**常用注解：**

@AutoWired 自动装配（DI），byName和byType最常见，需要在配置文件中注册AutoWiredAnnotationBeanPostProcessor。xml配置中有个对应的autowire属性，常见的有byType和byName两种，但是在找不到匹配Bean时不予理会，因此@Required应运而生，它需要注册RequiredAnnotationBeanPostProcessor。

@Inject，JSR330定义，可以作用在构造器、属性、方法上，可以设置required=false。

@Qualifier，当有多个符合条件的装配者时，使用该注解指定。相当于配置中的ref。更好的方式是使用JSR-250中的@Resource注解。要使这些注解生效，得在配置文件中注册相应的BeanFactoryPostProcessor，也可以简化配置为：<context:annotation-config/>，注意引入相应的命名空间。

JSR-330的@Named相当于Spring的@Component注解。

**Bean配置：**

静态工厂：

public class HelloApiStaticFactory {

//工厂方法

public static HelloApi newInstance(String message) {

//返回需要的Bean实例

return new HelloImpl2(message);

}

}

<!-- 使用静态工厂方法 -->

<bean id="bean3" class="cn.javass.spring.chapter2.HelloApiStaticFactory" factory-method="newInstance">

<constructor-arg index="0" value="Hello Spring!"/>

</bean>

通过factory-method方法创建实例，可以注入参数。问题：不知道创建出来的实例类型。

实例工厂：

package cn.javass.spring.chapter2;

public class HelloApiInstanceFactory {

public HelloApi newInstance(String message) {

return new HelloImpl2(message);

}

}

<!—1、定义实例工厂Bean -->

<bean id="beanInstanceFactory"

class="cn.javass.spring.chapter2.HelloApiInstanceFactory"/>

<!—2、使用实例工厂Bean创建Bean -->

<bean id="bean4"

factory-bean="beanInstanceFactory"

factory-method="newInstance">

<constructor-arg index="0" value="Hello Spring!"></constructor-arg>

</bean>

因为是实例方法，必须使用factory-bean单独指定工厂类。

**循环依赖：**

循环调用无解，死循环。

singleton的Bean可以循环依赖，因为Spring会将之缓存；prototype的Bean无法循环依赖，因为不缓存。对于singleton的循环依赖，Spring先将Bean实例化出来，不注入，缓存之，然后挨个注入。

**Resource和ResourceLoader**

Resource对各种资源提供统一接口，如ByteArrayResource，FileResource，URLResource等；而ResourceLoader则相当于Resource的一个类工厂，通过传入不同参数返回不同的Resource，而不需要我们自己判断。一般通过ApplicationContext的getResource方法来获取Resource，如ctx.getResource(“resource location”)，底层使用ResourceLoader来获取。

在Bean中，如果有property是Resource类型的，在配置时，可以只配置Resource所在路径。

**SpEL Spring Expression Language**

和OGNL、EL、JSTL等类似，不过天然与spring集成。需要注意的是，如果要修改模板开始结束符号，需要建立一个BeanFactoryPostProcessor，在它的postProcessBeanFactory方法中替换模板开始结束符号。如下：

StandardBeanExpressionResolver resolver = (StandardBeanExpressionResolver) beanFactory.getBeanExpressionResolver();

resolver.setExpressionPrefix("%{");

**AOP**

推荐使用自定义注解的方式来配置。类似于@Transactional，这样便于知道当前方法被哪些切面横切。注：@Transactional注解需要配置<tx:annotation-driven/>,需指定事务管理者

使用注解方式配置aop，需要配置<aop:aspectj-autoproxy />

**JDBC和ORM**

spring提供了大量的模板方法，关于jdbc的就不介绍了。

hibernate3：SessionFactory，用LocalSessionFactoryBean。另提供HibernateDaoSupport（可获取HibernateTemplate）。

iBatis：SqlMapClientFactoryBean。

**Web框架集成：**

在web.xml文件中

<context-param>

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

<param-value>

org.springframework.web.context.support.XmlWebApplicationContext

</param-value>

</context-param> 这个是默认的ApplicationContext实现，如果不更改则可以不配置。

<context-param>

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

<param-value>

classpath:chapter10/applicationContext-message.xml

</param-value>

</context-param> 这个用来声明spring的配置文件位置

<listener>

<listener-class>

org.springframework.web.context.ContextLoaderListener

</listener-class>

</listener> 加载Spring Web

需要servlet2.4以上版本支持。

获取Spring Web容器：

WebApplicationContextUtils.getRequiredWebApplicationContext(servletContext) 该方法会抛出异常，也应该抛出异常。

集成Struts2

Struts2通过ObjectFactory接口创建和获取Action的实例，与Spring集成的关键点在于如何创建ObjectFactory实现来从Spring容器中获取对应的Action Bean。

Struts2提供了一个StrutsSpringObjectFactory类，我们使用它作为中介，连结Struts2和Spring。

1、在Spring配置文件中声明Action Bean

<bean id=”helloAction” class=”…”>…</bean>

2、在Struts2配置文件中声明Action

<action name=”hello” class=”helloAction”>…</action> 红字部分是关键

3、在Struts2配置文件或者properties文件中配置

struts.objectFactory = org.apache.struts2.spring.StrutsSpringObjectFactory

**零配置**

<context:component-scan>会自动引入<context:annotation-config/>

**SpringMVC**

前端控制器(Front Controller)：为表现层提供统一访问点，相当于一个门面。可以为多个请求提供共用逻辑(如上下文)，将选择视图与视图的具体处理功能分离。

应用控制器(Application Controller)：前端控制器将选择视图和视图处理分离后，需要应用控制器来管理如何选择视图、如何进行功能处理。

页面控制器(Page Controller)：功能处理代码，收集参数，封装数据到模型。

上下文(Context)：以前我们都把模型数据放到Request中，现在可以放到上下文中，从而与Servlet剥离。一般使用ThreadLocal来实现。

对应到SpringMVC中，前端控制器是DispatcherServlet，应用控制器拆分为处理器映射器(Handler Mapping)和视图解析器(View Resolver)，页面控制器是Controller(接口或注解)。

SpringMVC支持本地化(Locale)解析，主题(Theme)解析，文件上传等。提供了非常灵活的数据验证、格式化和数据绑定机制。提供了强大的约定优于配置的契约式编程支持。

处理流程，参见本目录下SpringMVC核心架构。

**支持的注解**

@Controller 定义一个Controller，推荐使用这种方式

@RequestMapping 定义URI路径映射，可以放在类和方法上，支持占位符、Ant风格和正则表达式。不推荐使用Ant风格。带有几个参数，method：可以为post、get等；params：请求参数中有或者没有某参数，如params=”[!]create”表示有[没有]create参数，甚至可以指定参数值，如params=”submitFlag=create”；headers：请求头部，其他和params一样；生产者消费者：produces和consumes分别表示方法输出和接受什么类型的ContentType；除了生产者和消费者是覆盖类上的注解，其他都是继承。

@RequestParam 绑定单个请求参数，通过get和post都可以获取到

@ModelAttribute 绑定参数到命令对象，并且放到Model中

@SessionAttributes 绑定命令对象到session

@InitBind 被注解的Controller独享的PropertyEditor

@Value 绑定spEL表达式

**RESTful注解**

@CookieValue 绑定cookie数据

@RequestHeader 绑定请求头数据

@RequestBody 绑定请求内容数据。与HttpMessageConverter配合使用。

@RequestPart 绑定”multipart-formdata”数据，可以是从表单提交，也可以直接客户端

@ResponseBody 和@RequestBody一样。

@ResponseStatus 注解在处理方法上，忽略方法返回值，直接将HTTP状态码和信息返回给客户端

@ExceptionHandler Controller级别的异常处理

@PathVariable 绑定URI路径中的变量

HttpEntity<?>、ResponseEntity<?>和@RequestBody类似，但增加了对请求头、响应头的访问。

**一些配置**

<mvc:annotation-driven>，自动注册基于注解所需要的DefaultAnnotationHandlerMapping，AnnotationMethodHandlerAdaptor；支持ConversionService自动注册；支持JSR-303验证框架的自动探测和注册；自动注册相应的HttpMessageConverter(用于支持@RequestBody和@ResponseBody，如xml输入输出转换[只需将JAXB实现放到classpath下]，JSON输入输出转换[只需将Jackson放到classpath下])

<mvc:interceptors>，注册自定义的处理器拦截器。

<mvc:view-controller>，和ParameterizableViewController 类似，收到相应请求后直接选择相应的视图。

<mvc:resources>，逻辑静态资源路径到物理静态资源路径的支持。

<mvc:default-servlet-handler>，当在 web.xml 中 DispatcherServlet 使用<url-pattern>/</url-pattern> 映射时，这个配置中的Servlet能映射静态资源（当 Spring Web MVC 框架没有处理请求对应的控制器时[如一些静态资源]，转交给这个Servlet来响应静态文件，否则报404）

**Spring容器上下文**

ContextLoaderListener的参数里指定的配置文件不包含springmvc，它们都是springmvc的父容器。

DispatcherServlet初始化：①初始化Spring Web MVC的上下文容器，并指定父容器(如果有)②初始化DispatcherServlet使用的一些策略，如HandlerMapping、HandlerAdaptor等，这些默认策略配置在org.springframework.web.servlet.DispatcherServlet.properties文件中。

**HandlerInterceptor**

三个方法，preHandle处理前，如果返回false则不继续往后传递；postHandle处理后页面渲染前；afterCompletion页面渲染完毕后。注意Interceptor是单例的。

**可用的输入输出参数**：

输入：WebRequest/NativeRequest、ServletRequest/HttpServletRequest、MultipartRequest

HttpSession、Locale、Principal、InputStream/Reader、SessionStatus

输出：ServletResponse/HttpServletResponse、OutputStream/Writer、RedirectAttributes

UriComponentsBuilder、Map/ModelMap、命令/表单对象、Errors/BindingResult、HttpEntity

一些注意事项：

ModelAndView会覆盖Model中的同名参数；

不同于Struts2，SpringMVC无法绑定表单中的user.userName到user对象，即，一个页面只能绑定一个表单对象。

@RequestParam(“flag”) int[] flag，xxx?flag=1,2 会报错，将int[]改为String[]则不报错，这应该是bug。

Spring的表单标签库很难用，当前阶段应不予适用。

使用@Valid进行Bean Validate时，方法输入参数必须有BindingResult/Errors

ConversionService：

比PropertyEditor更通用

**Spring3.2 Reference**

001、<context:annotation-config/>自动注册AutowiredAnnotationBeanPostProcessor(支持@AutoWired注解，自动注入，找不到匹配的Bean时，啥都不干),

CommonAnnotationBeanPostProcessor(支持@Resource注解，作用和@AutoWired类似，但优先使用byName方式查找匹配；支持@PostConstruct和@PreDestroy注解，分别用于Bean初始化和销毁), PersistenceAnnotationBeanPostProcessor,

RequiredAnnotationBeanPostProcessor(支持@Required注解，自动注入，找不到匹配Bean时，抛出异常).注意：该配置只对它所在的ApplicationContext下的Bean起作用。

@Inject(JSR330定义)和@AutoWired作用一样。

@Qualifier，当有多个匹配的Bean时，用此注解唯一指定。

综上：<context:annotation-config/>的作用是支持Bean的属性注入。

002、<context:component-scan base-package="org.example"/>，扫描org.example包及其所有子包中的用 @Component、@Repository、@Service、@Controller注解的类并将其注册为Bean。它有子元素<context:exclude-filter/>和<context:include-filter/>用于排除和添加需要扫描的类。同时，该配置自动注册<context:annotation-config/>

综上：<context:component-scan/>的作用是自动扫描特定注解的类并将其注册成为Bean，同时支持注解形式的属性注入。

003、<context:property-placeholder location="classpath: jdbc.properties"/> 属性占位符，格式是${ }，从location指定的properties文件中获取资源。注意和messagesource(用于国际化)、SpEL的区别。

004、MessageSource，用于国际化。ApplicationContext实现了此接口，当ApplicationContext启动时，它会依次从自己、父容器中查找id/name为messagesource的Bean。我们一般使用ReloadableResourceBundleMessageSource这个实现类。

005、**校验/数据绑定/类型转换**

类型转换，spring3.0之前用的是PropertyEditor，注册自定义PropertyEditor如下：

<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">

<property name="customEditors">

<map>

<entry key="example.ExoticType" value="example.ExoticTypeEditor"/>

</map>

</property>

</bean>

3.0后，新增ConversionService，我们使用FormattingConversionServiceFactoryBean来注册，通过属性converters和formatters分别添加自定义类型转换。

**JSR303**校验启用，注册LocalValidatorFactoryBean，指定providerClass和validationMessageSource，分别表示Validator的实现类和messagesource。

@Controller级别的数据校验 使用@InitBind注解一个方法，输入参数为WebDataBinder，为这个DataBinder指定validator和propertyEditor。

006、**SpEL**

ApplicationContext中可直接使用SpEL，如设置bean属性。使用的符号是#{ }，注意和PropertyPlaceHolderConfigurer的区别。systemProperties变量是预定义的，可以直接使用。如#{systemProperties[‘user.region’]}。@Value注解可以使用SpEL。

007、**AOP**

<aop:aspectj-autoproxy/>自动启用基于AspectJ的AOP。@Aspect注解不能被自动扫描到，还得需要@Component。

008、**事务**

事务策略定义在PlatformTransactionManager中，支持JDBC、JTA、Hibernate等。

数据库操作需要配置dataSource、txManager，txManager引用dataSource。如下：

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"

destroy- method="close">

<property name="driverClassName" value="${jdbc.driverClassName}" />

<property name="url" value="${jdbc.url}" />

<property name="username" value="${jdbc.username}" />

<property name="password" value="${jdbc.password}" />

</bean>

<bean id="txManager"

class="org.springframework.jdbc.datasource.DataSourceTransactionManager">

<property name="dataSource" ref="dataSource" />

</bean>

如果是JNDI和JTA事务，则：

<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

<bean id="txManager" class="org.springframework.transaction.jta.JtaTransactionManager"/>

txManager中不需要配置dataSource，由容器负责。

基于注解的声明式事务：

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

@Transactional，默认配置：事务传播PROPAGATION\_REQUIRED，隔离级别ISOLATION\_DEFAULT，事务级别：Read/Write，超时：看底层，回滚：RuntimeException回滚，受检异常不回滚。

**009、OXM/Object Xml Mapping**

提供一致性接口Marshaller(Object to XML)和UnMarshaller(XML to Object)，具体实现可使用JAXB、JIBX、Castor、XmlBeans或其他类库。

**010、SpringMVC**

ViewResolver：forward和redirect，方法返回值类型为String，返回”forword:/web/reg”，第一个/有没有都是根目录。

ContentNegotiatingViewResolver：两种策略：一、根据后缀，如a.jsp，b.pdf，c.xml；二、根据Accept Header，如text/xml，application/pdf。

Locale：提供AcceptHeader、Cookie、Session等策略改变Locale。

文件上传：以commons-fileupload为例，需声明一个CommonsMultipartResolver。在处理方法上，使用MultipartFile类型来接收上传的文件。需要注意的是，当上传文件超过大小时，会抛出异常，需要自行处理。如想使用Controller上的@ExceptionHandler处理该异常，则需要把CommonsMultipartResolver的resolveLazily属性设为true，表示到解析时才抛出异常，这样才能进入Controller，从而被@ExceptionHandler捕获。

异常处理的三种方法：①、配置SimpleMappingExceptionResolver；②自行实现HandlerExceptionResolver接口或者继承其实现类，比如ExceptionHandlerExceptionResolver；③Controller中的@ExceptionHandler。捕获的时候先在①或②中查找匹配，找不到或处理不了再进入到③，再处理不了，抛出异常。对于返回状态码的异常，只能通过web.xml中的error-page元素来处理。

MVC命名空间：

< mvc:annotation-driven />，默认注册了RequestMappingHandlerMapping，RequestMapping -HandlerAdapter，ExceptionHandlerExceptionResolver，用以支持MVC下的各种注解。同时，开启了以下功能：

ConversionService，包括Converter和Formatter，如@Number、@DateTimeFormat

JSR303，如@Valid

HttpMessageConverter，注册了：ByteArray[HttpMessageConverter]，String\*，Resource\*(Spring的Resource接口)，Source\*( javax.xml.transform.Source，用于生成XML)，Form\*(将表单数据转换成一个MultiValueMap<String,String>)，Jaxb2RootElement\*(OXM，需要Jaxb2的支持)，MappingJackson2\*(JSON，需要Jackson支持)，AtomFeed\*(Atom Feed，需要Rome支持)，RssChannel\*(Rss Feed，也需要Rome支持)。

ContentNegotiation：后缀优先，再AcceptHeader。如果classpath下有相应jar包，如下类型会自动注册：.json，.xml，.atom，.rss，其他类型最好显式注册。

处理Excel和PDF：前者需要用到POI库，后者需要用到iText库。实现相应的View(可以简单继承AbstractExcelView和AbstractPdfView)就可以了，ViewResolver会使用InternalResourceViewResolver来解析。

**011、远程调用和Webservice**

RMI(Remoting Method Invocation)，Spring’s HTTP Invoker(Spring自己的实现)，Hessian(基于二进制协议)，Burlap(Hessian的XML变种)，JAX-RPC(JEE1.4的Webservice API)，JAX-WS(JEE5以上，JAX-RPC的继任者)，JMS(Java Message Service)。

**012、JMS**

JmsTemplate：和JdbcTemplate类似，线程安全，通过回调操作。自行释放资源。默认的knowledge是PTP。

标准的发送、接收消息流程，都要消费临时的ConnectionFactory、Connection、Session对象等，如果能将这些对象加以缓存，性能会好不少。

因此：SingleConnectionFactory和CachingConnectionFactory应运而生。前者每次都返回同一个Connection，忽略close()方法。CachingConnectionFactory缓存的对象更多。

Destination管理：和ConnectionFactory一样，可以作为JNDI资源来管理。也可以通过DestinationResolver来创建。

MessageListenerContainer，异步接收消息的Listener容器。注册消息消费者，参与事务管理，资源获取和释放，异常转换等等。SimpleMessageListenerContainer一般用于独立应用程序，DefaultMessageListenerContainer用于J2EE环境，支持XA事务和动态Destination。

事务处理：JmsTransactionManager，和Spring的其他事务管理模型保持一致。

MessageConverter：用于Java Object和JMS Message之间的转换。

接收消息：同步接收和异步接收。异步时，要保证MessageListener的线程安全性。Spring提供了更强大的类MessageListenerAdapter来将任意一个POJO适配成MessageListener。注意一个MessageListenerContainer只能注册一个MessageListener，因此当有多个Listener时，需要配置多个Container。

**013、JMX**

四大功能：

将任意spring bean自动注册为MBean；

为bean的管理接口提供灵活的管理机制；(窄化暴露的方法和属性)

声明式的服务暴露，支持远程和JSR160；

简化本地和远程MBean资源代理；

通过MBeanExporter将任意bean暴露为MBean，注意MBeanExporter不能延迟加载，即lazy-init必须为false。

**014、Task执行、计划**

两大接口：TaskExecutor和TaskScheduler，支持线程池，屏蔽了Java5、Java6和J2EE之间的差异。通过FactoryBean集成Quartz和JDK Timer。

SimpleAsyncTaskExecutor，没有线程池功能，每次都创建一个新线程；SyncTaskExecutor，仅用于测试；ConcurrentTaskExecutor，对java.util.concurrent.Executor的包装，一般也不使用，优先使用**ThreadPoolTaskExecutor**；SimpleThreadPoolTaskExecutor，是SimpleThreadPool的子类，使用场景，为Quartz和非Quartz组件提供线程池；TimerTaskExecutor，顾名思义，后台为TimerTask，对JDK Timer提供支持；WorkManagerTaskExecutor，对CommonJ提供支持。

注意：TaskExecutor是随着Spring容器关闭而关闭的，如果Spring容器关闭时前者没有一起关闭则有问题。

TaskScheduler在3.0后加入。它最重要的方法是：

ScheduledFuture schedule(Runnable task, Trigger trigger); Trigger接口本身非常简单：

Date nextExecutionTime(TriggerContext triggerContext);得到下一次执行时间即可。

TriggerContext记录了上一次执行的时间点信息，有最近计划执行时间，最近实际执行时间和最近执行完成时间。最有用的Trigger是Quartz提供的CronTrigger，可以参考其语法。

TaskScheduler最常用的实现是**ThreadPoolTaskScheduler**，它实现了TaskExecutor接口，同时代理了ScheduledExecutorService，因此它可以用来做异步的任务调度。

注解支持异步调用和任务调度：@Scheduled、@Async，在application中的命名空间配置为：<task:annotation-driven executor="x" scheduler="x"/>，executor和@Async对应，scheduler和@ Scheduled对应。

**特别注意**：@Scheduled标注的方法，不能带有返回值，也不能带有参数。只能将这二者作为该方法所在类的属性。@Async标注的方法1，被所在类的方法2调用时，即使方法2再被其他方法调用，也是没有异步效果的。

**015、动态语言支持**

Spring目前支持的动态语言有Groovy、Jruby和BeanShell。动态语言的优势在于：系统运行期间，修改代码后不用重启服务器或者重新部署。当然，启用这个特性需要设置一个参数：refresh-check-delay。需要注意的是，只有当一个新的方法调用到动态脚本时才会去检查是否更改。

动态语言的应用场景，官方文档上给出的是Controller和Validator。

**016、缓存(早期支持)**

Buffer vs Cache，Buffer是可见的，且是一次性的；Cache是透明的，可重复使用。

对于一个方法，如果给定输入，每次得到的输出都一样，则cache有用武之地。对于使用者来说，需要注意的有两点：1、声明Cache，哪些方法需要使用缓存，缓存策略是什么；2、配置Cache，即使用的是什么缓存组件，比如ehcache还是oscache，缓存存放在哪里，从哪里读取。

**@Cacheable**，标注在方法上，标明该方法的返回值可被缓存。可以指定多个缓存绑定名称，如@Cacheable({ "books", "isbns" })。缓存实际就是key-value对，因此，key的生成策略很重要，如果需要自己的生成策略，需要实现cache.KeyGenerator接口。这个策略是全局的，方法级的需要指定该注解的key参数，可以使用SpEL表达式，如@Cacheable(key=”#a”)，a是方法入参。如果只对某些情况使用缓存，可以指定condition参数，同样可以使用SpEL表达式，如@Cacheable(value="book", condition="#name.length < 32")，表示只有当name的长度小于32时才会将结果缓存，否则直接执行方法体。SpEL执行时有一个上下文，此处的上下文请参见spring的reference文档。

**@CachePut**，同@Cacheable一样，不过，它的作用在于更新缓存。即，每次都会执行方法，然后将结果存入缓存。

**@CacheEvict**，删除缓存，和@Cacheable一样，不过多了一个参数allEntries，可以用来删除所有。@CacheEvict标注的方法可以没有返回值。

注意：这三个注解不能混用。如果要在一个方法上多次标注同一个注解，可以使用下面的方法：@Caching(evict = { @CacheEvict("primary"), @CacheEvict(value = "secondary", key =

"#p0") })

配置：**<cache:annotation-driven />**。它有几个属性：cache-manager，默认ref到名为cacheManager的bean；mode，proxy和aspectj二选一，表示使用Spring AOP还是aspectj；proxy-target-class表示是基于接口(使用JDK代理)还是基于类。

建议：上面提到的三个注解，都只标注在实例类上，而不是接口上。另外，和@Async一样，被这几个注解标注的方法被同一个类中的其他方法调用是不会有缓存效果的，此时需要使用aspectj模式。

最后，这三个注解可以作为原注解被使用。

CacheManager，目前支持两种，JDK提供的ConcurrentMap(如ConcurrentHashMap)和EhCache。

**一些问题**

Controller级别的Converter：这个没有实现方法，倒是有该级别的ConversionService，但相比PropertyEditor太过麻烦。因此还是用PropertyEditor比较好。

JSR303验证：先注册一个名为validator的LocalValidatorFactoryBean，将其注入到一个名为webBindingInitializer的ConfigurableWebBindingInitializer，再将webBindingInitializer注入到HandlerAdapter。

Formatter：全局配置在ConversionService中，参见005。Controller级别的最好不用。字段级别的用注解，比如@DateTimeFormat、@NumberFormat等。在页面上，需要使用bind标签展示，例如<sp:bind path="user.birthday">${status.value}</sp:bind>。注意当没有user这个变量时会报错。

forward、redirect共享变量：forward时，目前只能使用HttpServletRequest，不考虑Session；redirect时，可以使用RedirectAttributes，对于对象类型，添加到FlashAttribute，对于字符串变量，添加到Attribute，该变量会附加到URL参数中。

静态资源处理：spring提供了<mvc:resoures/>配置来定位静态资源，但该方法的执行效率肯定不及Web容器自身。因此还是将静态资源放在WebContent目录下，动态资源才放在WebContent/Web-INF下，此时需要开启<mvc:default-servlet-handler />来处理。

StringHttpMessageConverter的默认编码为ISO-8859-1，比较恶心。因此直接修改了它的源文件，改为UTF-8。

**Profile和Environment**

这两个东西是为了解决在不同环境下相同作用的bean的不同配置而产生的，如在开发和生产环境下不同的dataSource。Profile用来标记不同环境，Environment用来指定Profile。

在XML配置下，顶层的beans元素下可以再配置beans元素，通过profile attribute来区分。示例如下：

<beans>

...

<beans profile=”dev”>

<bean ...>

<bean ...>

</beans>

</beans>

在编程环境下，可以通过ctx.getEnvironment().setActiveProfiles("profile")来指定profile。在Web环境下，可以通过系统环境变量、JVM系统属性、Servlet的context param等来设置spring.profiles.active，从而指定profile。如通过JVM来设置：-Dspring.profiles.active=”...”

**spring4.0 reference**

**校验、数据绑定、类型转换**

Validator接口用来校验，Errors接口用来记录校验错误结果在MessageSource中对应的代码。最后，从MessageSource中获取校验结果。

Spring在填充Bean时，先使用BeanWrapper对Bean包装，然后调用PropertyEditor来设置属性。PropertyEditor在String和Object之间进行类型转换。

Spring3开始，Spring的Converter接口登场，它提供任意两个不同类型之间的转换，远比PropertyEditor更方便。编写自己的Converter时，需要注意它必须是线程安全的。Spring使用ConversionService接口来管理Converter，然后，使用ConversionServiceFactory工厂来管理ConversionService。

Spring使用Converter SPI来完成Bean的属性绑定，也通过它来完成SpEL和DataBinder的字段绑定。

对于[反]格式化，Spring提供Formatter接口，该接口提供从类型T到String的格式化及反向解析。问题：格式化之后，放到哪里，再向页面输出？如果不使用spring页面标签，能否将该格式化与JSON结合？

在Web中，有了Validator还不够，还需要DataBinder来调用Validator进行校验。Controller级别的DataBinder可以使用@InitBinder注解来设置，如下：

@InitBinder

protected void initBinder(WebDataBinder binder){

binder.addValidators(new FooValidator());

}

主要作用是向DataBinder中设置一个Validator。

全局的Validator在<mvc:annotation-driven>中设置。

**测试**

Spring提供了自己的测试框架。

首先，它提供了大量的Mock对象，覆盖了Environment到JNDI到Servlet等。其次，它提供了一些通用工具，比如用于设置private属性的ReflectionTestUtils等。再次，在MVC上，它提供了ModelAndViewAssert断言，用来处理ModelAndView对象。

在集成测试方面，@ContextConfiguration注解将实例化ApplicationContext大大简化，注解@WebAppConfiguration与之结合使用，可加载Web环境下的配置(如MVC)。

为了提高效率，Spring将ApplicationContext、MockServletContext等缓存，key使用最多的是配置文件的location。

**MVC**

异步处理：方法返回值为Callable<?>，方法体使用回调。异常处理跟普通方法一样。

Redirect跳转方式，方法返回值为String，格式为redirect:url

Theme(主题)设置，方式跟国际化类似，将css路径写在资源文件里。然后通过ThemeResolver来获取，改变Theme通过ThemeChangeInterceptor来设置，参数可以放在request、session或者cookie中。

**WebSocket**

全双工，服务器--客户端双向连接。工作过程：使用“升级”后的HTTP协议建立握手，保持两端的TCP Socket开启，使用该Socket通信。

并非所有的浏览器都支持WebSocket，当遇到不支持的浏览器时，Spring会自动退回到SockJS协议。

WebSocket提供了一个子协议用来规定通信格式，Spring使用的是STOMP。

当容器是Tomcat7和GlassFish4以上时，Spring使用的是JSR356；当是Jetty9以上时，使用容器原生的WebSocket引擎。

创建WebSocket Server，只需要实现WebSocketHandler接口，或者继承TextWebSocketHandler/BinaryWebSocketHandler类(这两个类都实现了该接口)。

配置如下：

<websocket:handlers>

<websocket:mapping path="/myHandler" handler="myHandler"/>

</websocket:handlers>

<bean id="myHandler" class="org.springframework.samples.MyHandler"/>

**JMS**

生产者和消费者。

JmsTemplate用于生产消息和同步接收消息，异步接收使用的是Message-Driven风格的监听器(Listener)机制。JmsTemplate提供大量回调方法，屏蔽资源创建和释放过程，对外暴露JMS Session和Message Producer。

JMS抛出的checked异常都被转换为unchecked异常。

MessageConverter提供Java Object和JMS Message之间的转换。

destination包提供各种Destination管理策略，比如JNDI方式的Service Locator等。

ConnectionFactory用于非EE环境，JmsTransactionManager将JMS作为事务性资源无缝集成到Spring中。

JmsTemplate提供了两类发送消息的方法，带QOS和不带QOS的。QOS指标包括优先级、存活时间、超时等等。

JmsTemplate是线程安全的，但它是有状态的，它持有对ConnectionFactory的引用。ConnectionFactory是JMS工作的入口点，它是JMS Connection的创建工厂，同时提供大量特性选项， 比如SSL等。

发送一条消息的流程为ConnectionFactory --- Connection --- Session --- MessageProducer --- send，在这个过程中，有三个临时对象，发送完后会被销毁。为了减少资源开销，Spring提供了两种可复用的ConnectionFactory。SingleConnectionFactory每次都返回同一个Connection，忽略Connection的close方法，因为每次都是同一个connection，可参与任意事务。CachingConnectionFactory继承了SingleConnectionFactory的功能特性，同时对Session、MessageProducer、MessageConsumer提供了缓存。

当有大量的Destination时，使用JNDI方式非常笨重。此时，Destination的创建、查找被代理给DestinationResolver处理，DynamicDestinationResolver是默认实现，它用于创建动态的Destination，JndiDestinationResolver是更高级的实现，它可以查找JNDI形式的Destination，查找失败时退回到DynamicDestinationResolver。

Message Listener Container用于接收队列消息并驱动注入它当中的MessageListener。它是Message Producer和MDP(Message-Driven POJO，即MessageListener)之间的中介者，提供注册收取消息、参与事务、资源的获取与释放、异常转换等功能。

SimpleMessageListenerContainer不适用于EE环境，DefaultMessageListenerContainer支持运行时动态适配(Destination?)、能参与外部事务(XA)。

MDP的实现：实现MessageListener接口，保证线程安全(因为要从多个线程中接收消息)。然后配置Message Listener Container的Spring Bean。

事务处理：将MessageListenerContainer的sessionTransacted属性设为true。

JMS命名空间：<jms:listener>和<jms:listener-container>，其中listener-container支持Thread Pool，可配置TaskExecutor。

**任务调度**

任务使用TaskExecutor接口，调度使用TaskScheduler接口。具体实现根据JDK和EE环境的不同而不同。提供对Quartz的集成。

TaskExecutor和java.util.concurrent.Executor是等同的，之所以另起一个，是为了屏蔽对JDK版本的依赖。

Spring提供的TaskExecutor有：SimpleAsyncTaskExecutor，每个任务都新开一个线程，可以配置最大线程数；SyncTaskExecutor，同步单线程，仅用于测试；ConcurrentTaskExecutor是java.util.concurrent.Executor的包装，不推荐使用；SimpleThreadPoolTaskExecutor是Quartz的SimpleThreadPool的子类，用于Quartz和非Quartz组件之间的共享；ThreadPoolTaskExecutor是Java5以上最常用的Executor，它包装了ThreadPoolExecutor。

TaskScheduler接口提供的方法大部分都比较简单，统一返回ScheduledFuture类型。唯一复杂的是Trigger入参。Trigger接口本身也比较简单，只有一个方法，nextExecutionTime，返回下一次执行的时间，该方法的入参为TriggerContext，将所有的上下文信息都包含在内。

Spring提供的Scheduler有ThreadPoolTaskScheduler。

基于注解的Scheduler和异步调用：使用<task:annotation-driven>标签开启注解功能，注意：TaskExecutor和TaskScheduler还是需要配置的。@Scheduled和@Async注解都用在方法上，@Scheduled注解的方法必须没有入参，返回值为Void；

page705