# Spring

应用上下文即是Spring容器抽象的一种实现，对象整个的生命周期的管理——创建、装配、销毁

① AnnotationConfigApplicationContext:从一个或多个基于java的配置类中加载上下文定义，适用于java注解的方式；

② ClassPathXmlApplicationContext:从类路径下的一个或多个xml配置文件中加载上下文定义，适用于xml配置的方式；

③ FileSystemXmlApplicationContext:从文件系统下的一个或多个xml配置文件中加载上下文定义，也就是说系统盘符中加载xml配置文件；

④ AnnotationConfigWebApplicationContext:专门为web应用准备的，适用于注解方式；

⑤ XmlWebApplicationContext:从web应用下的一个或多个xml配置文件加载上下文定义，适用于xml配置方式。

ApplicationContext context = new AnnotationConfigApplicationContext(ManConfig.class);

//从容器中获取对象实例

Man man = context.getBean(Man.class);

### 依赖注入

1. 自动装配

实现自动装配需要两个动作

1. 声明bean:

在需要注入的类中采用@Component,@Serivice,@Controller等声明为组件类

1. 组件扫描

需要使用@ComponentScan扫描，它默认会扫描相同包以及子包查找带有@Component的类

c.自动注入

Spring会自动根据依赖关系注入bean

1. public interface Disc{//光碟
2. public void play();
3. }
5. @Component
6. public Peppers implements disc{
8. @override
9. public void play(){
10. system.out.pritIn("play music");
11. }
12. }
14. @Configuration
15. @ComponentScan
16. public class config{
18. }
20. public class Test{
22. @Autowired
23. public Disc d;
24. public static void main(String args[]){
25. d.play();
26. }
27. }

默认bean的ID为类名首字母小写，例如Peppers 的id为peppers。如果需要别的id的话，需要@Component(“otherName”)或者@Named(“otherName”)

设置扫描基础包

@ComponentScan默认会扫描自己所在的包以及子包，如果要扫描别的包，可以采用如下2种方式  
@ComponentScan(basePackages = "{"com.my.package.first","com.my.package.second"}")

@ComponentScan(basePackageClasses = "{"Class1","Class2"}")

第二种方法是扫描class1和class2所在包以及子包

自动装配通过注解@Autowired来完成

1. @Component
2. public CDPlayer implements Player{
3. Private Peppers pepper1;
4. Private Peppers pepper2;
5. @Autowired
6. public CDPlayer(Peppers pepper){
7. This.pepper1 = pepper;//通过构造函数注入
8. }
9. @Autowired
10. Public void SetPepper(Peppers pepper)  {
11. This.pepper1 = pepper;//通过setter函数注入
12. }

}

其实@Autowired可以在任何方法进行自动注入。@Autowired是Spring自动装配，@Inject是Java的自动装配，在Spring项目中两者没有大的区别。

1. Java装配  
   JavaConfig和xml文件进行装配都是属于显示配置，通常使用@Configuration注解实现java代码装配

要在JavaConfig中声明bean，采用注解@Bean

1. @Configuration
2. **public** **class** config{
3. @Bean
4. **public** Peppers peppers(){
5. **return** **new** Pepers;
6. }

}

@Bean注解会告诉Spring这个方法返回的对象要注册为Spring应用的上下文bean，该bean的id和带有@Bean的方法名是一样的。也可以通过name属性指定一个不同的名字

1. @Configuration
2. **public** **class** config{
3. @Bean(name="oterhName")
4. **public** Peppers peppers(){
5. **return** **new** Pepers;
6. }
7. }
8. Xml文件装配

声明一个简单的<bean>

<bean id=”student” **class**=”syst.student”>

构造器注入bean

1. <bean id=”cdPlayer” **class**=”sys.CDPlayer”>
2. <construct-arg ref=”compactDisc”>
3. </bean>

构造器注入字符串

1. <bean id=”cdPlayer” **class**=”sys.CDPlayer”>
2. <construct-arg value=”hello1”>
3. <construct-arg value=”hello2”>
4. </bean>

如果注入有包含数组的话

1. <bean id=”cdPlayer” **class**=”sys.CDPlayer”>
2. <list>
3. <value>hello1</value>
4. <value>hello2</value>
5. </list>
6. </bean>

如果数组是个对象数组的话

1. <bean id=”cdPlayer” **class**=”sys.CDPlayer”>
2. <list>
3. <ref bean=”student1”/>
4. <ref bean=”student2”/>
5. <ref bean=”student3”/>
6. </list>
7. </bean>

上面都是属于构造时注入，下面介绍利用属性注入bean

其中name的值即为CDPlayer声明的对象成员。

1. <bean id=”cdPlayer” **class**=”sys.CDPlayer”>
2. <property name=”compactDisc” ref=”compactDisc”>
3. </bean>

不同JavaConfig或xml集成导入

如果有个JavaConfig，即多个@Configuration标识的类相互之间需要引用，那么就要用到@Import。假设CompactDisc的bean在CDConfig.class类声明，那么

1. @Configuration
2. @Import(CDConfig.**class**)
3. Public **class** CDPlayerConfig{
4. @bean
5. Public CDPlayer cdPlayer(CompactDisc compactDisc){
6. Return **new** CDPlayer(compactDisc);
7. }
8. }

Spring在确定哪个profile处于激活状态时，需要依赖两个独立的属性：

Spring.profiles.active和spring.profiles.default

有条件配置bean

1. @Bean
2. @Conditional(MagicExistsCondition.**class**)
3. Public MagicBean magicBean(){
4. Return **new** MagicBean();
5. }

magicBean是否会被创建取决于MagicExistsCondition.class，其中MagicExistsCondition必须要实现Condition接口，实现matches()方法，@Conditional会根据matches()方法返回ture还是false决定是否创建magicBean

1. Public **class** MagicExistsCondition **implements** Condition{
2. @Override
3. Public **boolean** matches(ConditionContext context,AnnotatedTypeMetadata metadata){
4. Enviroment env = context.getEnvironment();
5. Return env.containsProperty(“magic”);//检查magic属性
6. }
7. }

限定自动装配

当要注入一个接口，但该接口有多个实现时就会发生歧义报错，例如

1. @Autowired
2. Public **void** setDessert(Dessert dessert){
3. This.dessert = dessert;
4. }

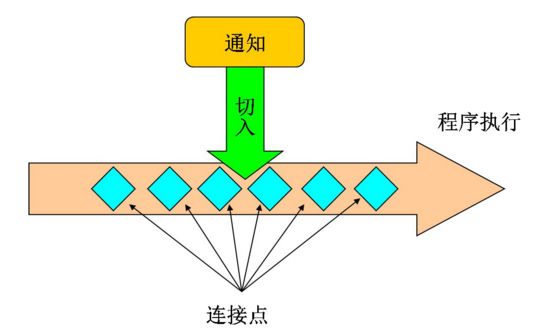
Dessert有三个实现类

1. @Component
2. Public **class** Cake **implements** Dessert {}
3. @Componet
4. Public **class** Cookies implemnets Dessert {}
5. @Componet
6. Public **class** IceCream implemnets Dessert {}

通过@Autowired注入时发现Dessert有三个可注入的bean时将无法进行，此时需要@Qualifier注解指定需要注入哪个bean

1. @Autowired
2. @Qualifier(“cake”)
3. Public **void** setDessert(Dessert dessert){
4. This.dessert = dessert;
5. }

## AOP



术语

通知：即工作的实际内容，可分为5种类型

* 前置通知
* 后置通知
* 返回通知
* 异常通知
* 环绕通知

连接点：能够插入切面的点，可以简单理解就是类的方法。

切点：连接点有多个，切点即定义哪几个点要织入通知。

切面：通知和切点的结合，定义了何时何处完成功能

织入：把切面应用到目标对象并创建新的代理对象的过程。切面在指定的连接点被织入到目标对象。

AOP现在主要有两种：Sring AOP和AspectJ

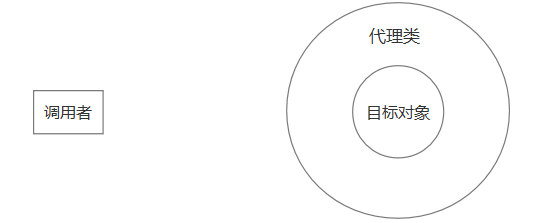
Spring提供4种类型的AOP支持

基于代理的经典AOP

纯POJO面

@AspectJ注解驱动的切面

注入式AspectJ切面



Spring的切面由包裹了目标对象的代理类实现。代理类处理方法的调用，执行额外的切面逻辑并调用目标方法。

编写切点

1. **package** connect;
2. **public** **interface** Performance{
3. **public** **void** perform();
4. }
5. execution(\* connect.performance.perform(..))
6. 使用AspectJ切点表达式来选择Performance的perform()方法
7. execution(\* connect.performance.perform(..)) && within(connect.\*)
8. 使用within()指示器限制切点范围
9. executeion(\* connect.performance.perform(..)) && bean('woodstock')
10. bean()使用bean ID或bean名称作为参数来限制切点只匹配特定的bean

定义切面

Spring使用AspectJ注解声明通知方法

|  |  |
| --- | --- |
| 注解 | 通知 |
| @After | 通知方法会在目标方法返回或抛出异常调用 |
| @AfterReturning | 通知方法会在目标方法返回后调用 |
| @AfterThrowing | 通知方法会在目标方法刨出异常后调用 |
| @Around | 通知方法会将目标方法封装起来 |
| @Before | 通知方法会在目标方法调用之前执行 |

1. @Aspect
2. **public** **class** Audience{
3. @Before("execution(\* connect.performance.perform(..))")
4. **public** **void** silenceCellPhones(){ ...}//执行perform()前触发切点
5. @After("execution(\* connect.performance.perform(..))")
6. **public** **void** applause() { ...}//执行perform()后触发切点
7. }

上面不同的方法都设置了切点，但切点的表达式都是一致的，采用@Pointcut就可以不必每次都编写同样的表达式。

1. @Aspect
2. **public** **class** Audience{
3. @Pointcut("execution(\* connect.performance.perform(..))")//定义切点
4. **public** **void** performance() {}
6. @Before("performance()")
7. **public** **void** silenceCellPhones(){ ...}//执行perform()前触发切点
8. @After("performance()")
9. **public** **void** applause() { ...}//执行perform()后触发切点
10. }

要使用aop的话需要在配置类使用@EnableAspectJAutoProxy

创建环绕通知

环绕通知是最为强大的通知，它能够将被通知的目标方法完全包装起来。实际上就像在一个通知方法同时编写前置通知和后置通知，环绕通知需传递一个参数ProceedingJoinPoint 。

1. @Aspect
2. **public** **class** Audience{
3. @Pointcut("execution(\* connect.performance.perform(..))")//定义切点
4. **public** **void** performance() {}
5. @Around("performance()")
6. **public** **void** watchPerformance(ProceedingJoinPoint jp){
7. **try**{
8. system.out.printLn("前置通知");
9. jp.proceed();
10. system.out.printLn("后置通知");
11. }**catch**(Exception e){
12. system.out.printLn("异常通知");
13. }
14. }
15. }

处理通知中的参数:

当需要获取被通知方法的入参时通知方法可通过下面方法获取参数到。Play(int)是被通知的方法，借由args(count)传到通知方法diskCounter(int count)

1. @Aspect
2. **public** **class** CdCounter {
3. **private** Map<Integer,Integer> counterMap = **new** HashMap<>();
4. @Pointcut("execution(\* Cd.Disk.play(int)) && args(count)")//定义切点
5. **public** **void** diskPlay(**int** count){ }//通知参数
6. @After("diskPlay(count)")
7. **public** **void** diskCounter(**int** count){//通知
8. **if**(counterMap.containsKey(count)){
9. counterMap.put(count,counterMap.get(count) + 1);
10. }**else**{
11. counterMap.put(count,1);
12. }
13. }
14. **public** **int** playCount(**int** i){
15. **return** counterMap.getOrDefault(i,0);
16. }
17. }

通过注解引入新功能:

简单来说就是不修改类的定义，利用AOP为该类的对象增加一个新方法或功能。

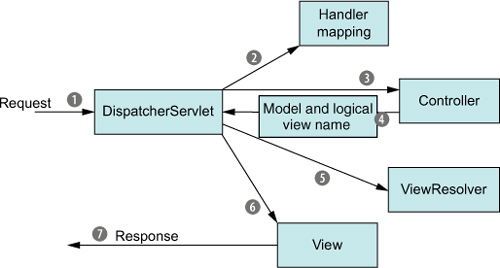
XML文件AOP

|  |  |
| --- | --- |
| <aop:advisor> |  |
| <aop:after> | 不管通知是否执行成功 |
| <aop:after-returnning> |  |
| <aop:after-throwing> |  |
| <aop:around> |  |
| <aop:aspect> |  |
| <aop:aspectj-autoproxy> | 启用@AspectJ注解驱动的切面 |
| <aop:before> |  |
| <aop:config> |  |
| <aop:declare-parents> |  |
| <aop:pointcut> |  |
|  |  |

1. <aop:config>
2. <aop:aspect ref="cdCounter">
3. <aop:before
4. pointcut = "execution(\* Cd.Disk.play(..))"
5. method = "diskCounter"/>
6. </aop:aspect>
7. </aop:config>

## Spring MVC

DispatcherServlet是Spring MVC的核心，负责接收前端的请求。通常采用web.xml配置，从servlet3.0和spring



3.1开始可以用Java配置

1. **import** org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
2. **public** **class** SpittrWebAppInitializer **extends** AbstractAnnotationConfigDispatcherServletInitializer {
3. @Override
4. **protected** String[] getServletMappings(){
5. **return** **new** String[] { "/" };
6. }
7. @Override
8. **protected**  Class<?>[] getRootConfigClasses(){
9. **return** **new** Class<?>[] {};
10. }
11. @Override
12. **protected**  Class<?>[] getServletConfigClasses(){
13. **return** **new** Class<?>[] {WebConfig.class};
14. }
15. }

实现AbstractAnnotationConfigDispatcherServletInitializer 接口都会自动配置DispatcherServlet和Spring应用上下文。Spring web通常有两个上下文。其中Spring上下文由DispatcherServlet创建，另外一个上下文由ContextLoaderListener创建。上面getServletConfigClasses()返回的带有@Configuration注解类用来定义DispatcherServlet应用上下文的bean。getRootConfigClasses()返回的带有@Configuration注解类用来定义ContextLoaderListener应用上下文的bean。

启动Spring MVC:

1. @Configuration
2. @EnableWebMvc
3. @ComponentScan("spitter.web")//@Controller类所在的包
4. **public** **class** WebConfig
5. **extends** WebMvcConfigurerAdapter {
7. @Bean
8. **public** ViewResolver viewResolver() {  //视图解析器
9. InternalResourceViewResolver resolver =
10. **new** InternalResourceViewResolver();
11. resolver.setPrefix("/WEB-INF/views/");
12. resolver.setSuffix(".jsp");
13. resolver.setExposeContextBeansAsAttributes(**true**);
14. **return** resolver;
15. }
17. @Override
18. **public** **void** configureDefaultServletHandling(
19. DefaultServletHandlerConfigurer configurer) {  //配置静态资源的处理
20. configurer.enable();
21. }
23. }

RootConfig配置类为：

1. @Configuration
2. @ComponentScan(basePackages={"spitter"},
3. excludeFilters={
4. @Filter(type=FilterType.ANNOTATION, value=EnableWebMvc.**class**)
5. })
6. **public** **class** RootConfig {
7. }

编写基本的控制器

1. @Controller
2. **public** **class** HomeController {
3. @RequestMapping(value="/", method=GET)
4. **public** String home() {
5. **return** "home";
6. }
7. }

鉴于我们配置InternalResourceViewResolver 的方式，视图名“home”将会解析为”/WEB-INF/views/home.jsp”

如果控制器需要传递数据给视图的话需要传入一个参数Model，它本质是一个map,如果没有指定key的话会根据值的对象类型确定。入下面的key为spittleList

1. @Controller
2. @RequestMapping("/spittles")
3. **public** **class** SpittleController {
4. @RequestMapping(method=RequestMethod.GET)
5. **public** String spittles(Model model) {
6. model.addAttribute(
7. spittleRepository.findSpittles(
8. Long.MAX\_VALUE, 20));
9. **return** "spittles";
10. }

当然也可以指定可以key

1. @Controller
2. @RequestMapping("/spittles")
3. **public** **class** SpittleController {
4. @RequestMapping(method=RequestMethod.GET)
5. **public** String spittles(Model model) {
6. model.put( "spittleList"
7. spittleRepository.findSpittles(
8. Long.MAX\_VALUE, 20));
9. **return** "spittles";
10. }
11. <?xml version="1.0" encoding="UTF-8"?>
12. <web-app version="2.5"
13. xmlns="http://java.sun.com/xml/ns/javaee"
14. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
15. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
16. http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">
18. <context-param>
19. <param-name>contextConfigLocation</param-name>
20. <param-value>/WEB-INF/spring/root-context.xml</param-value>
21. </context-param>
23. <listener>
24. <listener-**class**>
25. org.springframework.web.context.ContextLoaderListener
26. </listener-**class**>
27. </listener>
29. <servlet>
30. <servlet-name>appServlet</servlet-name>
31. <servlet-**class**>
32. org.springframework.web.servlet.DispatcherServlet
33. </servlet-**class**>
34. <load-on-startup>1</load-on-startup>
35. </servlet>
37. <servlet-mapping>
38. <servlet-name>appServlet</servlet-name>
39. <url-pattern>/</url-pattern>
40. </servlet-mapping>
42. </web-app>

## Spring DataBase

配置数据源

无论选择Spring的那种方式访问数据库，都需要配置数据源，包括：

* JDBC
* JNDI
* 连接池

用于生产环境的数据源通常采用连接池方式，常见的开源连接池有

* Apache Commons DBCP (http://jakarta.apache.org/commons/dbcp)
* c3p0 (http://sourceforge.net/projects/c3p0/)
* BoneCP (http://jolbox.com/)
* Druid(阿里的连接池)

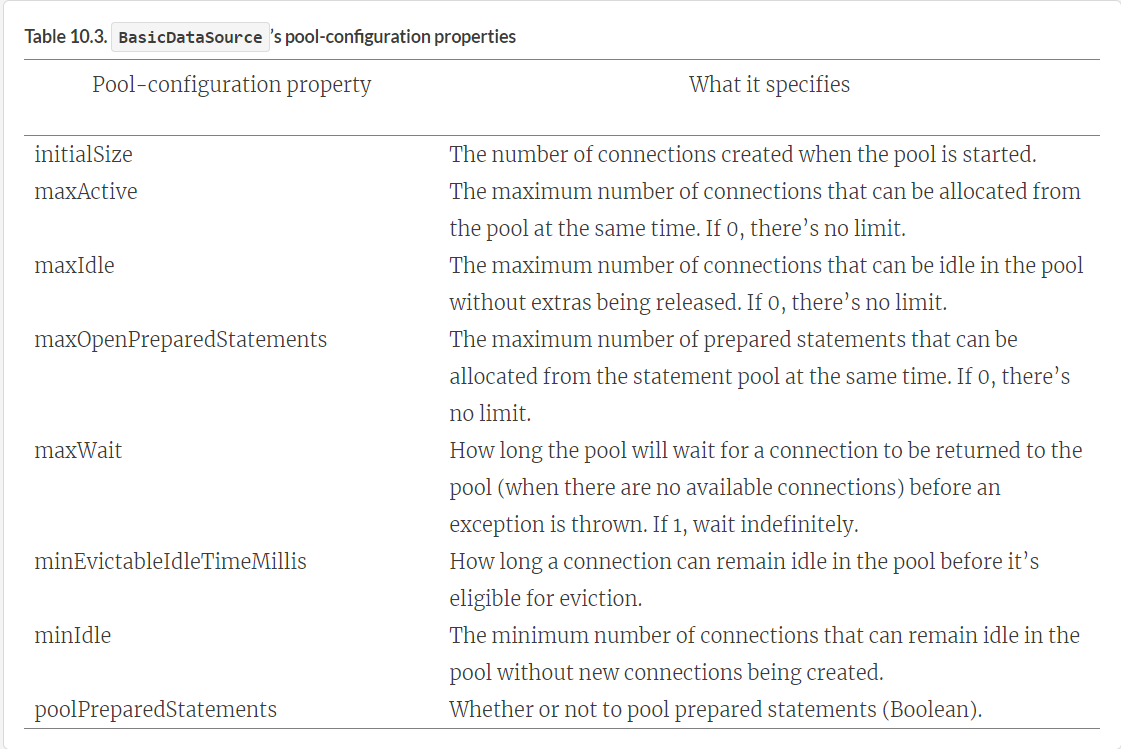
如下采用的是DBCP的方式:

1. <bean id="dataSource" **class**="org.apache.commons.dbcp.BasicDataSource"
2. p:driverClassName="org.h2.Driver"
3. p:url="jdbc:h2:tcp://localhost/~/spitter"
4. p:username="sa"
5. p:password=""
6. p:initialSize="5"
7. p:maxActive="10" />

JAVA配置

1. @Bean
2. **public** BasicDataSource dataSource() {
3. BasicDataSource ds = **new** BasicDataSource();
4. ds.setDriverClassName("org.h2.Driver");
5. ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
6. ds.setUsername("sa");
7. ds.setPassword("");
8. ds.setInitialSize(5);
9. ds.setMaxActive(10);
10. **return** ds;
11. }

常见的连接池配置参数如下表



JDBC方式

Spring提供三个数据源类

DriverManagerDataSource 每次请求都新建一个连接，消耗性能

SimpleDriverDataSource 有且只有一个连接，不适合生产环境，不合适多线程

SingleConnectionDataSource 每次请求返回同一个连接

1. @Bean
2. **public** DataSource dataSource() {
3. DriverManagerDataSource ds = **new** DriverManagerDataSource();
4. ds.setDriverClassName("org.h2.Driver");
5. ds.setUrl("jdbc:h2:tcp://localhost/~/spitter");
6. ds.setUsername("sa");
7. ds.setPassword("");
8. **return** ds;
9. }

Spring Data JPA

开启Srping Data JPA，Java配置

1. @Configuration
2. @EnableJpaRepositories(basePackages="com.habuma.spittr.db")
3. **public** **class** JpaConfiguration {
4. ...
5. }

xml文件配置

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xmlns:jpa="http://www.springframework.org/schema/data/jpa"
5. xsi:schemaLocation="http://www.springframework.org/schema/data/jpa
6. http://www.springframework.org/schema/data/jpa/spring-jpa-1.0.xsd">
8. <jpa:repositories base-package="com.habuma.spittr.db" />
10. ...
12. </beans>

@EnableJpaRepositories也要指定一个扫描包，它会扫到一个扩展Repository的接口并自动生成该接口的实现类。

1. @Configuration
2. @EnableJpaRepositories(basePackages="com.habuma.spittr.db")
3. **public** **class** JpaConfiguration {
4. ...
5. }

## Redis

Spring Data Ridis提供了四种客户端连接redis

* JedisConnectionFactory
* JredisConnectionFactory
* LettuceConnectionFactory
* SrpConnectionFactory

1. @Bean
2. **public** RedisConnectionFactory redisCF() {
3. JedisConnectionFactory cf = **new** JedisConnectionFactory();
4. cf.setHostName("redis-server");
5. cf.setPort(7379);
6. cf.setPassword("foobared");
7. **return** cf;
8. }

使用RedisTemplate读写redis,主要有两种template

RedisTemplate

StringRedisTemplate如果key和value都是String的话推荐使用这个

1. @Bean
2. **public** RedisTemplate<String, Product>
3. redisTemplate(RedisConnectionFactory cf) {
4. RedisTemplate<String, Product> redis =
5. **new** RedisTemplate<String, Product>();
6. redis.setConnectionFactory(cf);
7. **return** redis;
8. }
9. @Bean
10. **public** StringRedisTemplate
11. stringRedisTemplate(RedisConnectionFactory cf) {
12. **return** **new** StringRedisTemplate(cf);
13. }

RedisTemplate大多数操作都是基于下表的子API提供：



使用简单的值：

redis.opsForValue().set(product.getSku(), product);

Product product = redis.opsForValue().get("123456");

使用List类型的值：

在key为cart的list尾部添加一个值

redis.opsForList().rightPush("cart", product);

在key为cart的list头部添加一个值

redis.opsForList().leftPush("cart", product);

在key为cart的list头部取一个值

Product first = redis.opsForList().leftPop("cart");

在key为cart的list尾部取一个值

Product last = redis.opsForList().rightPop("cart");

去除下标2到12的值

List<Product> products = redis.opsForList().range("cart", 2, 12);

在Set上执行操作

redis.opsForSet().add("cart", product);

在多个set进行取其差异，交集和并集

List<Product> diff = redis.opsForSet().difference("cart1", "cart2");

List<Product> union = redis.opsForSet().union("cart1", "cart2");

List<Product> isect = redis.opsForSet().isect("cart1", "cart2");

redis.opsForSet().remove(product);

序列化器

* GenericToStringSerializer —使用Spring转换服务进行序列化
* JacksonJsonRedisSerializer —使用Jackson1进行序列化
* Jackson2JsonRedisSerializer —使用Jackson2进行序列化
* JdkSerializationRedisSerializer —Ovaa Icoc taliioirnezsa使用java序列化
* OxmSerializer —StringRedisSerializer —Slisazieer String hcko and svleua
* StringRedisSerializer String类型的序列化

1. @Bean
2. **public** RedisTemplate<String, Product>
3. redisTemplate(RedisConnectionFactory cf) {
4. RedisTemplate<String, Product> redis =
5. **new** RedisTemplate<String, Product>();
6. redis.setConnectionFactory(cf);
7. redis.setKeySerializer(**new** StringRedisSerializer());
8. redis.setValueSerializer(
9. **new** Jackson2JsonRedisSerializer<Product>(Product.**class**));
10. **return** redis;
11. }

### 缓存

使用缓存前添加@EnableCaching注解

1. @Configuration
2. @EnableCaching
3. **public** **class** CachingConfig {
5. @Bean
6. **public** CacheManager cacheManager() {
7. **return** **new** ConcurrentMapCacheManager();
8. }
10. }

缓存管理器是缓存的核心可以和许多流行缓存进行集成

Spring内置5个缓存管理器

* SimpleCacheManager
* NoOpCacheManager
* ConcurrentMapCacheManager
* CompositeCacheManager
* EhCacheCacheManager

Spring data提供了两个缓存管理器

* RedisCacheManager
* GemfireCacheManager

### 使用Ehcache缓存

1. @Configuration
2. @EnableCaching
3. **public** **class** CachingConfig {
4. @Bean
5. **public** EhCacheCacheManager cacheManager(CacheManager cm) {
6. **return** **new** EhCacheCacheManager(cm);
7. }
8. @Bean
9. **public** EhCacheManagerFactoryBean ehcache() {
10. EhCacheManagerFactoryBean ehCacheFactoryBean =
11. **new** EhCacheManagerFactoryBean();
12. ehCacheFactoryBean.setConfigLocation(
13. **new** ClassPathResource("com/habuma/spittr/cache/ehcache.xml"));
14. **return** ehCacheFactoryBean;
15. }
16. }

红色标注的是escache的cacheManager,需要被注入到spring的EhCacheCacheManager。值得注意的是红色CacheManager是要备注入的，也就是说它需要被声明成bean，然而实际被声明为bean的是EhCacheManagerFactoryBean ，由于EhCacheManagerFactoryBean 实现了Spring的FactoryBean接口，所以它实际返回的是CacheManager实例。

Ehcache.xml内容如下

<ehcache>

<cache name="spittleCache"

maxBytesLocalHeap="50m"

timeToLiveSeconds="100">

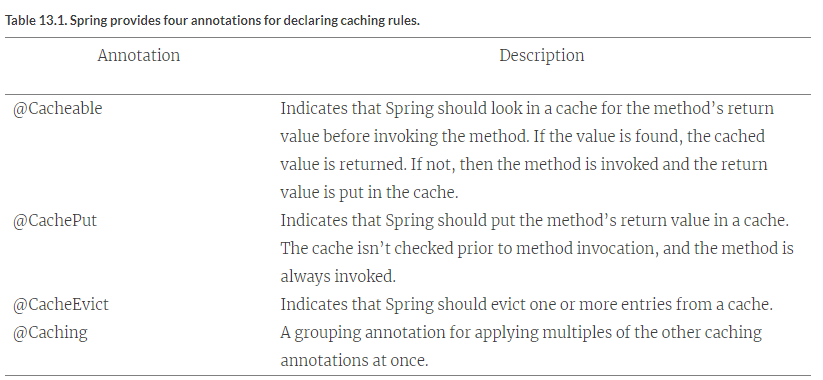
</cache>

</ehcache>

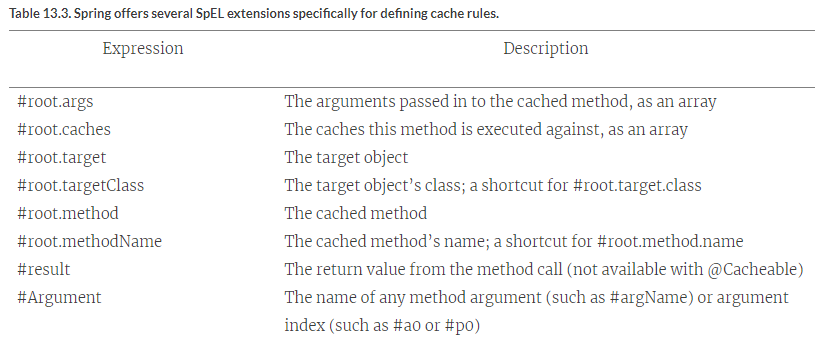
### Redis缓存

1. @Configuration
2. @EnableCaching
3. **public** **class** CachingConfig {
4. @Bean
5. **public** CacheManager cacheManager(RedisTemplate redisTemplate) {
6. **return** **new** RedisCacheManager(redisTemplate);
7. }
8. @Bean
9. **public** JedisConnectionFactory redisConnectionFactory() {
10. JedisConnectionFactory jedisConnectionFactory =
11. **new** JedisConnectionFactory();
12. jedisConnectionFactory.afterPropertiesSet();
13. **return** jedisConnectionFactory;
14. }
15. @Bean
16. **public** RedisTemplate<String, String> redisTemplate(
17. RedisConnectionFactory redisCF) {
18. RedisTemplate<String, String> redisTemplate =
19. **new** RedisTemplate<String, String>();
20. redisTemplate.setConnectionFactory(redisCF);
21. redisTemplate.afterPropertiesSet();
22. **return** redisTemplate;
23. }
25. }

方法缓存



1. @Cacheable("spittleCache")
2. **public** Spittle findOne(**long** id) {
3. **try** {
4. **return** jdbcTemplate.queryForObject(
5. SELECT\_SPITTLE\_BY\_ID,
6. **new** SpittleRowMapper(),
7. id);
8. } **catch** (EmptyResultDataAccessException e) {
9. **return** **null**;
10. }
11. }



@CachePut(value="spittleCache", key="#result.id")

Spittle save(Spittle spittle);

### AMQP



AMQP的生产者不会直接把消息发不到队列而是发不到Exchange，Exchange再根据不同的模式发不到队列中，有四种模式：

Direct:如果消息的routing key和binding key的routing key直接匹配的话消息将会路由到该队列。

Topic:如果消息的routint key和binding key的routing key符合通配符匹配的话消息将会路由到该队列中。

Headers:如果消息参数表中的头信息和值都与binding参数表中相匹配的话，消息会路由到该队列中。

Fanout:不管消息的routing key和参数表的头信息/值是什么，消息都会路由到所有的队列中。

## Spring boot

入口类

@SpringBootApplication

## Spring boot + redis

1. pom文件引入

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

1. 修改项目启动类，增加注解@EnableCaching，开启缓存功能，如下：

@SpringBootApplication

@EnableCaching

public class SpringbootApplication

1. application.properties配置

# Redis数据库索引（默认为0）

spring.redis.database=0

# Redis服务器地址

spring.redis.host=172.31.19.222

# Redis服务器连接端口

spring.redis.port=6379

# Redis服务器连接密码（默认为空）

spring.redis.password=

# 连接池最大连接数（使用负值表示没有限制）

spring.redis.pool.max-active=8

# 连接池最大阻塞等待时间（使用负值表示没有限制）

spring.redis.pool.max-wait=-1

# 连接池中的最大空闲连接

spring.redis.pool.max-idle=8

# 连接池中的最小空闲连接

spring.redis.pool.min-idle=0

# 连接超时时间（毫秒）

spring.redis.timeout=0

1. RedisConfig

因为springboot默认注入的redisTemplate不够强大，需自定义自己的RedisTemplate

1. @Configuration
2. **public** **class** RedisConfig{
3. //缓存管理器
4. @Bean
5. **public** CacheManager cacheManager(@SuppressWarnings("rawtypes") RedisTemplate redisTemplate) {
6. RedisCacheManager cacheManager = **new** RedisCacheManager(redisTemplate);
7. //设置缓存过期时间
8. cacheManager.setDefaultExpiration(10000);
9. **return** cacheManager;
10. }
11. @Bean
12. **public** RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory){
13. StringRedisTemplate template = **new** StringRedisTemplate(factory);
14. setSerializer(template);//设置序列化工具
15. template.afterPropertiesSet();
16. **return** template;
17. }
18. **private** **void** setSerializer(StringRedisTemplate template){
19. @SuppressWarnings({ "rawtypes", "unchecked" })
20. Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = **new**
21. Jackson2JsonRedisSerializer(Object.**class**);
22. ObjectMapper om = **new** ObjectMapper();
23. om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
24. om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON\_FINAL);
25. jackson2JsonRedisSerializer.setObjectMapper(om);
26. template.setValueSerializer(jackson2JsonRedisSerializer);
27. }
28. }

## 注册Servlet、filter、listener

当使用Servlet容器（tomcat,jetty等）时我们通过将servlet，filter和listener声明为Spring bean而达到注册效果，或者注册ServletRegistrationBean

1. @Bean
2. **public** XxServlet xxServlet(){
3. **return** **new** XxServlet();
4. }
5. @Bean
6. **public** YhFilter yyFilter(){
7. **return** **new** YfFilter();
8. }
9. @Bean
10. **public** ZzListener zzListener(){
11. **return** **new** ZzListener();
12. }

或者

1. @Bean
2. **public** ServletRegistrationBean servletRegistrationBean(){
3. **return** **new** ServletRegistrationBean(**new** XxServlet(),"/xx/");
4. }
5. @Bean
6. **public** FilterRegistrationBean filterRegistrationBean(){
7. FilterRegistrationBean registrationBean = **new** FilterRegistrationBean();
8. registrationBean.setFilter(**new** YyFilter());
9. registrationBean.setOrder(2);
10. **return** registrationBean;
11. }
12. @Bean
13. **public** ServletListenerRegistrationBean<ZzListerner> zzListenerServletRegistrationBean(){
14. **return** **new** ServletListenerRegistrationBean<ZzListener> (**new** Zzlistener());
15. }

使用jetty作为web容器

1. <dependency>
2. <groupId>org.springframework.boot</groupId>
3. <artifactId>spring-boot-starter-web</artifactId>
4. <exclusions>
5. <exclusion>
6. <groupId>org.springframework.boot</groupId>
7. <artifactId>spring-boot-starter-tomcat</artifactId>
8. </exclusion>
9. </exclusions>
10. </dependency>
11. <dependency>
12. <groupId>org.springframework.boot</groupId>
13. <artifactId>spring-boot-starter-jetty</artifactId>
14. </dependency>

## 事务

①、原子性（Atomicity）：事务是一个原子操作，由一系列动作组成。事务的原子性确保动作要么全部完成，要么完全不起作用。

②、一致性（Consistency）：一旦事务完成（不管成功还是失败），系统必须确保它所建模的业务处于一致的状态，而不会是部分完成部分失败。在现实中的数据不应该被破坏。

③、隔离性（Isolation）：可能有许多事务会同时处理相同的数据，因此每个事务都应该与其他事务隔离开来，防止数据损坏。

④、持久性（Durability）：一旦事务完成，无论发生什么系统错误，它的结果都不应该受到影响，这样就能从任何系统崩溃中恢复过来。通常情况下，事务的结果被写到持久化存储器中。

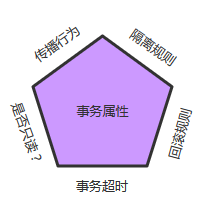
Spring提供了一个PlatformTransactionManager接口实现事务

|  |  |
| --- | --- |
| 数据库访问技术 | 实现 |
| JDBC | DataSourceTransactionManager |
| JPA | JpaTransactionManager |
| Hibernate | HibernateTransactionManager |
| JDO | JdoTransactionManager |
| 分布式事务 | JtaTransactionManager |

在程序中定义事务管理器的代码如下：

1. @Bean
2. **public** PlatformTransactionManager transactionManager(){
3. JpaTransactionManager transactionManager = **new** JpaTransactionManager();
4. transactionManager.setDataSource(dataSource());
5. **return** transactionManager;
6. }

事务属性可以理解成事务的一些基本配置，描述了事务策略如何应用到方法上。事务属性包含了5个方面，如图所示



### **2.2.1 传播行为**

事务的第一个方面是传播行为（propagation behavior）。当事务方法被另一个事务方法调用时，必须指定事务应该如何传播。例如：方法可能继续在现有事务中运行，也可能开启一个新事务，并在自己的事务中运行。Spring定义了七种传播行为，这里以A业务和B业务之间如何传播事务为例说明

| **传播行为** | **含义** |
| --- | --- |
| PROPAGATION\_REQUIRED | 默认值，A如果有事务，B将使用该事务；如果A没有事务，B将创建一个新的事务 |
| PROPAGATION\_SUPPORTS | A如果有事务，B将使用该事务；如果A没有事务，B将以非事务执行 |
| PROPAGATION\_MANDATORY | A如果有事务，B将使用该事务；如果A没有事务，B将抛异常。 |
| PROPAGATION\_REQUIRED\_NEW | 如果A有事务，将A的事务挂起，B创建一个新的事务；如果A没有事务，B创建一个新的事务 |
| PROPAGATION\_NOT\_SUPPORTED | 如果A有事务，将A的事务挂起，B将以非事务执行；如果A没有事务，B将以非事务执行 |
| PROPAGATION\_NEVER | 如果A有事务，B将抛异常；如果A没有事务，B将以非事务执行 |
| PROPAGATION\_NESTED | A和B底层采用保存点机制，形成嵌套事务 |

1. **隔离级别：定义了一个事务可能受其他并发事务影响的程度。**

**并发事务引起的问题：**

在典型的应用程序中，多个事务并发运行，经常会操作相同的数据来完成各自的任务。并发虽然是必须的，但可能会导致以下的问题。

①、脏读（Dirty reads）——脏读发生在一个事务读取了另一个事务改写但尚未提交的数据时。如果改写在稍后被回滚了，那么第一个事务获取的数据就是无效的。

②、不可重复读（Nonrepeatable read）——不可重复读发生在一个事务执行相同的查询两次或两次以上，但是每次都得到不同的数据时。这通常是因为另一个并发事务在两次查询期间进行了更新。

③、幻读（Phantom read）——幻读与不可重复读类似。它发生在一个事务（T1）读取了几行数据，接着另一个并发事务（T2）插入了一些数据时。在随后的查询中，第一个事务（T1）就会发现多了一些原本不存在的记录。

注意：不可重复读重点是修改，而幻读重点是新增或删除。

| **隔离级别** | **含义** |
| --- | --- |
| ISOLATION\_DEFAULT | 使用后端数据库默认的隔离级别 |
| ISOLATION\_READ\_UNCOMMITTED | 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读 |
| ISOLATION\_READ\_COMMITTED | 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生 |
| ISOLATION\_REPEATABLE\_READ | 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生 |
| ISOLATION\_SERIALIZABLE | 最高的隔离级别，完全服从ACID的隔离级别，确保阻止脏读、不可重复读以及幻读，也是最慢的事务隔离级别，因为它通常是通过完全锁定事务相关的数据库表来实现的 |

### **2.2.3 只读**

事务的第三个特性是它是否为只读事务。如果事务只对后端的数据库进行该操作，数据库可以利用事务的只读特性来进行一些特定的优化。通过将事务设置为只读，你就可以给数据库一个机会，让它应用它认为合适的优化措施。

### **2.2.4 事务超时**

为了使应用程序很好地运行，事务不能运行太长的时间。因为事务可能涉及对后端数据库的锁定，所以长时间的事务会不必要的占用数据库资源。事务超时就是事务的一个定时器，在特定时间内事务如果没有执行完毕，那么就会自动回滚，而不是一直等待其结束。

### **2.2.5 回滚规则**

事务五边形的最后一个方面是一组规则，这些规则定义了哪些异常会导致事务回滚而哪些不会。默认情况下，事务只有遇到运行期异常时才会回滚，而在遇到检查型异常时不会回滚（这一行为与EJB的回滚行为是一致的）   
但是你可以声明事务在遇到特定的检查型异常时像遇到运行期异常那样回滚。同样，你还可以声明事务遇到特定的异常不回滚，即使这些异常是运行期异常。

## 声明式事务

@Transactional注解支持声明式事务，

[此@Transactional注解来自org.springframework.transaction.annotation](mailto:此@Transactional注解来自org.springframework.transaction.annotation)

要开启事务需要@EnableTransactionManagement开启声明式事务

@EnableTransactionManagement

@Configuration

Public calss AppConfig(){ }

而在Spring boot中添加Spring-boot-starter-data-jpa依赖后默认开启@EnableTransactionManagementh和注入JpaTransactionManager