# 1.添加依赖

*<!--  
 包含支持UI模版（Velocity，FreeMarker，JasperReports），  
 邮件服务，  
 脚本服务(JRuby)，  
 缓存Cache（EHCache），  
 任务计划Scheduling（uartz）。  
 -->*<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context-support</**artifactId**>  
</**dependency**>  
  
*<!-- JPA操作数据库. -->*<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-jpa</**artifactId**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**scope**>test</**scope**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-data-redis</**artifactId**>  
</**dependency**>

# 2. 配置application.properties;

*########################################################  
###datasource \u914d\u7f6eMySQL\u6570\u636e\u6e90\uff1b  
########################################################***spring.datasource.url** = **jdbc:mysql://localhost:3306/internetredis  
spring.datasource.username** = **root  
spring.datasource.password** = **147094  
spring.datasource.driverClassName** = **com.mysql.jdbc.Driver  
spring.datasource.max-active**=**20  
spring.datasource.max-idle**=**8  
spring.datasource.min-idle**=**8  
spring.datasource.initial-size**=**10** *# database name***spring.redis.database**=**0***# server host1***spring.redis.host**=**127.0.0.1***# server password  
#spring.redis.password=   
#connection port***spring.redis.port**=**6379***# pool settings ...***spring.redis.pool.max-idle**=**8  
spring.redis.pool.min-idle**=**0  
spring.redis.pool.max-active**=**8  
spring.redis.pool.max-wait**=**-1***# name of Redis server  
#spring.redis.sentinel.master=   
# comma-separated list of host:port pairs  
#spring.redis.sentinel.nodes=***spring.jpa.database** = **MYSQL***# Show or not log for each sql query***spring.jpa.show-sql** = **true***# Hibernate ddl auto (create, create-drop, update)***spring.jpa.hibernate.ddl-auto** = **update***# Naming strategy***spring.jpa.hibernate.naming-strategy** = **org.hibernate.cfg.ImprovedNamingStrategy***# stripped before adding them to the entity manager)***spring.jpa.properties.hibernate.dialect** = **org.hibernate.dialect.MySQL5Dialect**

# 3. 编写RedisCacheConfig配置类；

## 解释：      缓存主要有几个要实现的类：

### 其一就是CacheManager缓存管理器；

### 其二就是具体操作实现类；

### 其三就是CacheManager工厂类（这个可以使用配置文件配置的进行注入，也可以通过编码的方式进行实现）；

### 其四就是缓存key生产策略（当然Spring自带生成策略，但是在Redis客户端进行查看的话是系列化的key,对于我们肉眼来说就是感觉是乱码了，这里我们先使用自带的缓存策略）。

## 1.缓存管理器

*/\*\*  
 \* redis 缓存配置;  
 \*   
 \* 注意：RedisCacheConfig这里也可以不用继承 ：CachingConfigurerSupport，也就是直接一个普通的Class就好了；  
 \*   
 \* 这里主要我们之后要重新实现 key的生成策略，只要这里修改KeyGenerator，其它位置不用修改就生效了。  
 \*   
 \* 普通使用普通类的方式的话，那么在使用@Cacheable的时候还需要指定KeyGenerator的名称;这样编码的时候比较麻烦。  
 \*   
 \** ***@author*** *Angel(QQ*:*412887952)  
 \** ***@version*** *v.0.1  
 \*/*@Configuration  
@EnableCaching *//启用缓存，这个注解很重要；***public class** RedisCacheConfig **extends** CachingConfigurerSupport {  
   
 */\*\*  
 \* 缓存管理器.  
 \** ***@param redisTemplate*** *\** ***@return*** *\*/* @Bean  
 **public** CacheManager cacheManager(RedisTemplate<?,?> redisTemplate) {  
 CacheManager cacheManager = **new** RedisCacheManager(redisTemplate);  
 **return** cacheManager;  
 }

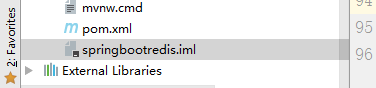
## 2.缓存操作类

*/\*\*  
 \* RedisTemplate缓存操作类,类似于jdbcTemplate的一个类;  
 \*   
 \* 虽然CacheManager也能获取到Cache对象，但是操作起来没有那么灵活；  
 \*   
 \* 这里在扩展下：RedisTemplate这个类不见得很好操作，我们可以在进行扩展一个我们  
 \*   
 \* 自己的缓存类，比如：RedisStorage类;  
 \*   
 \** ***@param factory*** *: 通过Spring进行注入，参数在application.properties进行配置；  
 \** ***@return*** *\*/*@Bean  
**public** RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {  
 RedisTemplate<String, String> redisTemplate = **new** RedisTemplate<String, String>();  
 redisTemplate.setConnectionFactory(factory);  
   
 *//key序列化方式;（不然会出现乱码;）,但是如果方法上有Long等非String类型的话，会报类型转换错误；  
 //所以在没有自己定义key生成策略的时候，以下这个代码建议不要这么写，可以不配置或者自己实现ObjectRedisSerializer  
 //或者JdkSerializationRedisSerializer序列化方式;* RedisSerializer<String> redisSerializer = **new** StringRedisSerializer();*//Long类型不可以会出现异常信息;* redisTemplate.setKeySerializer(redisSerializer);  
 redisTemplate.setHashKeySerializer(redisSerializer);  
   
 **return** redisTemplate;  
}

## 3.自定义的key

*/\*\*  
 \* 自定义key.  
 \* 此方法将会根据类名+方法名+所有参数的值生成唯一的一个key,即使@Cacheable中的value属性一样，key也会不一样。  
 \*/*@Override  
**public** KeyGenerator keyGenerator() {  
 System.***out***.println(**"RedisCacheConfig.keyGenerator()"**);  
 **return new** KeyGenerator() {  
 @Override  
 **public** Object generate(Object o, Method method, Object... objects) {  
 *// This will generate a unique key of the class name, the method name  
 //and all method parameters appended.* StringBuilder sb = **new** StringBuilder();  
 sb.append(o.getClass().getName());  
 sb.append(method.getName());  
 **for** (Object obj : objects) {  
 sb.append(obj.toString());  
 }  
 System.***out***.println(**"keyGenerator="** + sb.toString());  
 **return** sb.toString();  
 }  
 };  
}

## 4. CacheManager工厂类（这个可以使用配置文件



# 4,编写实体类

@Entity  
**public class** DemoInfo **implements** Serializable{  
 **private static final long *serialVersionUID*** = 1L;  
 @Id @GeneratedValue  
 **private long id**;  
 **private** String **name**;  
 **private** String **pwd**;

# 5.编写数据层

**public interface** DemoInfoRepository **extends** CrudRepository<DemoInfo,Long> {  
  
}

# 6.接口层，

**public interface** DemoInfoService {  
   
 **public** DemoInfo findById(**long** id);  
   
 **public void** deleteFromCache(**long** id);  
  
 **void** test();  
}

# 7.服务层重要

@Service  
**public class** DemoInfoServiceImpl **implements** DemoInfoService {  
   
 @Resource  
 **private** DemoInfoRepository **demoInfoRepository**;  
   
 @Resource  
 **private** RedisTemplate<String,String> **redisTemplate**;  
   
 @Override  
 **public void** test(){  
 ValueOperations<String,String> valueOperations = **redisTemplate**.opsForValue();  
 valueOperations.set(**"mykey4"**, **"random1="**+Math.*random*());  
 System.***out***.println(valueOperations.get(**"mykey4"**));  
 }  
  
 *//keyGenerator="myKeyGenerator"* @Cacheable(value=**"demoInfo"**) *//缓存,这里没有指定key.* @Override  
 **public** DemoInfo findById(**long** id) {  
 System.***err***.println(**"DemoInfoServiceImpl.findById()=========从数据库中进行获取的....id="**+id);  
 **return demoInfoRepository**.findOne(id);  
  
 }

@CacheEvict(value=**"demoInfo"**)  
 @Override  
 **public void** deleteFromCache(**long** id) {  
 System.***out***.println(**"DemoInfoServiceImpl.delete().从缓存中删除."**);  
 }  
   
}

## 解释

## 1. @Cacheable 支持如下几个参数：

value：缓存位置名称，不能为空，如果使用EHCache，就是ehcache.xml中声明的cache的name

key：缓存的key，默认为空，既表示使用方法的参数类型及参数值作为key，支持SpEL

condition：触发条件，只有满足条件的情况才会加入缓存，默认为空，既表示全部都加入缓存，支持SpEL

以下有一个例子：

1.  //将缓存保存进andCache，并使用参数中的userId加上一个字符串(这里使用方法名称)作为缓存的key

2.  @Cacheable(value="andCache",key="#userId + 'findById'")

3.  **public** SystemUser findById(String userId) {

4.      SystemUser user = (SystemUser) dao.findById(SystemUser.**class**, userId);

5.      **return** user ;

6.  }

7.  //将缓存保存进andCache，并当参数userId的长度小于32时才保存进缓存，默认使用参数值及类型作为缓存的key

8.  @Cacheable(value="andCache",condition="#userId.length < 32")

9.  **public** **boolean** isReserved(String userId) {

10.    System.out.println("hello andCache"+userId);

11.    **return** **false**;

12.}

## 2. @CacheEvict 支持如下几个参数：

value：缓存位置名称，不能为空，同上

key：缓存的key，默认为空，同上

condition：触发条件，只有满足条件的情况才会清除缓存，默认为空，支持SpEL

allEntries：true表示清除value中的全部缓存，默认为false

以下是一个小例子：

1.  //清除掉指定key的缓存

2.  @CacheEvict(value="andCache",key="#user.userId + 'findById'")

3.  **public** **void** modifyUserRole(SystemUser user) {

4.           System.out.println("hello andCache delete"+user.getUserId());

5.  }

6.

7.  //清除掉全部缓存

8.  @CacheEvict(value="andCache",allEntries=**true**)

9.  **public** **void** setReservedUsers() {

10.    System.out.println("hello andCache deleteall");

11.}

一般来说，我们的更新操作只需要刷新缓存中某一个值，所以定义缓存的key值的方式就很重要，最好是能够唯一，因为这样可以准确的清除掉特定的缓存，而不会影响到其它缓存值，

比如我这里针对用户的操作，使用(userId+方法名称)的方式设定key值，当然，你也可以找到更适合自己的方式去设定。

# 8.controller测试

@Controller  
**public class** DemoInfoController {  
   
 @Autowired  
 DemoInfoService **demoInfoService**;  
   
 @RequestMapping(**"/test"**)  
 **public** @ResponseBody  
 String test(){  
 DemoInfo loaded = **demoInfoService**.findById(1);  
 System.***out***.println(**"loaded="**+loaded);  
 DemoInfo cached = **demoInfoService**.findById(1);  
 System.***out***.println(**"cached="**+cached);  
 loaded = **demoInfoService**.findById(2);  
 System.***out***.println(**"loaded2="**+loaded);  
 **return "ok"**;  
 }  
  
 @RequestMapping(**"/delete"**)  
 **public** @ResponseBody  
 String delete(**long** id){  
 **demoInfoService**.deleteFromCache(id);  
 **return "ok"**;  
 }   
   
 @RequestMapping(**"/test1"**)  
 **public** @ResponseBody  
 String test1(){  
 **demoInfoService**.test();  
 System.***out***.println(**"DemoInfoController.test1()"**);  
 **return "ok"**;  
 }

# 8.测试

