Spring 框架
专业人士笔记

Spring® 框架
专业人士笔记

Chapter 6: Bean scopes
Section 6.1: Additional scopes in web-aware contexts
Chapter 10: RestTemplate
Section 10.1: Downloading a Large File
Section 10.2: Setting headers on Spring RestTemplate request
Chapter 15: JdbcTemplate
Section 15.1: Basic Query methods
Section 15.2: Query for List of Maps

50+ 页
专业提示和技巧

Spring® Framework
Notes for Professionals

Chapter 6: Bean scopes
Section 6.1: Additional scopes in web-aware contexts
Chapter 10: RestTemplate
Section 10.1: Downloading a Large File
Section 10.2: Setting headers on Spring RestTemplate request
Chapter 15: JdbcTemplate
Section 15.1: Basic Query methods
Section 15.2: Query for List of Maps

50+ pages
of professional hints and tricks

目录

Contents

关于

About

第1章：Spring框架入门

版本	发布日期
5.0.x	2017-10-24
4.3.x	2016-06-10
4.2.x	2015-07-31
4.1.x	2014-09-14
4.0.x	2013-12-12
3.2.x	2012-12-13
3.1.x	2011-12-13
3.0.x	2009-12-17
2.5.x	2007-12-25
2.0.x	2006-10-04
1.2.x	2005-05-13
1.1.x	2004-09-05
1.0.x	2003-03-24

第1.1节：设置（XML配置）

创建Hello Spring的步骤：

1. 调查 Spring Boot，看看它是否更适合您的需求。
2. 建立一个包含正确依赖项的项目。建议使用 Maven 或 Gradle。
3. 创建一个 POJO 类，例如Employee.java
4. 创建一个 XML 文件，在其中定义您的类和变量。例如beans.xml
5. 创建您的主类，例如Customer.java
6. 将spring-beans（及其传递依赖！）作为依赖项包含进来。

Employee.java:

```java
package com.test;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void displayName() {
        System.out.println(name);
    }
}
```

beans.xml:

Chapter 1: Getting started with Spring Framework

Version	Release Date
5.0.x	2017-10-24
4.3.x	2016-06-10
4.2.x	2015-07-31
4.1.x	2014-09-14
4.0.x	2013-12-12
3.2.x	2012-12-13
3.1.x	2011-12-13
3.0.x	2009-12-17
2.5.x	2007-12-25
2.0.x	2006-10-04
1.2.x	2005-05-13
1.1.x	2004-09-05
1.0.x	2003-03-24

Section 1.1: Setup (XML Configuration)

Steps to create Hello Spring:

1. Investigate Spring Boot to see if that would better suit your needs.
2. Have a project set up with the correct dependencies. It is recommended that you are using Maven or Gradle.
3. create a POJO class, e.g. `Employee.java`
4. create a XML file where you can define your class and variables. e.g `beans.xml`
5. create your main class e.g. `Customer.java`
6. Include spring-beans (and its transitive dependencies!) as a dependency.

`Employee.java`:

```java
package com.test;

public class Employee {

    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void displayName() {
        System.out.println(name);
    }
}
```

`beans.xml`:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/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-4.3.xsd">

    <bean id="employee" class="com.test.Employee">
        <property name="name" value="test spring"></property>
    </bean>

</beans>
```

Customer.java:

```java
package com.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Customer {
   public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

      Employee obj = (Employee) context.getBean("employee");
obj.displayName();
   }
}
```

第1.2节：通过示例展示Spring核心功能

描述

这是一个自包含的运行示例，包含/展示：所需的最小*依赖*，Java*配置*，
通过注解和Java配置声明Bean，通过构造函数和属性进行依赖注入，以及
*前置/后置*钩子。

依赖

类路径中需要以下依赖：

1.spring-core
2.spring-context
3.spring-beans
4.spring-aop
5.spring-expression
6.commons-logging

主类

从末尾开始，这是我们的主类，作为main()方法的占位符，该方法通过指向配置类来初始化应用上下文，并加载展示特定功能所需的各种bean。

```java
包 com.stackoverflow.documentation;

导入 org.springframework.context.ApplicationContext;
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/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-4.3.xsd">

    <bean id="employee" class="com.test.Employee">
        <property name="name" value="test spring"></property>
    </bean>

</beans>
```

Customer.java:

```java
package com.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Customer {
   public static void main(String[] args) {
      ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");

      Employee obj = (Employee) context.getBean("employee");
      obj.displayName();
   }
}
```

Section 1.2: Showcasing Core Spring Features by example

Description

This is a self-contained running example including/showcasing: minimum *dependencies* needed, Java *Configuration*, *Bean declaration* by annotation and Java Configuration, *Dependency Injection* by Constructor and by Property, and *Pre/Post* hooks.

Dependencies

These dependencies are needed in the classpath:

1. spring-core
2. spring-context
3. spring-beans
4. spring-aop
5. spring-expression
6. commons-logging

Main Class

Starting from the end, this is our Main class that serves as a placeholder for the `main()` method which initialises the Application Context by pointing to the Configuration class and loads all the various beans needed to showcase particular functionality.

```java
package com.stackoverflow.documentation;

import org.springframework.context.ApplicationContext;
```

```
导入 org.springframework.context.annotation.AnnotationConfigApplicationContext;


公共类 Main {

    public static void main(String[] args) {

        //每个应用程序只初始化一次应用上下文。
ApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(AppConfig.class);

        //通过注解注册的bean
BeanDeclaredByAnnotation beanDeclaredByAnnotation =
                applicationContext.getBean(BeanDeclaredByAnnotation.class);
        beanDeclaredByAnnotation.sayHello();

        //通过Java配置文件注册的Bean
BeanDeclaredInAppConfig beanDeclaredInAppConfig =
                applicationContext.getBean(BeanDeclaredInAppConfig.class);
        beanDeclaredInAppConfig.sayHello();

        //展示构造函数注入
BeanConstructorInjection beanConstructorInjection =
                applicationContext.getBean(BeanConstructorInjection.class);
        beanConstructorInjection.sayHello();

        //展示属性注入
BeanPropertyInjection beanPropertyInjection =
                applicationContext.getBean(BeanPropertyInjection.class);
        beanPropertyInjection.sayHello();

        //展示PreConstruct / PostDestroy钩子
BeanPostConstructPreDestroy beanPostConstructPreDestroy =
                applicationContext.getBean(BeanPostConstructPreDestroy.class);
        beanPostConstructPreDestroy.sayHello();
    }
}
```

应用配置文件

配置类使用@Configuration注解，并作为初始化的应用程序上下文（Application Context）的参数。配置类上的@ComponentScan注解指向一个包，用于扫描该包中通过注解注册的Bean和依赖。最后，@Bean注解用作配置类中的Bean定义。

```
package com.stackoverflow.documentation;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.stackoverflow.documentation")
public class AppConfig {

   @Bean
    public BeanDeclaredInAppConfig beanDeclaredInAppConfig() {
        return new BeanDeclaredInAppConfig();
    }
```

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


public class Main {

    public static void main(String[] args) {

        //initializing the Application Context once per application.
        ApplicationContext applicationContext =
                new AnnotationConfigApplicationContext(AppConfig.class);

        //bean registered by annotation
        BeanDeclaredByAnnotation beanDeclaredByAnnotation =
                applicationContext.getBean(BeanDeclaredByAnnotation.class);
        beanDeclaredByAnnotation.sayHello();

        //bean registered by Java configuration file
        BeanDeclaredInAppConfig beanDeclaredInAppConfig =
                applicationContext.getBean(BeanDeclaredInAppConfig.class);
        beanDeclaredInAppConfig.sayHello();

        //showcasing constructor injection
        BeanConstructorInjection beanConstructorInjection =
                applicationContext.getBean(BeanConstructorInjection.class);
        beanConstructorInjection.sayHello();

        //showcasing property injection
        BeanPropertyInjection beanPropertyInjection =
                applicationContext.getBean(BeanPropertyInjection.class);
        beanPropertyInjection.sayHello();

        //showcasing PreConstruct / PostDestroy hooks
        BeanPostConstructPreDestroy beanPostConstructPreDestroy =
                applicationContext.getBean(BeanPostConstructPreDestroy.class);
        beanPostConstructPreDestroy.sayHello();
    }
}
```

Application Configuration file

The configuration class is annotated by `@Configuration` and is used as a parameter in the initialised Application Context. The `@ComponentScan` annotation at the class level of the configuration class points to a package to be scanned for Beans and dependencies registered using annotations. Finally the `@Bean` annotation serves as a bean definition in the configuration class.

```
package com.stackoverflow.documentation;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@ComponentScan("com.stackoverflow.documentation")
public class AppConfig {

    @Bean
    public BeanDeclaredInAppConfig beanDeclaredInAppConfig() {
        return new BeanDeclaredInAppConfig();
    }
}
```

通过注解声明Bean

@Component注解用于将POJO标记为Spring Bean，以便在组件扫描期间注册。

```
@Component
public class BeanDeclaredByAnnotation {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredByAnnotation !");
    }
}
```

通过应用配置声明 Bean

请注意，我们不需要对 POJO 进行注解或其他标记，因为 Bean 的声明/定义是在应用配置类文件中完成的。

```
public class BeanDeclaredInAppConfig {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredInAppConfig !");
    }
}
```

构造函数注入

请注意，@Autowired 注解设置在构造函数级别。还要注意，除非显式按名称定义，否则默认的自动装配是基于 Bean 的类型进行的（在此例中为 BeanToBeInjected）。

```
package com.stackoverflow.documentation;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BeanConstructorInjection {

    private BeanToBeInjected 依赖;

    @Autowired
    public BeanConstructorInjection(BeanToBeInjected 依赖) {
        this.dependency = dependency;
    }

    public void sayHello() {
        System.out.print("来自 BeanConstructorInjection 的问候，依赖为: ");
       dependency.sayHello();
    }
}
```

属性注入

注意 @Autowired 注解标记了符合 JavaBeans 标准的 setter 方法。

```
}
```

Bean Declaration by Annotation

The `@Component` annotation serves to demarcate the POJO as a Spring bean available for registration during component scanning.

```
@Component
public class BeanDeclaredByAnnotation {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredByAnnotation !");
    }
}
```

Bean Declaration by Application Configuration

Notice that we don't need to annotate or otherwise mark our POJO since the bean declaration/definition is happening in the Application Configuration class file.

```
public class BeanDeclaredInAppConfig {

    public void sayHello() {
        System.out.println("Hello, World from BeanDeclaredInAppConfig !");
    }
}
```

Constructor Injection

Notice that the `@Autowired` annotation is set at the constructor level. Also notice that unless explicitely defined by name the default autowiring is happening *based on the type* of the bean (in this instance `BeanToBeInjected`).

```
package com.stackoverflow.documentation;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BeanConstructorInjection {

    private BeanToBeInjected dependency;

    @Autowired
    public BeanConstructorInjection(BeanToBeInjected dependency) {
        this.dependency = dependency;
    }

    public void sayHello() {
        System.out.print("Hello, World from BeanConstructorInjection with dependency: ");
        dependency.sayHello();
    }
}
```

Property Injection

Notice that the `@Autowired` annotation demarcates the setter method whose name follows the JavaBeans standard.

```java
package com.stackoverflow.documentation;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BeanPropertyInjection {

    private BeanToBeInjected 依赖;

    @Autowired
    public void setBeanToBeInjected(BeanToBeInjected beanToBeInjected) {
        this.dependency = beanToBeInjected;
    }

    public void sayHello() {
        System.out.println("Hello, World from BeanPropertyInjection !");
    }
}
```

PostConstruct / PreDestroy 钩子

我们可以通过 @PostConstruct 和 @PreDestroy 钩子拦截 Bean 的初始化和销毁。

```java
package com.stackoverflow.documentation;

import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class BeanPostConstructPreDestroy {

   @PostConstruct
    public void pre() {
        System.out.println("BeanPostConstructPreDestroy - PostConstruct");
    }

    public void sayHello() {
        System.out.println(" Hello World, BeanPostConstructPreDestroy !");
    }

    @PreDestroy
    public void post() {
        System.out.println("BeanPostConstructPreDestroy - PreDestroy");
    }
}
```

第1.3节：什么是Spring框架，为什么我们要选择它？

Spring是一个框架，提供了一堆类，使用它我们不需要在代码中编写样板逻辑，因此Spring在J2EE上提供了一个抽象层。

例如，在简单的JDBC应用程序中，程序员负责

1.加载驱动类

```java
package com.stackoverflow.documentation;


import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class BeanPropertyInjection {

    private BeanToBeInjected dependency;

    @Autowired
    public void setBeanToBeInjected(BeanToBeInjected beanToBeInjected) {
        this.dependency = beanToBeInjected;
    }

    public void sayHello() {
        System.out.println("Hello, World from BeanPropertyInjection !");
    }
}
```

PostConstruct / PreDestroy hooks

We can intercept initialisation and destruction of a Bean by the @PostConstruct and @PreDestroy hooks.

```java
package com.stackoverflow.documentation;

import org.springframework.stereotype.Component;

import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

@Component
public class BeanPostConstructPreDestroy {

    @PostConstruct
    public void pre() {
        System.out.println("BeanPostConstructPreDestroy - PostConstruct");
    }

    public void sayHello() {
        System.out.println(" Hello World, BeanPostConstructPreDestroy !");
    }

    @PreDestroy
    public void post() {
        System.out.println("BeanPostConstructPreDestroy - PreDestroy");
    }
}
```

Section 1.3: What is Spring Framework, why should we go for it?

Spring is a framework, which provides bunch of classes, by using this we don't need to write boiler plate logic in our code, so Spring provides an abstract layer on J2ee.

For Example in Simple JDBC Application programmer is responsible for

1. Loading the driver class

2.创建连接
3.创建语句对象
4.处理异常
5.创建查询
6.执行查询
7.关闭连接

这些被视为样板代码，因为每个程序员都会写相同的代码。为了简化，框架负责处理样板逻辑，程序员只需编写业务逻辑。因此，使用Spring框架我们可以用最少的代码快速开发项目，且无任何错误，开发成本和时间也得以减少。

那么为什么选择Spring，而不是Struts呢？

Strut 是一个仅针对 Web 方面提供解决方案的框架，而 Struts 具有侵入性。Spring 相较于 Struts 有许多功能，因此我们必须选择 Spring。

1. Spring 本质上是非侵入式的：这意味着你不需要继承任何类或实现任何接口
   接口到您的类。
2. Spring 是多功能的：这意味着它可以与您项目中的任何现有技术集成。
3. Spring 提供端到端项目开发：这意味着我们可以开发所有模块，如业务
   层，持久层。
4. Spring 是轻量级的：这意味着如果你想在特定模块上工作，那么你不需要学习
   完整的Spring，只学习特定模块（例如Spring Jdbc，Spring DAO）
5. Spring支持***依赖注入***。
6. Spring 支持多项目开发例如：核心 Java 应用程序、Web 应用程序、分布式
   应用程序、企业应用程序。

7.Spring 支持面向切面编程（AOP）以处理横切关注点。

所以最终我们可以说 Spring 是 Struts 的一种替代方案。但 Spring 并不是 J2EE API 的替代品，因为 Spring
内部使用的类是基于 J2EE API 类的。Spring 是一个庞大的框架，因此它被划分为多个模块。
除了 Spring Core 外，没有模块依赖于其他模块。一些重要的模块有

1. Spring Core
2. Spring JDBC
3. Spring AOP

4.Spring 事务管理

5. Spring ORM
6. Spring MVC

2. Creating the connection
3. Creating statement object
4. Handling the exceptions
5. Creating query
6. Executing query
7. Closing the connection

Which is treated as boilerplate code as every programmer write the same code. So for simplicity the framework takes care of boilerplate logic and the programmer has to write only business logic. So by using Spring framework we can develop projects rapidly with minimum lines of code, without any bug, the development cost and time also reduced.

So Why to choose Spring as struts is there

Strut is a framework which provide solution to web aspects only and struts is invasive in nature. Spring has many features over struts so we have to choose Spring.

1. Spring is ***Noninvasive*** in nature: That means you don't need to extend any classes or implement any interfaces to your class.
2. Spring is ***versatile***: That means it can integrated with any existing technology in your project.
3. Spring provides ***end to end*** project development: That means we can develop all the modules like business layer, persistence layer.
4. Spring is ***light weight***: That means if you want to work on particular module then , you don't need to learn complete spring, only learn that particular module(eg. Spring Jdbc, Spring DAO)
5. Spring supports ***dependency injection***.
6. Spring supports ***multiple project development*** eg: Core java Application, Web Application, Distributed Application, Enterprise Application.
7. Spring supports Aspect oriented Programming for cross cutting concerns.

So finally we can say Spring is an alternative to Struts. But Spring is not a replacement of J2EE API, As Spring supplied classes internally uses J2EE API classes. Spring is a vast framework so it has divided into several modules. No module is dependent to another except Spring Core. Some Important modules are

1. Spring Core
2. Spring JDBC
3. Spring AOP
4. Spring Transaction
5. Spring ORM
6. Spring MVC

第二章：Spring Core

第2.1节：Spring Core 介绍

Spring 是一个庞大的框架，因此 Spring 框架被划分为多个模块，这使得 Spring 轻量化。一些重要的模块有：

1.Spring 核心
2. Spring AOP
3. Spring JDBC
4.Spring 事务
5. Spring ORM
6. Spring MVC

除了 Spring 核心之外，所有 Spring 模块都是相互独立的。由于 Spring 核心是基础模块，因此在所有模块中都必须使用 Spring 核心

Spring 核心

Spring 核心主要讲述依赖管理。也就是说，如果向 Spring 提供任意类，Spring 可以管理其依赖关系。

什么是依赖：
从项目角度来看，一个项目或应用中有多个具有不同功能的类，每个类都需要其他类的一些功能。

示例：

```
class Engine {

  public void start() {
    System.out.println("引擎启动");
  }
}

class Car {

  public void move() {
    // 由于需要移动 engine 类的 start() 方法
  }
}
```

这里 Engine 类被 Car 类所依赖，所以我们可以说 Engine 类依赖于 Car 类，因此我们可以通过继承或创建对象来管理这些依赖，如下所示。

通过继承：

```
class Engine {

  public void start() {
    System.out.println("引擎已启动");
  }
}

class Car extends Engine {
```

Chapter 2: Spring Core

Section 2.1: Introduction to Spring Core

Spring is a vast framework, so the Spring framework has been divided in several modules which makes spring lightweight. Some important modules are:

1. Spring Core
2. Spring AOP
3. Spring JDBC
4. Spring Transaction
5. Spring ORM
6. Spring MVC

All the modules of Spring are independent of each other except Spring Core. As Spring core is the base module, so in all module we have to use Spring Core

Spring Core

Spring Core talking all about dependency management.That means if any arbitrary classes provided to spring then Spring can manage dependency.

What is a dependency:
From project point of view, in a project or application multiple classes are there with different functionality. and each classes required some functionality of other classes.

Example:

```
class Engine {

  public void start() {
    System.out.println("Engine started");
  }
}

class Car {

  public void move() {
    // For moving start() method of engine class is required
  }
}
```

Here class Engine is required by class car so we can say class engine is dependent to class Car, So instead of we managing those dependency by Inheritance or creating object as fallows.

By Inheritance:

```
class Engine {

  public void start() {
    System.out.println("Engine started");
  }
}

class Car extends Engine {
  public void move() {
start(); // 调用父类的 start 方法，
  }
}
```

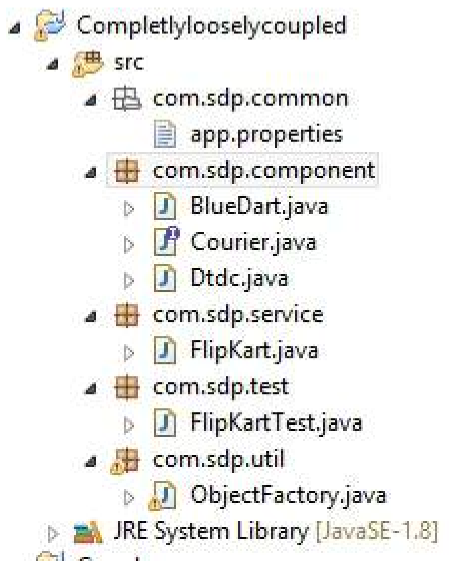
通过创建依赖类的对象：

```
class Engine {

    public void start() {
        System.out.println("引擎启动");
    }
}

class Car {

Engine eng = new Engine();

    public void move() {
      eng.start();
    }
}
```

因此，与其我们自己管理类之间的依赖关系，不如让 Spring Core 来负责依赖管理。但有一些规则，类必须采用某种设计技术来设计，即策略设计模式（Strategy design pattern）。

第2.2节：理解Spring如何管理依赖？

让我写一段代码，展示完全松耦合的情况，这样你就能轻松理解Spring核心如何在内部管理依赖。考虑一个场景，在线业务Flipkart存在，它有时使用DTDC或Blue Dart快递服务，所以让我设计一个展示完全松耦合的应用程序。Eclipse目录如下：



```
Completlylooselycoupled
  src
    com.sdp.common
      app.properties
    com.sdp.component
      BlueDart.java
      Courier.java
      Dtdc.java
    com.sdp.service
      FlipKart.java
    com.sdp.test
      FlipKartTest.java
    com.sdp.util
      ObjectFactory.java
  JRE System Library [JavaSE-1.8]
```

```
//接口
package com.sdp.component;

public interface Courier {
    public String deliver(String iteams,String address);

}
```

```
    public void move() {
      start(); //Calling super class start method,
    }
}
```

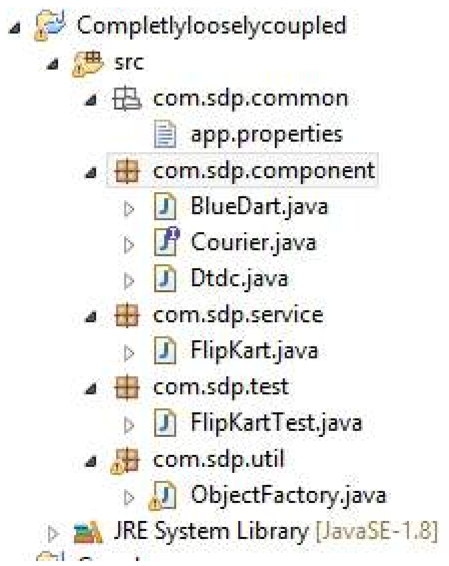
By creating object of dependent class:

```
class Engine {

    public void start() {
        System.out.println("Engine started");
    }
}

class Car {

    Engine eng = new Engine();

    public void move() {
      eng.start();
    }
}
```

So instead of we managing dependency between classes spring core takes the responsibility dependency management. But Some rule are there, The classes must be designed with some design technique that is Strategy design pattern.

Section 2.2: Understanding How Spring Manage Dependency?

Let me write a piece of code which shows completely loosely coupled, Then you can easily understand how Spring core manage the dependency internally. Consider an scenario, Online business Flipkart is there, it uses some times DTDC or Blue Dart courier service , So let me design a application which shows complete loosely coupled. The Eclipse Directory as fallows:



```
Completlylooselycoupled
  src
    com.sdp.common
      app.properties
    com.sdp.component
      BlueDart.java
      Courier.java
      Dtdc.java
    com.sdp.service
      FlipKart.java
    com.sdp.test
      FlipKartTest.java
    com.sdp.util
      ObjectFactory.java
  JRE System Library [JavaSE-1.8]
```

```
//Interface
package com.sdp.component;

public interface Courier {
    public String deliver(String iteams,String address);

}
```

//实现类

```
  package com.sdp.component;

public class BlueDart implements Courier {


    public String deliver(String iteams, String address) {

        return iteams+ "已发货至地址 "+address +"通过 BlueDart";
    }

}

package com.sdp.component;

public class Dtdc implements Courier {


    public String deliver(String iteams, String address) {
        return iteams+ "Shiped to Address "+address +"Through Dtdc";     }

}
```

//组件类

```
package com.sdp.service;

import com.sdp.component.Courier;

public class FlipKart {
    private Courier courier;

    public void setCourier(Courier courier) {
        this.courier = courier;
    }
    public void shopping(String iteams,String address)
    {
        String status=courier.deliver(iteams, address);
        System.out.println(status);
    }

}
```

//工厂类用于创建和返回对象

```
package com.sdp.util;

import java.io.IOException;
import java.util.Properties;

import com.sdp.component.Courier;

public class ObjectFactory {
private static Properties props;
static{

props=new Properties();
    try {
```

//implementation classes

```
  package com.sdp.component;

public class BlueDart implements Courier {


    public String deliver(String iteams, String address) {

        return iteams+ "Shiped to Address "+address +"Through BlueDart";
    }

}

package com.sdp.component;

public class Dtdc implements Courier {


    public String deliver(String iteams, String address) {
        return iteams+ "Shiped to Address "+address +"Through Dtdc";     }

}
```

//Component classe

```
package com.sdp.service;

import com.sdp.component.Courier;

public class FlipKart {
    private Courier courier;

    public void setCourier(Courier courier) {
        this.courier = courier;
    }
    public void shopping(String iteams,String address)
    {
        String status=courier.deliver(iteams, address);
        System.out.println(status);
    }

}
```

//Factory classes to create and return Object

```
package com.sdp.util;

import java.io.IOException;
import java.util.Properties;

import com.sdp.component.Courier;

public class ObjectFactory {
private static Properties props;
static{

    props=new Properties();
    try {
```

```
props.load(ObjectFactory.class.getClassLoader().getResourceAsStream("com//sdp//common//app.properti
es"));
    } catch (IOException e) {
        // TODO 自动生成的 catch 块
e.printStackTrace();
    }

}
public static Object getInstance(String logicalclassName)
{
    Object obj = null;
    String originalclassName=props.getProperty(logicalclassName);
    try {
obj=Class.forName(originalclassName).newInstance();
    } catch (InstantiationException e) {
        // TODO 自动生成的 catch 块
e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO 自动生成的 catch 块
e.printStackTrace();
    } catch (ClassNotFoundException e) {
        // TODO 自动生成的 catch 块
e.printStackTrace();
    }
    return obj;
}

    }
```

//属性文件

```
BlueDart.class=com.sdp.component.BlueDart
Dtdc.class=com.sdp.component.Dtdc
FlipKart.class=com.sdp.service.FlipKart
```

//测试类

```
package com.sdp.test;

import com.sdp.component.Courier;
import com.sdp.service.FlipKart;
import com.sdp.util.ObjectFactory;

public class FlipKartTest {
    public static void main(String[] args) {
Courier courier=(Courier)ObjectFactory.getInstance("Dtdc.class");
        FlipKart flipkart=(FlipKart)ObjectFactory.getInstance("FlipKart.class");
        flipkart.setCourier(courier);
flipkart.shopping("Hp Laptop", "SR Nagar,Hyderabad");

    }

}
```

如果我们编写这段代码，那么我们可以手动实现松耦合，这适用于所有类都需要BlueDart或Dtdc的情况，但如果有些类需要BlueDart，而另一些类需要Dtdc，那么它们又会紧耦合。因此，不是由我们来创建和管理依赖注入，而是由Spring核心负责创建和管理bean。希望这对你有帮助，下一例子中我们将看到Spring核心的第一个应用及其详细内容。

```
props.load(ObjectFactory.class.getClassLoader().getResourceAsStream("com//sdp//common//app.properti
es"));
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

}
public static Object getInstance(String logicalclassName)
{
    Object obj = null;
    String originalclassName=props.getProperty(logicalclassName);
    try {
         obj=Class.forName(originalclassName).newInstance();
    } catch (InstantiationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (ClassNotFoundException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
    return obj;
}

    }
```

//properties file

```
BlueDart.class=com.sdp.component.BlueDart
Dtdc.class=com.sdp.component.Dtdc
FlipKart.class=com.sdp.service.FlipKart
```

//Test class

```
package com.sdp.test;

import com.sdp.component.Courier;
import com.sdp.service.FlipKart;
import com.sdp.util.ObjectFactory;

public class FlipKartTest {
    public static void main(String[] args) {
        Courier courier=(Courier)ObjectFactory.getInstance("Dtdc.class");
        FlipKart flipkart=(FlipKart)ObjectFactory.getInstance("FlipKart.class");
        flipkart.setCourier(courier);
        flipkart.shopping("Hp Laptop", "SR Nagar,Hyderabad");

    }

}
```

If we write this code then we can manually achieve loose coupling,this is applicable if all the classes want either BlueDart or Dtdc , But if some class want BlueDart and some other class want Dtdc then again it will be tightly coupled, So instead of we creating and managing the dependency injection Spring core takes the responsibility of creating and managing the beans, Hope This will helpful, in next example we wil see the !st application on Spring core with deitals

第3章：Spring表达式语言 (SpEL)

第3.1节：语法参考

您可以使用@Value("#{expression}")在运行时注入值，其中 expression是一个SpEL表达式。

字面表达式

支持的类型包括字符串、日期、数值（整数、实数和十六进制）、布尔值和null。

```
"#{'Hello World'}"  //字符串
"#{3.1415926}"      //数值（双精度）
"#{true}"           //布尔值
"#{null}"           //null
```

内联列表

```
"#{1,2,3,4}"              //数字列表
"#{{'a','b'},{'x','y'}}"  //列表的列表
```

内联映射

```
"#{name:'尼古拉',dob:'1856年7月10日'}"
"#{name:{first:'尼古拉',last:'特斯拉'},dob:{day:10,month:'七月',year:1856}}" //map of maps
```

调用方法

```
"#{'abc'.length()}"      //计算结果为3
"#{f('hello')}"   //f是该表达式所属类中的一个方法，它有一个字符串参数
```

Chapter 3: Spring Expression Language (SpEL)

Section 3.1: Syntax Reference

You can use `@Value("#{expression}")` to inject value at runtime, in which the `expression` is a SpEL expression.

Literal expressions

Supported types include strings, dates, numeric values (int, real, and hex), boolean and null.

```
"#{'Hello World'}"  //strings
"#{3.1415926}"      //numeric values (double)
"#{true}"           //boolean
"#{null}"           //null
```

Inline list

```
"#{1,2,3,4}"              //list of number
"#{{'a','b'},{'x','y'}}"  //list of list
```

Inline Maps

```
"#{name:'Nikola',dob:'10-July-1856'}"
"#{name:{first:'Nikola',last:'Tesla'},dob:{day:10,month:'July',year:1856}}" //map of maps
```

Invoking Methods

```
"#{'abc'.length()}"      //evaluates to 3
"#{f('hello')}"   //f is a method in the class to which this expression belongs, it has a string
parameter
```

第4章：从SimpleJdbcCall获取SqlRowSet

本文介绍如何使用SimpleJdbcCall直接获取SqlRowSet，适用于数据库中带有游标输出参数的存储过程

我正在使用Oracle数据库，尝试创建一个示例，应该适用于其他数据库，
我的Oracle示例详细说明了Oracle相关的问题。

第4.1节：SimpleJdbcCall的创建

通常，您会希望在服务层创建SimpleJdbcCall。

本示例假设您的存储过程只有一个输出参数，且为游标；您需要调整declareParameters以匹配您的存储过程。

```
@Service
public class MyService() {

@Autowired
    private DataSource dataSource;

    // 自动装配您的配置，例如
   @Value("${db.procedure.schema}")
    String schema;

    private SimpleJdbcCall myProcCall;

    // 在属性配置完成后创建 SimpleJdbcCall
    @PostConstruct
    void initialize() {
        this.myProcCall = new SimpleJdbcCall(dataSource)
                        .withProcedureName("my_procedure_name")
                        .withCatalogName("my_package")
.withSchemaName(schema)
                        .declareParameters(new SqlOutParameter(
                            "out_param_name",
                            Types.REF_CURSOR,
                            new SqlRowSetResultSetExtractor()));
    }

    public SqlRowSet myProc() {
Map<String, Object> out = this.myProcCall.execute();
        return (SqlRowSet) out.get("out_param_name");
    }

}
```

这里有许多可供选择的选项：

- **withoutProcedureColumnMetaDataAccess()** 如果你有重载的存储过程名称，或者只是不想让SimpleJdbcCall去数据库验证，则需要使用此方法。
- withReturnValue() 如果存储过程有返回值。传递给declareParameters的第一个值定义了返回值。此外，如果你的存储过程是函数，执行时请使用withFunctionName和executeFunction。

- withNamedBinding() 如果你想使用参数名而不是位置来传递参数。

Chapter 4: Obtaining a SqlRowSet from SimpleJdbcCall

This describes how to directly obtain a **SqlRowSet** using **SimpleJdbcCall** with a stored procedure in your database that has a **cursor output parameter**,

I am working with an Oracle database, I've attempted to create an example that should work for other databases, my Oracle example details issues with Oracle.

Section 4.1: SimpleJdbcCall creation

Typically, you will want to create your SimpleJdbcCalls in a Service.

This example assumes your procedure has a single output parameter that is a cursor; you will need to adjust your declareParameters to match your procedure.

```
@Service
public class MyService() {

@Autowired
    private DataSource dataSource;

    // Autowire your configuration, for example
    @Value("${db.procedure.schema}")
    String schema;

    private SimpleJdbcCall myProcCall;

    // create SimpleJdbcCall after properties are configured
    @PostConstruct
    void initialize() {
        this.myProcCall = new SimpleJdbcCall(dataSource)
                        .withProcedureName("my_procedure_name")
                        .withCatalogName("my_package")
                        .withSchemaName(schema)
                        .declareParameters(new SqlOutParameter(
                            "out_param_name",
                            Types.REF_CURSOR,
                            new SqlRowSetResultSetExtractor()));
    }

    public SqlRowSet myProc() {
        Map<String, Object> out = this.myProcCall.execute();
        return (SqlRowSet) out.get("out_param_name");
    }

}
```

There are many options you can use here:

- **withoutProcedureColumnMetaDataAccess()** needed if you have overloaded procedure names or just don't want SimpleJdbcCall to validate against the database.
- **withReturnValue()** if procedure has a return value. First value given to declareParameters defines the return value. Also, if your procedure is a function, use **withFunctionName** and **executeFunction** when executing.
- **withNamedBinding()** if you want to give arguments using names instead of position.

- useInParameterNames() 定义参数顺序。我认为如果你传入的参数是列表而不是参数名到值的映射，可能需要使用此方法。不过这可能仅在你使用withoutProcedureColumnMetaDataAccess()时才需要。

第4.2节：Oracle数据库

以下是解决Oracle相关问题的方法。

假设你的存储过程输出参数是ref cursor，你会遇到以下异常。

> java.sql.SQLException: 无效的列类型：2012

因此，在simpleJdbcCall.declareParameters()中，将Types.REF_CURSOR改为OracleTypes.CURSOR

以支持OracleTypes。

你可能只需要在数据中存在某些列类型时才进行此更改。

我遇到的下一个问题是，专有类型如`oracle.sql.TIMESTAMPTZ`导致了SqlRowSetResultSetExtractor中的此错误：

> 列的SQL类型无效；嵌套异常是java.sql.SQLException：列的SQL类型无效

因此我们需要创建一个支持Oracle类型的ResultSetExtractor。
我将在这段代码之后解释密码的原因。

```
package com.boost.oracle;

import oracle.jdbc.rowset.OracleCachedRowSet;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet;
import org.springframework.jdbc.support.rowset.SqlRowSet;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
* OracleTypes可能导致{@link org.springframework.jdbc.core.SqlRowSetResultSetExtractor}
 * 因为Oracle SQL类型不在标准{@link java.sql.Types}中而失败。
 *
* 此外，诸如{@link oracle.sql.TIMESTAMPTZ}之类的类型在处理
 * ResultSet时需要Connection；{@link OracleCachedRowSet#getConnectionInternal()}需要JNDI
* 数据源名称或要设置的用户名和密码。
 *
* 目前我决定只设置密码，因为将 SpringBoot 更改为 JNDI 数据源配置有点复杂。

 *
* 由 Arlo White 于 2017 年 2 月 23 日创建。
 */
public class OracleSqlRowSetResultSetExtractor implements ResultSetExtractor<SqlRowSet> {

    private String oraclePassword;

    public OracleSqlRowSetResultSetExtractor(String oraclePassword) {
```

- **useInParameterNames()** defines the argument order. I think this may be required if you pass in your arguments as a list instead of a map of argument name to value. Though it may only be required if you use withoutProcedureColumnMetaDataAccess()

Section 4.2: Oracle Databases

Here's how to resolve issues with Oracle.

Assuming your procedure output parameter is `ref cursor`, you will get this exception.

> java.sql.SQLException: Invalid column type: 2012

So change `Types.REF_CURSOR` to `OracleTypes.CURSOR` in **simpleJdbcCall.declareParameters()**

Supporting OracleTypes

You may only need to do this if you have certain column types in your data.

The next issue I encountered was that proprietary Types such as `oracle.sql.TIMESTAMPTZ` caused this error in SqlRowSetResultSetExtractor:

> Invalid SQL type for column; nested exception is java.sql.SQLException: Invalid SQL type for column

So we need to create a **ResultSetExtractor** that supports Oracle types.
I will explain the reason for password after this code.

```
package com.boost.oracle;

import oracle.jdbc.rowset.OracleCachedRowSet;
import org.springframework.dao.DataAccessException;
import org.springframework.jdbc.core.ResultSetExtractor;
import org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet;
import org.springframework.jdbc.support.rowset.SqlRowSet;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * OracleTypes can cause {@link org.springframework.jdbc.core.SqlRowSetResultSetExtractor}
 * to fail due to a Oracle SQL type that is not in the standard {@link java.sql.Types}.
 *
 * Also, types such as {@link oracle.sql.TIMESTAMPTZ} require a Connection when processing
 * the ResultSet; {@link OracleCachedRowSet#getConnectionInternal()} requires a JNDI
 * DataSource name or the username and password to be set.
 *
 * For now I decided to just set the password since changing SpringBoot to a JNDI DataSource
 * configuration is a bit complicated.
 *
 * Created by Arlo White on 2/23/17.
 */
public class OracleSqlRowSetResultSetExtractor implements ResultSetExtractor<SqlRowSet> {

    private String oraclePassword;

    public OracleSqlRowSetResultSetExtractor(String oraclePassword) {
        this.oraclePassword = oraclePassword;
    }

    @Override
    public SqlRowSet extractData(ResultSet rs) throws SQLException, DataAccessException {
        OracleCachedRowSet cachedRowSet = new OracleCachedRowSet();
        // 允许 getConnectionInternal 获取 TIMESTAMPTZ 的连接
cachedRowSet.setPassword(oraclePassword);
        cachedRowSet.populate(rs);
        return new ResultSetWrappingSqlRowSet(cachedRowSet);
    }

}
```

某些 Oracle 类型需要通过连接从 ResultSet 中获取列值。TIMESTAMPTZ 就是其中之一。
因此当调用 `rowSet.getTimestamp(colIndex)` 时，会抛出以下异常：

> 原因：java.sql.SQLException：未设置一个或多个认证的 RowSet 属性
> at oracle.jdbc.rowset.OracleCachedRowSet.getConnectionInternal(OracleCachedRowSet.java:560)
> at oracle.jdbc.rowset.OracleCachedRowSet.getTimestamp(OracleCachedRowSet.java:3717)
> at org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet.getTimestamp

如果深入查看这段代码，你会发现 OracleCachedRowSet 需要密码或 JNDI DataSource 名称来获取连接。
如果你更喜欢使用 JNDI 查找，只需确认 OracleCachedRowSet 已设置 DataSourceName。

所以在我的服务中，我通过 Autowire 注入密码，并这样声明输出参数：

```
new SqlOutParameter("cursor_param_name", OracleTypes.CURSOR, new
OracleSqlRowSetResultSetExtractor(oraclePassword))
```

```
        this.oraclePassword = oraclePassword;
    }

    @Override
    public SqlRowSet extractData(ResultSet rs) throws SQLException, DataAccessException {
        OracleCachedRowSet cachedRowSet = new OracleCachedRowSet();
        // allows getConnectionInternal to get a Connection for TIMESTAMPTZ
        cachedRowSet.setPassword(oraclePassword);
        cachedRowSet.populate(rs);
        return new ResultSetWrappingSqlRowSet(cachedRowSet);
    }

}
```

Certain Oracle types require a Connection to obtain the column value from a ResultSet. TIMESTAMPTZ is one of these types. So when `rowSet.getTimestamp(colIndex)` is called, you will get this exception:

> Caused by: java.sql.SQLException: One or more of the authenticating RowSet properties not set at
> oracle.jdbc.rowset.OracleCachedRowSet.getConnectionInternal(OracleCachedRowSet.java:560) at
> oracle.jdbc.rowset.OracleCachedRowSet.getTimestamp(OracleCachedRowSet.java:3717) at
> org.springframework.jdbc.support.rowset.ResultSetWrappingSqlRowSet.getTimestamp

If you dig into this code, you will see that the OracleCachedRowSet needs the password or a JNDI DataSource name to get a Connection. If you prefer the JNDI lookup, just verify that OracleCachedRowSet has DataSourceName set.

So in my Service, I Autowire in the password and declare the output parameter like this:

```
new SqlOutParameter("cursor_param_name", OracleTypes.CURSOR, new
OracleSqlRowSetResultSetExtractor(oraclePassword))
```

第5章：创建和使用 Bean

第5.1节：自动装配特定类型的所有 Bean

如果你有同一个接口的多个实现，Spring 可以将它们全部自动装配到一个集合对象中。我将使用一个示例，使用验证器模式1

Foo类：

```
public class Foo {
        private String name;
        private String emailAddress;
        private String errorMessage;
        /** 省略了Getter和Setter **/
}
```

接口：

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

名称验证器类：

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //验证逻辑写在这里。
    }
}
```

邮箱验证器类：

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //不同的验证逻辑写在这里。
    }
}
```

你现在可以将这些验证器单独或一起自动装配到一个类中。

接口：

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

类：

```
@Service
public class FooServiceImpl implements FooService {
    /** 自动装配所有实现了 FooValidator 接口的类**/
    @Autowired
```

Chapter 5: Creating and using beans

Section 5.1: Autowiring all beans of a specific type

If you've got multiple implementations of the same interface, Spring can autowire them all into a collection object. I'm going to use an example using a Validator pattern1

Foo Class:

```
public class Foo {
     private String name;
     private String emailAddress;
     private String errorMessage;
     /** Getters & Setters omitted **/
}
```

Interface:

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

Name Validator Class:

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Validation logic goes here.
    }
}
```

Email Validator Class:

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Different validation logic goes here.
    }
}
```

You can now autowire these validators individually or together into a class.

Interface:

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

Class:

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire all classes implementing FooValidator interface**/
    @Autowired
    private List<FooValidator> allValidators;

    @Override
    public void handleFoo(Foo foo) {
        /**你可以使用列表中的所有实例**/
        for(FooValidator validator : allValidators) {
            foo = validator.validate(foo);
        }
    }
}
```

值得注意的是，如果在Spring IoC容器中有多个接口的实现，并且没有使用@Qualifier注解指定要使用哪一个，Spring在尝试启动时会抛出异常，因为它不知道应该使用哪个实例。

1：这不是进行如此简单验证的正确方法。这是一个关于自动装配的简单示例。如果你想了解一种更简单的验证方法，可以查阅Spring如何使用注解进行验证。

第5.2节：基本注解自动装配

接口：

```
public interface FooService {
    public int doSomething();
}
```

类：

```
@Service
public class FooServiceImpl implements FooService {
    @Override
    public int doSomething() {
        //在这里执行一些操作
        return 0;
    }
}
```

需要注意的是，类必须实现接口，Spring才能自动装配该类。虽然有一种方法允许Spring通过加载时织入自动装配独立类，但这超出了本示例的范围。

你可以在任何由Spring IoC容器实例化的类中使用@Autowired注解访问该bean。

用法：

```
@Autowired([required=true])
```

@Autowired注解会首先尝试按类型自动装配，如果存在歧义，则回退到按bean名称自动装配。

该注解可以通过多种不同方式应用。

构造器注入：

```
public class BarClass() {
    private FooService fooService
```

```
    private List<FooValidator> allValidators;

    @Override
    public void handleFoo(Foo foo) {
        /**You can use all instances from the list**/
        for(FooValidator validator : allValidators) {
            foo = validator.validate(foo);
        }
    }
}
```

It's worth noting that if you have more than one implementation of an interface in the Spring IoC container and don't specify which one you want to use with the `@Qualifier` annotation, Spring will throw an exception when trying to start, because it won't know which instance to use.

1: This is not the right way to do such simple validations. This is a simple example about autowiring. If you want an idea of a much easier validation method look up how Spring does validation with annotations.

Section 5.2: Basic annotation autowiring

Interface:

```
public interface FooService {
    public int doSomething();
}
```

Class:

```
@Service
public class FooServiceImpl implements FooService {
    @Override
    public int doSomething() {
        //Do some stuff here
        return 0;
    }
}
```

It should be noted that a class must implement an interface for Spring to be able to autowire this class. There is a method to allow Spring to autowire stand-alone classes using load time weaving, but that is out of scope for this example.

You can gain access to this bean in any class that instantiated by the Spring IoC container using the `@Autowired` annotation.

Usage:

```
@Autowired([required=true])
```

The `@Autowired` annotation will first attempt to autowire by type, and then fall back on bean name in the event of ambiguity.

This annotation can be applied in several different ways.

Constructor injection:

```
public class BarClass() {
    private FooService fooService
    @Autowired
    public BarClass(FooService fooService) {
        this.fooService = fooService;
    }
}
```

字段注入：

```
public class BarClass() {
    @Autowired
    private FooService fooService;
}
```

Setter注入：

```
public class BarClass() {
    private FooService fooService;

    @Autowired
    public void setFooService(FooService fooService) {
        this.fooService = fooService;
    }
}
```

第5.3节：使用FactoryBean进行动态Bean实例化

为了动态决定注入哪些Bean，我们可以使用FactoryBean。这些类实现了工厂方法模式，为容器提供Bean实例。Spring能够识别它们，并且可以透明地使用，无需知道Bean来自工厂。例如：

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    // 此方法确定Bean的类型以供自动装配使用
    @Override
    public Class<?> getObjectType() {
        return String.class;
    }

    // 此工厂方法生成实际的Bean
    @Override
    protected String createInstance() throws Exception {
        // 你返回的内容可以动态定义，
        // 即从文件、数据库、网络读取，或者
        // 如果你愿意，也可以简单地随机生成。
        return "Something from factory";
    }
}
```

配置：

```
@Configuration
public class ExampleConfig {
    @Bean
    public FactoryBean<String> fromFactory() {
        return new ExampleFactoryBean();
    }
}
```

```
    @Autowired
    public BarClass(FooService fooService) {
        this.fooService = fooService;
    }
}
```

Field injection:

```
public class BarClass() {
    @Autowired
    private FooService fooService;
}
```

Setter injection:

```
public class BarClass() {
    private FooService fooService;

    @Autowired
    public void setFooService(FooService fooService) {
        this.fooService = fooService;
    }
}
```

Section 5.3: Using FactoryBean for dynamic bean instantiation

In order to dynamically decide what beans to inject, we can use `FactoryBeans`. These are classes which implement the factory method pattern, providing instances of beans for the container. They are recognized by Spring and can be used transparently, without need to know that the bean comes from a factory. For example:

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    // This method determines the type of the bean for autowiring purposes
    @Override
    public Class<?> getObjectType() {
        return String.class;
    }

    // this factory method produces the actual bean
    @Override
    protected String createInstance() throws Exception {
        // The thing you return can be defined dynamically,
        // that is read from a file, database, network or just
        // simply randomly generated if you wish.
        return "Something from factory";
    }
}
```

Configuration:

```
@Configuration
public class ExampleConfig {
    @Bean
    public FactoryBean<String> fromFactory() {
        return new ExampleFactoryBean();
    }
}
```

获取 Bean：

```
AbstractApplicationContext context = new AnnotationConfigApplicationContext(ExampleConfig.class);
String exampleString = (String) context.getBean("fromFactory");
```

要获取实际的 FactoryBean，请在 Bean 名称前加上 & 符号：

```
FactoryBean<String> bean = (FactoryBean<String>) context.getBean("&fromFactory");
```

请注意，您只能使用 `prototype` 或 `singleton` 作用域——要将作用域更改为 `prototype` ，请重写 `isSingleton` 方法：

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    @Override
    public boolean isSingleton() {
        return false;
    }

    // 为了可读性省略其他方法
}
```

请注意，作用域指的是实际创建的实例，而不是工厂 Bean 本身。

第5.4节：声明Bean

要声明一个Bean，只需用`@Bean`注解一个方法，或用`@Component`注解一个类（也可以使用注解`@Service`、`@Repository`、`@Controller`）。

当JavaConfig遇到这样的方法时，它会执行该方法，并将返回值注册为BeanFactory中的一个Bean。默认情况下，Bean的名称将是方法名。

我们可以通过以下三种方式之一创建Bean：

1. 使用基于Java的配置：在配置文件中需要使用@Bean注解声明Bean

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

2. 使用基于XML的配置：对于基于XML的配置，我们需要在应用程序配置XML中声明Bean配置XML，即

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

3. 基于注解的组件：对于基于注解的组件，我们需要在想要声明为bean的类上添加@Component注解。需要在想要声明为bean的类上添加@Component注解。

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
```

Getting the bean:

```
AbstractApplicationContext context = new AnnotationConfigApplicationContext(ExampleConfig.class);
String exampleString = (String) context.getBean("fromFactory");
```

To get the actual FactoryBean, use the ampersand prefix before the bean's name:

```
FactoryBean<String> bean = (FactoryBean<String>) context.getBean("&fromFactory");
```

Please note that you can only use `prototype` or `singleton` scopes - to change the scope to `prototype` override `isSingleton` method:

```
public class ExampleFactoryBean extends AbstractFactoryBean<String> {
    @Override
    public boolean isSingleton() {
        return false;
    }

    // other methods omitted for readability reasons
}
```

Note that scoping refers to the actual instances being created, not the factory bean itself.

Section 5.4: Declaring Bean

To declare a bean, simply annotate a method with the `@Bean` annotation or annotate a class with the `@Component` annotation (annotations `@Service`, `@Repository`, `@Controller` could be used as well).

When JavaConfig encounters such a method, it will execute that method and register the return value as a bean within a BeanFactory. By default, the bean name will be that of the method name.

We can create bean using one of three ways:

1. **Using Java based Configuration**: In Configuration file we need to declare bean using @bean annotation

```
@Configuration
public class AppConfig {
    @Bean
    public TransferService transferService() {
        return new TransferServiceImpl();
    }
}
```

2. **Using XML based configuration**: For XML based configuration we need to create declare bean in application configuration XML i.e.

```
<beans>
    <bean name="transferService" class="com.acme.TransferServiceImpl"/>
</beans>
```

3. **Annotation-Driven Component**: For annotation-driven components, we need to add the @Component annotation to the class we want to declare as bean.

```
@Component("transferService")
public class TransferServiceImpl implements TransferService {
    ...
}
```

现在，所有名为transferService的三个bean都可以在BeanFactory或ApplicationContext中使用。

第5.5节：使用@Qualifier自动装配特定的bean实例

如果你有同一接口的多个实现，Spring需要知道应该自动装配哪个实现到某个类中。这个例子中我将使用验证器（Validator）模式。1

Foo类：

```
public class Foo {
     private String name;
     private String emailAddress;
     private String errorMessage;
     /** 省略了Getter和Setter **/
}
```

接口：

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

名称验证器类：

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //验证逻辑写在这里。
    }
}
```

邮箱验证器类：

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //不同的验证逻辑写在这里。
    }
}
```

你现在可以将这些验证器分别自动装配到某个类中。

接口：

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

类：

```
    ...
}
```

Now all three beans with name `transferService` are available in `BeanFactory` or `ApplicationContext`.

Section 5.5: Autowiring specific bean instances with @Qualifier

If you've got multiple implementations of the same interface, Spring needs to know which one it should autowire into a class. I'm going to use a Validator pattern in this example.1

Foo Class:

```
public class Foo {
     private String name;
     private String emailAddress;
     private String errorMessage;
     /** Getters & Setters omitted **/
}
```

Interface:

```
public interface FooValidator {
    public Foo validate(Foo foo);
}
```

Name Validator Class:

```
@Component(value="FooNameValidator")
public class FooNameValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Validation logic goes here.
    }
}
```

Email Validator Class:

```
@Component(value="FooEmailValidator")
public class FooEmailValidator implements FooValidator {
    @Override
    public Foo validate(Foo foo) {
        //Different validation logic goes here.
    }
}
```

You can now autowire these validators individually into a class.

Interface:

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

Class:

```
@Service
public class FooServiceImpl implements FooService {
    /** 单独自动装配验证器 **/
    @Autowired
    /*
* 注意这里的字符串值如何与@Component注解上的值匹配？这就是Sprin
g知道要自动装配哪个实例的方式。

     */
@Qualifier("FooNameValidator")
    private FooValidator nameValidator;

    @Autowired
@Qualifier("FooEmailValidator")
    private FooValidator emailValidator;

    @Override
    public void handleFoo(Foo foo) {
        /**如果需要，你也可以只使用一个实例**/
       foo = nameValidator.validate(foo);
    }
}
```

值得注意的是，如果在Spring IoC容器中有多个接口的实现，并且没有使用@Qualifier注解指定要使用哪一个，Spring在尝试启动时会抛出异常，因为它不知道应该使用哪个实例。

1：这不是进行如此简单验证的正确方法。这是一个关于自动装配的简单示例。如果你想了解一种更简单的验证方法，可以查阅Spring如何使用注解进行验证。

第5.6节：使用泛型类型参数自动装配特定类的实例

如果你有一个带泛型类型参数的接口，Spring可以利用它只自动装配实现了你指定类型参数的实现类。

接口：

```
公共接口 GenericValidator<T> {
    公共 T 验证(T 对象);
}
```

Foo 验证器类：

```
@Component
公共类 FooValidator 实现 GenericValidator<Foo> {
    @Override
    公共 Foo 验证(Foo foo) {
        //此处为验证 foo 对象的逻辑。
    }
}
```

Bar 验证器类：

```
@Component
公共类 BarValidator 实现 GenericValidator<Bar> {
    @Override
    公共 Bar 验证(Bar bar) {
```

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire validators individually **/
    @Autowired
    /*
     * Notice how the String value here matches the value
     * on the @Component annotation? That's how Spring knows which
     * instance to autowire.
     */
    @Qualifier("FooNameValidator")
    private FooValidator nameValidator;

    @Autowired
    @Qualifier("FooEmailValidator")
    private FooValidator emailValidator;

    @Override
    public void handleFoo(Foo foo) {
        /**You can use just one instance if you need**/
        foo = nameValidator.validate(foo);
    }
}
```

It's worth noting that if you have more than one implementation of an interface in the Spring IoC container and don't specify which one you want to use with the `@Qualifier` annotation, Spring will throw an exception when trying to start, because it won't know which instance to use.

1: This is not the right way to do such simple validations. This is a simple example about autowiring. If you want an idea of a much easier validation method look up how Spring does validation with annotations.

Section 5.6: Autowiring specific instances of classes using generic type parameters

If you've got an interface with a generic type parameter, Spring can use that to only autowire implementations that implement a type parameter you specify.

Interface:

```
public interface GenericValidator<T> {
    public T validate(T object);
}
```

Foo Validator Class:

```
@Component
public class FooValidator implements GenericValidator<Foo> {
    @Override
    public Foo validate(Foo foo) {
        //Logic here to validate foo objects.
    }
}
```

Bar Validator Class:

```
@Component
public class BarValidator implements GenericValidator<Bar> {
    @Override
    public Bar validate(Bar bar) {
        //此处为 Bar 验证逻辑
    }
}
```

现在你可以使用类型参数自动装配这些验证器，以决定自动装配哪个实例。

接口：

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

类：

```
@Service
公共类 FooServiceImpl 实现 FooService {
    /** 自动装配 Foo 验证器 **/
    @Autowired
    private GenericValidator<Foo> fooValidator;

    @Override
    public void handleFoo(Foo foo) {
        foo = fooValidator.validate(foo);
    }
}
```

第5.7节：将原型作用域的Bean注入单例Bean中

容器只创建一次单例Bean并注入其协作者。当单例Bean拥有原型作用域的协作者时，这不是期望的行为，因为原型作用域的Bean应在每次通过访问器访问时注入。

对此问题有几种解决方案：

1. 使用查找方法注入
2. 通过javax.inject.Provider获取原型作用域的Bean
3. 通过org.springframework.beans.factory.ObjectFactory获取原型作用域的Bean（#2的等价方法，但使用Spring特定的类）
   通过实现ApplicationContextAware接口使单例Be
4. an感知容器

方法#3和#4通常不推荐使用，因为它们会使应用程序强绑定于Spring框架。因此，本示例中不涉及这两种方法。

通过XML配置和抽象方法实现查找方法注入

Java 类

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // 每次调用创建新实例
        ...
    }
```

```
        //Bar validation logic here
    }
}
```

You can now autowire these validators using type parameters to decide which instance to autowire.

Interface:

```
public interface FooService {
    public void handleFoo(Foo foo);
}
```

Class:

```
@Service
public class FooServiceImpl implements FooService {
    /** Autowire Foo Validator **/
    @Autowired
    private GenericValidator<Foo> fooValidator;

    @Override
    public void handleFoo(Foo foo) {
        foo = fooValidator.validate(foo);
    }
}
```

Section 5.7: Inject prototype-scoped beans into singletons

The container creates a singleton bean and injects collaborators into it only once. This is not the desired behavior when a singleton bean has a prototype-scoped collaborator, since the prototype-scoped bean should be injected every time it is being accessed via accessor.

There are several solutions to this problem:

1. Use lookup method injection
2. Retrieve a prototype-scoped bean via `javax.inject.Provider`
3. Retrieve a prototype-scoped bean via `org.springframework.beans.factory.ObjectFactory` (an equivalent of #2, but with the class that is specific to Spring)
4. Make a singleton bean container aware via implementing `ApplicationContextAware` interface

Approaches #3 and #4 are generally discouraged, since they strongly tie an app to Spring framework. Thus, they are not covered in this example.

Lookup method injection via XML configuration and an abstract method

Java Classes

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }
    protected abstract Window createNewWindow(); // 查找方法
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <lookup-method name="createNewWindow" bean="window"/>
</bean>
```

通过Java配置和@Component进行查找方法注入

Java 类

```
public class Window {
}

@Component
public class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // 每次调用创建新实例
        ...
    }

    @Lookup
    protected Window createNewWindow() {
        throw new UnsupportedOperationException();
    }
}
```

Java配置

```
@Configuration
@ComponentScan("somepackage") // WindowGenerator所在的包
public class MyConfiguration {

    @Bean
    @Lazy
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Window window() {
        return new Window();
    }
}
```

通过Java配置的手动查找方法注入

Java 类

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // 每次调用创建新实例
        ...
    }
```

```
    protected abstract Window createNewWindow(); // lookup method
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <lookup-method name="createNewWindow" bean="window"/>
</bean>
```

Lookup method injection via Java configuration and @Component

Java Classes

```
public class Window {
}

@Component
public class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }

    @Lookup
    protected Window createNewWindow() {
        throw new UnsupportedOperationException();
    }
}
```

Java configuration

```
@Configuration
@ComponentScan("somepackage") // package where WindowGenerator is located
public class MyConfiguration {

    @Bean
    @Lazy
    @Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Window window() {
        return new Window();
    }
}
```

Manual lookup method injection via Java configuration

Java Classes

```
public class Window {
}

public abstract class WindowGenerator {

    public Window generateWindow() {
        Window window = createNewWindow(); // new instance for each call
        ...
    }
    protected abstract Window createNewWindow(); // 查找方法
}
```

Java配置

```
@Configuration
public class MyConfiguration {

    @Bean
    @Lazy
@Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Window window() {
        return new Window();
    }

    @Bean
    public WindowGenerator windowGenerator(){
        return new WindowGenerator() {
            @Override
            protected Window createNewWindow(){
                return window();
            }
        };
    }
}
```

通过javax.inject.Provider

Java类将原型作用域的Bean注入单例

```
public class Window {
}

public class WindowGenerator {

    private final Provider<Window> windowProvider;

    public WindowGenerator(final Provider<Window> windowProvider) {
        this.windowProvider = windowProvider;
    }

    public Window generateWindow() {
        Window window = windowProvider.get(); // 每次调用创建新实例
        ...
    }
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <constructor-arg>
        <bean class="org.springframework.beans.factory.config.ProviderCreatingFactoryBean">
            <property name="targetBeanName" value="window"/>
        </bean>
    </constructor-arg>
</bean>
```

相同的方法也可以用于其他作用域（例如，将请求作用域的bean注入到单例中）。

```
    protected abstract Window createNewWindow(); // lookup method
}
```

Java configuration

```
@Configuration
public class MyConfiguration {

    @Bean
    @Lazy
    @Scope(scopeName = ConfigurableBeanFactory.SCOPE_PROTOTYPE)
    public Window window() {
        return new Window();
    }

    @Bean
    public WindowGenerator windowGenerator(){
        return new WindowGenerator() {
            @Override
            protected Window createNewWindow(){
                return window();
            }
        };
    }
}
```

Injection of a protoype-scoped bean into singleton via javax.inject.Provider

Java classes

```
public class Window {
}

public class WindowGenerator {

    private final Provider<Window> windowProvider;

    public WindowGenerator(final Provider<Window> windowProvider) {
        this.windowProvider = windowProvider;
    }

    public Window generateWindow() {
        Window window = windowProvider.get(); // new instance for each call
        ...
    }
}
```

XML

```
<bean id="window" class="somepackage.Window" scope="prototype" lazy-init="true"/>

<bean id="windowGenerator" class="somepackage.WindowGenerator">
    <constructor-arg>
        <bean class="org.springframework.beans.factory.config.ProviderCreatingFactoryBean">
            <property name="targetBeanName" value="window"/>
        </bean>
    </constructor-arg>
</bean>
```

The same approaches can be used for other scopes as well (e.g. for injection a request-scoped bean into singleton).

第6章：Bean作用域

第6.1节：Web感知上下文中的附加作用域

在Web感知的应用上下文中，有几种作用域是专门可用的：

- request - 每个HTTP请求创建一个新的bean实例session -
- 每个HTTP会话创建一个新的bean实例application - 每个S
- ervletContext创建一个新的bean实例globalSession - 在Portlet
- 环境中每个全局会话创建一个新的bean实例（在Servlet环境中，全局会话作用域等同于会话作用域）

- websocket - 每个WebSocket会话创建一个新的bean实例

在Spring Web MVC环境中，声明和访问Web作用域的bean无需额外设置。

XML 配置

```xml
<bean id="myRequestBean" class="OneClass" scope="request"/>
<bean id="mySessionBean" class="AnotherClass" scope="session"/>
<bean id="myApplicationBean" class="YetAnotherClass" scope="application"/>
<bean id="myGlobalSessionBean" class="OneMoreClass" scope="globalSession"/>
```

Java 配置（Spring 4.3 之前）

```java
@Configuration
public class MyConfiguration {

   @Bean
     @Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
      @Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
@Scope(value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }

    @Bean
@Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public OneMoreClass myGlobalSessionBean() {
        return new OneMoreClass();
    }
}
```

Java 配置（Spring 4.3 之后）

```java
@Configuration
public class MyConfiguration {
```

Chapter 6: Bean scopes

Section 6.1: Additional scopes in web-aware contexts

There are several scopes that are available only in a web-aware application context:

- **request** - new bean instance is created per HTTP request
- **session** - new bean instance is created per HTTP session
- **application** - new bean instance is created per `ServletContext`
- **globalSession** - new bean instance is created per global session in Portlet environment (in Servlet environment global session scope is equal to session scope)
- **websocket** - new bean instance is created per WebSocket session

No additional setup is required to declare and access web-scoped beans in Spring Web MVC environment.

XML Configuration

```xml
<bean id="myRequestBean" class="OneClass" scope="request"/>
<bean id="mySessionBean" class="AnotherClass" scope="session"/>
<bean id="myApplicationBean" class="YetAnotherClass" scope="application"/>
<bean id="myGlobalSessionBean" class="OneMoreClass" scope="globalSession"/>
```

Java Configuration (prior to Spring 4.3)

```java
@Configuration
public class MyConfiguration {

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_APPLICATION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }

    @Bean
    @Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
    public OneMoreClass myGlobalSessionBean() {
        return new OneMoreClass();
    }
}
```

Java Configuration (after Spring 4.3)

```java
@Configuration
public class MyConfiguration {
```

```
@Bean
    @RequestScope
    public OneClass myRequestBean() {
        return new OneClass();
    }

@Bean
    @SessionScope
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @ApplicationScope
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }
}
```

注解驱动组件

```
@Component
@RequestScope
public class OneClass {
    ...
}

@Component
@SessionScope
public class AnotherClass {
    ...
}

@组件
@应用范围
public class YetAnotherClass {
    ...
}

@Component
@范围(范围名称 = WebApplicationContext.SCOPE_GLOBAL_SESSION, 代理模式 =
ScopedProxyMode.TARGET_CLASS)
public class OneMoreClass {
    ...
}

@Component
@范围(范围名称 = "websocket", 代理模式 = ScopedProxyMode.TARGET_CLASS)
public class AndOneMoreClass {
    ...
}
```

第6.2节：原型范围

原型范围的Bean不会在Spring容器启动时预先创建。相反，每次请求获取该Bean时，都会创建一个新的实例。该范围推荐用于有状态对象，因为其状态不会被其他组件共享。

为了定义一个原型范围的Bean，我们需要添加@Scope注解，指定我们想要的范围类型。

```
    @Bean
    @RequestScope
    public OneClass myRequestBean() {
        return new OneClass();
    }

    @Bean
    @SessionScope
    public AnotherClass mySessionBean() {
        return new AnotherClass();
    }

    @Bean
    @ApplicationScope
    public YetAnotherClass myApplicationBean() {
        return new YetAnotherClass();
    }
}
```

Annotation-Driven Components

```
@Component
@RequestScope
public class OneClass {
    ...
}

@Component
@SessionScope
public class AnotherClass {
    ...
}

@Component
@ApplicationScope
public class YetAnotherClass {
    ...
}

@Component
@Scope(scopeName = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode =
ScopedProxyMode.TARGET_CLASS)
public class OneMoreClass {
    ...
}

@Component
@Scope(scopeName = "websocket", proxyMode = ScopedProxyMode.TARGET_CLASS)
public class AndOneMoreClass {
    ...
}
```

Section 6.2: Prototype scope

A prototype-scoped bean is not pre-created on Spring container startup. Instead, a new fresh instance will be created every time a request to retrieve this bean is sent to the container. This scope is recommended for stateful objects, since its state won't be shared by other components.

In order to define a prototype-scoped bean, we need to add the @Scope annotation, specifying the type of scope we want.

给定以下 MyBean 类：

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
LOGGER.info("Initializing {} bean...", property);
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

我们定义一个bean定义，声明其作用域为原型（prototype）：

```
@Configuration
public class PrototypeConfiguration {

    @Bean
@Scope("prototype")
    public MyBean prototypeBean() {
        return new MyBean("prototype");
    }
}
```

为了了解其工作原理，我们从Spring容器中获取bean，并为其属性字段设置一个不同的值。接下来，我们将再次从容器中获取该bean并查看其值：

```
MyBean prototypeBean1 = context.getBean("prototypeBean", MyBean.class);
prototypeBean1.setProperty("changed property");

MyBean prototypeBean2 = context.getBean("prototypeBean", MyBean.class);

logger.info("Prototype bean 1 property: " + prototypeBean1.getProperty());
logger.info("Prototype bean 2 property: " + prototypeBean2.getProperty());
```

从以下结果可以看出，每次请求bean时都会创建一个新的实例：

```
正在初始化原型bean...
正在初始化原型bean...
原型bean 1 属性: changed property
原型bean 2 属性: prototype
```

一个常见的错误是认为bean是每次调用或每个线程重新创建的，事实并非如此。
实际上，每次注入（或从上下文中检索）都会创建一个实例。如果一个原型作用域的bean仅被注入到单个单例bean中，那么该原型作用域的bean将永远只有一个实例。

Spring不管理原型bean的完整生命周期：容器实例化、配置、装饰并组装原型对象，将其交给客户端，之后不再管理该原型实例。

Given the following MyBean class:

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
        LOGGER.info("Initializing {} bean...", property);
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

We define a bean definition, stating its scope as prototype:

```
@Configuration
public class PrototypeConfiguration {

    @Bean
    @Scope("prototype")
    public MyBean prototypeBean() {
        return new MyBean("prototype");
    }
}
```

In order to see how it works, we retrieve the bean from the Spring container and set a different value for its property field. Next, we will again retrieve the bean from the container and look up its value:

```
MyBean prototypeBean1 = context.getBean("prototypeBean", MyBean.class);
prototypeBean1.setProperty("changed property");

MyBean prototypeBean2 = context.getBean("prototypeBean", MyBean.class);

logger.info("Prototype bean 1 property: " + prototypeBean1.getProperty());
logger.info("Prototype bean 2 property: " + prototypeBean2.getProperty());
```

Looking at the following result, we can see how a new instance has been created on each bean request:

```
Initializing prototype bean...
Initializing prototype bean...
Prototype bean 1 property: changed property
Prototype bean 2 property: prototype
```

A common mistake is to assume that the bean is recreated per invocation or per thread, this is **NOT** the case. Instead an instance is created PER INJECTION (or retrieval from the context). If a Prototype scoped bean is only ever injected into a single singleton bean, there will only ever be one instance of that Prototype scoped bean.

Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of that prototype instance.

第6.3节：单例作用域

如果一个 Bean 被定义为单例作用域，则在 Spring 容器中只会初始化一个单一的对象实例。对该 Bean 的所有请求都会返回相同的共享实例。这是定义 Bean 时的默认作用域。

给定以下 MyBean 类：

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
LOGGER.info("Initializing {} bean...", property);
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

我们可以使用 @Bean 注解定义一个单例 Bean：

```
@Configuration
public class SingletonConfiguration {

   @Bean
    public MyBean singletonBean() {
        return new MyBean("singleton");
    }
}
```

以下示例从 Spring 上下文中两次获取相同的 Bean：

```
MyBean singletonBean1 = context.getBean("singletonBean", MyBean.class);
singletonBean1.setProperty("changed property");

MyBean singletonBean2 = context.getBean("singletonBean", MyBean.class);
```

当记录 singletonBean2 的属性时，会显示消息 "changed property"，因为我们刚刚获取的是相同的共享实例。

由于该实例在不同组件间共享，建议为无状态对象定义单例作用域。

延迟加载的单例 Bean

默认情况下，单例 Bean 会被预先实例化。因此，共享的对象实例会在 Spring 容器创建时被创建。如果我们启动应用程序，将会显示"Initializing singleton bean..."消息。

如果我们不希望 Bean 被预先实例化，可以在 Bean 定义中添加 @Lazy 注解。这将阻止 Bean 在首次请求之前被创建。

Section 6.3: Singleton scope

If a bean is defined with singleton scope, there will only be one single object instance initialized in the Spring container. All requests to this bean will return the same shared instance. This is the default scope when defining a bean.

Given the following MyBean class:

```
public class MyBean {
    private static final Logger LOGGER = LoggerFactory.getLogger(MyBean.class);
    private String property;

    public MyBean(String property) {
        this.property = property;
        LOGGER.info("Initializing {} bean...", property);
    }

    public String getProperty() {
        return this.property;
    }

    public void setProperty(String property) {
        this.property = property;
    }
}
```

We can define a singleton bean with the @Bean annotation:

```
@Configuration
public class SingletonConfiguration {

    @Bean
    public MyBean singletonBean() {
        return new MyBean("singleton");
    }
}
```

The following example retrieves the same bean twice from the Spring context:

```
MyBean singletonBean1 = context.getBean("singletonBean", MyBean.class);
singletonBean1.setProperty("changed property");

MyBean singletonBean2 = context.getBean("singletonBean", MyBean.class);
```

When logging the singletonBean2 property, the message *"changed property"* will be shown, since we just retrieved the same shared instance.

Since the instance is shared among different components, it is recommended to define singleton scope for stateless objects.

Lazy singleton beans

By default, singleton beans are pre-instantiated. Hence, the shared object instance will be created when the Spring container is created. If we start the application, the *"Initializing singleton bean..."* message will be shown.

If we don't want the bean to be pre-instantiated, we can add the @Lazy annotation to the bean definition. This will prevent the bean from being created until it is first requested.

```
@Bean
@Lazy
public MyBean lazySingletonBean() {
    return new MyBean("lazy singleton");
}
```

现在，如果我们启动 Spring 容器，不会出现"Initializing lazy singleton bean..."消息。该 Bean 直到首次被请求时才会被创建：

```
logger.info("Retrieving lazy singleton bean...");
context.getBean("lazySingletonBean");
```

如果我们同时定义了单例和延迟单例 Bean 并运行应用程序，将会产生以下消息：

```
Initializing singleton bean...
Retrieving lazy singleton bean...
正在初始化延迟单例 Bean...
```

```
@Bean
@Lazy
public MyBean lazySingletonBean() {
    return new MyBean("lazy singleton");
}
```

Now, if we start the Spring container, no *"Initializing lazy singleton bean..."* message will appear. The bean won't be created until it is requested for the first time:

```
logger.info("Retrieving lazy singleton bean...");
context.getBean("lazySingletonBean");
```

If we run the application with both singleton and lazy singleton beans defined, It will produce the following messages:

```
Initializing singleton bean...
Retrieving lazy singleton bean...
Initializing lazy singleton bean...
```

第7章：Spring中的条件Bean注册

第7.1节：仅在指定属性或值时注册Bean

Spring Bean可以配置为仅在具有特定值或满足指定属性时注册。为此，实现Condition.matches来检查属性/值：

```
public class PropertyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("propertyName") != null;
        // 可选地检查属性值
    }
}
```

在Java配置中，使用上述实现作为条件来注册Bean。注意使用了@Conditional
注解。

```
@Configuration
public class MyAppConfig {

    @Bean
@Conditional(PropertyCondition.class)
    public MyBean myBean() {
      return new MyBean();
    }
}
```

在PropertyCondition中，可以评估任意数量的条件。但建议将每个条件的实现分开，以保持它们的松耦合。例如：

```
@Configuration
public class MyAppConfig {

    @Bean
@Conditional({PropertyCondition.class, SomeOtherCondition.class})
    public MyBean myBean() {
      return new MyBean();
    }
}
```

第7.2节：条件注解

除了主要的@Conditional注解外，还有一组类似的注解用于不同的情况。

类条件

@ConditionalOnClass和@ConditionalOnMissingClass注解允许基于特定类的存在或缺失来包含配置。

例如，当OObjectDatabaseTx.class被添加到依赖中且没有OrientWebConfigurer bean时，我们创建该配置器。

Chapter 7: Conditional bean registration in Spring

Section 7.1: Register beans only when a property or value is specified

A spring bean can be configured such that it will register *only* if it has a particular value *or* a specified property is met. To do so, implement `Condition.matches` to check the property/value:

```
public class PropertyCondition implements Condition {
    @Override
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return context.getEnvironment().getProperty("propertyName") != null;
        // optionally check the property value
    }
}
```

In Java config, use the above implementation as a condition to register the bean. Note the use of @Conditional annotation.

```
@Configuration
public class MyAppConfig {

    @Bean
    @Conditional(PropertyCondition.class)
    public MyBean myBean() {
      return new MyBean();
    }
}
```

In `PropertyCondition`, any number of conditions can be evaluated. However it is advised to separate the implementation for each condition to keep them loosely coupled. For example:

```
@Configuration
public class MyAppConfig {

    @Bean
    @Conditional({PropertyCondition.class, SomeOtherCondition.class})
    public MyBean myBean() {
      return new MyBean();
    }
}
```

Section 7.2: Condition annotations

Except main `@conditional` annotation there are set of similar annotation to be used for different cases.

Class conditions

The `@ConditionalOnClass` and `@ConditionalOnMissingClass` annotations allows configuration to be included based on the presence or absence of specific classes.

E.g. when `OObjectDatabaseTx.class` is added to dependencies and there is no `OrientWebConfigurer` bean we create the configurer.

```
@Bean
@ConditionalOnWebApplication
@ConditionalOnClass(OObjectDatabaseTx.class)
@ConditionalOnMissingBean(OrientWebConfigurer.class)
public OrientWebConfigurer orientWebConfigurer() {
    return new OrientWebConfigurer();
}
```

Bean 条件

@ConditionalOnBean 和 @ConditionalOnMissingBean 注解允许根据特定 Bean 的存在或不存在来包含一个 Bean 。你可以使用 value 属性按类型指定 Bean，或使用 name 按名称指定 Bean。search 属性允许你限制在搜索 Bean 时应考虑的 ApplicationContext 层级。

参见上面的示例，当我们检查是否没有定义的 Bean 时。

属性条件

@ConditionalOnProperty 注解允许基于 Spring Environment 属性来包含配置。使用 prefix 和 name 属性指定应检查的属性。默认情况下，任何存在且不等于 false 的属性都会匹配。你也可以使用 havingValue 和 matchIfMissing 属性创建更高级的检查。

```
@ConditionalOnProperty(value='somebean.enabled', matchIfMissing = true, havingValue="yes")
@Bean
public SomeBean someBean(){
}
```

资源条件

@ConditionalOnResource 注解允许仅在特定资源存在时包含配置。

```
@ConditionalOnResource(resources = "classpath:init-db.sql")
```

Web 应用条件

@ConditionalOnWebApplication 和 @ConditionalOnNotWebApplication 注解允许根据应用是否为“Web 应用”来包含配置。

```
@Configuration
@ConditionalOnWebApplication
public class MyWebMvcAutoConfiguration {...}
```

SpEL 表达式条件

@ConditionalOnExpression 注解允许根据 SpEL 表达式的结果来包含配置。

```
@ConditionalOnExpression("${rest.security.enabled}==false")
```

```
@Bean
@ConditionalOnWebApplication
@ConditionalOnClass(OObjectDatabaseTx.class)
@ConditionalOnMissingBean(OrientWebConfigurer.class)
public OrientWebConfigurer orientWebConfigurer() {
    return new OrientWebConfigurer();
}
```

Bean conditions

The @ConditionalOnBean and @ConditionalOnMissingBean annotations allow a bean to be included based on the presence or absence of specific beans. You can use the value attribute to specify beans by type, or name to specify beans by name. The search attribute allows you to limit the ApplicationContext hierarchy that should be considered when searching for beans.

See the example above when we check whether there is no defined bean.

Property conditions

The @ConditionalOnProperty annotation allows configuration to be included based on a Spring Environment property. Use the prefix and name attributes to specify the property that should be checked. By default any property that exists and is not equal to **false** will be matched. You can also create more advanced checks using the havingValue and matchIfMissing attributes.

```
@ConditionalOnProperty(value='somebean.enabled', matchIfMissing = true, havingValue="yes")
@Bean
public SomeBean someBean(){
}
```

Resource conditions

The @ConditionalOnResource annotation allows configuration to be included only when a specific resource is present.

```
@ConditionalOnResource(resources = "classpath:init-db.sql")
```

Web application conditions

The @ConditionalOnWebApplication and @ConditionalOnNotWebApplication annotations allow configuration to be included depending on whether the application is a 'web application'.

```
@Configuration
@ConditionalOnWebApplication
public class MyWebMvcAutoConfiguration {...}
```

SpEL expression conditions

The @ConditionalOnExpression annotation allows configuration to be included based on the result of a SpEL expression.

```
@ConditionalOnExpression("${rest.security.enabled}==false")
```

第8章：Spring JSR 303 Bean 验证

Spring 支持 JSR303 Bean 验证。我们可以使用它来进行输入 Bean 验证。使用 JSR303 将验证逻辑与业务逻辑分离。

第8.1节：使用 @Valid 验证嵌套的 POJO

假设我们有一个需要验证的POJO类User。

```
public class User {

    @NotEmpty
@Size(min=5)
   @Email
    private String email;
}
```

以及一个用于验证用户实例的控制器方法

```
public String registerUser(@Valid User user, BindingResult result);
```

让我们扩展 User，添加一个需要验证的嵌套 POJO 地址（Address）。

```
public class Address {

    @NotEmpty
@Size(min=2, max=3)
    private String countryCode;
}
```

只需在地址字段上添加 @Valid 注解即可运行嵌套 POJO 的验证。

```
public class User {

    @NotEmpty
@Size(min=5)
   @Email
    private String email;

    @Valid
    private Address address;
}
```

第8.2节：Spring JSR 303 验证 - 自定义错误消息

假设我们有一个带有验证注解的简单类

```
public class UserDTO {
    @NotEmpty
    private String name;

@Min(18)
    private int age;

//getters/setters
```

Chapter 8: Spring JSR 303 Bean Validation

Spring has JSR303 bean validation support. We can use this to do input bean validation. Separate validation logic from business logic using JSR303.

Section 8.1: @Valid usage to validate nested POJOs

Suppose we have a POJO class User we need to validate.

```
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String email;
}
```

and a controller method to validate the user instance

```
public String registerUser(@Valid User user, BindingResult result);
```

Let's extend the User with a nested POJO Address we also need to validate.

```
public class Address {

    @NotEmpty
    @Size(min=2, max=3)
    private String countryCode;
}
```

Just add `@Valid` annotation on address field to run validation of nested POJOs.

```
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String email;

    @Valid
    private Address address;
}
```

Section 8.2: Spring JSR 303 Validation - Customize error messages

Suppose we have a simple class with validation annotations

```
public class UserDTO {
    @NotEmpty
    private String name;

    @Min(18)
    private int age;

//getters/setters
}
```

用于检查UserDTO有效性的控制器。

```
@RestController
public class ValidationController {

@RequestMapping(value = "/validate", method = RequestMethod.POST)
    public ResponseEntity<String> check(@Valid @RequestBody UserDTO userDTO,
           BindingResult bindingResult) {
        return new ResponseEntity<>("ok" , HttpStatus.OK);
    }
}
```

还有一个测试。

```
@Test
public void testValid() throws Exception {
    TestRestTemplate template = new TestRestTemplate();
    String url = base + contextPath + "/validate";
    Map<String, Object> params = new HashMap<>();
    params.put("name", "");
params.put("age", "10");

MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
    headers.add("Content-Type", "application/json");

HttpEntity<Map<String, Object>> request = new HttpEntity<>(params, headers);
    String res = template.postForObject(url, request, String.class);

assertThat(res, equalTo("ok"));
}
```

姓名和年龄均无效，因此在BindingResult中我们有两个验证错误。每个错误都有一组代码。

最小值检查的代码

```
0 = "Min.userDTO.age"
1 = "Min.age"
2 = "Min.int"
3 = "Min"
```

以及非空检查的代码

```
0 = "NotEmpty.userDTO.name"
1 = "NotEmpty.name"
2 = "NotEmpty.java.lang.String"
3 = "NotEmpty"
```

让我们添加一个 custom.properties 文件来替换默认消息。

```
@SpringBootApplication
@Configuration
public class DemoApplication {

@Bean(name = "messageSource")
    public MessageSource messageSource() {
ReloadableResourceBundleMessageSource bean = new ReloadableResourceBundleMessageSource();
        bean.setBasename("classpath:custom");
```

```
}
```

A controller to check the UserDTO validity.

```
@RestController
public class ValidationController {

    @RequestMapping(value = "/validate", method = RequestMethod.POST)
    public ResponseEntity<String> check(@Valid @RequestBody UserDTO userDTO,
           BindingResult bindingResult) {
        return new ResponseEntity<>("ok" , HttpStatus.OK);
    }
}
```

And a test.

```
@Test
public void testValid() throws Exception {
    TestRestTemplate template = new TestRestTemplate();
    String url = base + contextPath + "/validate";
    Map<String, Object> params = new HashMap<>();
    params.put("name", "");
    params.put("age", "10");

    MultiValueMap<String, String> headers = new LinkedMultiValueMap<>();
    headers.add("Content-Type", "application/json");

    HttpEntity<Map<String, Object>> request = new HttpEntity<>(params, headers);
    String res = template.postForObject(url, request, String.class);

    assertThat(res, equalTo("ok"));
}
```

Both name and age are invalid so in the BindingResult we have two validation errors. Each has array of codes.

Codes for Min check

```
0 = "Min.userDTO.age"
1 = "Min.age"
2 = "Min.int"
3 = "Min"
```

And for NotEmpty check

```
0 = "NotEmpty.userDTO.name"
1 = "NotEmpty.name"
2 = "NotEmpty.java.lang.String"
3 = "NotEmpty"
```

Let's add a custom.properties file to substitute default messages.

```
@SpringBootApplication
@Configuration
public class DemoApplication {

    @Bean(name = "messageSource")
    public MessageSource messageSource() {
        ReloadableResourceBundleMessageSource bean = new ReloadableResourceBundleMessageSource();
        bean.setBasename("classpath:custom");
bean.setDefaultEncoding("UTF-8");
        return bean;
    }

@Bean(name = "validator")
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

如果我们在 custom.properties 文件中添加一行

```
NotEmpty=该字段不能为空！
```

错误信息将显示新的值。为了解决消息，验证器会从
开头开始查找合适的消息代码。

因此，当我们在 .properties 文件中为所有使用 @NotEmpty 注解的情况定义 `NotEmpty` 键时，
我们的消息将被应用。

如果我们定义一条消息

```
Min.int=这里是一些自定义消息。
```

所有对整数值应用最小值检查的注解都使用新定义的消息。

如果我们需要本地化验证错误消息，也可以应用相同的逻辑。

第8.3节：基于JSR303注解的Spring验证示例

将任何JSR 303实现添加到您的类路径中。常用的是Hibernate的Hibernate Validator。

```
<dependency >
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

假设系统中有一个用于创建用户的REST API

```
@RequestMapping(value="/registeruser", method=RequestMethod.POST)
public String registerUser(User user);
```

输入的JSON示例如下

```
{"username" : "abc@abc.com", "password" : "password1", "password2":"password1"}
```

User.java

```
        bean.setDefaultEncoding("UTF-8");
        return bean;
    }

    @Bean(name = "validator")
    public LocalValidatorFactoryBean validator() {
        LocalValidatorFactoryBean bean = new LocalValidatorFactoryBean();
        bean.setValidationMessageSource(messageSource());
        return bean;
    }

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

If we add to the custom.properties file the line

```
NotEmpty=The field must not be empty!
```

The new value is shown for the error. To resolve message validator looks through the codes starting from the beginning to find proper messages.

Thus when we define `NotEmpty` key in the .properties file for all cases where the `@NotEmpty` annotation is used our message is applied.

If we define a message

```
Min.int=Some custom message here.
```

All annotations where we app min check to integer values use the newly defined message.

The same logic could be applied if we need to localize the validation error messages.

Section 8.3: JSR303 Annotation based validations in Springs examples

Add any JSR 303 implementation to your classpath. Popular one used is Hibernate validator from Hibernate.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

Lets say the there is a rest api to create user in the system

```
@RequestMapping(value="/registeruser", method=RequestMethod.POST)
public String registerUser(User user);
```

The input json sample would look like as below

```
{"username" : "abc@abc.com", "password" : "password1", "password2":"password1"}
```

User.java

```
public class User {

    private String username;
    private String password;
    private String password2;

getXXX 和 setXXX

}
```

我们可以在 User 类上定义如下的 JSR 303 校验。

```
public class User {

    @NotEmpty
@Size(min=5)
   @Email
    private String username;

@NotEmpty
    private String password;

@NotEmpty
    private String password2;

}
```

我们可能还需要一个业务验证器，比如密码和密码确认（password2）是否相同，为此
我们可以添加如下自定义验证器。编写一个自定义注解来标注数据字段。

```
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PasswordValidator.class)
public @interface GoodPassword {
    String message() default "密码不匹配。";
   Class<?>[] groups() default {};
   Class<? extends Payload>[] payload() default {};
}
```

编写一个验证器类以应用验证逻辑。

```
public class PastValidator implements ConstraintValidator<GoodPassword, User> {
   @Override
    public void initialize(GoodPassword annotation) {}

    @Override
    public boolean isValid(User user, ConstraintValidatorContext context) {
        return user.getPassword().equals(user.getPassword2());
    }
}
```

将此验证添加到用户类中

```
@GoodPassword
public class User {

    @NotEmpty
@Size(min=5)
   @Email
```

```
public class User {

    private String username;
    private String password;
    private String password2;

    getXXX and setXXX

}
```

We can define JSR 303 validations on User Class as below.

```
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String username;

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;

}
```

We may also need to have a business validator like password and password2(confirm password) are same, for this we can add a custom validator as below. Write a custom annotation for annotating the data field.

```
@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = PasswordValidator.class)
public @interface GoodPassword {
    String message() default "Passwords won't match.";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

Write a Validator class for applying Validation logic.

```
public class PastValidator implements ConstraintValidator<GoodPassword, User> {
    @Override
    public void initialize(GoodPassword annotation) {}

    @Override
    public boolean isValid(User user, ConstraintValidatorContext context) {
        return user.getPassword().equals(user.getPassword2());
    }
}
```

Adding this validation to User Class

```
@GoodPassword
public class User {

    @NotEmpty
    @Size(min=5)
    @Email
    private String username;

@NotEmpty
    private String password;

@NotEmpty
    private String password2;
}
```

@Valid 在 Spring 中触发验证。BindingResult 是由 Spring 注入的对象，包含验证后的错误列表。

```
public String registerUser(@Valid User user, BindingResult result);
```

JSR 303 注解具有 message 属性，可用于提供自定义消息。

```
@GoodPassword
public class User {

@NotEmpty(message="用户名不能为空")
    @Size(min=5, message="用户名不能少于5个字符")
    @Email(message="应为邮箱格式")
    private String username;

@NotEmpty(message="密码不能为空")
    private String password;

@NotEmpty(message="确认密码不能为空")
    private String password2;

}
```

```
    private String username;

    @NotEmpty
    private String password;

    @NotEmpty
    private String password2;
}
```

@Valid triggers validation in Spring. BindingResult is an object injected by spring which has list of errors after validation.

```
public String registerUser(@Valid User user, BindingResult result);
```

JSR 303 annotation has message attributes on them which can be used for providing custom messages.

```
@GoodPassword
public class User {

    @NotEmpty(message="Username Cant be empty")
    @Size(min=5, message="Username cant be les than 5 chars")
    @Email(message="Should be in email format")
    private String username;

    @NotEmpty(message="Password cant be empty")
    private String password;

    @NotEmpty(message="Password2 cant be empty")
    private String password2;

}
```

第9章：ApplicationContext配置

第9.1节：自动装配

自动装配是使用sterotype注解来指定哪些类将成为 ApplicationContext 中的 Bean ApplicationContext，并使用Autowired和Value注解来指定 Bean 的依赖。自动装配的独特之处在于没有外部的ApplicationContext定义，所有内容都在作为 Bean 的类内部完成。

```
@Component // 指定将此类作为 Bean
           // 包含在 ApplicationContext 中的注解
class Book {

@Autowired // 将下面定义的 Author 实例注入到该 Bean 中的注解

Author author;

    String title = "It";

Author getAuthor() { return author; }
    String getTitle() { return title; }
}

@Component // 指定将此类作为 Bean 包含在 ApplicationContext
           中的注解
class Author {
    String firstName = "Steven";
    String lastName = "King";

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }
}
```

第9.2节：引导 ApplicationContext

Java 配置

配置类只需是应用程序类路径中且对应用程序主类可见的类。

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext(MyConfig.class);

        // 准备从 appContext 中获取 bean，例如 myObject。
    }
}

@Configuration
class MyConfig {
   @Bean
MyObject myObject() {
        // ...配置 myObject...
    }
```

Chapter 9: ApplicationContext Configuration

Section 9.1: Autowiring

Autowiring is done using a *sterotype* annotation to specify what classes are going to be beans in the ApplicationContext, and using the Autowired and Value annotations to specify bean dependencies. The unique part of autowiring is that there is no external ApplicationContext definition, as it is all done within the classes that are the beans themselves.

```
@Component // The annotation that specifies to include this as a bean
           // in the ApplicationContext
class Book {

    @Autowired // The annotation that wires the below defined Author
               // instance into this bean
    Author author;

    String title = "It";

    Author getAuthor() { return author; }
    String getTitle() { return title; }
}

@Component // The annotation that specifies to include
           // this as a bean in the ApplicationContext
class Author {
    String firstName = "Steven";
    String lastName = "King";

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }
}
```

Section 9.2: Bootstrapping the ApplicationContext

Java Config

The configuration class needs only to be a class that is on the classpath of your application and visible to your applications main class.

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext(MyConfig.class);

        // ready to retrieve beans from appContext, such as myObject.
    }
}

@Configuration
class MyConfig {
    @Bean
    MyObject myObject() {
        // ...configure myObject...
    }
    // ...定义更多的 bean...
}
```

Xml 配置

配置的 xml 文件只需放在应用程序的类路径下即可。

```
class MyApp {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext appContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        // 准备从 appContext 中获取 bean，例如 myObject。
    }
}

<?xml 版本="1.0" 编码="UTF-8"?>
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/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.xsd">
    <bean id="myObject" class="com.example.MyObject">
        <!-- ...配置 myObject... -->
    </bean>

    <!-- ...定义 更多的bean... -->
</beans>
```

自动装配

自动装配需要知道扫描哪些基础包以查找带注解的bean（@Component）。这通过#scan(String...) 方法指定。

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext();
        appContext.scan("com.example");
        appContext.refresh();

        // 准备从 appContext 中获取 bean，例如 myObject。
    }
}

// 假设此类位于 com.example 包中。
@Component
class MyObject {
    // ...myObject 定义...
}
```

第9.3节：Java配置

Java配置通常通过对类应用@Configuration注解来完成，以表明该类包含bean定义。bean定义是通过对返回对象的方法应用@Bean注解来指定的。

```
@Configuration // 该注解告诉ApplicationContext此类
```

```
    // ...define more beans...
}
```

Xml Config

The configuration xml file needs only be on the classpath of your application.

```
class MyApp {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext appContext =
            new ClassPathXmlApplicationContext("applicationContext.xml");

        // ready to retrieve beans from appContext, such as myObject.
    }
}

<?xml version="1.0" encoding="UTF-8"?>
<!-- applicationContext.xml -->
<beans xmlns="http://www.springframework.org/schema/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.xsd">
    <bean id="myObject" class="com.example.MyObject">
        <!-- ...configure myObject... -->
    </bean>

    <!-- ...define more beans... -->
</beans>
```

Autowiring

Autowiring needs to know which base packages to scan for annotated beans (`@Component`). This is specified via the `#scan(String...)` method.

```
class MyApp {
    public static void main(String[] args) throws Exception {
        AnnotationConfigApplicationContext appContext =
            new AnnotationConfigApplicationContext();
        appContext.scan("com.example");
        appContext.refresh();

        // ready to retrieve beans from appContext, such as myObject.
    }
}

// assume this class is in the com.example package.
@Component
class MyObject {
    // ...myObject definition...
}
```

Section 9.3: Java Configuration

Java configuration is typically done by applying the `@Configuration` annotation to a class to suggest that a class contains bean definitions. The bean definitions are specified by applying the `@Bean` annotation to a method that returns an object.

```
@Configuration // This annotation tells the ApplicationContext that this class
```

```
                // 包含bean定义。
类 AppConfig {
    /**
* 使用默认构造函数创建的Author
     * 未设置任何属性
     */
@Bean // 该注解标记定义bean的方法
   Author author1() {
        return new Author();
    }

    /**
* 作者通过初始化名称字段的构造函数创建

     */
    @Bean
Author author2() {
        return new Author("Steven", "King");
    }

    /**
* 使用默认构造函数创建的作者，     * 然后使用属性设置器指定姓名字段


    @Bean
Author author3() {
        Author author = new Author();
        author.setFirstName("George");
        author.setLastName("Martin");
        return author;
    }

    /**
* 通过构造函数参数引用上面创建的author2创建的书籍。     * 该依赖通过以
普通Java方式调用方法来满足。

     */
@Bean
    Book book1() {
        return new Book(author2(), "It");
    }

    /**
* 通过属性设置器引用上面创建的author3创建的书籍。     * 该依赖通过以普
通Java方式调用方法来满足。

     */
    @Bean
    Book book2() {
        Book book = new Book();
       book.setAuthor(author3());
       book.setTitle("权力的游戏");
        return book;
    }
}
```

```
// 上面正在初始化和连接的类...
class Book { // 假设包名为 org.springframework.example
   Author author;
    String title;
```

```
                // contains bean definitions.
class AppConfig {
    /**
     * An Author created with the default constructor
     * setting no properties
     */
    @Bean // This annotation marks a method that defines a bean
    Author author1() {
        return new Author();
    }

    /**
     * An Author created with the constructor that initializes the
     * name fields
     */
    @Bean
    Author author2() {
        return new Author("Steven", "King");
    }

    /**
     * An Author created with the default constructor, but
     * then uses the property setters to specify name fields
     */
    @Bean
    Author author3() {
        Author author = new Author();
        author.setFirstName("George");
        author.setLastName("Martin");
        return author;
    }

    /**
     * A Book created referring to author2 (created above) via
     * a constructor argument.  The dependency is fulfilled by
     * invoking the method as plain Java.
     */
    @Bean
    Book book1() {
        return new Book(author2(), "It");
    }

    /**
     * A Book created referring to author3 (created above) via
     * a property setter.  The dependency is fulfilled by
     * invoking the method as plain Java.
     */
    @Bean
    Book book2() {
        Book book = new Book();
        book.setAuthor(author3());
        book.setTitle("A Game of Thrones");
        return book;
    }
}
```

```
// The classes that are being initialized and wired above...
class Book { // assume package org.springframework.example
    Author author;
    String title;
    Book() {} // 默认构造函数
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}

class Author { // 假设包 org.springframework.example
    String firstName;
    String lastName;

Author() {} // 默认构造函数
   Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

第9.4节：Xml配置

Xml配置通常通过在xml文件中定义bean来完成，使用Spring特定的beans模式。在根元素beans下，典型的bean定义会使用bean子元素。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/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.xsd">
    <!-- 使用默认构造函数创建的Author
       未设置任何属性 -->
    <bean id="author1" class="org.springframework.example.Author" />

    <!-- 使用初始化名称字段的构造函数创建的作者 -->

    <bean id="author2" class="org.springframework.example.Author">
        <constructor-arg index="0" value="Steven" />
        <constructor-arg index="1" value="King" />
    </bean>
```

```
    Book() {} // default constructor
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

    Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}

class Author { // assume package org.springframework.example
    String firstName;
    String lastName;

    Author() {} // default constructor
    Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Section 9.4: Xml Configuration

Xml configuration is typically done by defining beans within an xml file, using Spring's specific beans schema. Under the root beans element, typical bean definition would be done using the bean subelement.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/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.xsd">
    <!-- An Author created with the default constructor
       setting no properties -->
    <bean id="author1" class="org.springframework.example.Author" />

    <!-- An Author created with the constructor that initializes the
       name fields -->
    <bean id="author2" class="org.springframework.example.Author">
        <constructor-arg index="0" value="Steven" />
        <constructor-arg index="1" value="King" />
    </bean>
    <!-- 使用默认构造函数创建的作者，
        然后使用属性设置器指定名称字段 -->
    <bean id="author3" class="org.springframework.example.Author">
        <property name="firstName" value="George" />
        <property name="lastName" value="Martin" />
    </bean>

    <!-- 通过构造函数参数引用上面创建的author2创建的书籍 -->

    <bean id="book1" class="org.springframework.example.Book">
        <constructor-arg index="0" ref="author2" />
        <constructor-arg index="1" value="It" />
    </bean>

    <!-- 通过属性设置器引用上面创建的author3创建的书籍 -->

    <bean id="book1" class="org.springframework.example.Book">
        <property name="author" ref="author3" />
        <property name="title" value="权力的游戏" />
    </bean>
</beans>
```

```
// 上面正在初始化和连接的类...
class Book { // 假设包名为 org.springframework.example
    Author author;
    String title;

    Book() {} // 默认构造函数
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}

class Author { // 假设包 org.springframework.example
    String firstName;
    String lastName;

Author() {} // 默认构造函数
    Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }
```

```
    <!-- An Author created with the default constructor, but
        then uses the property setters to specify name fields -->
    <bean id="author3" class="org.springframework.example.Author">
        <property name="firstName" value="George" />
        <property name="lastName" value="Martin" />
    </bean>

    <!-- A Book created referring to author2 (created above) via
        a constructor argument -->
    <bean id="book1" class="org.springframework.example.Book">
        <constructor-arg index="0" ref="author2" />
        <constructor-arg index="1" value="It" />
    </bean>

    <!-- A Book created referring to author3 (created above) via
        a property setter -->
    <bean id="book1" class="org.springframework.example.Book">
        <property name="author" ref="author3" />
        <property name="title" value="A Game of Thrones" />
    </bean>
</beans>
```

```
// The classes that are being initialized and wired above...
class Book { // assume package org.springframework.example
    Author author;
    String title;

    Book() {} // default constructor
    Book(Author author, String title) {
        this.author = author;
        this.title= title;
    }

    Author getAuthor() { return author; }
    String getTitle() { return title; }

    void setAuthor(Author author) {
        this.author = author;
    }

    void setTitle(String title) {
        this.title= title;
    }
}

class Author { // assume package org.springframework.example
    String firstName;
    String lastName;

    Author() {} // default constructor
    Author(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    String getFirstName() { return firstName; }
    String getLastName() { return lastName; }

    void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

```
    void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

第10章：RestTemplate

第10.1节：下载大文件

RestTemplate的getForObject和getForEntity方法会将整个响应加载到内存中。这不适合下载大文件，因为可能导致内存溢出异常。此示例展示了如何流式处理GET请求的响应。

```
RestTemplate restTemplate // = ...;

// 可选的Accept头
RequestCallback requestCallback = request -> request.getHeaders()
        .setAccept(Arrays.asList(MediaType.APPLICATION_OCTET_STREAM, MediaType.ALL));

// 流式处理响应，而不是全部加载到内存中
ResponseExtractor<Void> responseExtractor = response -> {
    // 这里我将响应写入文件，但你可以按需处理
Path path = Paths.get("some/path");
    Files.copy(response.getBody(), path);
    return null;
};
restTemplate.execute(URI.create("www.something.com"), HttpMethod.GET, requestCallback,
responseExtractor);
```

请注意，您不能简单地从提取器返回InputStream，因为当execute方法返回时，底层连接和流已经关闭。

第10.2节：在Spring RestTemplate请求中设置头信息

RestTemplate的exchange方法允许您指定一个HttpEntity，该实体将在执行方法时写入请求。您可以向该实体添加头信息（如用户代理、引用来源等）：

```
public void testHeader(final RestTemplate restTemplate){
        //设置您需要发送的头信息
        final HttpHeaders headers = new HttpHeaders();
        headers.set("User-Agent", "eltabo");

        //创建一个新的HttpEntity
        final HttpEntity<String> entity = new HttpEntity<String>(headers);

        //执行方法，将您的 HttpEntity 写入请求
ResponseEntity<Map> response = restTemplate.exchange("https://httpbin.org/user-agent",
HttpMethod.GET, entity, Map.class);
        System.out.println(response.getBody());
    }
```

如果需要向多个请求添加相同的头信息，也可以为你的RestTemplate添加一个拦截器：

```
public void testHeader2(final RestTemplate restTemplate){
    //向RestTemplate添加一个ClientHttpRequestInterceptor
    restTemplate.getInterceptors().add(new ClientHttpRequestInterceptor(){
        @Override
        public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution) throws IOException {
            request.getHeaders().set("User-Agent", "eltabo");//为每个请求设置头信息
              return execution.execute(request, body);
        }
```

Chapter 10: RestTemplate

Section 10.1: Downloading a Large File

The getForObject and getForEntity methods of RestTemplate load the entire response in memory. This is not suitable for downloading large files since it can cause out of memory exceptions. This example shows how to stream the response of a GET request.

```
RestTemplate restTemplate // = ...;

// Optional Accept header
RequestCallback requestCallback = request -> request.getHeaders()
        .setAccept(Arrays.asList(MediaType.APPLICATION_OCTET_STREAM, MediaType.ALL));

// Streams the response instead of loading it all in memory
ResponseExtractor<Void> responseExtractor = response -> {
    // Here I write the response to a file but do what you like
    Path path = Paths.get("some/path");
    Files.copy(response.getBody(), path);
    return null;
};
restTemplate.execute(URI.create("www.something.com"), HttpMethod.GET, requestCallback,
responseExtractor);
```

Note that you cannot simply return the InputStream from the extractor, because by the time the execute method returns, the underlying connection and stream are already closed.

Section 10.2: Setting headers on Spring RestTemplate request

The exchange methods of RestTemplate allows you specify a HttpEntity that will be written to the request when execute the method. You can add headers (such user agent, referrer...) to this entity:

```
public void testHeader(final RestTemplate restTemplate){
        //Set the headers you need send
        final HttpHeaders headers = new HttpHeaders();
        headers.set("User-Agent", "eltabo");

        //Create a new HttpEntity
        final HttpEntity<String> entity = new HttpEntity<String>(headers);

        //Execute the method writing your HttpEntity to the request
        ResponseEntity<Map> response = restTemplate.exchange("https://httpbin.org/user-agent",
HttpMethod.GET, entity, Map.class);
        System.out.println(response.getBody());
    }
```

Also you can add an interceptor to your RestTemplate if you need to add the same headers to multiple requests:

```
public void testHeader2(final RestTemplate restTemplate){
    //Add a ClientHttpRequestInterceptor to the RestTemplate
    restTemplate.getInterceptors().add(new ClientHttpRequestInterceptor(){
        @Override
        public ClientHttpResponse intercept(HttpRequest request, byte[] body,
ClientHttpRequestExecution execution) throws IOException {
            request.getHeaders().set("User-Agent", "eltabo");//Set the header for each request
            return execution.execute(request, body);
        }
    });

ResponseEntity<Map> response = restTemplate.getForEntity("https://httpbin.org/user-agent",
Map.class);
    System.out.println(response.getBody());

ResponseEntity<Map> response2 = restTemplate.getForEntity("https://httpbin.org/headers",
Map.class);
    System.out.println(response2.getBody());
}
```

第10.3节：Spring RestTemplate的泛型结果

为了让RestTemplate理解返回内容的泛型，我们需要定义结果类型引用。

org.springframework.core.ParameterizedTypeReference自3.2版本起引入

```
Wrapper<Model> response = restClient.exchange(url,
                          HttpMethod.GET,
                          null,
                          new ParameterizedTypeReference<Wrapper<Model>>() {}).getBody();
```

例如，从控制器获取List<User>可能会很有用。

第10.4节：在RestTemplate和HttpClient中使用预先基本认证

预先基本认证是指在服务器返回401响应请求认证之前，提前发送HTTP基本认证凭据（用户名和密码）的做法。当调用已知需要基本认证的REST API时，这可以节省一次请求往返。

如Spring文档所述，可以使用Apache HttpClient作为底层实现，通过HttpComponentsClientHttpRequestFactory创建HTTP请求。HttpClient可以配置为执行预先基本认证。

以下类扩展了HttpComponentsClientHttpRequestFactory以提供预先的基本身份验证。

```
/**
* {@link HttpComponentsClientHttpRequestFactory} 使用预先的基本认证 * 以避免
不必要的首次 401 响应请求凭证。

 * <p>
* 仅预先向指定主机发送给定的凭证， * 并可选择性地向其子域发送。匹配子域对于使用多
个子域且不总是事先知道的 API 非常有用。

 * <p>
* {@link HttpClient} 的其他配置不受影响（例如默认凭证提供者）。

 */
public class PreAuthHttpComponentsClientHttpRequestFactory extends
HttpComponentsClientHttpRequestFactory {

    private String hostName;
    private boolean matchSubDomains;
    private Credentials credentials;

    /**
```

```
    });

    ResponseEntity<Map> response = restTemplate.getForEntity("https://httpbin.org/user-agent",
Map.class);
    System.out.println(response.getBody());

    ResponseEntity<Map> response2 = restTemplate.getForEntity("https://httpbin.org/headers",
Map.class);
    System.out.println(response2.getBody());
}
```

Section 10.3: Generics results from Spring RestTemplate

To let `RestTemplate` understand generic of returned content we need to define result type reference.

`org.springframework.core.ParameterizedTypeReference` has been introduced since 3.2

```
Wrapper<Model> response = restClient.exchange(url,
                          HttpMethod.GET,
                          null,
                          new ParameterizedTypeReference<Wrapper<Model>>() {}).getBody();
```

Could be useful to get e.g. `List<User>` from a controller.

Section 10.4: Using Preemptive Basic Authentication with RestTemplate and HttpClient

Preemptive basic authentication is the practice of sending http basic authentication credentials (username and password) *before* a server replies with a `401` response asking for them. This can save a request round trip when consuming REST apis which are known to require basic authentication.

As described in the Spring documentation, Apache HttpClient may be used as the underlying implementation to create HTTP requests by using the `HttpComponentsClientHttpRequestFactory`. HttpClient can be configured to do preemptive basic authentication.

The following class extends `HttpComponentsClientHttpRequestFactory` to provide preemptive basic authentication.

```
/**
 * {@link HttpComponentsClientHttpRequestFactory} with preemptive basic
 * authentication to avoid the unnecessary first 401 response asking for
 * credentials.
 * <p>
 * Only preemptively sends the given credentials to the given host and
 * optionally to its subdomains. Matching subdomains can be useful for APIs
 * using multiple subdomains which are not always known in advance.
 * <p>
 * Other configurations of the {@link HttpClient} are not modified (e.g. the
 * default credentials provider).
 */
public class PreAuthHttpComponentsClientHttpRequestFactory extends
HttpComponentsClientHttpRequestFactory {

    private String hostName;
    private boolean matchSubDomains;
    private Credentials credentials;

    /**
```

```
     * @param httpClient
     *            client
     * @param hostName
*           主机名
* @param matchSubDomains
    *           是否匹配主机的子域名
     * @param userName
*           基本认证用户名
     * @param password
*           基本认证密码
     */
        public PreAuthHttpComponentsClientHttpRequestFactory(HttpClient httpClient, String hostName,
            boolean matchSubDomains, String userName, String password) {
        super(httpClient);
        this.hostName = hostName;
        this.matchSubDomains = matchSubDomains;
        credentials = new UsernamePasswordCredentials(userName, password);
    }

    @Override
    protected HttpContext createHttpContext(HttpMethod httpMethod, URI uri) {
        // 将 AuthCache 添加到执行上下文中
HttpClientContext context = HttpClientContext.create();
        context.setCredentialsProvider(new PreAuthCredentialsProvider());
        context.setAuthCache(new PreAuthAuthCache());
        return context;
    }

    /**
     * @param host
*           主机名
* @return 是否应为给定的
   *          主机使用配置的凭据
     */
    protected boolean hostNameMatches(String host) {
        return host.equals(hostName) || (matchSubDomains && host.endsWith("." + hostName));
    }

    private class PreAuthCredentialsProvider extends BasicCredentialsProvider {
       @Override
        public Credentials getCredentials(AuthScope authscope) {
            if (hostNameMatches(authscope.getHost())) {
                // 模拟凭据提供者中的基本认证凭据条目。

                return credentials;
            }
            return super.getCredentials(authscope);
        }
    }

    private class PreAuthAuthCache extends BasicAuthCache {
       @Override
        public AuthScheme get(HttpHost host) {
            if (hostNameMatches(host.getHostName())) {
                // 模拟该主机的缓存条目。这指示
                // HttpClient 对该主机使用基本认证。
                return new BasicScheme();
            }
            return super.get(host);
        }
    }
}
```

```
     * @param httpClient
     *            client
     * @param hostName
     *            host name
     * @param matchSubDomains
     *            whether to match the host's subdomains
     * @param userName
     *            basic authentication user name
     * @param password
     *            basic authentication password
     */
    public PreAuthHttpComponentsClientHttpRequestFactory(HttpClient httpClient, String hostName,
            boolean matchSubDomains, String userName, String password) {
        super(httpClient);
        this.hostName = hostName;
        this.matchSubDomains = matchSubDomains;
        credentials = new UsernamePasswordCredentials(userName, password);
    }

    @Override
    protected HttpContext createHttpContext(HttpMethod httpMethod, URI uri) {
        // Add AuthCache to the execution context
        HttpClientContext context = HttpClientContext.create();
        context.setCredentialsProvider(new PreAuthCredentialsProvider());
        context.setAuthCache(new PreAuthAuthCache());
        return context;
    }

    /**
     * @param host
     *            host name
     * @return whether the configured credentials should be used for the given
     *         host
     */
    protected boolean hostNameMatches(String host) {
        return host.equals(hostName) || (matchSubDomains && host.endsWith("." + hostName));
    }

    private class PreAuthCredentialsProvider extends BasicCredentialsProvider {
        @Override
        public Credentials getCredentials(AuthScope authscope) {
            if (hostNameMatches(authscope.getHost())) {
                // Simulate a basic authenticationcredentials entry in the
                // credentials provider.
                return credentials;
            }
            return super.getCredentials(authscope);
        }
    }

    private class PreAuthAuthCache extends BasicAuthCache {
        @Override
        public AuthScheme get(HttpHost host) {
            if (hostNameMatches(host.getHostName())) {
                // Simulate a cache entry for this host. This instructs
                // HttpClient to use basic authentication for this host.
                return new BasicScheme();
            }
            return super.get(host);
        }
    }
}
```

这可以按如下方式使用：

```
HttpClientBuilder builder = HttpClientBuilder.create();
ClientHttpRequestFactory requestFactory =
    new PreAuthHttpComponentsClientHttpRequestFactory(builder.build(),
        "api.some-host.com", true, "api", "my-key");
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

第10.5节：使用Basic认证与HttpComponent的HttpClient

使用HttpClient作为RestTemplate的底层实现来创建HTTP请求，可以在与API交互时自动处理基本认证请求（HTTP 401响应）。本示例展示了如何配置RestTemplate以实现此功能。

```
// 凭证存储在这里
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(
        // AuthScope 可以进行更广泛的配置，以限制凭据将用于哪些主机/端口/协
        议等。
        new AuthScope("somehost", AuthScope.ANY_PORT),
        new UsernamePasswordCredentials("username", "password"));

// 使用凭证提供者
HttpClientBuilder builder = HttpClientBuilder.create();
builder.setDefaultCredentialsProvider(credsProvider);

// 配置 RestTemplate 使用 HttpComponent 的 HttpClient
ClientHttpRequestFactory requestFactory =
        new HttpComponentsClientHttpRequestFactory(builder.build());
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

This can be used as follows:

```
HttpClientBuilder builder = HttpClientBuilder.create();
ClientHttpRequestFactory requestFactory =
    new PreAuthHttpComponentsClientHttpRequestFactory(builder.build(),
        "api.some-host.com", true, "api", "my-key");
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

Section 10.5: Using Basic Authentication with HttpComponent's HttpClient

Using `HttpClient` as `RestTemplate`'s underlying implementation to create HTTP requests allows for automatic handling of basic authentication requests (an http 401 response) when interacting with APIs. This example shows how to configure a `RestTemplate` to achieve this.

```
// The credentials are stored here
CredentialsProvider credsProvider = new BasicCredentialsProvider();
credsProvider.setCredentials(
        // AuthScope can be configured more extensively to restrict
        // for which host/port/scheme/etc the credentials will be used.
        new AuthScope("somehost", AuthScope.ANY_PORT),
        new UsernamePasswordCredentials("username", "password"));

// Use the credentials provider
HttpClientBuilder builder = HttpClientBuilder.create();
builder.setDefaultCredentialsProvider(credsProvider);

// Configure the RestTemplate to use HttpComponent's HttpClient
ClientHttpRequestFactory requestFactory =
        new HttpComponentsClientHttpRequestFactory(builder.build());
RestTemplate restTemplate = new RestTemplate(requestFactory);
```

第11章：任务执行与调度

第11.1节：启用调度

Spring 提供了有用的任务调度支持。要启用它，只需在任意 `@Configuration` 类上
添加 `@EnableScheduling` 注解：

```
@Configuration
 @EnableScheduling
 public class MyConfig {

     // 这里是你的配置
 }
```

第11.2节：Cron表达式

Cron表达式由六个连续字段组成——

```
秒，分钟，小时，月份中的天，月份，星期中的天(s)
```

声明方式如下

```
@Scheduled(cron = "* * * * * *")
```

我们也可以设置时区为——

```
@Scheduled(cron="* * * * * *", zone="Europe/Istanbul")
```

注意事项：-

```
语法          含义                示例                说明
------------------------------------------------------------------------------------
*               匹配任意            "* * * * * *"           总是执行
*/x             每隔 x              "*/5 * * * * *"         每隔五秒执行一次
?               无指定      "0 0 0 25 12 ?"         每年圣诞节执行一次
```

示例：-

```
语法                       含义
------------------------------------------------------------------------------------
"0 0 * * * *"               每天每小时的整点执行。
"*/10 * * * * *"            每十秒执行一次。
"0 0 8-10 * * *"            每天的8点、9点和10点执行。
"0 0/30 8-10 * * *"         每天8:00、8:30、9:00、9:30和10点执行。
"0 0 9-17 * * MON-FRI"        每周一至周五9点到17点整点执行
"0 0 0 25 12 ?"             每年圣诞节午夜执行
```

用@Scheduled()声明的方法会在每个匹配的情况下被显式调用。

如果我们希望在满足cron表达式时执行某段代码，则必须在注解中指定：

```
@Component
public class MyScheduler{
```

Chapter 11: Task Execution and Scheduling

Section 11.1: Enable Scheduling

Spring provides a useful task scheduling support. To enable it, just annotate any of your `@Configuration` classes with `@EnableScheduling`:

```
@Configuration
@EnableScheduling
public class MyConfig {

    // Here it goes your configuration
}
```

Section 11.2: Cron expression

A Cron expression consists of six sequential fields -

```
second, minute, hour, day of month, month, day(s) of week
```

and is declared as follows

```
@Scheduled(cron = "* * * * * *")
```

We can also set the timezone as -

```
@Scheduled(cron="* * * * * *", zone="Europe/Istanbul")
```

Notes: -

```
syntax           means              example              explanation
------------------------------------------------------------------------------------
*                match any          "* * * * * *"        do always
*/x              every x            "*/5 * * * * *"      do every five seconds
?                no specification   "0 0 0 25 12 ?"      do every Christmas Day
```

Example: -

```
syntax                      means
------------------------------------------------------------------------------------
"0 0 * * * *"               the top of every hour of every day.
"*/10 * * * * *"            every ten seconds.
"0 0 8-10 * * *"            8, 9 and 10 o'clock of every day.
"0 0/30 8-10 * * *"         8:00, 8:30, 9:00, 9:30 and 10 o'clock every day.
"0 0 9-17 * * MON-FRI"      on the hour nine-to-five weekdays
"0 0 0 25 12 ?"             every Christmas Day at midnight
```

A method declared with `@Scheduled()` is called explicitly for every matching case.

If we want some code to be executed when a cron expression is met, then we have to specify it in the annotation:

```
@Component
public class MyScheduler{
```

```
@Scheduled(cron="*/5 * * * * MON-FRI")
    public void doSomething() {
        // 这将在工作日执行
    }
}
```

如果我们想每隔5秒在控制台打印当前时间 -

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;


@Component
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

@Scheduled(cron = "*/5 * * * * *")
    public void currentTime() {
log.info("当前时间      = {}", dateFormat.format(new Date()));
    }

}
```

使用 XML 配置示例：

示例类：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;


@Component("schedulerBean")
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    public void currentTime() {
log.info("当前时间      = {}", dateFormat.format(new Date()));
    }

}
```

示例 XML(task-context.xml)：

```
  <?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
    @Scheduled(cron="*/5 * * * * MON-FRI")
    public void doSomething() {
        // this will execute on weekdays
    }
}
```

If we want to print current time in our console for every after 5 seconds -

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;


@Component
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(cron = "*/5 * * * * *")
    public void currentTime() {
        log.info("Current Time      = {}", dateFormat.format(new Date()));
    }

}
```

Example using XML configuration:

Example class:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.text.SimpleDateFormat;
import java.util.Date;


@Component("schedulerBean")
public class Scheduler {

    private static final Logger log = LoggerFactory.getLogger(Scheduler.class);
    private static final SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    public void currentTime() {
        log.info("Current Time      = {}", dateFormat.format(new Date()));
    }

}
```

Example XML(task-context.xml):

```
  <?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:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
        http://www.springframework.org/schema/task
http://www.springframework.org/schema/task/spring-task-4.1.xsd">


    <task:scheduled-tasks scheduler="scheduledTasks">
        <task:scheduled ref="schedulerBean" method="currentTime" cron="*/5 * * * * MON-FRI" />
    </task:scheduled-tasks>

    <task:scheduler id="scheduledTasks" />

</beans>
```

第11.3节：固定延迟

如果我们希望某段代码在前一次执行完成后周期性执行，应使用固定延迟（以毫秒为单位）：

```
@Component
public class MyScheduler{

@Scheduled(fixedDelay=5000)
    public void doSomething() {
        // 这段代码将在前一次执行完成后周期性执行
    }
}
```

第11.4节：固定利率

如果我们想让某个操作周期性执行，这段代码将会按照我们指定的毫秒值触发一次：

```
@Component
public class MyScheduler{

   @Scheduled(fixedRate=5000)
    public void doSomething() {
        // 这将周期性执行
    }
}
```

```
    xmlns:task="http://www.springframework.org/schema/task"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-4.1.xsd
        http://www.springframework.org/schema/task
        http://www.springframework.org/schema/task/spring-task-4.1.xsd">


    <task:scheduled-tasks scheduler="scheduledTasks">
        <task:scheduled ref="schedulerBean" method="currentTime" cron="*/5 * * * * MON-FRI" />
    </task:scheduled-tasks>

    <task:scheduler id="scheduledTasks" />

</beans>
```

Section 11.3: Fixed delay

If we want some code to be executed periodically after the execution which was before is finished, we should use fixed delay (measured in milliseconds):

```
@Component
public class MyScheduler{

    @Scheduled(fixedDelay=5000)
    public void doSomething() {
        // this will execute periodically, after the one before finishes
    }
}
```

Section 11.4: Fixed Rate

If we want something to be executed periodically, this code will be triggered once per the value in milliseconds we specify:

```
@Component
public class MyScheduler{

    @Scheduled(fixedRate=5000)
    public void doSomething() {
        // this will execute periodically
    }
}
```

第12章：Spring延迟初始化

第12.1节：Spring中延迟初始化的示例

@Lazy注解允许我们指示IOC容器延迟初始化一个bean。默认情况下，bean会在IOC容器创建时立即实例化，@Lazy允许我们改变这个实例化过程。

spring中的lazy-init是bean标签的一个属性。lazy-init的取值为true和false。如果lazy-init为true，则该bean会在请求该bean时初始化。该bean不会在spring容器初始化时初始化。如果lazy-init为false，则bean会随着spring容器的初始化而初始化，这也是默认行为。

app-conf.xml

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

<bean id="testA" class="com.concretepage.A"/>
<bean id="testB" class="com.concretepage.B" lazy-init="true"/>
```

A.java

```
package com.concretepage;
public class A {
public A(){
    System.out.println("Bean A is initialized");
   }
}
```

B.java

```
package com.concretepage;
public class B {
public B(){
    System.out.println("Bean B is initialized");
   }
}
```

SpringTest.java

```
package com.concretepage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class SpringTest {
public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("app-conf.xml");
    System.out.println("获取 Bean B.");
context.getBean("testB");
   }
}
```

Chapter 12: Spring Lazy Initialization

Section 12.1: Example of Lazy Init in Spring

The @Lazy allow us to instruct the IOC container to delay the initialization of a bean. By default, beans are instantiated as soon as the IOC container is created, The @Lazy allow us to change this instantiation process.

lazy-init in spring is the attribute of bean tag. The values of lazy-init are true and false. If lazy-init is true, then that bean will be initialized when a request is made to bean. This bean will not be initialized when the spring container is initialized. If lazy-init is false then the bean will be initialized with the spring container initialization and this is the default behavior.

app-conf.xml

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

<bean id="testA" class="com.concretepage.A"/>
<bean id="testB" class="com.concretepage.B" lazy-init="true"/>
```

A.java

```
package com.concretepage;
public class A {
public A(){
    System.out.println("Bean A is initialized");
   }
}
```

B.java

```
package com.concretepage;
public class B {
public B(){
    System.out.println("Bean B is initialized");
   }
}
```

SpringTest.java

```
package com.concretepage;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public class SpringTest {
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("app-conf.xml");
    System.out.println("Feth bean B.");
    context.getBean("testB");
   }
}
```

输出

```
Bean A 已初始化
获取 Bean B。
Bean B 已初始化
```

第12.2节：关于组件扫描和自动装配

```
@Component
@Lazy
public class Demo {
    ....
    ....
}

@Component
public class B {

    @Autowired
                    @Lazy // 如果这里没有这个注解，Demo仍会被急切实例化以满足此请求。
    private Demo demo;

    .......
 }
```

第12.3节：配置类中的延迟初始化

```
@Configuration
// @Lazy - 使所有Bean延迟加载
public class AppConf {

   @Bean
    @Lazy
    public Demo demo() {
        return new Demo();
    }
}
```

Output

```
Bean A is initialized
Feth bean B.
Bean B is initialized
```

Section 12.2: For component scanning and auto-wiring

```
@Component
@Lazy
public class Demo {
    ....
    ....
}

@Component
public class B {

    @Autowired
    @Lazy // If this is not here, Demo will still get eagerly instantiated to satisfy this request.
    private Demo demo;

    .......
 }
```

Section 12.3: Lazy initialization in the configuration class

```
@Configuration
// @Lazy - For all Beans to load lazily
public class AppConf {

    @Bean
    @Lazy
    public Demo demo() {
        return new Demo();
    }
}
```

第13章：属性源

第13.1节：使用PropertyPlaceholderConfigurer的示例xml配置

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
     <list>
           <value>classpath:ReleaseBundle.properties</value>
     </list>
</bean>
```

第13.2节：注解

示例属性文件：nexus.properties

示例属性文件内容：

```
nexus.user=admin
nexus.pass=admin
nexus.rest.uri=http://xxx.xxx.xxx.xxx:xxxx/nexus/service/local/artifact/maven/content
```

示例上下文文件xml配置

```
<context:property-placeholder location="classpath:ReleaseBundle.properties" />
```

使用注解的示例属性Bean

```
@Component
@PropertySource(value = { "classpath:nexus.properties" })
public class NexusBean {

@Value("${" + NexusConstants.NEXUS_USER + "}")
    private String user;

@Value("${" + NexusConstants.NEXUS_PASS + "}")
    private String pass;

@Value("${" + NexusConstants.NEXUS_REST_URI + "}")
    private String restUri;
}
```

示例常量类

```
public class NexusConstants {
    public static final String NexusConstants.NEXUS_USER="";
    public static final String NexusConstants.NEXUS_PASS="";
    public static final String NexusConstants.NEXUS_REST_URI="";
}
```

Chapter 13: Property Source

Section 13.1: Sample xml configuration using PropertyPlaceholderConfigurer

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
     <list>
           <value>classpath:ReleaseBundle.properties</value>
     </list>
</bean>
```

Section 13.2: Annotation

Sample property file : nexus.properties

Sample property file content:

```
nexus.user=admin
nexus.pass=admin
nexus.rest.uri=http://xxx.xxx.xxx.xxx:xxxx/nexus/service/local/artifact/maven/content
```

Sample Context File xml configuration

```
<context:property-placeholder location="classpath:ReleaseBundle.properties" />
```

Sample Property Bean using annotations

```
@Component
@PropertySource(value = { "classpath:nexus.properties" })
public class NexusBean {

    @Value("${" + NexusConstants.NEXUS_USER + "}")
    private String user;

    @Value("${" + NexusConstants.NEXUS_PASS + "}")
    private String pass;

    @Value("${" + NexusConstants.NEXUS_REST_URI + "}")
    private String restUri;
}
```

Sample Constant class

```
public class NexusConstants {
    public static final String NexusConstants.NEXUS_USER="";
    public static final String NexusConstants.NEXUS_PASS="";
    public static final String NexusConstants.NEXUS_REST_URI="";
}
```

第14章：依赖注入（DI）和控制反转（IoC）

第14.1节：通过Java配置自动装配依赖

通过Java配置的构造函数注入也可以利用自动装配，例如：

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo(Bar bar) { return new Foo(bar); }
}
```

第14.2节：通过XML配置自动装配依赖

使用Spring框架的组件扫描功能时，可以自动装配依赖。为了使自动装配生效，必须进行以下XML配置：

```
<context:annotation-config/>
<context:component-scan base-package="[base package]"/>
```

其中，base-package是Spring应执行组件扫描的完整Java包名。

> 构造函数注入

依赖可以通过类构造函数注入，如下所示：

```
@Component
class Bar { ... }

@Component
class Foo {
  private Bar bar;

  @Autowired
  public Foo(Bar bar) { this.bar = bar; }
}
```

这里，@Autowired 是 Spring 特有的注解。Spring 也支持 JSR-299，以便应用程序能够移植到其他基于 Java 的依赖注入框架。这允许将 @Autowired 替换为 @Inject，如下所示：

```
@Component
class Foo {
  private Bar bar;

  @Inject
  public Foo(Bar bar) { this.bar = bar; }
}
```

Chapter 14: Dependency Injection (DI) and Inversion of Control (IoC)

Section 14.1: Autowiring a dependency through Java configuration

Constructor injection through Java configuration can also utilize autowiring, such as:

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo(Bar bar) { return new Foo(bar); }
}
```

Section 14.2: Autowiring a dependency through XML configuration

Dependencies can be autowired when using the component scan feature of the Spring framework. For autowiring to work, the following XML configuration must be made:

```
<context:annotation-config/>
<context:component-scan base-package="[base package]"/>
```

where, `base-package` is the fully-qualified Java package within which Spring should perform component scan.

> Constructor injection

Dependencies can be injected through the class constructor as follows:

```
@Component
class Bar { ... }

@Component
class Foo {
  private Bar bar;

  @Autowired
  public Foo(Bar bar) { this.bar = bar; }
}
```

Here, `@Autowired` is a Spring-specific annotation. Spring also supports JSR-299 to enable application portability to other Java-based dependency injection frameworks. This allows `@Autowired` to be replaced with `@Inject` as:

```
@Component
class Foo {
  private Bar bar;

  @Inject
  public Foo(Bar bar) { this.bar = bar; }
}
```

> 属性注入

依赖也可以通过 setter 方法注入，如下所示：

```
@Component
class Foo {
  private Bar bar;

  @Autowired
  public void setBar(Bar bar) { this.bar = bar; }
}
```

> 字段注入

自动装配还允许直接初始化类实例中的字段，如下所示：

```
@Component
class Foo {
  @Autowired
  private Bar bar;
}
```

对于 Spring 版本 4.1 及以上，您可以使用Optional来处理可选依赖。

```
@Component
class Foo {

    @Autowired
    private Optional<Bar> bar;
}
```

同样的方法也可以用于构造函数依赖注入（DI）。

```
@Component
class Foo {
    private Optional<Bar> bar;

    @Autowired
Foo(Optional<Bar> bar) {
        this.bar = bar;
    }
}
```

第14.3节：通过XML配置手动注入依赖

考虑以下Java类：

```
class Foo {
  private Bar bar;

  public void foo() {
    bar.baz();
  }
}
```

> Property injection

Dependencies can also be injected using setter methods as follows:

```
@Component
class Foo {
  private Bar bar;

  @Autowired
  public void setBar(Bar bar) { this.bar = bar; }
}
```

> Field injection

Autowiring also allows initializing fields within class instances directly, as follows:

```
@Component
class Foo {
  @Autowired
  private Bar bar;
}
```

For Spring versions 4.1+ you can use Optional for optional dependencies.

```
@Component
class Foo {

    @Autowired
    private Optional<Bar> bar;
}
```

The same approach can be used for constructor DI.

```
@Component
class Foo {
    private Optional<Bar> bar;

    @Autowired
    Foo(Optional<Bar> bar) {
        this.bar = bar;
    }
}
```

Section 14.3: Injecting a dependency manually through XML configuration

Consider the following Java classes:

```
class Foo {
  private Bar bar;

  public void foo() {
    bar.baz();
  }
}
```

如图所示，类Foo需要在其方法foo中调用另一个类Bar的实例方法baz才能成功运行。由于Foo无法在没有Bar实例的情况下正常工作，因此Bar被称为Foo的依赖。

> 构造函数注入

在使用Spring框架的XML配置来定义Spring管理的bean时，可以按如下方式配置一个类型为Foo的bean：

```
<bean class="Foo">
  <constructor-arg>
    <bean class="Bar" />
  </constructor-arg>
</bean>
```

或者，另一种方式（更详细）：

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <constructor-arg ref="bar" />
</bean>
```

在这两种情况下，Spring框架首先创建一个Bar的实例，并将其注入到Foo的实例中。此示例假设类Foo有一个构造函数可以接受一个Bar实例作为参数，即：

```
class Foo {
  private Bar bar;

  public Foo(Bar bar) { this.bar = bar; }
}
```

这种方式被称为构造函数注入，因为依赖（Bar实例）是通过类的构造函数注入的。

> 属性注入

将Bar依赖注入到Foo的另一种方法是：

```
<bean class="Foo">
  <property name="bar">
    <bean class="Bar" />
  </property>
</bean>
```

或者，另一种方式（更详细）：

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <property name="bar" ref="bar" />
</bean>
```

这要求Foo类具有一个接受Bar实例的setter方法，例如：

As can be seen, the class Foo needs to call the method baz on an instance of another class Bar for its method foo to work successfully. Bar is said to be a dependency for Foo since Foo cannot work correctly without a Bar instance.

> Constructor injection

When using XML configuration for Spring framework to define Spring-managed beans, a bean of type Foo can be configured as follows:

```
<bean class="Foo">
  <constructor-arg>
    <bean class="Bar" />
  </constructor-arg>
</bean>
```

or, alternatively (more verbose):

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <constructor-arg ref="bar" />
</bean>
```

In both cases, Spring framework first creates an instance of Bar and injects it into an instance of Foo. This example assumes that the class Foo has a constructor that can take a Bar instance as a parameter, that is:

```
class Foo {
  private Bar bar;

  public Foo(Bar bar) { this.bar = bar; }
}
```

This style is known as **constructor injection** because the dependency (Bar instance) is being injected into through the class constructor.

> Property injection

Another option to inject the Bar dependency into Foo is:

```
<bean class="Foo">
  <property name="bar">
    <bean class="Bar" />
  </property>
</bean>
```

or, alternatively (more verbose):

```
<bean id="bar" class="bar" />

<bean class="Foo">
  <property name="bar" ref="bar" />
</bean>
```

This requires the Foo class to have a setter method that accepts a Bar instance, such as:

```
class Foo {
  private Bar bar;

  public void setBar(Bar bar) { this.bar = bar; }
}
```

第14.4节：通过Java配置手动注入依赖

与上面使用XML配置的示例相同，可以用Java配置重写如下。

> 构造函数注入

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() { return new Foo(bar()); }
}
```

> 属性注入

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());

    return foo;
  }
}
```

```
class Foo {
  private Bar bar;

  public void setBar(Bar bar) { this.bar = bar; }
}
```

Section 14.4: Injecting a dependency manually through Java configuration

The same examples as shown above with XML configuration can be re-written with Java configuration as follows.

> Constructor injection

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() { return new Foo(bar()); }
}
```

> Property injection

```
@Configuration
class AppConfig {
  @Bean
  public Bar bar() { return new Bar(); }

  @Bean
  public Foo foo() {
    Foo foo = new Foo();
    foo.setBar(bar());

    return foo;
  }
}
```

第15章：JdbcTemplate

JdbcTemplate 类执行 SQL 查询、更新语句和存储过程调用，遍历 ResultSet 并提取返回的参数值。它还会捕获 JDBC 异常并将其转换为 org.springframework.dao 包中定义的通用且更具信息性的异常层次结构。

JdbcTemplate 类的实例在配置完成后是线程安全的，因此可以安全地将此共享引用注入到多个 DAO 中。

第 15.1 节：基本查询方法

JdbcTemplate 中的一些 queryFor* 方法适用于执行 CRUD 操作的简单 SQL 语句。

查询日期

```
String sql = "SELECT create_date FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, java.util.Date.class, customerId);
```

查询整数

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, Integer.class, customerId);
```

或者

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForInt(sql, customerId);                // 在 spring-jdbc 4 中已弃用
```

查询字符串

```
String sql = "SELECT first_Name FROM customer WHERE customer_id = ?";
String firstName = jdbcTemplate.queryForObject(sql, String.class, customerId);
```

查询列表

```
String sql = "SELECT first_Name FROM customer WHERE store_id =  ?";
List<String> firstNameList = jdbcTemplate.queryForList(sql, String.class, storeId);
```

第15.2节：查询映射列表

```
int storeId = 1;
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer WHERE store_id = ?";
List<Map<String, Object>> mapList = jdbcTemplate.queryForList(sql, storeId);

for(Map<String, Object> entryMap : mapList)
{
  for(Entry<String, Object> entry : entryMap.entrySet())
  {
       System.out.println(entry.getKey() + " / " + entry.getValue());
  }
  System.out.println("---");
```

Chapter 15: JdbcTemplate

The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.

Instances of the JdbcTemplate class are threadsafe once configured so it can be safely inject this shared reference into multiple DAOs.

Section 15.1: Basic Query methods

Some of the queryFor* methods available in JdbcTemplate are useful for simple sql statements that perform CRUD operations.

Querying for Date

```
String sql = "SELECT create_date FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, java.util.Date.class, customerId);
```

Querying for Integer

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForObject(sql, Integer.class, customerId);
```

OR

```
String sql = "SELECT store_id FROM customer WHERE customer_id = ?";
int storeId = jdbcTemplate.queryForInt(sql, customerId);                //Deprecated in spring-
jdbc 4
```

Querying for String

```
String sql = "SELECT first_Name FROM customer WHERE customer_id = ?";
String firstName = jdbcTemplate.queryForObject(sql, String.class, customerId);
```

Querying for List

```
String sql = "SELECT first_Name FROM customer WHERE store_id =  ?";
List<String> firstNameList = jdbcTemplate.queryForList(sql, String.class, storeId);
```

Section 15.2: Query for List of Maps

```
int storeId = 1;
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer WHERE store_id = ?";
List<Map<String, Object>> mapList = jdbcTemplate.queryForList(sql, storeId);

for(Map<String, Object> entryMap : mapList)
{
  for(Entry<String, Object> entry : entryMap.entrySet())
  {
       System.out.println(entry.getKey() + " / " + entry.getValue());
  }
  System.out.println("---");
}
```

第15.3节：SQLRowSet

```
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer";
SqlRowSet rowSet = jdbcTemplate.queryForRowSet(sql);

while(rowSet.next())
{
  String firstName = rowSet.getString("first_name");
  String lastName = rowSet.getString("last_name");
  System.out.println("名字: " + firstName);
  System.out.println("姓氏: " + lastName);
  System.out.println("---");
}
```

或者

```
String sql = "SELECT * FROM customer";
List<Customer> customerList = jdbcTemplate.query(sql, new RowMapper<Customer>()    {

  @Override
  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException
  {
Customer customer = new Customer();
    customer.setFirstName(rs.getString("first_Name"));
    customer.setLastName(rs.getString("first_Name"));
    customer.setEmail(rs.getString("email"));

    return customer;
  }

});
```

第15.4节：批量操作

JdbcTemplate 还提供了方便的方法来执行批量操作。

批量插入

```
final ArrayList<Student> list = // 获取要插入的学生列表..
String sql = "insert into student (id, f_name, l_name, age, address) VALUES (?, ?, ?, ?, ?)"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
ps.setString(1, s.getId());
ps.setString(2, s.getF_name());
        ps.setString(3, s.getL_name());
        ps.setInt(4, s.getAge());
ps.setString(5, s.getAddress());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
```

```
}
```

Section 15.3: SQLRowSet

```
DataSource dataSource = ... //
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
String sql = "SELECT * FROM customer";
SqlRowSet rowSet = jdbcTemplate.queryForRowSet(sql);

while(rowSet.next())
{
  String firstName = rowSet.getString("first_name");
  String lastName = rowSet.getString("last_name");
  System.out.println("Vorname: " + firstName);
  System.out.println("Nachname: " + lastName);
  System.out.println("---");
}
```

OR

```
String sql = "SELECT * FROM customer";
List<Customer> customerList = jdbcTemplate.query(sql, new RowMapper<Customer>()    {

  @Override
  public Customer mapRow(ResultSet rs, int rowNum) throws SQLException
  {
    Customer customer = new Customer();
    customer.setFirstName(rs.getString("first_Name"));
    customer.setLastName(rs.getString("first_Name"));
    customer.setEmail(rs.getString("email"));

    return customer;
  }

});
```

Section 15.4: Batch operations

JdbcTemplate also provides convenient methods to execute batch operations.

Batch Insert

```
final ArrayList<Student> list = // Get list of students to insert..
String sql = "insert into student (id, f_name, l_name, age, address) VALUES (?, ?, ?, ?, ?)"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
        ps.setString(1, s.getId());
        ps.setString(2, s.getF_name());
        ps.setString(3, s.getL_name());
        ps.setInt(4, s.getAge());
        ps.setString(5, s.getAddress());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
});
```

批量更新

```
final ArrayList<Student> list = // 获取要更新的学生列表..
String sql = "update student set f_name = ?, l_name = ?, age = ?, address = ? where id = ?"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
ps.setString(1, s.getF_name());
        ps.setString(2, s.getL_name());
        ps.setInt(3, s.getAge());
ps.setString(4, s.getAddress());
        ps.setString(5, s.getId());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
});
```

还有其他接受对象数组列表作为输入参数的 batchUpdate 方法。这些方法
内部使用 BatchPreparedStatementSetter 将数组列表中的值设置到 SQL 语句中。

第15.5节：NamedParameterJdbcTemplate 对 JdbcTemplate 的扩展

> NamedParameterJdbcTemplate 类增加了使用命名参数编写 JDBC 语句的支持，区别于仅使用经典占位符（'?'）参数编写 JDBC 语句。
> NamedParameterJdbcTemplate 类封装了一个 JdbcTemplate，并将大部分工作委托给被封装的 JdbcTemplate 来完成。

```
DataSource dataSource = ... //
NamedParameterJdbcTemplate jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

String sql = "SELECT count(*) FROM customer WHERE city_name=:cityName";
Map<String, String> params = Collections.singletonMap("cityName", cityName);
int count = jdbcTemplate.queryForObject(sql, params, Integer.class);
```

```
});
```

Batch Update

```
final ArrayList<Student> list = // Get list of students to update..
String sql = "update student set f_name = ?, l_name = ?, age = ?, address = ? where id = ?"
jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter(){
    @Override
    public void setValues(PreparedStatement ps, int i) throws SQLException {
        Student s = l.get(i);
        ps.setString(1, s.getF_name());
        ps.setString(2, s.getL_name());
        ps.setInt(3, s.getAge());
        ps.setString(4, s.getAddress());
        ps.setString(5, s.getId());
    }

    @Override
    public int getBatchSize() {
        return l.size();
    }
});
```

There are further batchUpdate methods which accept List of object array as input parameters. These methods internally use BatchPreparedStatementSetter to set the values from the list of arrays into sql statement.

Section 15.5: NamedParameterJdbcTemplate extension of JdbcTemplate

> The `NamedParameterJdbcTemplate` class adds support for programming JDBC statements using named parameters, as opposed to programming JDBC statements using only classic placeholder ('?') arguments. The `NamedParameterJdbcTemplate` class wraps a `JdbcTemplate`, and delegates to the wrapped `JdbcTemplate` to do much of its work.

```
DataSource dataSource = ... //
NamedParameterJdbcTemplate jdbcTemplate = new NamedParameterJdbcTemplate(dataSource);

String sql = "SELECT count(*) FROM customer WHERE city_name=:cityName";
Map<String, String> params = Collections.singletonMap("cityName", cityName);
int count = jdbcTemplate.queryForObject(sql, params, Integer.class);
```

第16章：SOAP Web服务的使用

第16.1节：使用基本认证调用SOAP Web服务

创建你自己的WSMessageSender：

```
import java.io.IOException;
import java.net.HttpURLConnection;

import org.springframework.ws.transport.http.HttpUrlConnectionMessageSender;

import sun.misc.BASE64Encoder;

public class CustomWSMessageSender extends HttpUrlConnectionMessageSender{

   @Override
    protected void prepareConnection(HttpURLConnection connection)
            throws IOException {

BASE64Encoder enc = new sun.misc.BASE64Encoder();
        String userpassword = "yourUser:yourPassword";
        String encodedAuthorization = enc.encode( userpassword.getBytes() );
        connection.setRequestProperty("Authorization", "Basic " + encodedAuthorization);

        super.prepareConnection(connection);
    }
}
```

在您的WS配置类中设置刚刚创建的MessageSender：

```
myWSClient.setMessageSender(new CustomWSMessageSender());
```

Chapter 16: SOAP WS Consumption

Section 16.1: Consuming a SOAP WS with Basic auth

Create your own WSMessageSender:

```
import java.io.IOException;
import java.net.HttpURLConnection;

import org.springframework.ws.transport.http.HttpUrlConnectionMessageSender;

import sun.misc.BASE64Encoder;

public class CustomWSMessageSender extends HttpUrlConnectionMessageSender{

    @Override
    protected void prepareConnection(HttpURLConnection connection)
            throws IOException {

        BASE64Encoder enc = new sun.misc.BASE64Encoder();
        String userpassword = "yourUser:yourPassword";
        String encodedAuthorization = enc.encode( userpassword.getBytes() );
        connection.setRequestProperty("Authorization", "Basic " + encodedAuthorization);

        super.prepareConnection(connection);
    }
}
```

In your WS configuration class set the MessageSender you just created:

```
myWSClient.setMessageSender(new CustomWSMessageSender());
```

第17章：Spring配置文件

第17.1节：Spring配置文件允许配置某些环境可用的部分

任何@Component或@Configuration都可以用@Profile注解标记

```
@Configuration
@Profile(\"production\")
public class ProductionConfiguration {

    // ...
}
```

XML配置中的相同内容

```
<beans profile="dev">
    <bean id="dataSource" class="<some data source class>" />
</beans>
```

活动配置文件可以在application.properties文件中配置

```
spring.profiles.active=dev,production
```

或者从命令行指定

```
--spring.profiles.active=dev,hsqldb
```

或者在SpringBoot中

```
SpringApplication.setAdditionalProfiles("dev");
```

可以使用注解@ActiveProfiles("dev")在测试中启用配置文件

Chapter 17: Spring profile

Section 17.1: Spring Profiles allows to configure parts available for certain environment

Any @Component or @Configuration could be marked with @Profile annotation

```
@Configuration
@Profile("production")
public class ProductionConfiguration {

    // ...
}
```

The same in XML config

```
<beans profile="dev">
    <bean id="dataSource" class="<some data source class>" />
</beans>
```

Active profiles could be configured in the application.properties file

```
spring.profiles.active=dev,production
```

or specified from command line

```
--spring.profiles.active=dev,hsqldb
```

or in SpringBoot

```
SpringApplication.setAdditionalProfiles("dev");
```

It is possible to enable profiles in Tests using the annotation @ActiveProfiles("dev")

第18章：理解dispatcher-servlet.xml

在Spring Web MVC中，DispatcherServlet类作为前端控制器。它负责管理Spring MVC应用程序的流程。

DispatcherServlet 也像普通的 servlet 一样需要在 web.xml 中配置

第18.1节：dispatcher-servlet.xml

这是一个重要的配置文件，我们需要在其中指定 ViewResolver 和视图组件。

context:component-scan 元素定义了 DispatcherServlet 将搜索控制器类的基础包
。

这里，InternalResourceViewResolver 类被用作 ViewResolver。

前缀 + 控制器返回的字符串 + 后缀 页面将被调用作为视图组件。

该 xml 文件应位于 WEB-INF 目录下。

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   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/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd>

    <context:component-scan base-package="com.srinu.controller.Employee" />

    <bean
       class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/views/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>

    </bean>
</beans>
```

第18.2节：web.xml中的调度器Servlet配置

在此XML文件中，我们指定了作为Spring Web MVC前端控制器的servlet类DispatcherServlet。所有针对HTML文件的传入请求都将被转发到DispatcherServlet。

```
<?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">
 <servlet>  </servlet>
    <servlet-name>spring</servlet-name>
```

Chapter 18: Understanding the dispatcher-servlet.xml

In Spring Web MVC, DispatcherServlet class works as the front controller. It is responsible for managing the flow of the spring MVC application.

DispatcherServlet is also like normal servlet need to be configured in web.xml

Section 18.1: dispatcher-servlet.xml

This is the important configuration file where we need to specify the ViewResolver and View components.

The context:component-scan element defines the base-package where DispatcherServlet will search the controller class.

Here, the InternalResourceViewResolver class is used for the ViewResolver.

The prefix+string returned by controller+suffix page will be invoked for the view component.

This xml file should be located inside the WEB-INF directory.

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:context="http://www.springframework.org/schema/context"
   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/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="com.srinu.controller.Employee" />

    <bean
       class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix">
            <value>/WEB-INF/views/</value>
        </property>
        <property name="suffix">
            <value>.jsp</value>
        </property>
    </bean>
</beans>
```

Section 18.2: dispatcher servlet configuration in web.xml

In this XML file, we are specifying the servlet class DispatcherServlet that acts as the front controller in Spring Web MVC. All the incoming request for the HTML file will be forwarded to the DispatcherServlet.

```
<?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">
 <servlet>
    <servlet-name>spring</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
</web-app>
```

```
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>spring</servlet-name>
    <url-pattern>*.html</url-pattern>
</servlet-mapping>
</web-app>
```

鸣谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可以发送至web@petercv.com以发布或更新新内容

AdamIJK	第12章
阿洛	第4章
伯尼	第10章
邦德	第7章
科林D	第5章
康斯坦丁	第5章和第6章
大卫·R	第12章
迪米特里斯利	第一章
埃尔塔博	第10章
高塔姆·乔斯	第十三章
吉列11	第十一章和第十六章
哈沙尔·帕蒂尔	第5章
希特什·库马尔	第一章
ipsi	第一章
JamesENL	第5章
Johir	第11章
manish	第14章
莫谢·阿拉德	第12章
mszymborski	第5章
nicholas.hauschild	第9章
黑豹	第1章、第6章和第13章
普拉尼思·拉梅什	第8章
拉贾尼坎塔·普拉丹	第1章和第2章
谢尔吉·比希尔	第14章
塞图	第15章
smichel	第15章
斯里尼瓦斯·加迪利	第18章
斯坦尼斯拉夫L	第5、7、8、10、15和17章
斯特凡·伊塞勒	第5章
泰勒	第6章
蒂姆·汤	第5章和第6章
沃尔什	第3章
xpadro	第6章
Xtreme Biker	第11章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

AdamIJK	Chapter 12
Arlo	Chapter 4
bernie	Chapter 10
Bond	Chapter 7
CollinD	Chapter 5
Constantine	Chapters 5 and 6
DavidR	Chapter 12
dimitrisli	Chapter 1
eltabo	Chapter 10
Gautam Jose	Chapter 13
guille11	Chapters 11 and 16
Harshal Patil	Chapter 5
Hitesh Kumar	Chapter 1
ipsi	Chapter 1
JamesENL	Chapter 5
Johir	Chapter 11
manish	Chapter 14
Moshe Arad	Chapter 12
mszymborski	Chapter 5
nicholas.hauschild	Chapter 9
Panther	Chapters 1, 6 and 13
Praneeth Ramesh	Chapter 8
Rajanikanta Pradhan	Chapters 1 and 2
Sergii Bishyr	Chapter 14
Setu	Chapter 15
smichel	Chapter 15
Srinivas Gadilli	Chapter 18
StanislavL	Chapters 5, 7, 8, 10, 15 and 17
Stefan Isele	Chapter 5
Taylor	Chapter 6
Tim Tong	Chapters 5 and 6
walsh	Chapter 3
xpadro	Chapter 6
Xtreme Biker	Chapter 11