vmware | SPRING & CLOUD FOUNDRY
开发者大会

中国北京 - 2012年12月7日-8日

Spring 3.1、3.2 和 3.3更新

Josh Long

VMWare 旗下 SpringSource 部门的 ***Spring*** *开发人员技术布道师*

http://www.joshlong.com || @starbuxman || josh.long@springsource.com

关于 springsource

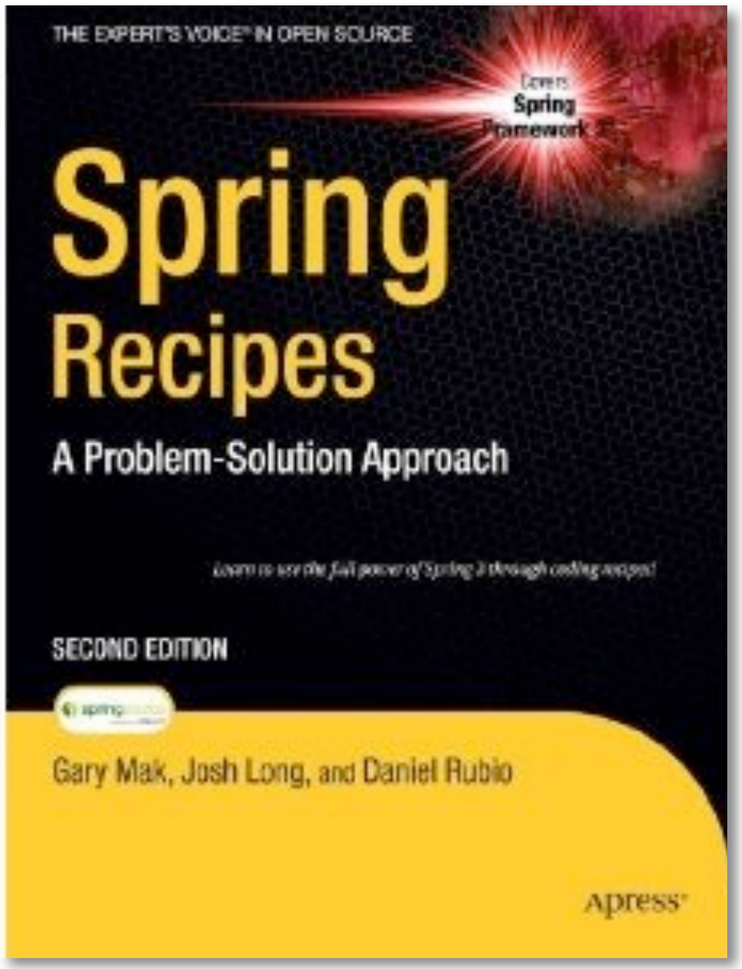
- **SpringSource** 是开发出业界领先的企业 Java 框架 **Spring 框架**的组织。
- **VMware** 在 2009 年收购了 **SpringSource**
- **VMware** 和 **SpringSource** 带您开始云中旅行，并以“构建、运行、管理”为使命
  - 已与 **Adobe**、**SalesForce** 和 **Google** 等行业巨头建立了合作伙伴关系，帮助大家跨多平台为 Spring 用户提供最佳体验

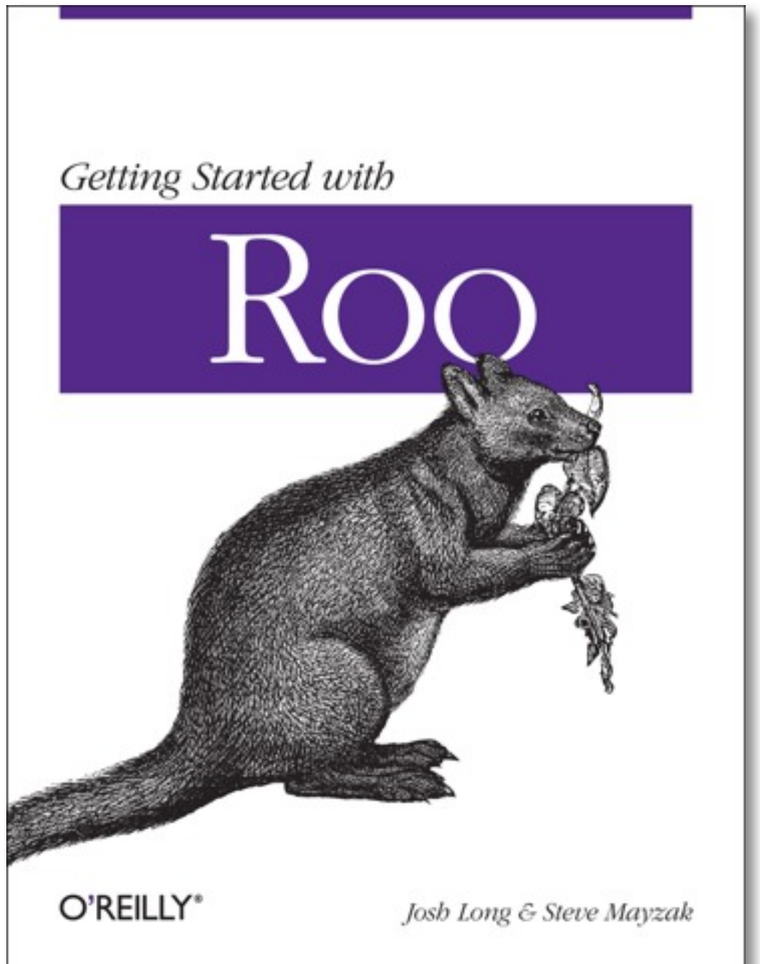
- 是 **Apache HTTPD** 和 **Apache Tomcat** 等项目的主要贡献者

Josh Long (龙之春) 介绍

Spring 开发人员技术布道师

twitter: @starbuxman
weibo: @springsource
josh.long@springsource.com





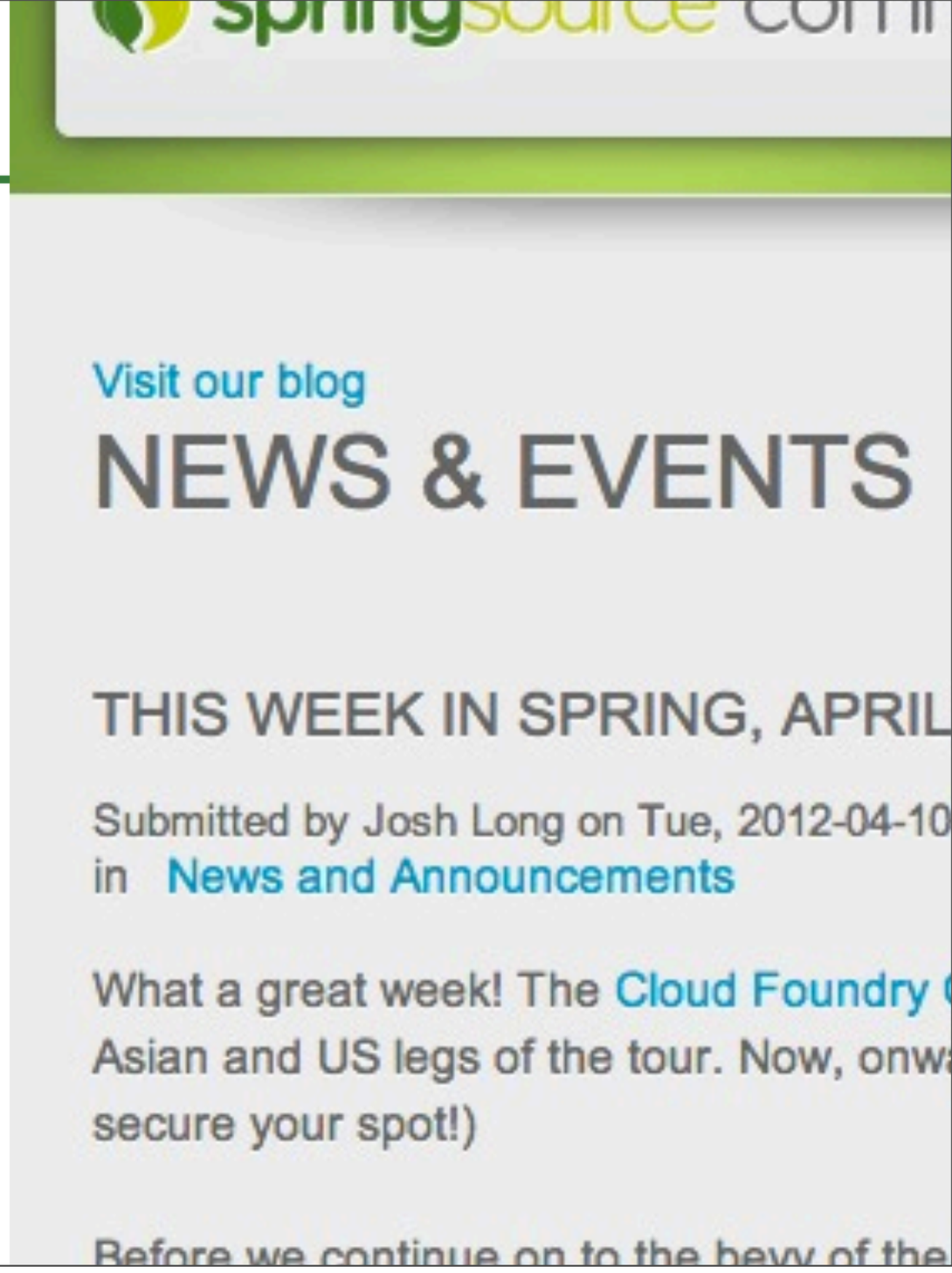
Josh Long 介绍

Spring 开发人员技术布道师

twitter: @starbuxman
josh.long@springsource.com

•参与贡献的项目：

- Spring Integration
- Spring Batch
- Spring Hadoop
- Activiti 工作流程引擎
- 使用 Akka Spring 模块



我们为什么在这里...

我们为什么在这里？

> 软件实体（类、模块、功能等）
> 应该可以扩展，而不允许修改。

-Bob Martin

我们为什么在这里？

千万不要
彻底改造车轮！

Spring 的目标：

让 java 开发更简便

Web 层 与 RIA	服务层	批处理	集成 与 消息传递	数据访问 / NoSQL / 大数据	移动	安全

Spring 框架

云：	轻量型	传统
CloudFoundry Google App Engine Amazon BeanStalk	tc Server Tomcat Jetty	WebSphere JBoss AS WebLogic （基于旧版本、工具）

Spring 框架

框架	说明
Spring Core	基础
Spring @MVC	Web 为主导的架构（伴随核心架构）
Spring Security	提供身份验证、授权的可扩展框架
Spring Webflow	适用于构建多页面流的卓越 Web
Spring Web 服务	契约优先、文档为中心的 SOAP 和 XML Web 服务
Spring Batch	强大的批处理框架
Spring Integration	实施企业集成模式
Spring BlazeDS	支持 Adobe BlazeDS
Spring AMQP	具有 AMQP 消息代理的界面，如 RabbitMQ
Spring Data	NoSQL 选项：MongoDB、Redis、 Riak、CouchDB、Neo4J 等
Spring Social	可集成 Twitter、Facebook、Tripit、 MySpace、LinkedIn 等
Spring Hadoop	提供以 POJO 为中心的方法，来构建 Hadoop 应用。支持 Hbase 及其他应用
Spring Mobile、Spring Android	为 iPhone、Android 服务创建和消费提供一流的支持
Spring GemFire	为 GemFire 企业网格技术提供最简便的界面

数据和数据存储多种多样

- **并非所有数据都位于关系数据库中**
  - 出于可扩展性原因，云环境通常建议您采用备用数据库
  - Hbase、Redis、Mongo 等
- **分布式缓存也会增加难题**
  - 尤其就应用程序级访问模式而言
  - GemFire、Coherence 等
- **几乎无标准化可用**
  - JSR-107（适用于缓存）已经很长时间没有取得任何进展
  - 最终到了 Java EE 7 才有所好转，但仍只适用于缓存
  - 备用数据存储空间太繁杂，无法标准化

Web 客户端多种多样

- **越来越多的客户端 Web 技术**
  - HTML 5 作为新一代浏览器标准
- **服务器端状态越来越少或甚至被完全删除**
  - 特别是，无服务器端用户界面状态
  - 严格受控的用户会话状态
- **JSF 以状态为中心的方法已经非常不适用**
  - 除了特殊类型的应用程序外（对于这些应用程序仍然很实用）
  - Web 应用程序后端和基于 JAX-RS / MVC 风格的 Web 服务
  - 不过，JSF 在不断演变 – 2012 年 JSF 2.2 问世

当前产品和即将面世的产品：3.1、3.2 与 3.3

- **Spring Framework 3.1（2012 年 12 月）**
  - 环境配置文件、基于 Java 的配置、声明式缓存
  - Java 7 初始支持、基于 Servlet 3.0 的部署
- **Spring Framework 3.2（2012 年 12 月）**
  - 基于 Gradle 的 Build、基于 GitHub 的 贡献模型
  - 完全面向 Java 7、 Servlet 3.0 上的异步 MVC 处理
- **Spring Framework 3.3（2013 年第四季度）**
  - 全面的 Java SE 8 支持（包括 lambda 表达式）
    - Spring 中的单一抽象法类型处于有利地位
  - 支持 Java EE 7 API 级和 WebSocket

Spring 3.1

当前产品和即将面世的产品：3.1、3.2 与 3.3

- **Spring Framework 3.1（2012 年 12 月）**
  - 环境配置文件、基于 Java 的配置、声明式缓存
  - Java 7 初始支持、基于 Servlet 3.0 的部署
- **Spring Framework 3.2（2012 年 12 月）**
  - 基于 Gradle 的 Build、基于 GitHub 的 贡献模型
  - 完全面向 Java 7、 Servlet 3.0 上的异步 MVC 处理
- **Spring Framework 3.3（2013 年第四季度）**
  - 全面的 Java SE 8 支持（包括 lambda 表达式）
    - Spring 中的单一抽象法类型处于有利地位
  - 支持 Java EE 7 API 级和 WebSocket

Spring Framework 3.1：精选功能

- 环境抽象和配置文件
- 基于 Java 的应用程序配置
- 对测试环境框架的全面检查

- 缓存抽象与声明式缓存
- 基于 Servlet 3.0 的 Web 应用程序
- @MVC 处理与 Flash 属性

- 优化的 JPA 支持
- Hibernate 4.0 与 Quartz 2.0
- 支持 Java SE 7

环境抽象

环境抽象

- 将 Bean 定义分组以便在特定环境中激活

环境抽象

- 将 Bean 定义分组以便在特定环境中激活
  - 例如，开发、测试、生产

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境
  - 属性源的层次结构

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境
  - 属性源的层次结构

- 可注入的环境抽象 API

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境
  - 属性源的层次结构

- 可注入的环境抽象 API
  - org.springframework.core.env.Environment

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境
  - 属性源的层次结构

- 可注入的环境抽象 API
  - org.springframework.core.env.Environment
- 统一属性解析 SPI

环境抽象

- **将 Bean 定义分组以便在特定环境中激活**
  - 例如，开发、测试、生产
  - 可能不同的部署环境
- **占位符自定义解析**
  - 取决于实际环境
  - 属性源的层次结构

- 可注入的环境抽象 API
  - org.springframework.core.env.Environment
- 统一属性解析 SPI
  - org.springframework.core.env.PropertyResolver

Bean 定义配置文件

```
<beans profile="production">
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
          destroy-method="close">
        <property name="driverClass" value="${database.driver}"/>
        <property name="jdbcUrl" value="${database.url}"/>
        <property name="username" value="${database.username}"/>
        <property name="password" value="${database.password}"/>
    </bean>
</beans>

<beans profile="embedded">
    <jdbc:embedded-database id="dataSource" type="H2">
        <jdbc:script location="/WEB-INF/database/schema-member.sql"/>
        <jdbc:script location="/WEB-INF/database/schema-activity.sql"/>
        <jdbc:script location="/WEB-INF/database/schema-event.sql"/>
        <jdbc:script location="/WEB-INF/database/data.sql"/>
    </jdbc:embedded-database>
</beans>
```

环境配置

环境配置

- 将特定 Bean 定义与特定环境相关联

环境配置

- 将特定 Bean 定义与特定环境相关联
  - <beans> 元素上的 XML“配置文件”属性
  - 配置类上的配置文件注释

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**
  - 例如，通过系统属性

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**
  - 例如，通过系统属性
    - -Dspring.profiles.active=**development**

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**
  - 例如，通过系统属性
    - -Dspring.profiles.active=**development**
  - 或部署单元以外的其他方法

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**
  - 例如，通过系统属性
    - -Dspring.profiles.active=**development**
  - 或部署单元以外的其他方法
    - 根据环境公约

环境配置

- **将特定 Bean 定义与特定环境相关联**
  - <beans> 元素上的 **XML**“配置文件”属性
  - 配置类上的配置文件注释
  - 单个组件类上的配置文件注释

- **根据名称激活特定的配置文件**
  - 例如，通过系统属性
    - -Dspring.profiles.active=**development**
  - 或部署单元以外的其他方法
    - 根据环境公约

- **理想情况：跨不同阶段/环境无需接触部署单元**

基于 Java 的应用程序配置

基于 Java 的应用程序配置

- 特定于应用程序的容器配置

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件

- **相当于 XML 命名空间功能**
  - 但不属于一对一映射

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件

- **相当于 XML 命名空间功能**
  - 但不属于一对一映射
  - 从以注释为导向的角度看，“自然”容器配置

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件
- **相当于 XML 命名空间功能**
  - 但不属于一对一映射
  - 从以注释为导向的角度看，“自然”容器配置
- **典型的基础架构设置**

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件
- **相当于 XML 命名空间功能**
  - 但不属于一对一映射
  - 从以注释为导向的角度看，“自然”容器配置
- **典型的基础架构设置**
  - 事务

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件
- **相当于 XML 命名空间功能**
  - 但不属于一对一映射
  - 从以注释为导向的角度看，“自然”容器配置
- **典型的基础架构设置**
  - 事务
  - 调度

基于 Java 的应用程序配置

- **特定于应用程序的容器配置**
  - 与 Spring 3.0 的 @Configuration 风格相同
  - 注重自定义基于注释的 Spring 处理组件
- **相当于 XML 命名空间功能**
  - 但不属于一对一映射
  - 从以注释为导向的角度看，“自然”容器配置
- **典型的基础架构设置**
  - 事务
  - 调度
  - MVC 定制

那么，所有这些的代码形式是什么样呢？

我想要数据库访问 ... 具有 Hibernate 4 支持

```
@Service
public class CustomerService {

    public Customer getCustomerById( long customerId)    {
        ...
    }

    public Customer createCustomer( String firstName, String lastName, Date date){
        ...
    }

}
```

我想要数据库访问 ... 具有 Hibernate 4 支持

```
@Service
public class CustomerService {

   @Inject
   private SessionFactory sessionFactory;

   public Customer createCustomer(String firstName,
                                  String lastName,
                                  Date signupDate)    {
      Customer customer = new Customer();
      customer.setFirstName(firstName);
      customer.setLastName(lastName);
      customer.setSignupDate(signupDate);

      sessionFactory.getCurrentSession().save(customer);
      return customer;
   }

}
```

我想要数据库访问 ... 具有 Hibernate 4 支持

```
@Service
public class CustomerService {

   @Inject
   private SessionFactory sessionFactory;

   @Transactional
   public Customer createCustomer(String firstName,
                                  String lastName,
                                  Date signupDate)    {
      Customer customer = new Customer();
      customer.setFirstName(firstName);
      customer.setLastName(lastName);
      customer.setSignupDate(signupDate);

      sessionFactory.getCurrentSession().save(customer);
      return customer;
   }

}
```

我想要轻松的终端...

```
@Service
public class CustomerService {

    @Inject
    private SessionFactory sessionFactory;

    @Transactional(readOnly = true)
    @Cacheable(“customers”)
    public Customer getCustomerById( long customerId)    {
        ...
    }

    ...

        ...

}
```

我想要轻松的终端...

```
package org.springsource.examples.spring31.web;
..

@Controller
public class CustomerController {

   @Inject
   private CustomerService customerService;

   @RequestMapping(value = "/customer/{id}" )
   @ResponseBody
   public Customer customerById( @PathVariable("id") Integer id  ) {
      return customerService.getCustomerById(id);
   }
    ...
}
```

...但是 SessionFactory 源自哪里？

Spring ApplicationContext

- **Spring 会管理您委托其管理的 Bean（隐式、显式）**
  - 使用注释（JSR 250、JSR 330、本机）
  - XML
  - Java 配置
  - 组件扫描
- **当然可以全盘采用，混搭！**

- **所有配置形式会告知 ApplicationContext 如何管理您的 Bean**

Spring ApplicationContext

- **注释（组件扫描）**
  - 在您希望 Spring 圆满解决问题时使用最佳，无需显式配置
- **Java 配置**
  - 类型安全且显式 – 所有配置集中化。提供对应用程序的概观视图
- **XML**
  - 显式 –命名空间在多数情况下仍最为有效
- **同时采用**
  - 采用适用于常规第三方 Bean（如 DataSource） 的 Java 配置
  - DSL 的 XML 命名空间和更高级别功能（Spring 集成、批处理等）
  - 组件注释（@Service 或 Spring MVC @Controller）

Spring ApplicationContext

在 XML 中：

```
public class Main {
  public static void main(String [] args) throws Throwable {
    ApplicationContext ctx =
      new ClassPathXmlApplication(  “my-config.xml” );
    CustomerService serviceReference =  ctx.getBean( CustomerService.class );
    Customer customer = serviceReference.createCustomer( "Juergen", "Hoeller");
  }
}
```

Spring ApplicationContext

在 XML 中：

```
public class Main {
 public static void main(String [] args) throws Throwable {
   ApplicationContext ctx =
     new ClassPathXmlApplication(  “my-config.xml” );
   CustomerService serviceReference =  ctx.getBean( CustomerService.class );
   Customer customer = serviceReference.createCustomer( "Juergen", "Hoeller");
 }
}
```

在 Java 配置中

```
public class Main {
 public static void main(String [] args) throws Throwable {
   ApplicationContext ctx =
     new AnnotationConfigApplicationContext( ServicesConfiguration.class );
   CustomerService serviceReference =  ctx.getBean( CustomerService.class );
   Customer customer = serviceReference.createCustomer( "Juergen", "Hoeller");
 }
}
```

在 Spring 3.1 中进行配置的快速入门

```
 ....
<beans>

  <tx:annotation-driven transaction-manager = "txManager"  />

  <context:component-scan base-package = "org.springsource.examples.spring31.services" />

  <context:property-placeholder properties = "config.properties" />

  <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
      <property name = "sessionFactory" ref = "sessionFactory" />
  </bean>

   <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    ...
   </bean>

   <bean id = "dataSource" class  = "..SimpleDriverDataSource">
      <property name= "userName" value = "${ds.user}"/>
       ...
   </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
....
<beans>

 <tx:annotation-driven transaction-manager = "txManager"  />

 <context:component-scan base-package = "org.springsource.examples.spring31.services" />

 <context:property-placeholder properties = "config.properties" />

 <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
     <property name = "sessionFactory" ref = "sessionFactory" />
 </bean>

  <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   ...
  </bean>

   <bean id = "dataSource" class  = "..SimpleDriverDataSource">
      <property name= "userName" value = "${ds.user}"/>
       ...
   </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
@Configuration
@PropertySource("/config.properties")
@EnableTransactionManagement
@ComponentScan(basePackageClasses = {CustomerService.class})
public class ServicesConfiguration {

   @Inject private Environment environment;

   @Bean
   public PlatformTransactionManager txManager() throws Exception {
      return new HibernateTransactionManager(this.sessionFactory());
   }

   @Bean
   public SessionFactory sessionFactory()  { ... }

   @Bean
   public DataSource dataSource(){
      SimpleDriverDataSource sds = new SimpleDriverDataSource();
      sds.setUserName( this.environment.getProperty( “ds.user”));
      // ...
      return sds;
   }

}
```

```
ApplicationContext ctx =
  new AnnotationConfigApplicationContext( ServicesConfiguration.class );
```

在 Spring 3.1 中进行配置的快速入门

```
....
<beans>

 <tx:annotation-driven transaction-manager = "txManager"  />

 <context:component-scan base-package = "org.springsource.examples.spring31.services" />

 <context:property-placeholder properties = "config.properties" />

 <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
     <property name = "sessionFactory" ref = "sessionFactory" />
 </bean>

  <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   ...
  </bean>

  <bean id = "dataSource" class  = "..SimpleDriverDataSource">
     <property name= "userName" value = "${ds.user}"/>
      ...
  </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
@Configuration
@PropertySource("/config.properties")
@EnableTransactionManagement
@ComponentScan(basePackageClasses = {CustomerService.class})
public class ServicesConfiguration {

   @Inject private Environment environment;

   @Bean
   public PlatformTransactionManager txManager() throws Exception {
      return new HibernateTransactionManager(this.sessionFactory());
   }

   @Bean
   public SessionFactory sessionFactory()  { ... }

   @Bean
   public DataSource dataSource(){
      SimpleDriverDataSource sds = new SimpleDriverDataSource();
      sds.setUserName( this.environment.getProperty( “ds.user”));
      // ...
      return sds;
   }

}
```

```
ApplicationContext ctx =
  new AnnotationConfigApplicationContext( ServicesConfiguration.class );
```

在 Spring 3.1 中进行配置的快速入门

```
....
<beans>

 <tx:annotation-driven transaction-manager = "txManager"  />

 <context:component-scan base-package = "org.springsource.examples.spring31.services" />

 <context:property-placeholder properties = "config.properties" />

 <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
     <property name = "sessionFactory" ref = "sessionFactory" />
 </bean>

  <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   ...
  </bean>

  <bean id = "dataSource" class  = "..SimpleDriverDataSource">
     <property name= "userName" value = "${ds.user}"/>
      ...
  </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
@Configuration
@PropertySource("/config.properties")
@EnableTransactionManagement
@ComponentScan(basePackageClasses = {CustomerService.class})
public class ServicesConfiguration {

   @Inject private Environment environment;

   @Bean
   public PlatformTransactionManager txManager() throws Exception {
      return new HibernateTransactionManager(this.sessionFactory());
   }

   @Bean
   public SessionFactory sessionFactory()  { ... }

   @Bean
   public DataSource dataSource(){
      SimpleDriverDataSource sds = new SimpleDriverDataSource();
      sds.setUserName( this.environment.getProperty( “ds.user”));
      // ...
      return sds;
   }

}
```

```
ApplicationContext ctx =
  new AnnotationConfigApplicationContext( ServicesConfiguration.class );
```

在 Spring 3.1 中进行配置的快速入门

```
....
<beans>

 <tx:annotation-driven transaction-manager = "txManager"  />

 <context:component-scan base-package = "org.springsource.examples.spring31.services" />

 <context:property-placeholder properties = "config.properties" />

 <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
     <property name = "sessionFactory" ref = "sessionFactory" />
 </bean>

  <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
   ...
  </bean>

  <bean id = "dataSource" class  = "..SimpleDriverDataSource">
     <property name= "userName" value = "${ds.user}"/>
      ...
  </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
@Configuration
@PropertySource("/config.properties")
@EnableTransactionManagement
@ComponentScan(basePackageClasses = {CustomerService.class})
public class ServicesConfiguration {

   @Inject private Environment environment;

   @Bean
   public PlatformTransactionManager txManager() throws Exception {
      return new HibernateTransactionManager(this.sessionFactory());
   }

   @Bean
   public SessionFactory sessionFactory()  { ... }

   @Bean
   public DataSource dataSource(){
      SimpleDriverDataSource sds = new SimpleDriverDataSource();
      sds.setUserName( this.environment.getProperty( “ds.user”) );
      // ...
      return sds;
   }

}
```

```
ApplicationContext ctx =
  new AnnotationConfigApplicationContext( ServicesConfiguration.class );
```

在 Spring 3.1 中进行配置的快速入门

```
....
<beans>

 <tx:annotation-driven transaction-manager = "txManager"  />

 <context:component-scan base-package = "org.springsource.examples.spring31.services" />

 <context:property-placeholder properties = "config.properties" />

 <bean id = "txManager"  class = "org.springframework.orm.hibernate4.HibernateTransactionManager">
     <property name = "sessionFactory" ref = "sessionFactory" />
 </bean>

 <bean id = "sessionFactory"  class = "org.springframework.orm.hibernate4.LocalSessionFactoryBean">
  ...
 </bean>

 <bean id = "dataSource" class  = "..SimpleDriverDataSource">
    <property name= "userName" value = "${ds.user}"/>
     ...
 </bean>

</beans>
```

```
ApplicationContext ctx =
  new ClassPathXmlApplication( “service-config.xml” );
```

在 Spring 3.1 中进行配置的快速入门

```
@Configuration
@PropertySource("/config.properties")
@EnableTransactionManagement
@ComponentScan(basePackageClasses = {CustomerService.class})
public class ServicesConfiguration {

   @Inject private Environment environment;

   @Bean
   public PlatformTransactionManager txManager() throws Exception {
      return new HibernateTransactionManager(this.sessionFactory());
   }

   @Bean
   public SessionFactory sessionFactory() { ... }

   @Bean
   public DataSource dataSource(){
      SimpleDriverDataSource sds = new SimpleDriverDataSource();
      sds.setUserName( this.environment.getProperty( “ds.user”));
      // ...
      return sds;
   }

}
```

```
ApplicationContext ctx =
  new AnnotationConfigApplicationContext( ServicesConfiguration.class );
```

测试环境框架

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(
   loader=AnnotationConfigContextLoader.class,
   classes={TransferServiceConfig.class, DataConfig.class})
@ActiveProfiles("dev")
public class TransferServiceTest {

   @Autowired
   private TransferService transferService;

   @Test
   public void testTransferService() {
         ...
   }
}
```

“c:” 命名空间

- **新的 XML 命名空间用于 Bean 配置**
  - <constructor-arg> 的快捷方式
    - 内联参数值
    - 与现有 “p:” 命名空间类似
  - 使用构造函数参数名称
    - 因易读性原因而推荐
    - 应用程序类文件中的调试符号必须可用

```
<bean class="..." c:age="10" c:name="myName"/>

<bean class="..." c:name-ref="nameBean" c:spouse-ref="spouseBean"/>
```

缓存抽象

缓存抽象

- 缓存管理器与缓存抽象

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要
    - 尤其是在云环境中

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要
    - 尤其是在云环境中

- **适用于 EhCache、GemFire、Coherence 等的后端适配器**

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要
    - 尤其是在云环境中

- **适用于 EhCache、GemFire、Coherence 等的后端适配器**
  - EhCache 适配器与 Spring Core 一同提供

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要
    - 尤其是在云环境中

- **适用于 EhCache、GemFire、Coherence 等的后端适配器**
  - EhCache 适配器与 Spring Core 一同提供

- **每个环境特定的缓存设置 – 通过配置文件？**

缓存抽象

- **缓存管理器与缓存抽象**
  - 在 org.springframework.cache 中
    - 直到仅包含 EhCache 支持的 3.0 版本为止
  - 对于分布式缓存的兴起特别重要
    - 尤其是在云环境中

- **适用于 EhCache、GemFire、Coherence 等的后端适配器**
  - EhCache 适配器与 Spring Core 一同提供

- **每个环境特定的缓存设置 – 通过配置文件？**
  - 实际上可能要适应一种提供运行时的服务

缓存抽象

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);
```

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);

@Cacheable(condition="name.length < 10")
```

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);

@Cacheable(condition="name.length < 10")
public Owner loadOwner(String name);
```

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);

@Cacheable(condition="name.length < 10")
public Owner loadOwner(String name);
```

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);

@Cacheable(condition="name.length < 10")
public Owner loadOwner(String name);

@CacheEvict
```

缓存抽象

```
@Cacheable
public Owner loadOwner(int id);

@Cacheable(condition="name.length < 10")
public Owner loadOwner(String name);

@CacheEvict
public void deleteOwner(int id);
```

基于 Servlet 3.0 的 Web 应用程序

- **对于 Servlet 3.0 容器的显式支持**
  - 如 Tomcat 7 和 GlassFish 3
  - 同时，保持与 Servlet 2.4+ 兼容

- **支持无 XML Web 应用程序设置（无 web.xml）**
  - Servlet 3.0 的 Servlet 容器初始化程序机制
  - 与 Spring 3.1 的 注释配置 Web 应用程序环境
  - 以及 Spring 3.1 的环境抽象相结合

- **Spring MVC 中的 Servlet 3.0 本机功能列举**
  - 分段解析抽象后的标准 Servlet 3.0 文件上传
  - 即将面世的 Spring 3.2 支持异步请求处理

Web 应用程序初始化程序示例

```
/**
 * Automatically detected and invoked on startup by Spring's
   ServletContainerInitializer. May register listeners, filters,
   servlets etc against the given Servlet 3.0 ServletContext.
 */
public class MyWebAppInitializer implements WebApplicationInitializer {


   public void onStartup(ServletContext sc) throws ServletException {
      // Create the 'root' Spring application context
      AnnotationConfigWebApplicationContext root =
            new AnnotationConfigWebApplicationContext();
      root.scan("com.mycompany.myapp");
      root.register(FurtherConfig.class);

      // Manages the lifecycle of the root application context
      sc.addListener(new ContextLoaderListener(root));
      ...
   }
}
```

@MVC 处理与 Flash 属性

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

- FlashMap 支持与 FlashMapManager 抽象

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

- **FlashMap 支持与 FlashMapManager 抽象**
  - 以 RedirectAttributes 作为新的 @MVC 处理程序方法参数类型

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

- **FlashMap 支持与 FlashMapManager 抽象**
  - 以 RedirectAttributes 作为新的 @MVC 处理程序方法参数类型
    - 显式调用 addFlashAttribute 为输出 FlashMap 添加值

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

- **FlashMap 支持与 FlashMapManager 抽象**
  - 以 RedirectAttributes 作为新的 @MVC 处理程序方法参数类型
    - 显式调用 addFlashAttribute 为输出 FlashMap 添加值
  - 将向用户会话临时添加一个传出FlashMap

@MVC 处理与 Flash 属性

- 请求方法处理程序适配器
  - 跨多个控制器对处理程序方法进行任意映射
  - 更优异的处理程序方法参数自定义
    - HandlerMethodArgumentResolver
    - HandlerMethodReturnValueHandler
    - 等等

- **FlashMap 支持与 FlashMapManager 抽象**
  - 以 RedirectAttributes 作为新的 @MVC 处理程序方法参数类型
    - 显式调用 addFlashAttribute 为输出 FlashMap 添加值
  - 将向用户会话临时添加一个传出FlashMap
  - 请求的传入 FlashMap 将自动显示给模型

优化的 JPA 支持

- **无 persistence.xml 包扫描**
  - LocalContainerEntityManagerFactoryBean 上的“packagesToScan”功能
    - 在特定应用程序包内根据 @Entity 类构建持久保留单元
  - 与 Hibernate 3 AnnotationSessionFactoryBean 类似
    - 合并至核心的 Hibernate 4 LocalSessionFactoryBean
  - 非常适用于具有单一默认持久保留单元的应用程序

- **根据持久保留单元名进行一致性 JPA 设置**
  - JPA 规范完全建立在持久保留单元名称概念的基础之上
    - @PersistenceContext / @PersistenceUnit 即为持久保留单元名称
  - 现在， Spring 的 JpaTransactionManager 和根据单元名称进行的 co. 支持设置以及
    - 备选设置直接指代 EntityManagerFactory Bean
    -

第三方支持更新

- **Hibernate 4.0**
  - 本机并通过 JPA
  - 专用 **org.springframework.orm.hibernate4 包本机支持**
    - 在 Hibernate API 中处理包的重新布局
  - 保持与 Hibernate 3.2+ in o.sf.orm.hibernate3 兼容

- **Quartz 2.0**
  - JobDetail 和 Trigger 现在属于 Quartz 2.0 API 中的界面
    - 而之前他们传统意义上属于 Quartz 1.x 中的基类
  - SchedulerFactoryBean 现在自动适应 Quartz 2.0（如存在）
    - Quartz 2.0 的 JobDetail/CronTrigger/SimpleTrigger**Factory**Bean 变量
  - 在同一支持包中，保持与 Quartz 1.5+ 兼容

Java SE 7

Java SE 7

- **Spring 3.1 引入 Java SE 7 支持**
  - 在运行时充分利用 JRE 7
  - 支持 JDBC 4.1
  - 支持 Fork-Join 框架

Java SE 7

- **Spring 3.1 引入 Java SE 7 支持**
  - 在运行时充分利用 JRE 7
  - 支持 JDBC 4.1
  - 支持 Fork-Join 框架

- **2011 年夏发布的 Oracle OpenJDK 7**
  - IBM JDK 7 紧随其后推出
  - Java 7 将很快成为基于新 Java 的项目的标准

Java SE 7

- **Spring 3.1 引入 Java SE 7 支持**
  - 在运行时充分利用 JRE 7
  - 支持 JDBC 4.1
  - 支持 Fork-Join 框架

- **2011 年夏发布的 Oracle OpenJDK 7**
  - IBM JDK 7 紧随其后推出
  - Java 7 将很快成为基于新 Java 的项目的标准

Java SE 7

- **Spring 3.1 引入 Java SE 7 支持**
  - 在运行时充分利用 JRE 7
  - 支持 JDBC 4.1
  - 支持 Fork-Join 框架

- **2011 年夏发布的 Oracle OpenJDK 7**
  - IBM JDK 7 紧随其后推出
  - Java 7 将很快成为基于新 Java 的项目的标准

- **Spring 框架尚未基于 Java 7 构建**
  - Build 升级将在 Spring 3.3 中实施d projects soon

github.com/SpringSource

github Search or Type a Command Explore Gist Blog Help joshlong

SpringSource
SpringSource

http://www.springsource.org
Joined on Jun 29, 2010

121 public repos **16** private repos **211** members

Repositories Members You are a member of this Organization!

Find a Repository… All Public Private Sources Forks Mirrors

spring-data-solr
Spring Data - Solr integration
Last updated 4 hours ago
Java ★ 15 10

spring-net-amqp
Spring AMQP for .NET
Last updated 7 hours ago
C# ★ 14 5

spring-data-jdbc-ext
Spring Data JDBC Extensions. Support for database specific extensions to standard JDBC including support for Oracle RAC fast connection failover, AQ JMS support and support for using advanced data types.
Last updated 10 hours ago
Java ★ 24 5

spring-insight-plugins
Public Repository of Plugins for Spring Insight
Last updated 20 hours ago
Java ★ 19 5

spring-framework
The Spring Framework
Java ★ 1,302 504

Spring 3.2

当前产品和即将面世的产品：3.1、3.2 与 3.3

- **Spring Framework 3.1（2012 年 12 月）**
  - 环境配置文件、基于 Java 的配置、声明式缓存
  - Java 7 初始支持、基于 Servlet 3.0 的部署
- **Spring Framework 3.2（2012 年 12 月）**
  - 基于 Gradle 的 Build、基于 GitHub 的 贡献模型
  - 完全面向 Java 7、 Servlet 3.0 上的异步 MVC 处理
- **Spring Framework 3.3（2013 年第四季度）**
  - 全面的 Java SE 8 支持（包括 lambda 表达式）
    - Spring 中的单一抽象法类型处于有利地位
  - 支持 Java EE 7 API 级和 WebSocket

计划变更

- **我们原本要在 Spring 3.2 中推出 Java SE 8 和 Java EE 7 主题**
- **但是，由于一再地精简 Java EE 7：2013 年第二季度**
  - 最终移除部分功能并延迟推出（不再重点关注云）
- **Java SE 8 (OpenJDK 8) 也重新计划：2013 年 9 月**
  - 再一次移除部分功能并延迟推出（不再包含模块系统）
  - 功能完备的开发程序预览版有望在 2013 年 2 月推出
- **我们的解决方案：于 2012 年第四季度推出进行了核心框架优化的 Spring 3.2**
  - 支持 Java SE 8 和 Java EE 7 的 Spring 3.3 将在 2013 年第四季度推出

3.2 具备哪些新功能？

- 基于 Gradle 的 Build
- 基于 Java 7 构建的二进制
- 内联 ASM 4.0 和 CGLIB 3.0
- Servlet 3.0 上的异步 MVC 处理
- Spring MVC 测试支持
- MVC 配置优化
- SpEL 优化
- 还包括许多运行时优化
  - 部分移植到 3.1.2/3.1.3
- 一般的 Spring MVC 优化细节
  - Servlet 3 异步支持
  - REST 方案中支持错误报告
  - 内容协商策略
  - 矩阵变量

Spring 3.2

- **Spring 3.1 仅具备对 Java 7 的前期支持**
  - JDBC 4.1、ForkJoinPool 等
  - 框架本身仍在 Java 6 上进行编译
- **Spring 3.2 现在基于 Java 7 构建**
  - CI build 在 Java 5、6 和 7 上运行
- **通过我们全新的 Gradle Build 实现**
- **Spring 3.2 配备 ASM 4.0 和 CGLIB 3.0**
  - 全面支持 Java 7 字节码格式
  - ASM 和 CGLIB 现已内联至 Spring 模块 Jar

异步 MVC 处理：可调用

```
@RequestMapping(name =“/upload”,
                        method=RequestMethod.POST)
 public Callable<String> processUpload(MultipartFile file) {

    return new Callable<String>() {
       public String call() throws Exception {
          // ...
          return "someView";
       }
    };

}
```

- **Spring MVC 管理的线程**
- 对长时间运行的数据库作业、第三方 REST API 调用等有益

异步 MVC 处理：DeferredResult

```
@RequestMapping("/quotes")
@ResponseBody
public DeferredResult quotes() {
    DeferredResult deferredResult = new DeferredResult();
    // Add deferredResult to a Queue or a Map...
    return deferredResult;
}

// In some other thread:
// Set the return value on the DeferredResult deferredResult.set(data);
```

- **在 Spring MVC 之外管理的线程**
- JMS 或 AMQP 消息侦听器、其他 HTTP 请求等

异步 MVC 处理：AsyncTask

```
@RequestMapping(name ="/upload", method=RequestMethod.POST)
 public AsyncTask<Foo> processUpload(MultipartFile file) {

    TaskExecutor asyncTaskExecutor  = new AsyncTaskExecutor(...);

    return new AsyncTask<Foo>(
          1000L,                          // timeout
           asyncTaskExecutor,             // thread pool
           new Callable<Foo>(){ ..}  // thread
      );

}
```

- 与可调用相同，具有额外功能
- 覆盖异步处理的超时值
- 允许您指定一个特定的 AsyncTaskExecutor

内容协商策略

ContentNegotiationStrategy

- 按 'Accept' 标头
- 按 URL 扩展名（.xml、.json 等）
- 按请求参数，例如 /accounts/1?format=json
- 固定内容类型，如备用选项

•ContentNegotiationManager

- 具有一个或多个 ContentNegotiationStrategy 实例
- 可与下列实例共用：

  RequestMappingHandlerMapping、
  RequestMappingHandlerAdapter、
  ExceptionHandlerExceptionResolver
  ContentNegotiatingViewResolver

“每个路径段都可能包含一系列参数，以分号‘;’符表示。

这些参数对于相对参照解析并无重要影响。”

RFC 2396, 3.3 节

“分号 (“;”) 和等号 (“=”) 保留字符通常用于分隔适用于该段的参数和参数值。逗号 (“,”) 保留字符通常用于类似用途。”

RFC 3986, 3.3 节

矩阵变量

- 两种使用类型
- 路径段名/值对

```
/qa-releases;buildNumber=135;revision=3.2
```

- 作为分隔列表路径段

```
/answers/id1;id2;id3;id4/comments
```

矩阵变量：常见实例

```
// GET /pets/42;q=11;r=22

@RequestMapping(value = "/pets/{petId}")
public void findPet(
    @PathVariable String petId, @MatrixVariable int q) {

   // petId == 42
   // q == 11

}
```

矩阵变量：获取所有矩阵变量

```
// GET /owners/42;q=11;r=12/pets/21;q=22;s=23

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}")
public void findPet(
    @MatrixVariable Map<String, String> matrixVars) {

    // matrixVars: ["q" : [11,22], "r" : 12, "s" : 23]

}
```

矩阵变量：限定路径段

```
// GET /owners/42;q=11/pets/21;q=22

@RequestMapping(value = "/owners/{ownerId}/pets/{petId}")
public void findPet(
    @MatrixVariable(value="q", pathVar="ownerId") int q1,
    @MatrixVariable(value="q", pathVar="petId") int q2) {

  // q1 == 11
  // q2 == 22

}
```

Spring 3.3...

当前产品和即将面世的产品：3.1、3.2 与 3.3

- **Spring Framework 3.1（2012 年 12 月）**
  - 环境配置文件、基于 Java 的配置、声明式缓存
  - Java 7 初始支持、基于 Servlet 3.0 的部署
- **Spring Framework 3.2（2012 年 12 月）**
  - 基于 Gradle 的 Build、基于 GitHub 的 贡献模型
  - 完全面向 Java 7、 Servlet 3.0 上的异步 MVC 处理
- **Spring Framework 3.3（2013 年第四季度）**
  - 全面的 Java SE 8 支持（包括 lambda 表达式）
    - Spring 中的单一抽象法类型处于有利地位
  - 支持 Java EE 7 API 级和 WebSocket

Spring Framework 3.3（2013 年第四季度）

- **全面支持 Java 8**
  - 支持 Java EE 7 API 级
  - 重点关注以消息为导向的体系结构
  - 注释驱动型 JMS 终端模型
- **改进的应用程序事件机制**
- **在 Spring MVC 中支持 WebSocket**
- **新一代 Groovy 支持**
- **Grails bean 生成器最终完全纳入 Spring**

- **很可能（但不确定）：将诞生 Spring 4.0**

Java SE 8 说明

- **全面支持 Java 8**
  - lambda 表达式 a.k.a. 闭包
  - 日期和时间 API (JSR-310)
  - 基于 NIO 的 HTTP 客户端 API
  - 参数名称查找
  - java.util.concurrent 改进
- **保留对 Java 5 及更高版本的支持**
  - Java 6 和 7 通用级别
  - Java 8 可能很快变得热门...

Java EE 7 说明

- **支持 Java EE 7 API 级**
  - JCache 1.0
  - JMS 2.0
  - JPA2.1
  - JTA 1.2 (@Transactional)
  - Bean Validation 1.1
  - Servlet3.1
  - JSF 2.2
- **保留对 Java EE 5 及更高版本的支持**

注释驱动型 JMS 终端

```
@JmsListener(destination="myQueue")
public void handleMessage(TextMessage payload);

@JmsListener(destination="myQueue", selector="...")
public void handleMessage(String payload);

@JmsListener(destination="myQueue")
public String handleMessage(String payload);
```

WebSocket 支持

- 当前服务器中对 WebSocket 的支持尚未完全标准化
- 在 Java EE 7 时间框架内即将出现 JSR-356
- Jetty 9 和 co. 的接受速度很快
- Spring 框架 3.3 研究了所有 WebSocket 支持选项
  - 主要是 Spring MVC
  - 通常也用于以消息为导向的体系结构

当前产品和即将面世的产品：3.1、3.2 与 3.3

- **Spring Framework 3.1（2012 年 12 月）**
  - 环境配置文件、基于 Java 的配置、声明式缓存
  - Java 7 初始支持、基于 Servlet 3.0 的部署
- **Spring Framework 3.2（2012 年 12 月）**
  - 基于 Gradle 的 Build、基于 GitHub 的 贡献模型
  - 完全面向 Java 7、 Servlet 3.0 上的异步 MVC 处理
- **Spring Framework 3.3（2013 年第四季度）**
  - 全面的 Java SE 8 支持（包括 lambda 表达式）
    - Spring 中的单一抽象法类型处于有利地位
  - 支持 Java EE 7 API 级和 WebSocket

@SpringSource @Starbuxman

问题？