用一个代理类，通过该代理类在运行期间修改类的字节码，改变一个类的功能，从而实现AOP的功能。

1. **数据访问和集成**

任何应用程序的核心问题是对数据的访问和操作。数据有多种表现形式，如数据表、XML、消息等，而每种数据形式又拥有不同的数据访问技术（如数据表的访问既可以直接通过JDBC，也可以通过Hibernate或MyBatis）。

首先，Spring站在DAO的抽象层面，建立了一套面向DAO层的同一的异常体系，同时将各种访问数据的检查型异常转换为非检查型异常，为整合各种持久化层框架提供基础。其次，Spring通过模板化技术对各种数据访问技术进行了薄层封装，将模式化的代码隐藏起来，使数据访问的程序得到大幅简化。这样Spring就建立起了和数据形式及访问技术无关的统一的DAO层，借助AOP技术，Spring提供了声明式事务的功能。

1. **Web及远程操作**

该模块建立在Application Context模块之上，提供了Web应用的各种工具类，如通过Listener或Servlet初始化Spring容器，将Spring容器注册到Web容器中。该模块还提供了多项面向Web的功能，如透明化文件上传、Velocity、FreeMarker、SXLT的支持。此外，Spring可以整个Struts、WebWork等MVC框架。

1. **Web及远程访问**

Spring自己提供了一个完整的类似于Struts的MVC框架，称为Spring MVC。当然，如果不希望使用Spring MVC，那么Spring对Struts、WebWork等MVC框架的整合，一定也可以带来方便。相对于Servlet的MVC，Spring在简化Portlet的开发商也做了很多工作。

1. **WebSocket**

WebSocket提供了一个在Web应用中该校、双向的通信，需要考虑到客户端和服务器之间的高频率和低时延消息交换。一般的应用场景有在线交易、游戏、协作、数据可视化等。

## Spring Web项目实例

本节通过一个简单的登录功能模块例子展现开发Spring Web应用的整体过程。实例应用按持久层、业务层和展现层进行组织，从底层DAO到Web展现，一步步地搭建起一个完整的实例。

* **实例功能简介**

登录模块的功能很简单，首先登录页面提供一个带用户名/密码的输入表单，用户填写并提交表单后，服务器端程序检查是否有匹配的用户名/密码。如果用户名/密码不匹配，则返回登录页面，并给出提示；如果用户名/密码匹配，则记录用户的成功登录日志，更新用户的最后登录时间和IP，并给用户增加5个积分，然后重定向到欢迎页面。

在持久层拥有两个DAO类，分别是UserDao和LoginLogDao，在业务层对应一个业务类UserService。在展现层拥有一个LoginController类和两个JSP页面，分别是登录页面login.jsp和欢迎页面main.jsp。

* **环境准备**

在进入实例的具体开发之前，需要做一些环境的准备工作，其中包括数据库表的创建、项目工程的创建、规划Spring配置文件等。

1. **建立工程**

本实例使用Maven来构建项目，管理项目中的依赖关系。在IDE中新创建一个Maven工程，在pom.xml配置文件中配置好项目中所需要的jar包。

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>com.uc</groupId>

<artifactId>Spring</artifactId>

<packaging>war</packaging>

<version>0.0.1-SNAPSHOT</version>

<name>Spring Maven Webapp</name>

<url>http://maven.apache.org</url>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.12</version>

<scope>test</scope>

</dependency>

<!-- 依赖的Spring模块类库 -->

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-core</artifactId>

<version>5.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-beans</artifactId>

<version>5.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-jdbc</artifactId>

<version>5.0.0.RELEASE</version>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.0.0.RELEASE</version>

</dependency>

<!-- 依赖的数据库驱动类库 -->

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.40</version>

</dependency>

<!-- 依赖的连接池驱动类库 -->

<dependency>

<groupId>commons-dbcp</groupId>

<artifactId>commons-dbcp</artifactId>

<version>1.4</version>

</dependency>

<!-- 依赖的Web类库 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

<version>1.2</version>

</dependency>

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>servlet-api</artifactId>

<version>3.0-alpha-1</version>

</dependency>

</dependencies>

<build>

<finalName>Spring</finalName>

</build>

</project>

<artifactId>servlet-api</artifactId>

<version>3.0-alpha-1</version>

</dependency>

</dependencies>

<build>

<finalName>Spring</finalName>

</build>

</project>

1. **创建库表**

运行如下脚本，创建实例对应的数据库。

DROP DATABASE IF EXISTS sampledb;

CREATE DATABASE sampledb DEFAULT CHARACTER SET utf-8;

USE sampledb;

数据库名为sampledb，默认字符集采用UTF-8。

创建实例所用的两张表：

#创建用户表

CREATE TABLE t\_user(

user\_id INT AUTO\_INCREMENT PRIMARY KEY,

user\_name VARCHAR(30),

credits INT,

password VARCHAR(32),

last\_visit datetime,

last\_ip VARCHAR(23)

)ENGINE=INNODB;

#创建用户登录日志表

CREATE TABLE t\_login\_log(

login\_log\_id INT AUTO\_INCREMENT PRIMARY KEY,

user\_id INT,

ip VARCHAR(23),

login\_datetime datetime

)ENGINE=INNODB;

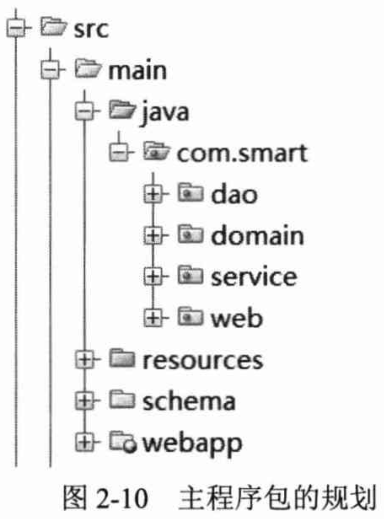
初始化一条数据，用户名/密码为admin/123456。

USE sampledb;

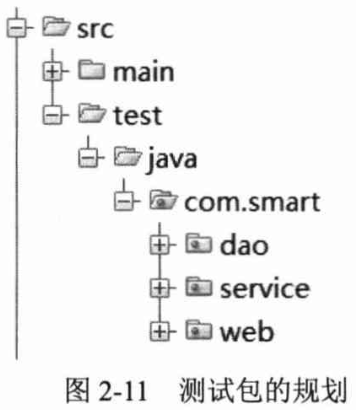
INSERT INTO t\_user(user\_name,password) VALUES('admin','123456');

1. **类包及Spring配置文件规划**
2. 类包规划

类包以分层的方式进行组织，共划分为4个包，分别是dao、domain、service和web。



单元测试的类包和程序的类包对应，但放置在不同的文件夹下。本实例仅对业务层的业务类进行单元测试。将程序类和测试类放置在物理不同的文件夹下，方便将来程序的打包盒分发，因为测试类仅在开发时有用，无须包含到部署包中。



1. Spring配置文件规划

Spring可以将所有的配置信息统一到一个文件中，也可以放置到多个文件中。对于简单的应用来说，由于配置信息少，仅用一个配置文件就足够应付。随着应用规模的扩大、配置信息量的增多，仅使用一个配置文件往往难以满足要求。如果不进行仔细规划，则将给配置信息的查看和团队协作带来负面影响。

配置文件在团队协作时是资源争用的焦点，对于大型应用一般要按模块进行划分，以在一定程度上降低争用，减少团队协作的版本控制冲突。由于我们的应用比较小，所以直接采用一个配置文件spring-context.xml即可。

* **持久层**

持久层负责数据的访问和操作，DAO类被上层的业务类调用。Spring本身支持多种流行的ORM框架，这里使用Spring JDBC作为持久层的实现技术。

1. **建立实体类**

实体类代表业务的状态，且贯穿展示层、业务层和持久层，并最终被持久化到数据库中。实体类使数据库表操作以面向对象的方式进行，为程序扩展带来了更大的灵活性。实体类不一定等同于数据库表，不过对于简单的应用来说，实体类对象往往拥有对应的数据库表。

持久层的主要工作就是从数据库表中加载数据并实例化实体类对象，或将实体类对象持久化到数据库表中。该实例中需要两个实体类：User和LoginLog，前者代表用户信息，后者代表日志信息，分别对应t\_user和t\_login\_log数据库表，实体类的包为com.smart.domain。

User.java

**package** com.smart.domain;

**import** java.io.Serializable;

**import** java.util.Date;

**public** **class** User **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** userId;

**private** String userName;

**private** String password;

**private** **int** credits;

**private** String lastIp;

**private** Date lastVisit;

//省略getXxx/setXxx方法

...

}

LoginLog.java

**package** com.smart.domain;

**import** java.io.Serializable;

**import** java.util.Date;

**public** **class** LoginLog **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** loginLogId;

**private** **int** userId;

**private** String ip;

**private** Date loginDate;

//省略getXxx/setXxx方法

...

}

实体类一般要实现Serializable接口，以方便可序列化。

1. **UserDao**

首先来定义访问User的DAO，它包括3个方法。

getMatchCount()：根据用户名和密码获取匹配的用户数。等于1表示用户名/密码正确；等于0表示用户名或密码错误（这是最简单的用户身份认证方法，在实际应用中需要采用诸如密码加密等安全策略）。

findUserByUserName()：根据用户名获取User对象。

updateLoginInfo()：更新用户积分、最后登录IP及最后登录时间。

下面通过Spring JDBC技术实现这个DAO类。

**package** com.smart.dao;

**import** java.util.List;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.stereotype.Repository;

**import** com.smart.domain.User;

@Repository //通过Spring注解定义 一个DAO

**public** **class** UserDao {

**private** JdbcTemplate jdbcTemplatel;

@Autowired //自动注入JdbcTemplate的Bean

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplatel = jdbcTemplate;

}

**public** **int** getMatchCount(String userName, String password) {

String sqlStr = "SELECT COUNT(\*) FROM t\_user WHERE user\_name=? AND password=?";

List<User> useres = jdbcTemplatel.queryForList(sqlStr, **new** Object[] {userName, password}, User.**class**);

**return** useres.size();

}

...

}

在Spring2.5以后，可以使用注解的方式定义Bean。较之于XML配置方式，注解配置方法的简单性非常明显。这里我们用@Repository定义了一个DAO Bean，使用@Autowired将Spring容器中的Bean注入进来。

传统的JDBC API太底层，即使用户执行一条最简单的数据查询操作，都必须执行如下过程：获取连接🡪创建Statement🡪执行数据操作🡪获取结果🡪关闭Statement🡪关闭结果🡪关闭连接，除此之外还需要进行异常处理的操作。如果使用传统的JDBC API进行数据访问操作，则可能会产生1/3以上单调乏味的重复代码。

Spring JDBC对传统的JDBC API进行了薄层封装，将样板式的代码和哪些必不可少的代码进行了分离，用户仅需要编写哪些必不可少的代码，剩余的哪些单调乏味的重复性工作则交由Spring JDBC框架处理。Spring JDBC通过一个模板类org.springframework.jdbc.core.JdbcTemplate封装了样板式的代码，用户通过模板类就可以轻松地完成大部分数据访问操作。

通过JdbcTemplate的支持，我们可以轻松地实现UserDao的另外两个方法。

@Repository //通过Spring注解定义 一个DAO

**public** **class** UserDao {

...

//根据用户名查询用户的sql语句

**private** **final** **static** String ***MATCH\_COUNT\_SQL*** = "SELECT COUNT(\*) FROM t\_user WHERE user\_name=? AND password=?";

**private** **final** **static** String ***UPDATE\_LOGIN\_INFO\_SQL*** = "UPDATE t\_user SET last\_visit=?,last\_ip=?,credits=? WHERE user\_id=?";

**public** User findUserByUserName(**final** String userName) {

**final** User user = **new** User();

jdbcTemplatel.query(***MATCH\_COUNT\_SQL***, **new** Object[] {userName, **new** RowCallbackHandler() {

**public** **void** processRow(ResultSet rs)**throws** SQLException{

user.setUserId(rs.getInt("user\_id"));

user.setUserName(userName);

user.setCredits(rs.getInt("credits"));

}

}});

**return** user;

}

**public** **void** updateInfo(User user) {

jdbcTemplatel.update(***UPDATE\_LOGIN\_INFO\_SQL***, **new** Object[] {user.getLastVisit(), user.getLastIp(), user.getCredits(), user.getUserId()});

}

}

findUserByUserName()方法稍微复杂一些。这里，我们使用了JdbcTemplate的query()方法，该方法的签名为query(String sql,Object[] args,RowCallbackHandler rch)，它有3个入参。

sql：查询的SQL语句，允许使用带“?”的参数占位符。

args：SQL语句中占位符对应的参数数组。

RowCallbackHandler：查询结果的处理回调接口，该回调接口有一个方法processRow(ResultSet rs)，负责将查询的结果从ResultSet装载到类似于实体类的对象实例中。

updateLoginInfo()方法比较简单，主要通过JdbcTemplate的update(String sql,Object[] args)进行数据的更新操作。

1. **LoginLogDao**

LoginLogDao负责记录用户的登录日志，它仅有一个insertLoginLog()方法。与UserDao相似，其实现类也通过JdbcTemplate的update(String sql,Object[] args)方法完成登录日志插入的操作。

**package** com.smart.dao;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.jdbc.core.JdbcTemplate;

**import** org.springframework.stereotype.Repository;

**import** com.smart.domain.LoginLog;

@Repository

**public** **class** LoginLogDao {

**private** JdbcTemplate jdbcTemplate;

//保存登录日志SQL

**private** **final** **static** String ***INSERT\_LOGIN\_LOG\_SQL*** = "INSERT INTO t\_login\_log(user\_id,ip,login\_datetime)VALUES(?,?,?)";

**public** **void** insertLoginLog(LoginLog loginLog) {

Object[] args = {loginLog.getUserId(),loginLog.getIp(),loginLog.getLoginDate()};

jdbcTemplate.update(***INSERT\_LOGIN\_LOG\_SQL***, args);

}

@Autowired

**public** **void** setJdbcTemplate(JdbcTemplate jdbcTemplate) {

**this**.jdbcTemplate = jdbcTemplate;

}

}

1. **在Spring中装配DAO**

在编写DAO接口的实现类时有一个问题：在以上两个DAO实现类中都没有打开/释放Connection的代码，DAO类究竟如何访问数据库呢？前面介绍过，样板式的操作都被JdbcTemplate封装起来了，JdbcTemplate本身需要一个DataSource，这样它就可以根据需要从DataSource中获取或返回连接。UserDao和LoginLog都提供了一个带@Autowired注解的JdbcTemplate变量，所以我们必须事先声明一个数据源，然后定义一个JdbcTemplate Bean，通过Spring的容器上下文自动绑定机制进行Bean的注入。

在项目工程的src/resources目录下创建一个名为smart-context.xml的Spring配置文件。该配置文件的基本结构如下：

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"* >

...

</beans>

在IDE中，刷新工程目录树，在resources文件夹下即可看到该配置文件。双击smart-context.xml文件，在这个文件中添加如下代码：

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"* >

<!-- 1.扫描类包，将标注Spring注解的类自动转化为Bean,同时完成Bean的注入 -->

<context:component-scan base-package=*"com.smart.dao"* />

<!-- 2.定义一个使用DBCP实现的数据源 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*

p:driverClassName=*"com.mysql.jdbc.Driver"*

p:url=*"jdbc:mysql://localhost:3306/sampledb"*

p:username=*"root"*

p:password=*"123456"* />

<!-- 3.定义JDBC模板Bean -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"* p:dataSource-ref=*"dataSource"* />

</beans>

在1处我们使用Spring的<context:component-scan>扫描指定类包下的所有类，这样在类中定义的Spring注解（如@Repository、@Autowired等）才能产生作用。在2处，我们使用Jakarta的DBCP开源数据源实现方案定义了一个数据源，数据库驱动类为com.mysql.jdbc.Driver。数据库的服务端口号为3306。在3处配置了JdbcTemplate，将2处声明的dataSource注入JdbcTemplate中，而这个JdbcTemplate Bean将通过@Autowired自动注入LoginLogDao和UserDao的Bean中。

* **业务层**

业务层仅哟一个业务类，即UserService。UserService负责将持久层UserDao和LoginLogDao组织起来，完成用户名/密码认证、登录日志记录等操作。

1. **UserService**

UserService业务类有3个业务方法，其中hasMatchUser()方法用于检查用户名/密码的正确性；findUserByUserName()方法以用户名为条件加载User对象；loginSuccess()方法在用户登录成功后调用，更新用户最后登录时间和IP信息，同时记录用户登录日志。

**package** com.smart.service;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** org.springframework.transaction.annotation.Transactional;

**import** com.smart.dao.LoginLogDao;

**import** com.smart.dao.UserDao;

**import** com.smart.domain.LoginLog;

**import** com.smart.domain.User;

@Service //将UserService标注为一个服务层的Bean

**public** **class** UserService {

**private** UserDao userDao;

**private** LoginLogDao loginLogDao;

@Autowired //将UserDao注入进来

**public** **void** setUserDao(UserDao userDao) {

**this**.userDao = userDao;

}

@Autowired //将LoginLogDao注入进来

**public** **void** setLoginLogDao(LoginLogDao loginLogDao) {

**this**.loginLogDao = loginLogDao;

}

**public** User findUserByUserName(String userName) {

**return** userDao.findUserByUserName(userName);

}

@Transactional //将该方法标注为事务注解，让其运行在事务环境中

**public** **void** loginSuccess(User user) {

user.setCredits(5 + user.getCredits());

LoginLog loginLog = **new** LoginLog();

loginLog.setUserId(user.getUserId());

loginLog.setIp(user.getLastIp());

loginLog.setLoginDate(user.getLastVisit());

userDao.updateInfo(user);

loginLogDao.insertLoginLog(loginLog);

}

}

1. **在Spring中装配Service**

事务管理的代码虽然无需出现在程序代码中，但我们必须以某种方式告诉Spring哪些业务类需要工作在事务环境下及事务的规则等内容，以便Spring根据这些信息自动为目标业务类添加事务管理的功能。

打开原来的smart-context.xml文件，修改配置文件信息。

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"* >

<!-- 扫描类包，将标注Spring注解的类自动转化为Bean,同时完成Bean的注入 -->

<context:component-scan base-package=*"com.smart.dao"* />

<!-- 扫描service类包，应用Spring的注解配置 -->

<context:component-scan base-package=*"com.smart.service"* />

<!-- 定义一个使用DBCP实现的数据源 -->

<bean id=*"dataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*

p:driverClassName=*"com.mysql.jdbc.Driver"*

p:url=*"jdbc:mysql://localhost:3306/sampledb"*

p:username=*"root"*

p:password=*"123456"* />

<!-- 定义JDBC模板Bean -->

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"* p:dataSource-ref=*"dataSource"* />

<!-- 配置事务管理器 -->

<bean id=*"transactionManager"* class=*"org.springframework.jdbc.datasource.DataSourceTransactionManager"* p:dataSource-ref=*"dataSource"* />

<!-- 通过AOP配置提供事务增强，让service包下所有Bean的所有方法拥有事务 -->

<aop:config proxy-target-class=*"true"*>

<aop:pointcut id=*"serviceMethod"* expression=*"(execution(\* com.smart.service..\*(..))) and (@annotation(org.springframework.transaction.annotation.Transaction))"*/>

<aop:advisor pointcut-ref=*"serviceMethod"* advice-ref=*"txAdvice"*/>

</aop:config>

<tx:advice id=*"txAdvice"* transaction-manager=*"transactionManager"*>

<tx:attributes>

<tx:method name=*"\*"*/>

</tx:attributes>

</tx:advice>

</beans>

* **展示层**

1. **配置Spring MVC框架**

首先需要对web.xml文件进行配置，以便Web容器启动时能够自动启动Spring容器。

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app version=*"3.0"* 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"*>

<display-name>Archetype Created Web Application</display-name>

<!-- 1.从类路径下加载Spring配置文件，classpath关键字特指类路径下加载 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:smart-context.xml</param-value>

</context-param>

<!-- 2.负责启动Spring容器的监听器，它将引用1处的上下文参数获得Spring配置文件的地址 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

…

</web-app>

然后通过Web容器上下文参数指定Spring配置文件的地址，如1所示。多个配置文件可以用逗号或空格分隔。然后再2处指定Spring所提供的ContextLoaderListener的Web容器监听器，该监听器在Web容器启动时自动运行，它会根据contextConfigLocation Web容器参数获取Spring配置文件，并启动Spring容器。注意，需要将log4J.properties日志配置文件放置在类路径下，以便日志引擎自动生效。

最后需要配置Spring MVC相关的信息。Spring MVC像Struts一样，也通过一个Servlet来获取URL请求，然后再进行相关的处理。

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app version=*"3.0"* 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"*>

<display-name>Archetype Created Web Application</display-name>

<!-- 1.从类路径下加载Spring配置文件，classpath关键字特指类路径下加载 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:smart-context.xml</param-value>

</context-param>

<!-- 2.负责启动Spring容器的监听器，它将引用1处的上下文参数获得Spring配置文件的地址 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

<!-- 3.Spring MVC的主控Servlet -->

<servlet>

<servlet-name>smart</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>2</load-on-startup>

</servlet>

<!-- 4.Spring MVC处理的URL -->

<servlet-mapping>

<servlet-name>smart</servlet-name>

<url-pattern>\*.html</url-pattern>

</servlet-mapping>

</web-app>

在3处声明了一个Servlet，Spring MVC也拥有一个Spring配置文件，该配置文件的文件名和此处定义的Servlet名有一个契约，即采用Servlet名-servlet.xml的形式。在这里Servlet名为smart，则在/WEB-INF目录下必须提供一个名为smart-servlet.xml的Spring MVC配置文件，但这个配置文件无需通过web.xml的contextConfigLocation上下文参数进行声明，因为Spring MVC的Servlet会自动将smart-servlet.xml文件和Spring的其他配置文件进行拼装。

在4处对这个Servlet的URL路径映射进行定义，在这里让所有以.html为后缀的URL都能被smart Servlet截获，进而转由Spring MVC框架进行处理。请求给Spring MVC截获后，首先根据请求的URL查找到目标的处理控制器，并将请求参数封装“命令”对象一起传给控制器处理；然后控制器调用Spring容器中的业务Bean完成业务处理工作并返回结果视图。

1. **处理登录请求**
2. **POJO控制器类**

首先需要编写的是LoginController，负责处理登录请求，完成登录业务，并根据登录成功与否转向欢迎页面或失败页面。

**package** com.smart.web;

**import** java.util.Date;

**import** javax.servlet.http.HttpServletRequest;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Controller;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.servlet.ModelAndView;

**import** com.smart.domain.User;

**import** com.smart.service.UserService;

@Controller //1.标注成为一个Spring MVC的Controller

**public** **class** LoginController {

**private** UserService userService;

//2.负责处理/index.html的请求

@RequestMapping(value="/index.html")

**public** String loginPage() {

**return** "login";

}

//3.负责处理/loginCheck.html的请求

@RequestMapping(value="/loginCheck.html")

**public** ModelAndView loginCheck(HttpServletRequest request, LoginCommand loginCommand) {

**boolean** isValidUser = userService.hasMatchUser(loginCommand.getUserName(), loginCommand.getPassword());

**if**(!isValidUser) {

**return** **new** ModelAndView("login", "error", "用户名或密码错误。");

}**else** {

User user = userService.findUserByUserName(loginCommand.getUserName());

user.setLastIp(request.getLocalAddr());

user.setLastVisit(**new** Date());

userService.loginSuccess(user);

request.getSession().setAttribute("user", user);

**return** **new** ModelAndView("main");

}

}

@Autowired

**public** **void** setUserService(UserService userService) {

**this**.userService = userService;

}

}

在1处通过Spring MVC的@Controller注解可以将任何一个POJO的类标注为Spring MVC的控制器，处理HTTP的请求。当然，标注了@Controller的类首先会是一个Bean，所以可以使用@Autowired进行Bean注入。

一个控制器可以拥有多个处理映射不同HTTP请求路径的方法，通过@RequestMapping指定方法如何映射请求路径，如2和3处所示。

请求参数会根据参数名称默认契约自动绑定到相应方法的入参中。例如在3处的loginCheck(HttpServletRequest request, LoginCommand loginCommand)方法中，请求参数会按名称匹配绑定到loginCommand的入参中。

请求响应方法可以返回一个ModelAndView，或直接返回一个字符串，Spring MVC会解析并转向目标响应页面。

1. **Spring MVC配置文件**

编写好LoginController后需要在smart-servlet.xml中声明控制器，扫描Web路径，指定Spring MVC的视图解析器。

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"* >

<!-- 1.扫描Web包，应用Spring的注解 -->

<context:component-scan base-package=*"com.smart.web"* />

<!-- 2.配置视图解析器，将ModelAndView及字符串解析为具体的页面 -->

<bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*

p:viewClass=*"org.springframework.web.servlet.view.JstlView"*

p:prefix=*"WEB-INFF/jsp"*

p:suffix=*".jsp"*

/>

</beans>

1. **JSP视图页面**

登录模块共包含两个JSP页面，分别是登录页面/WEB-INF/jsp/login.jsp和欢迎页面/WEB-INF/jsp/main.jsp。

1. 登录页面login.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

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

<title>登录页面</title>

</head>

<body>

<c:if test=*"*${!**empty** error}*"*>

<font color=*"red"*><c:out value=*"*${error}*"* /></font>

</c:if>

<form action=*"*<c:url value=*"/ loginCheck.html "*/>*"* 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"* />

<input type=*"reset"* value=*"重置"* />

</form>

</body>

</html>

1. 欢迎页面main.jsp

<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*

pageEncoding=*"UTF-8"*%>

<%@ taglib uri=*"http://java.sun.com/jsp/jstl/core"* prefix=*"c"* %>

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>

<head>

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

<title>欢迎页面</title>

</head>

<body>

${user.userName}，欢迎进入\*\*\*欢迎页面，您当前积分为${user.credits};

</body>

</html>

# Spring核心

## IoC容器

### IoC概述

IoC（Inverse of Control，控制反转）是Spring容器的内核，AOP、声明式事务等功能都建立在IoC基础之上。

IoC是随着近年来轻量级容器（Lightweight Container)的兴起而逐渐被很多人提起的一个名词，它的全称为Inversion of Control，中文通常翻译为“控制反转”，它还有一个别名叫做依赖注入（DependencyInjection）。它不是一种编程语言，而是一种编程方式。也是对面向对象的一种补充完善。它实现了解耦各业务对象间依赖关系的对象绑定方式。

为什么需要IoC，IoC的具体意义是什么？

在以往的Java开发中，如果在一个类中的属性依赖于另一个类，那么我们会在该类中定义一个依赖类的引用。当我们需要使用依赖类中的方法或数据的时候，必须显示 的通过new关键字实例化依赖类的对象，才能去使用依赖类所提供的服务。回头想想，这是否真的必要？我们最终所要做的，其实就是直接调用依赖对象所提供的某项服务而已。只要用到这个依赖对象的时候，它能够准备就绪，我们完全可以不管这个对象是如何创建的。

IoC就是为了帮助我们避免之前的“大费周折”，而提供了更加轻松简洁的方式。它的反转，就是反转在让你从原来的通过new关键字实例化对象后才能使用对象的服务，转变为直接使用对象所提供的服务，无需自己实例化对象，实例化对象的工作直接交由IoC框架来完成。

通常情况下，被注入对象会直接依赖于被依赖对象。但是在IoC的场景中，二者之间通过IoC Service Provider来打交道，所有的被注入对象和依赖对象现在由IoC Service Provider统一管理。被注入对象需要什么，直接通过IoC Service Provider来获取，后者就会把相应的被依赖对象注入到被注入对象中，从而达到IoC Service Provide为被注入对象服务的目的。IoC Service Provider在这里就是通常的IoC容器所充当的角色。从被注入对象的角度看，与之前直接寻求依赖对象相比，依赖对象的取得方式发生了反转，控制也从被注入对象转到了IoC Service Provider里。

### IoC的类型

从注入方法上看，IoC主要可以划分为3种类型：构造函数注入、属性注入和接口注入。Spring支持构造函数注入和属性注入。

#### 构造函数注入

在构造函数注入中，通过调用类的构造函数，将接口实现类通过构造函数变量传入。构造函数不必关心传入的具体对象是什么。

**public** **class** MoAttack {

**private** User user;

//构造函数注入

**public** MoAttack(User user) {

**this**.user = user;

}

**public** **void** sayName() {

user.getUserName();

}

}

#### 属性注入

属性注入是通过JavaBean的Setter方法实现调用类所需要依赖的注入，这种方式更加灵活方便。

**public** **class** MoAttack {

**private** User user;

//调用属性的Setter方法注入

**public** **void** setUser(User user) {

**this**.user = user;

}

**public** **void** sayName() {

user.getUserName();

}

}

#### 接口注入

接口注入是将调用类所有依赖注入的方式抽取到一个接口中，调用类通过实现该接口提供相应的注入方法。

定义一个实现依赖关系的接口

**public** **interface** ActorArrangable {

**void** setUser(User user);

}

定义一个接口的实现类

**public** **class** MoAttack **implements** ActorArrangable {

**private** User user;

//实现接口的方法

@Override

**public** **void** setUser(User user) {

**this**.user = user;

}

**public** **void** sayName() {

user.getUserName();

}

}

由于通过接口注入需要额外声明一个接口，增加了类的数目，而且它的效果和属性注入并无本质区别，因此不提倡采用这种注入方式。

#### 三种注入方式的比较

1. 接口注入：从注入方式的使用上来说，接口注入是现在不甚提倡的一种方式，基本处于“退役状态”。因为它强制被注入对象实现不必要的接口，带有侵入性。而构造方法注入和setter方法注入则不需要如此。
2. 构造方法注入：这种注入方式的优点就是，对象在构造完成之后，即已进入就绪状态，可以马上使用。缺点就是，当依赖对象比较多的时候，构造方法的参数列表会比较长。而通过反射构造对象的时候，对相同类型的参数的处理会比较困难，维护和使用上也比较麻烦。而且在Java中，构造方法无法被继承，无法设置默认值。对于非必须的依赖处理，可能需要引入多个构造方法，而参数数量的变动可能造成维护上的不便。
3. setter方法注入：因为方法可以命名，所以setter方法注入在描述性上要比构造方法注入好一些。另外，setter方法可以被继承，允许设置默认值，而且有良好的IDE支持。缺点当然就是对象无法在构造完成后马上进入就绪状态。

综上所述，构造方法注入和setter方法注入因为其侵入性较弱，且易于理解和使用，所以是现在使用最多的注入方式；而接口注入因为侵入性较强，近年来已经不 流行了。

## 资源访问

### 资源抽象接口

JDK所提供的访问资源的类（如java.net.URL、File等）并不能很好地满足各种底层资源的访问需求，比如缺少从类路径或者Web容器的上下文中获取资源的操作类。鉴此Spring设计了一个Resource接口，它为应用提供了更强的底层资源访问能力。该接口拥有对应不同资源类型的实现类。

Resource接口的主要方法：

boolean exists()：资源是否存在

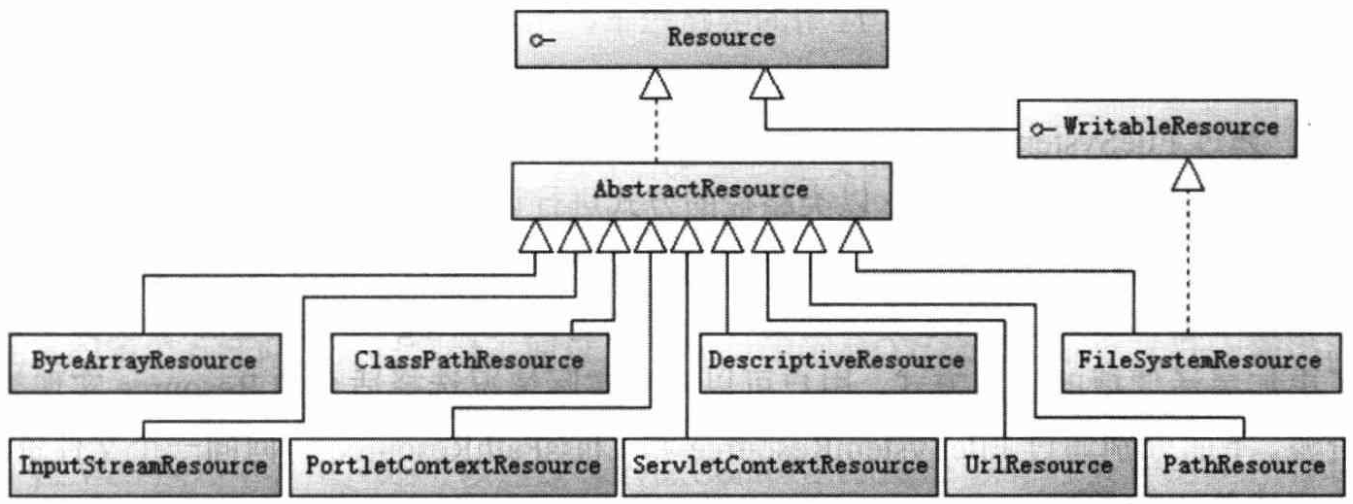
boolean isOpen()：资源是否打开

URL getURL() throws IOException：如果底层资源可以标称成URL，则该方法返回对应的URL对象。

File getFile() throws IOException：如果底层资源对应一个文件，则该方法返回对应的File对象。

InputStream getInputStream() throws IOException：返回资源对应的输入流。

Resource在Spring框架中起着不可或缺的作用，Spring框架使用Resource装载各种资源，包括配置文件资源、国际化属性文件资源等。



**WritableResource**：可写资源接口，是Spring3.1版本新加的接口，有两个实现类，即FileSystemResource和PathResource，其中PathResource是Spring4.0提供的实现类。

**ByteArrayResource**：二进制数组表示的资源，二进制数组资源可以在内存中通过程序构造。

**ClassPathResource**：类路径下的资源，资源以相对于类路径的方式表示。

**FileSystemResource**：文件系统资源，资源以文件系统路径的方式表示，如D:/conf/bean.xml等。

**InputStreamResource**：以输入流返回表示的资源。

**ServletContextResource**：为访问Web容器上下文中的资源而设计的类，负责以相对于Web应用根目录的路径加载资源。它支持以流和URL的方式访问，在WAR解包的情况下，也可以通过File方式访问。该类还可以直接从JAR包中访问资源。

**UrlResource**：URL封装了java.net.URL，它使用户能够访问任何可以通过URL表示的资源，如文件系统的资源、HTTP资源、FTP资源等。

**PathResource**：Spring4.0提供的读取资源文件的新类。Path封装了java.net.URL、java.nio.file.Path（Java7.0提供）、文件系统资源，它使用户能够访问任何可以通过URL、Path、系统文件路径表示的资源，如文件系统的资源、HTTP资源、FTP资源等。

有了这个抽象的资源类后，就可以将Spring的配置信息放置在任何地方（如数据库、LDAP中），只要最终可以通过Resource接口返回配置信息即可。

假设有一个文件位于Web应用的类路径下，用户可以通过以下方式对这个文件资源进行访问：

1. 通过FileSystemResource以文件系统绝对路径的方式进行访问。
2. 通过ClassPathResource以类路径的方式进行访问
3. 通过ServletContextResource以相对于Web应用根目录的方式进行访问。

相比于通过JDK的File类访问文件资源的方式，Spring的Resource实现类无疑提供了更加灵活便捷的访问方式，用户可以根据实际情况选择合适的Resource实现类访问资源。

**package** com.smart.domain;

**import** java.io.ByteArrayOutputStream;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.OutputStream;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.PathResource;

**import** org.springframework.core.io.Resource;

**import** org.springframework.core.io.WritableResource;

**public** **class** FileSourceExample {

**public** **static** **void** main(String[] args) {

**try** {

String filePath = "D:/masterSpring/code/chapter4/src/main/resource/conf/file1.txt";

//使用系统文件路径方式加载文件

WritableResource res1 = **new** PathResource(filePath);

//使用类路径方式加载文件

Resource res2 = **new** ClassPathResource("conf/file.txt");

//使用WritableResource接口写资源文件

OutputStream stream1 = res1.getOutputStream();

stream1.write("欢迎光临".getBytes());

stream1.close();

//使用Resource接口读资源文件

InputStream ins1 = res1.getInputStream();

InputStream ins2 = res2.getInputStream();

ByteArrayOutputStream baos = **new** ByteArrayOutputStream();

**int** i;

**while**((i = ins1.read()) != -1) {

baos.wait(i);

}

System.***out***.println(baos.toString());

System.***out***.println("res1:" + res1.getFilename());

System.***out***.println("res2:" + res2.getFilename());

}**catch**(IOException e) {

e.printStackTrace();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

在获取资源后，用户就可以通过Resource接口定义的多个方法访问文件的数据和其他信息。

资源加载时默认采用系统编码读取资源内容。如果资源文件采用特殊的编码格式，可以通过EncodedResource对资源进行编码，以保证资源内容操作的正确性。

**package** com.smart.domain;

**import** java.io.IOException;

**import** org.springframework.core.io.ClassPathResource;

**import** org.springframework.core.io.Resource;

**import** org.springframework.core.io.support.EncodedResource;

**import** org.springframework.util.FileCopyUtils;

**public** **class** EncodedResourceExample {

**public** **static** **void** main(String[] args) **throws** IOException {

Resource res = **new** ClassPathResource("conf/file1.txt");

EncodedResource encRes = **new** EncodedResource(res, "UTF-8");

String content = FileCopyUtils.*copyToString*(encRes.getReader());

System.***out***.println(content);

}

}

### 资源加载

为了访问不同类型的资源，必须使用相应的Resource实现类，这是比较麻烦的。Spring提供了一个强大的加载资源的机制，不但能够通过“classpath”、“file”等资源地址前缀识别不同的资源类型，还支持Ant风格带通配符的资源地址。

1. **资源地址表达式**

|  |  |  |
| --- | --- | --- |
| 地址前缀 | 实例 | 对应的资源类型 |
| classpath: | classpath:com/smart/beanfactory/beans.xml | 从类路径中加载资源，classpath:和classpath:/是等价的，都是相对于类的根路径。资源文件可以在标准的文件系统中，也可以在JAR或ZIP的类包中。 |
| file: | file:/conf/com/smart/beanfactory/beans.xml | 使用UrlResource从文件系统目录中装载资源，可采用绝对或相对路径。 |
| http:// | http://www.smart.com/resource/beans.xml | 使用UrlResource从Web服务器中装载资源 |
| ftp:// | ftp://www.smart.com/resource/beans.xml | 使用UrlResource从FTP服务器中装载资源 |
| 没有前缀 | com/smart/beanfactory/beans.xml | 根据ApplicationContext的具体实现类采用对应类型的Resource |

其中和“classpath:”对应的还有另一种比较难理解的“classpath\*:”前缀。假设有多个JAR包或文件系统类路径都拥有一个相同的包名（如com.smart）。“classpath:”只会在第一个加载的com.smart包的类路径下查找，而“classpath\*:”会扫描所有这些JAR包及类路径下出现的com.smart类路径。

这对于分模块打包的应用非常有用。假设一个名为smart的应用共分成3个模块，一个模块对应一个配置文件，分别是module1.xml、module2.xml、module4.xml，都放到com.smart目录下，每个模块单独打成JAR包。使用“classpath\*:com/smart/module\*.xml”可以成功加载这3个模块的配置文件，而使用“classpath:com/smart/module\*.xml”只会加载一个模块的配置文件。

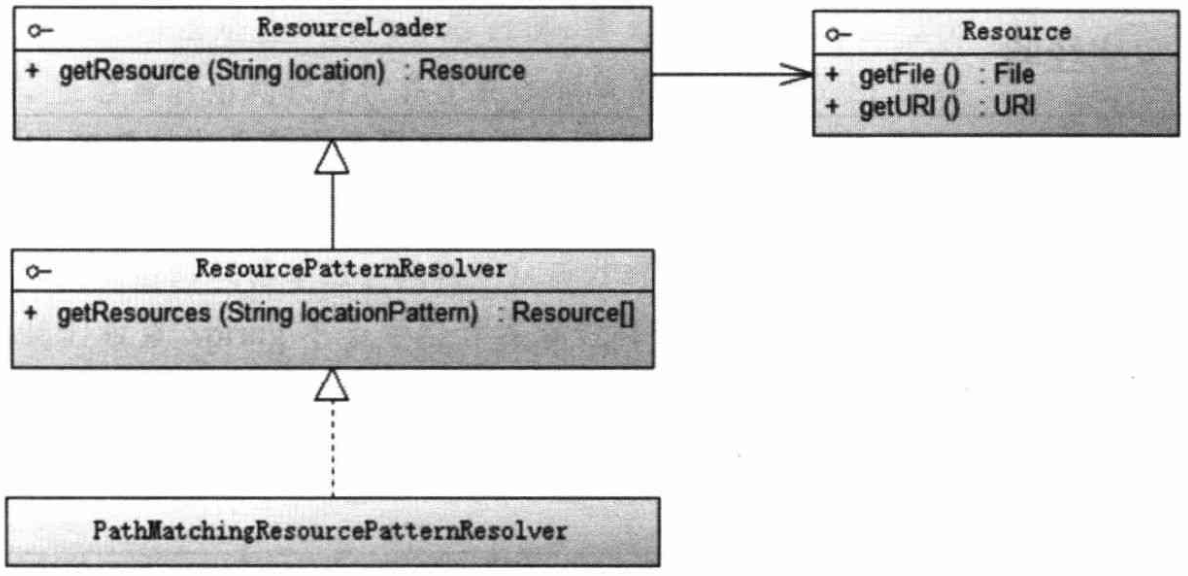
Ant风格的资源地址支持3种匹配符：

1. ?：匹配文件名中的一个字符
2. \*：匹配文件名中的任意字符
3. \*\*：匹配多层路径

下面时候几个Ant风格的资源路径的示例：

1. classpath:com/t?st.xml：匹配com类路径下的com/test.xml、com/tast.xml或者com/txst.xml文件。
2. file:D:/conf/\*.xml：匹配文件系统D:/conf目录下所有以.xml为后缀的文件。
3. classpath:com/\*\*/test.xml：匹配com类路径下（当前目录及其子孙目录）的test.xml文件
4. classpath:org/springframework/\*\*/\*.xml：匹配类路径org/springframework下所有以.xml为后缀的文件。
5. classpath:org/\*\*/servlet/bla.xml：不仅匹配类路径org/springframework/servlet/bla.xml，也匹配org/springframework/testing/servlet/bla.xml，还匹配org/servlet/bla.xml。
6. **资源加载器**

Spring定义了一套资源加载的接口，并提供了实现类



ResourceLoader接口仅有一个getResource(String location)方法，可以根据一个资源地址加载文件资源。不过，资源地址仅支持带资源类型前缀的表达式，不支持Ant风格的资源路径表达式。ResourcePatternResolver扩展ResourceLoader接口，定义了一个新的接口方法getResources(String locationPattern)，该方法支持带资源类型前缀及Ant风格的资源路径表达式。PathMatchingResourcePatternResolver是Spring提供的标准实现类。

**package** com.smart.domain;

**import** **static** org.junit.Assert.*assertNotNull*;

**import** org.junit.Test;

**import** org.springframework.core.io.Resource;

**import** org.springframework.core.io.support.PathMatchingResourcePatternResolver;

**import** org.springframework.core.io.support.ResourcePatternResolver;

**public** **class** PatternResolverTest {

@Test

**public** **void** getResources() **throws** Throwable {

ResourcePatternResolver resolver = **new** PathMatchingResourcePatternResolver();

//加载所有类包com.smart(及子孙包)下以.xml为后缀的资源

Resource resources[] = resolver.getResources("classpath\*:com/smart/\*\*/\*.xml");

*assertNotNull*(resources);

**for** (Resource resource : resources) {

System.***out***.println(resource.getDescription());

}

}

}

由于资源路径是“classpath\*:”，所以PathMatchingResourcePatternResolver将扫描所有类路径下及JAR包中对应com.smart类包下的路径，读取所有以.xml为后缀的文件资源。

提示：用Resource操作文件时，如果资源配置文件在项目发布时会被打包到JAR中，那么不能使用Resource的getFile()方法，会抛出FileNotFoundException。但可以使用Resource的getInputStream()方法读取。

## BeanFactory和ApplicationContext

Spring通过一个配置文件描述Bean与Bean之间的依赖关系，利用Java语言的反射功能实例化Bean并建立Bean之间的依赖关系。Spring的IoC容器在完成这些底层工作的基础上，还提供了Bean实例缓存、生命周期管理、Bean实例代理、实践发布、资源装载等高级服务。

Bean工厂（com.springframework.beans.factory.BeanFactory）是Spring框架最核心的接口，它提供了高级IoC的配置机制。BeanFactory使管理不同类型的Java对象成为可能，应用上下文（com.springframework.context.ApplicationContext）建立在BeanFactory基础之上，提供了更多面向应用的功能，它提供了国际化支持和框架事件体系，更易于创建实际应用。我们一般称BeanFactory为IoC容器，而称ApplicationContext为应用上下文。

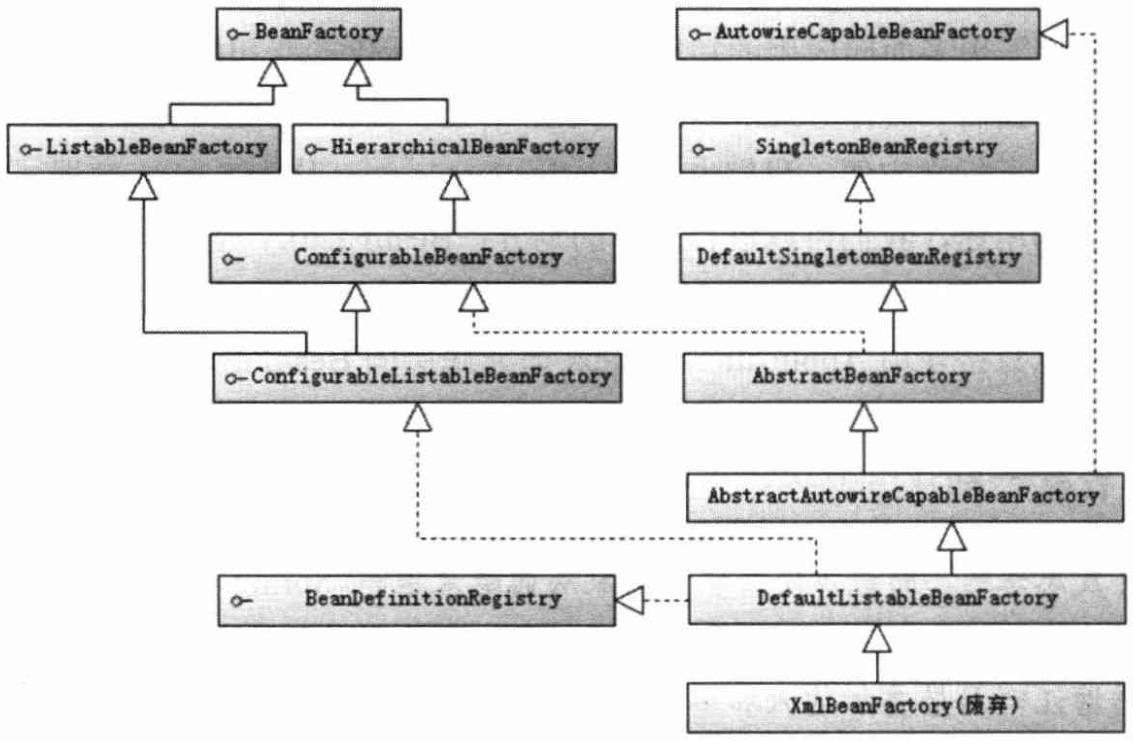
BeanFactory是Spring框架的基础，面向Spring本身；ApplicationContext面向使用Spring框架的开发者，几乎所有的应用场合都可以直接使用ApplicationContext而非底层的BeanFactory。

### BeanFactory

BeanFactory是一个类工厂，但和传统的类工厂不同，传统的类工厂仅负责构造一个或几个类的实例；而BeanFactory是累的通用工厂，它可以创建并管理各种类的对象。这些可被创建和管理的对象本身没有什么特别之处，仅是一个POJO，Spring称这些被创建和管理的Java对象为Bean。

#### BeanFactory的类体系结构

Spring为BeanFactory提供了多种实现，最常用的是XmlBeanFactory，但在Spring3.2中已被废弃，建议使用XmlBeanDefinitionReader、DefaultListableBeanFactory替代。



BeanFactory接口位于类结构树的顶端，它最主要的方法是getBean(String beanName)，该方法从容器中返回特定名称的Bean。BeanFactory的功能通过其他接口得到不断扩展。

1. ListableBeanFactory：该接口定义了访问容器中Bean基本信息的若干方法，如查看Bean的个数、获取某一类型Bean的配置名、查看容器中是否包括某一Bean等。
2. HierarchicalBeanFactory：父子级联IoC容器的接口，子容器可以通过接口方法访问父容器。
3. ConfigurableBeanFactory：这是一个重要的接口，增强了IoC容器的可定制性。它定义了设置类装载器、属性编辑器、容器初始化后置处理器等方法。
4. AutowireCapableBeanFactory：定义了将容器中的Bean按某种规则（如按名字匹配、按类型匹配等）进行自动装配的方法。
5. SingletonBeanRegistry：定义了允许在运行期向容器注册单实例Bean的方法。
6. BeanDefinitionRegistry：Spring配置文件中每一个<bean>节点元素在Spring容器里都通过一个BeanDefinition对象表示，它描述了Bean的配置信息。而BeanDefinitionRegistry接口提供了向容器手工注册BeanDefinition对象的方法。

#### 初始化BeanFactory

下面使用Spring配置文件为Car提供配置信息，然后通过BeanFactory装载配置文件，启动Spring IoC容器。

beans.xml：Car的配置文件

<?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:p=*"http://www.springframework.org/schema/p"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:aop=*"http://www.springframework.org/schema/aop"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.0.xsd*

*http://www.springframework.org/schema/context*

*http://www.springframework.org/schema/context/spring-context-4.0.xsd*

*http://www.springframework.org/schema/tx*

*http://www.springframework.org/schema/tx/spring-tx-4.0.xsd*

*http://www.springframework.org/schema/aop*

*http://www.springframework.org/schema/aop/spring-aop-4.0.xsd"* >

<bean id=*"car1"* class=*"com.smart.Car"*>

<property name=*"brand"* value=*"红旗CA72"* />

<property name=*"color"* value=*"黑色"* />

<property name=*"maxSpeed"* value=*"200"* />

</bean>

</beans>

下面通过XmlBeanDefinitionReader、DefaultListableBeanFactory实现类启动Spring IoC容器。

**package** com.smart.domain;

**import** org.apache.naming.factory.BeanFactory;

**import** org.junit.Test;

**import** org.springframework.beans.factory.support.DefaultListableBeanFactory;

**import** org.springframework.beans.factory.xml.XmlBeanDefinitionReader;

**import** org.springframework.core.io.Resource;

**import** org.springframework.core.io.support.PathMatchingResourcePatternResolver;

**import** org.springframework.core.io.support.ResourcePatternResolver;

**public** **class** BeanFactoryTest {

@Test

**public** **void** getBean() **throws** Throwable {

ResourcePatternResolver resolver = **new** PathMatchingResourcePatternResolver();

Resource res = resolver.getResource("classpath:com/smart/beanfactory/beans.xml");

System.***out***.println(res.getURI());

//被废弃，不建议使用

//BeanFactory bf = new XmlBeanFactory(res);

DefaultListableBeanFactory factory = **new** DefaultListableBeanFactory();

XmlBeanDefinitionReader reader = **new** XmlBeanDefinitionReader(factory);

reader.loadBeanDefinitions(res);

System.***out***.println("init BeanFactory.");

Car car = factory.getBean("car", Car.**class**);

System.***out***.println("car bean is ready for use!");

car.introduce();

}

}

XmlBeanDefinitionReader通过Resource装载Spring配置信息并启动IoC容器，然后就可以通过BeanFactory的getBean(beanName)方法从IoC容器中获取Bean。通过BeanFactory启动IoC容器时，并不会初始化配置文件中的Bean，初始化动作发生在第一次调用时。对于单实例的Bean来说，BeanFactory会缓存Bean实例，所以第二次使用getBean()获取Bean时，将直接从IoC容器的缓存中获取Bean实例。

Spring在DefaultSingletonBeanRegistry类中提供了一个用于缓存单实例Bean的缓存器，它是一个用HashMap实现的缓存器，单实例的Bean以beanName为键保存在这个HashMap中。

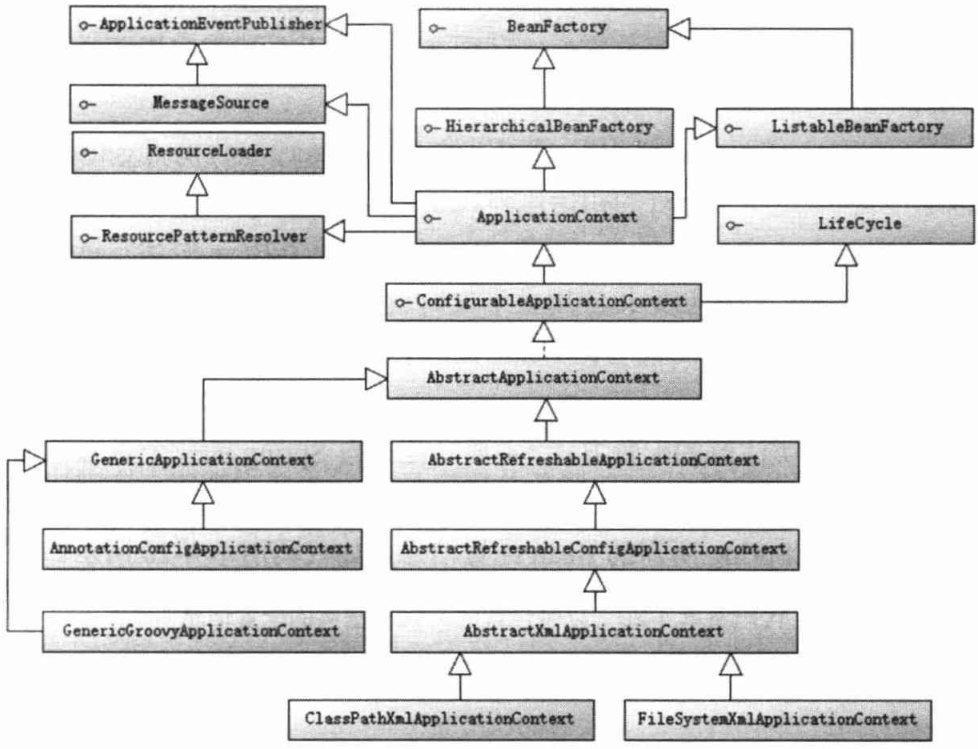
值得一提的是，在初始化BeanFactory时，必须为其提供一种日志框架，我们使用Log4J，即在类路径下提供Log4J配置文件，这样启动Spring容器才不会报错。

### ApplicationContext

ApplicationContext由BeanFactory派生而来，提供了更多面向实际应用的功能。在BeanFactory中，很多功能需要以编程的方式实现，而在ApplicationContext中则可以通过配置的方式实现。

#### ApplicationContext类体系结构

ApplicationContext的主要实现类是ClassPathXmlApplicationContext和FileSystemXmlApplicationContext，前者默认从类路径加载配置文件，后者默认从文件系统中加载配置文件。



从上图可以看出，ApplicationContext继承了HierarchicalBeanFactory和ListableBeanFactory接口，在此基础上，还通过多个其他的接口扩展了BeanFactory的功能。这些接口入下：

1. ApplicationEventPublisher：让容器拥有发布应用上下文事件的功能，包括容器启动事件、关闭事件等。实现了ApplicationListener事件监听接口的Bean可以接收到容器事件，并对事件进行响应处理。在ApplicationConntext抽象实现类AbstractApplicationContext中存在一个ApplicationEventMulticaster，它负责保存所有的监听器，以便在容器产生上下文事件时通知这些事件监听者。
2. MessageSource：为应用提供i18n国际化消息访问的功能。
3. ResourcePatternResolver：所有ApplicationContext实现类都实现了类似于PathMatchingResourcePatternResolver的功能，可以通过带前缀的Ant风格的资源文件路径装载Spring的配置文件。
4. LifeCycle：该接口提供了start()和stop()两个方法，主要用于控制异步处理过程。在具体使用时，该接口同时被ApplicationContext实现及具体Bean实现，ApplicationContext会将start/stop的消息传递给容器中所有实现了该接口的Bean，以达到管理和控制JMX、任务调度等目的。

ConfigurableApplicationContext扩展于ApplicationContext，它新增了两个主要的方法：refresh()和close()，让AppliccationContext具有启动、刷新和关闭应用上下文的能力。在应用上下文关闭的情况下调用refresh()即可启动应用上下文，在已经启动的状态下调用refresh()则可清除缓存并重新装载配置信息，而调用close()则可关闭应用上下文。

和BeanFactory初始化相似，ApplicationContext的初始化也很简单。如果配置文件放置在类路径下，则可以优先考虑使用ClassPathXmlApplicationContext实现类。

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("com/smart/context/beans.xml");

对于ClassPathXmlApplicationContext来说“com/smart/context/beans.xml”等同于“classpath:com/smart/context/beans.xml”。

如果配置文件放置在文件系统的路径下，则可以优先考虑使用FileSystemXmlApplicationContext实现类。

ApplicationContext ctx = **new** FileSystemXmlApplicationContext("com/smart/context/beans.xml");

对于FileSystemXmlApplicationContext来说，“com/smart/context/beans.xml”等同于“file:com/smart/context/beans.xml”。

还可以指定一组配置文件，Spring会自动将多个配置文件在内存中“整合”成一个配置文件。

ApplicationContext ctx = **new** ClassPathXmlApplicationContext(**new** String[] {"com/smart/context/beans2.xml"});

当然，FileSystemXmlApplicationContext和ClassPathXmlApplicationContext都可以显示使用带资源类型前缀的路径，它们的区别在于如果不显示指定资源类型前缀，则分别将路径解析为文件系统路径和类路径。

在获取ApplicationContext实例后，就可以像BeanFactory一样调用getBean(beanName)返回Bean了。ApplicationContext的初始化与BeanFactory不同的在于ApplicationContext在初始化上下文时就已经实例化所有单实例的Bean。而BeanFactory则在调用的时候才实例化Bean，所以ApplicationContext的初始化时间比BeanFactory稍长一些。

Spring支持基于类注解的配置方式，主要功能来自Spring的一个名为JavaConfig的子项目。JavaConfig现已升级为Spring核心框架的一部分。一个标注@Configuration注解的POJO即可提供Spring所需的Bean配置信息。

**package** com.smart.domain;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.Configuration;

@Configuration //表示是一个配置信息提供类

**public** **class** Beans {

//定义一个Bean

@Bean(name="car")

**public** Car buildCar() {

Car car = **new** Car();

car.setBrand("红旗CA72");

car.setMaxSpeed(200);

**return** car;

}

}

#### ApplicationContext初始化

和基于XML文件的配置方式相比，类注解的配置方式可以很容易地让开发者控制Bean的初始化过程，比基于XML文件的配置方式更加灵活。

Spring为基于注解类的配置提供了专门的ApplicationContext实现类：AnnotationConfigApplicationContext。下面是使用AnnotationConfigApplicationContext启动Spring容器的示例。

**package** com.smart.domain;

**import** **static** org.junit.Assert.*assertNotNull*;

**import** org.junit.Test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**public** **class** AnnotationConfigApplicationContextTest {

@Test

**public** **void** getBean() {

//通过一个带@Configuration的POJO装载Bean配置

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(Beans.**class**);

Car car = ctx.getBean("car", Car.**class**);

*assertNotNull*(car);

}

}

AnnotationConfigApplicationContext将加载Beans.class中的Bean定义并调用Beans.class中的方法实例化Bean，启动容器并装配Bean。

Spring4.0支持使用Groovy DSL来进行Bean定义配置。其与基于XML文件的配置类似，只不过基于Groovy脚本语言，可以实现复杂、灵活的Bean配置逻辑。

package com.smart.context;

import com.smart.Car;

beans{

car(Car){ //名字（类型）

brand = "红旗CA72" //注入属性

maxSpeed = "200"

color = "red"

}

}

基于Groovy的配置方式可以很容易地让开发者配置复杂Bean的初始化过程，比基于XML文件、注解的配置方式更加灵活。

Spring为基于Groovy的配置提供了专门的ApplicationContext实现类：GenericGroovyApplicationContext。下面是使用GenericGroovyApplicationContext启动Spring容器的示例。

**package** com.smart.domain;

**import** **static** org.junit.Assert.*assertEquals*;

**import** **static** org.junit.Assert.*assertNotNull*;

**import** org.junit.Test;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.GenericGroovyApplicationContext;

**public** **class** GenericGroovyApplicationContextTest {

@Test

**public** **void** getBean() {

ApplicationContext ctx = **new** GenericGroovyApplicationContext("classpath:com/smart/context/groovy-beans.groovy");

Car car = (Car)ctx.getBean("car");

*assertNotNull*(car);

*assertEquals*(car.getColor(), "red");

}

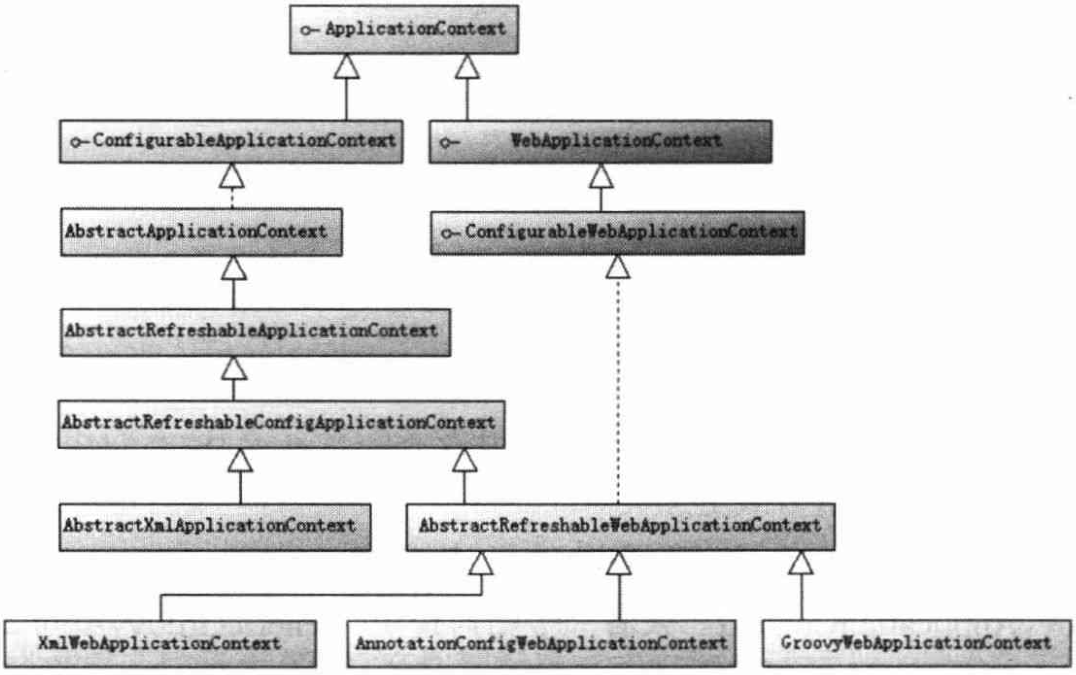
}

#### WebApplicationContext类体系结构

WebApplicationContext是专门为Web应用准备的，它允许从相对于Web根目录的路径中装载配置文件完成初始化工作。从WebApplicationContext中可以获得ServletContext的引用，整个Web应用上下文对象将作为属性放置到ServletContext中，以便Web应用环境可以访问Spring应用上下文。Spring专门为此提供了一个工具类WebApplicationContextUtils，通过该类的getWebApplicationContext(ServletContext sc)方法，可以从ServletContext中获取WebApplicationContext实例。

在非Web应用的环境下，Bean只有singleton和prototype两种作用域。WebApplicationContext为Bean添加了三个新的作用域：request、session和global session。

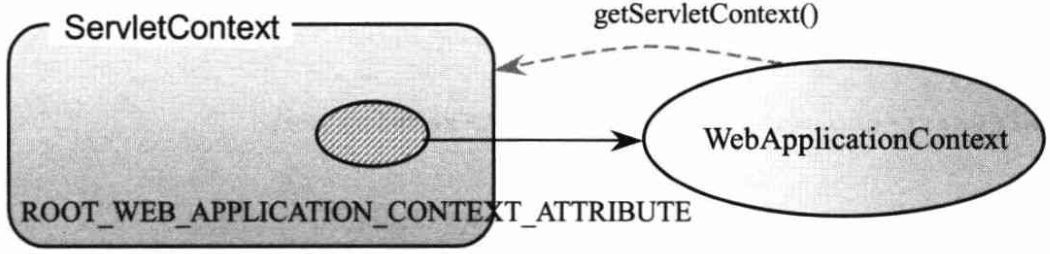
下图是WebApplicationContext的类继承体系：



由于Web应用比一般的应用拥有更多的特性，因为WebApplicationContext扩展了ApplicationContext。WebApplicationContext定义了一个常量ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE，在上下文启动时，WebApplicationContext实例即以此为键放置在ServletContext的属性列表中，可以通过下面的语句从Web容器中获取WebApplicationContext：

WebApplicationContext wac = (WebApplicationContext) servletContext.getAttribute(WebApplicationContext.***ROOT\_WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE***);

这正是前面提到的WebApplicationContextUtils工具类getWebApplicationContext(ServletContext sc)方法的内部实现方式。这样Spring的Web应用上下文和Web容器的上下文应用就可以实现互访，二者实现了融合。



ConfigurableWebApplicationContext扩展了WebApplicationContext，它允许通过配置的方式实例化WebApplicationContext，同时定义了两个重要的方法：

1. setServletContext(ServletContext servletContext)：为Spring设置Web应用上下文，以便二者整合。
2. setConfigLocations(String[] configLocations)：设置Spring配置文件地址，一般情况下，配置文件地址是相对于Web根目录的地址，如/WEB-INF/smart-dao.xml、/WEB-INF/smart-service.xml等。用户也可以使用带资源类型前缀的地址，如classpath:com/smart/beans.xml等。

#### WebApplicationContext初始化

WebApplicationContext的初始化方式和BeanFactory、ApplicationContext有所区别，因为WebApplicationContext需要ServletContext实例，也就是说，它必须在拥有Web容器的前提下才能完成启动工作。可以在web.xml中配置自启动的Servlet或定义Web容器监听器，借助二者中的任何一个，就可以完成启动Spring Web应用上下文的工作。

Spring分别提供了用于启动WebApplicationContext的Servlet和Web容器监听器：

1. org.springframework.web.context.ContextLoaderServlet。
2. org.springframework.web.context.ContextLoaderListener。

二者的内部都实现了启动WebApplicationContext实例的逻辑，只要根据Web容器的情况下选择二者之一，并在web.xml中完成配置即可。

下面示例是使用ContextLoaderListener启动WebApplicationContext的具体配置：

.....

<!-- 指定配置文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/smart-dao.xml, /WEB-INF/smart-service.xml</param-value>

</context-param>

<!-- 声明Web容器监听器 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

ContextLoaderListener通过Web容器上下文参数contextConfigLocation获取Spring配置文件的位置。用户可以指定多个配置文件，用逗号、空格或冒号分隔均可。对于未带资源类型前缀的配置文件路径，WebApplicationContext默认这些路径相对于Web的部署根路径。也可以采用带资源类型前缀的路径配置：“classpath\*:/smart-\*.xml”和上面的配置是等效的。

下面是使用ContextLoaderServlet完成相同的工作：

<!-- 指定配置文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/smart-dao.xml, /WEB-INF/smart-service.xml</param-value>

</context-param>

<!-- 声明自启动的Servlet -->

<servlet>

<servlet-name>springContextLoaderServlet</servlet-name>

<servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

注意：在Spring3.0版本以后移除了ContextLoaderServlet，所以不建议使用此方法获取WebApplicationContext。

由于WebApplicationContext需要使用日志功能，所以可以将Log4J的配置文件放置在类路径/WEB-INF下，这时Log4J引擎即可顺利启动。如果Log4J配置文件放置在其他位置，那么必须在web.xml中指定Log4J配置文件的位置。Spring为启动Log4J引擎提供了两个类似于启动WebApplicationContext的实现类：Log4jConfigServlet和Log4jConfigListener。不管采用哪种方式，都必须保证能够在装载Spring配置文件前先装载Log4J配置信息。

<!-- 指定Log4J配置文件的位置 -->

<context-param>

<param-name>log4jConfigLocation</param-name>

<param-value>/WEB-INF/log4j.properties</param-value>

</context-param>

<!-- 装载Log4J配置文件的监听器 -->

<listener>

<listener-class>org.springframework.web.util.Log4jConfigListener</listener-class>

</listener>

<!-- 指定配置文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/smart-dao.xml, /WEB-INF/smart-service.xml</param-value>

</context-param>

<!-- 声明Web容器监听器 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

注意：但是在Spring5.0之后已经废除了Log4jConfigServlet和Log4jConfigListener实现类，Spring建议用log4j2 来替换这个类。

如果使用标注@Configuration的Java类提供配置信息，则web.xml需要按一下方式配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* version=*"3.0"*>

<display-name>Archetype Created Web Application</display-name>

<!-- 通过指定context参数，让Spring使用AnnotationConfigWebApplicationContext而非XmlWebApplicationContext启动容器 -->

<context-param>

<param-name>contextClass</param-name>

<param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>

</context-param>

<!-- 指定标注了@Configuration的配置类，多个可以使用逗号或空格分隔 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>com.smart.AppConfig1, com.smart.AppConfig2</param-value>

</context-param>

<!-- ContextLoaderListener监听器将根据上面的配置使用AnnotationConfigWebApplicationContext根据contextConfigLocation指定的配置类启动Spring容器 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

</web-app>

如果使用Groovy DSL配置Bean信息，则web.xml需要按以下方式配置：

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<web-app xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xmlns=*"http://java.sun.com/xml/ns/javaee"* xsi:schemaLocation=*"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd"* version=*"3.0"*>

<display-name>Archetype Created Web Application</display-name>

<!-- 通过指定context参数，让Spring使用GroovyWebApplicationContext而非XmlWebApplicationContext或AnnotationConfigWebApplicationContext启动容器 -->

<context-param>

<param-name>contextClass</param-name>

<param-value>org.springframework.web.context.support.GroovyWebApplicationContext</param-value>

</context-param>

<!-- 指定标注了Groovy的配置文件 -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath\*:conf/spring-mvc.groovy</param-value>

</context-param>

<!-- ContextLoaderListener监听器将根据上面的配置使用GroovyWebApplicationContext根据contextConfigLocation指定的配置类启动Spring容器 -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

</web-app>

### 父子容器

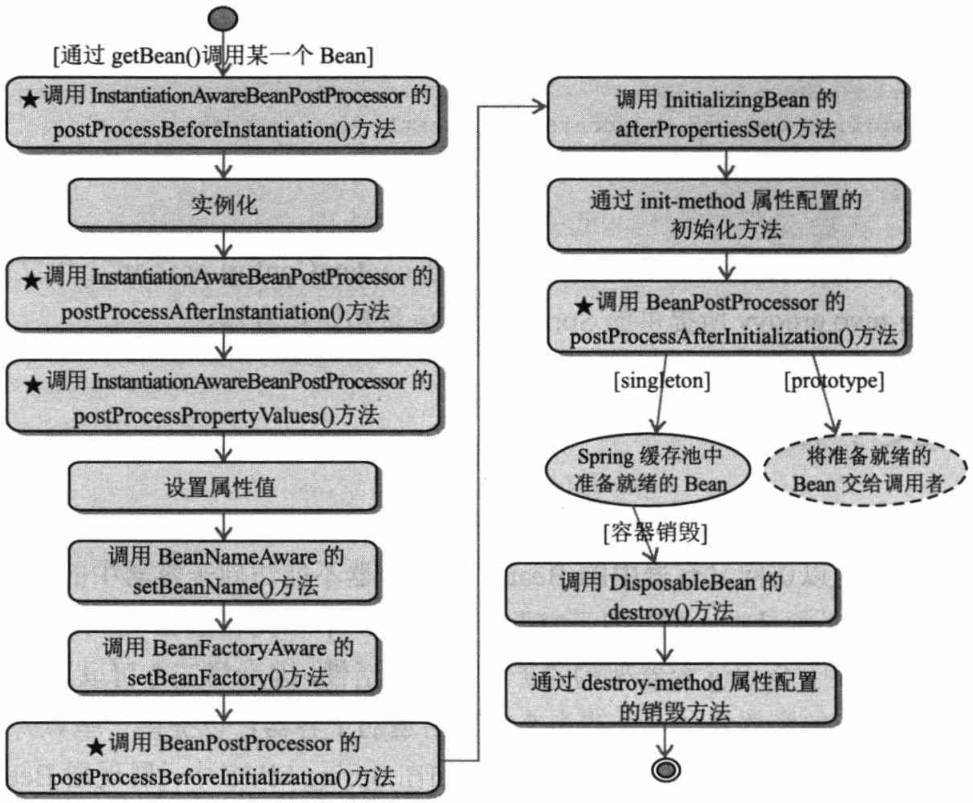
通过HierarchicalBeanFactory接口，Spring的IoC容器可以建立父子层级关联的容器体系，子容器可以访问父容器中的Bean，但父容器不能访问子容器中的Bean。在容器内，Bean的id必须是唯一的，但子容器可以拥有一个和父容器id相同的Bean。父子容器层级体系增强了Spring容器架构的扩展性和灵活性，因为第三方可以通过编程的方式为一个已经存在的容器添加一个或多个特殊用途的子容器，以提供一些额外的功能。

Spring使用父子容器实现了很多功能，比如在Spring MVC中，展现层Bean位于一个子容器中，而业务层和持久层Bean位于父容器中。这样展现层Bean就可以引用业务层和持久层Bean，而业务层和持久层Bean则看不到展现层Bean。

### Bean的生命周期

#### BeanFactory中Bean的生命周期

1. **生命周期图解**



具体过程如下：

1. 当调用者通过getBean(beanName)向容器请求某个Bean时，如果容器注册了org.springframework.beans.factory.config.InstantiationAwareBeanPostProcessor接口，则在实例化Bean之前，将调用 接口的postProcessBeforeInstantiation()方法。
2. 根据配置情况调用Bean构造函数或工厂方法实例化Bean。
3. 如果容器注册了InstantiationAwareBeanPostProcessor接口，那么在实例化Bean之后，调用该接口的postProcessAfterInstantiation()方法，对实例化的对象进行一些操作。
4. 如果Bean配置了属性信息，那么容器在这一步着手将配置值设置到Bean对应的属性中，不过在设置每个属性之前将先调用InstantiationAwareBeanPostProcessor接口的postProcessPropertyValues()方法。

#### ApplicationContext中Bean的生命周期

## 在IoC容器中装配Bean

### Spring配置概述

要使应用程序中的Spring容器成功启动，需要同时具备以下三方面的条件：

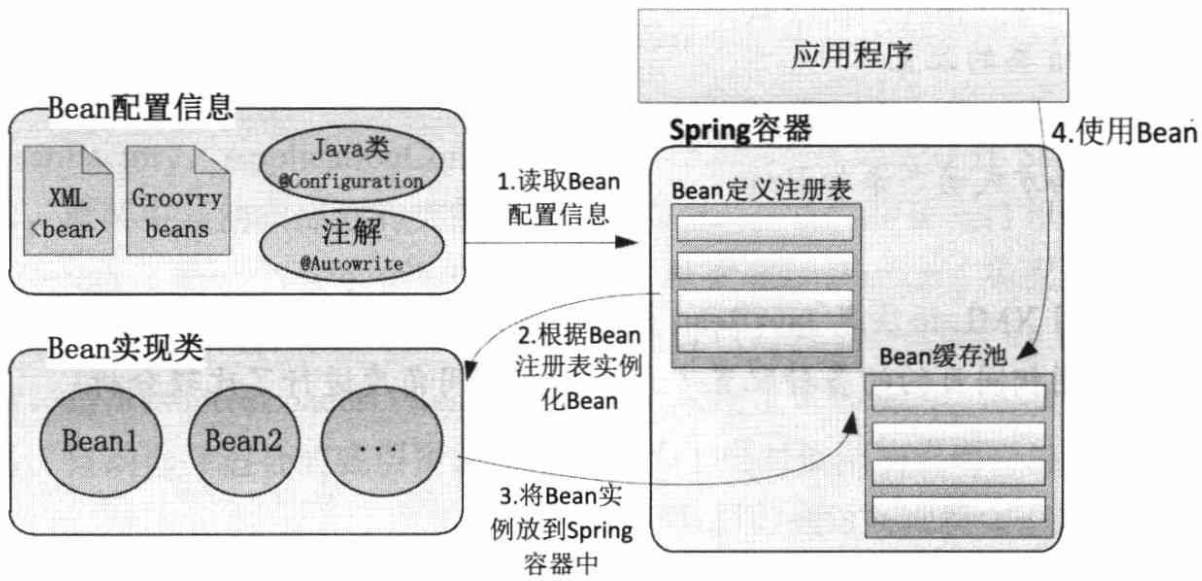
1. Spring框架的类包都已经放到应用程序的类路径下。
2. 应用程序为Spring提供了完备的Bean配置信息。
3. Bean的类都已经放到应用程序的类路径下。

Spring启动时读取应用程序提供的Bean配置信息，并在Spring容器中生成一份相应的Bean配置注册表，然后根据这张注册表实例化Bean，装配好Bean之间的依赖关系，为上层应用提供准备就绪的运行环境。

Bean配置信息时Bean的元数据信息，它由以下4个方面组成：

1. Bean的实现类
2. Bean的属性信息，如数据源的连接数、用户名、密码等
3. Bean的依赖关系，Spring根据依赖关系配置完成Bean之间的装配
4. Bean的行为配置，如生命周期范围及生命周期各过程的回调函数等。

Bean元数据信息在Spring容器中的内部对应物是一个个BeanDefinition形成的Bean注册表，Spring实现了Bean元数据信息内部表示和外部定义的解耦。Spring支持多种形式的Bean配置方式：XML配置、注解配置、Java类配置、Groovy动态语言配置。



### 基于XML的配置

#### Bean基本配置

#### 依赖注入

#### 注入参数详解

#### 方法注入

#### <bean>之间的关系

#### 整合多个配置文件

#### Bean作用域

### 基于注解的配置

### 基于Java类的配置

### 基于Groovy DSL的配置

### 通过编码方式动态添加Bean

### 不同配置方式比较

## Spring容器高级主题

### Spring容器技术内幕

### 属性编辑器

### 使用外部属性文件

### 国际化信息

### 容器事件

## Spring AOP

### AOP概述

### 基础知识

### 创建增强类

### 创建切面

### 自动创建代理

## 基于@AspectJ和Schema的AOP

## Spring SpEL

# Spring数据操作

# Spring整合其他框架

# Spring Web