一、springboot-mybatis整合代码

1.controller代码

```
@RestController
@RequestMapping(value="/emp")
public class EmpController {
    @Autowired
    EmployeeService employeeService;
    //测试地址   http://localhost:8080/emp/emp/1
    @GetMapping(value="/emp/{id}")
    public Employee getEmpById(@PathVariable("id") Integer id) {

        Employee emp = employeeService.getEmpById(id);
        return emp;
    }

//  @CachePut(value="emp",key="#employee.id")
    @RequestMapping(value="/upd")
    public Employee updateEmployee(Employee employee) {
        Employee updateEmployee = employeeService.updateEmployee(employee);
        return updateEmployee;
```

```
    }
    @RequestMapping(value="/add")
    public Employee insertEmployee(Employee employee) {

employeeService.insertEmployee(employee);
        return employee;
    }
    @RequestMapping(value="/del")
    public Integer deleteEmployee(Integer id) {
        System.out.println(id);
        employeeService.deleteEmployee(id);
        return id;
    }
}

```

2.service代码

```
@Service
public class EmployeeService {
    @Autowired
    EmployeeMapper employeeMapper;

    public Employee getEmpById(Integer id) {
        Employee emp = employeeMapper.getEmpById(id);
        return emp;
```

```
    }

    public void updateEmployee(Employee employee) {
        employeeMapper.updateEmployee(employee);
    }

    public void insertEmployee(Employee employee) {
        employeeMapper.insertEmployee(employee);
    }

    public void deleteEmployee(Integer id) {
        employeeMapper.deleteEmployee(id);
    }
}
```

3.mapper代码

```
@Mapper
public interface EmployeeMapper {
    //根据Id查询员工的接口
    @Select("select * from employee where id=#{id}")
    public Employee getEmpById(Integer id);
    @Update("update employee set lastName=#{lastName},email=#{email},gender=#{gender},d_id=#{dId} where id=#{id}")
```

```
    public void updateEmployee(Employee employee);

    @Update("insert into employee set lastName=#{lastName},email=#{email},gender=#{gender},d_id=#{dId}")
    public void insertEmployee(Employee employee);

    @Delete("delete from employee where id=#{id}")
    public void deleteEmployee(Integer id);
}
```

4.yaml配置文件

```
spring:
  datasource:
    url: jdbc:mysql://47.105.103.45:3306/mybatis
    username: root
    password: 123456
    driver-class-name: com.mysql.jdbc.Driver
#配置开启驼峰命名法
mybatis:
  configuration:
    map-underscore-to-camel-case: true
#设置日志级别，使他打印sql语句
```

```yaml
logging:
  level:
    com.haoge.cache.mapper: debug
```

5。实体类

```java
public class Employee {

    private Integer id;
    private String lastName;
    private String email;
    private Integer gender; //性别 1男  0女
    private Integer dId;
}
```

6.建表语句

```
CREATE TABLE `employee` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `lastName` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `gender` int(2) DEFAULT NULL,
  `d_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8;
```

二、缓存cache

springboot开启缓存功能需要在主启动类上加上@EnableCaching注解.

```
@SpringBootApplication
@EnableCaching
public class SpringbootCacheApplication {
    public static void main(String[] args) {

SpringApplication.run(SpringbootCacheApplication.class, args);
    }
}
```

1.Cacheable注解

```
    /**
     * Cacheable注解作用：将方法的返回结果进行缓存，以后如果是获取相同的数据，则从缓存中获取，不需要查询数据库
     * cacheManager：管理cache组件的，真正对缓存的crud操作在缓存组件中进行，每一个缓存组件都有自己对应的名字
     * Cacheable中的几个属性对应的意思：
     *      cacheNames/value:用来指定缓存组件的名字
```

```
6       *       key:缓存数据时使用的key值可以用这个属性来指定，默认是使用方法参数的值，如这个方法中使用的是id为key值
7       *           我们也可以编写SpEL表达式来执行key值。如 #id,   #a0   #p0   #root.args[0]
8       *       keyGenerator:key值的生成器，我们可以自己指定key的生成器的组件ID。key和keyGenerator:key不能同时使用
9       *       cacheManager:指定缓存管理器  ;  cacheResolver:指定缓存解析器
10      *       condition:指定在符合情况下才能进行缓存
11      *       unless:否定缓存，当unless的条件结果为true时，返回结果不进行缓存。也可以获取到结果再进行缓存。
12      *               如unless="#result==null"即返回结果为空的时候不进行缓存
13      *       sync:是否适用异步模式，如果设置sync属性为true,即使用异步模式，此时unless属性不受支持
14      *
15      */
16     @Cacheable(cacheNames= {"emp"})
17     @GetMapping(value="/emp/{id}")
18     public Employee getEmpById(@PathVariable("id") Integer id) {
19
20         Employee emp = employeeService.getEmpById(id);
21         return emp;
```

```
22      }
```

2.缓存原理

1. 缓存自动配置类 CacheAutoConfiguration
2. springboot已经缓存的配置类如下

   1530757947830
3. springboot中默认生效的缓存类为 SimpleCacheConfiguration
4. SimpleCacheConfiguration会给容器中注册一个缓存管理器：ConcurrentMapCacheManager
5. ConcurrentMapCacheManager这个缓存管理器会获取ConcurrentMapCache类型的缓存，它的缓存数据存在ConcurrentMap中。

   ## 3.Cacheable注解的运行流程

```
    @Cacheable(cacheNames= {"emp"})
    @GetMapping(value="/emp/{id}")
    public Employee
getEmpById(@PathVariable("id") Integer id) {

        Employee emp =
employeeService.getEmpById(id);
        return emp;
    }
```

如代码所示：标注了Cacheable注解的方法运行流程如下：

1. 方法运行之前先查询cache,按照cacheNames指定的名字进行获取对应的缓存；cacheManager先获取相应的缓存，第一次获取的时候如果没有对应的缓存，就会进行创建对应的缓存。
2. 去cache中获取对应的缓存的内容，默认的key是方法的参数。key是按照某种生成策略生成的，默认的生成策略是KeyGenerator生成的，默认使用SimpleKeyGenerator生成key

   SimpleKeyGenerator生成key的策略

   如果请求没有参数的话，key=new SimpleKey();

   有一个参数：key=参数的值

   有多个参数：key=new SimpleKey(params);
3. 如果在缓存中没有查到数据就调用目标方法

4. 将目标方法返回的数据放进缓冲中

总结：Cacheable标注的方法执行之前先来缓存中检查有没有这个数据，默认按照参数的值作为key去查询，如果没有查询到结果就调用目标方法并将最终结果放入缓存中，以后再次调用该方法的时候就直接从缓存中获取数据。

4.Cacheable中属性详解

1. key :如下代码拼接处的key 为 getEmpById[2]

```
    @Cacheable(cacheNames={"emp"},key="#root.methodName+'['+#id+']'")
    @GetMapping(value="/emp/{id}")
    public Employee getEmpById(@PathVariable("id") Integer id) {
```

2. 指定我们自己编写的keyGenerator，并且在方法上使用

```
@Configuration
public class MyCacheConfig {
    //将我们自定义的keyGenerator加入到容器中，并且制定id
    @Bean("myKeyGenerator")
    public KeyGenerator keyGenerator() {
        return new KeyGenerator() {

            @Override
```

```
            public Object generate(Object target, Method method, Object... params) {
                // TODO Auto-generated method stub
                return method.getName()+Arrays.asList(params).toString();
            }
        };
    }
}
```

```
    @Cacheable(cacheNames={"emp"},keyGenerator="myKeyGenerator")
```

3. condition属性

```
    @Cacheable(cacheNames={"emp"},keyGenerator="myKeyGenerator",condition="#a0>1")
```

作用：当第一个参数>1的时候，结果才会进行缓存。

condition 还支持如下这种格式

```
condition="#a0>1 and #root.methodName eq 'aaa'
```

4. unless属性，如下如果第一次参数结果=2就不进行缓存。

```
    @Cacheable(cacheNames=
{"emp"},keyGenerator="myKeyGenerator",condition="
#a0>1 and #root.methodName eq
'aaa'",unless="#a0==2")
```

5.@CachePut注解

1. 作用：既调用了方法又同时更新了缓存。比如修改方法，我们在修改数据库之后，会将我们返回的结果同时在cache中进行更新。
2. 运行时机：先调用目标方法，之后再将目标方法返回的结果进行缓存。

```
    @Cacheable(cacheNames= {"emp"})
    @GetMapping(value="/emp/{id}")
    public Employee
getEmpById(@PathVariable("id") Integer id) {

        Employee emp =
employeeService.getEmpById(id);
        return emp;
    }
    //key="#result.id"也是对应数据的ID
    @CachePut(value="emp",key="#employee.id")
    @RequestMapping(value="/upd")
    public Employee updateEmployee(Employee
employee) {
```

```
12          Employee updateEmployee =
employeeService.updateEmployee(employee);
13          return updateEmployee;
14      }
```

注意：查询方法默认缓存的key是id，所以在修改方法上我们也要指定key为对应数据的ID，才可以达到修改同时更新缓存的效果。

6.@CacheEvict注解

1. 作用：清除缓存
2. 几个重要属性：
    - beforeInvocation=false,判断清除缓存操作是否在方法之前执行，默认为false,是在方法之后执行，如果方法出现异常，则清除缓存操作不会执行。
    - beforeInvocation=true 代表清除缓存操作在方法之前执行，不论执行方法是否会出现异常，缓存都会被清除。
    - allEntries=true 清除指定缓存中的所有数据

```
@CacheEvict(value="emp",beforeInvocation=true,allEntries=true)
    @RequestMapping(value="/del")
    public Integer deleteEmployee(Integer id) {
        System.out.println(id);
        employeeService.deleteEmployee(id);
        return id;
    }
```

7.@Caching作用

```
//@Caching注解的作用：定义复杂的缓存规则
    @Caching(
            cacheable= {

@Cacheable(value="emp",key="#lastName")
            },
            put= {

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

@CachePut(value="emp",key="#result.email")
            }
    )
    public Employee getEmpBylastName(String lastName) {
```

```
12          Employee
    employee=employeeMapper.getEmpBylastName(lastNam
    e);
13          return employee;
14      }
```

8.@CacheConfig注解

抽取该类中的关于缓存的公共注解，放在类上。

```
1 @CacheConfig(cacheNames="emp")
2 @RestController
3 @RequestMapping(value="/emp")
4 public class EmpController {
```

三、springboot整合redis

1.使用docker安装redis

- 下载镜像 docker pull redis
- 启动redis容器

  docker run -d -p 6379:6379 --name myredis docker.io/redis

2.在项目中引入redis启动器

```
<dependency>

 <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

3.配置redis

```
#配置redis的主机地址
spring:
  redis:
    host: 47.105.103.45
```

4.redis自动配置

- redis自动配置类：RedisAutoConfiguration
- RedisAutoConfiguration为我们配了两个Template，用来进行和redis交互,分别是StringRedisTemplate和RedisTemplate，源码如下

```
    /**
     * Standard Redis configuration.
     */
    @Configuration
    protected static class RedisConfiguration {
```

```
6
7           @Bean
8           @ConditionalOnMissingBean(name =
"redisTemplate")
9           public RedisTemplate<Object, Object>
redisTemplate(
10                  RedisConnectionFactory
redisConnectionFactory)
11                  throws UnknownHostException {
12              RedisTemplate<Object, Object>
template = new RedisTemplate<Object, Object>
();
13
template.setConnectionFactory(redisConnectionF
actory);
14              return template;
15          }
16
17          @Bean
18
@ConditionalOnMissingBean(StringRedisTemplate.
class)
19          public StringRedisTemplate
stringRedisTemplate(
20                  RedisConnectionFactory
redisConnectionFactory)
21                  throws UnknownHostException {
```

```
            StringRedisTemplate template = new StringRedisTemplate();

template.setConnectionFactory(redisConnectionFactory);
            return template;
        }

    }
```

5.对RedisTemplate进行测试

```
//对RedisTemplate进行测试
@RunWith(SpringRunner.class)
@SpringBootTest
public class SpringbootCacheApplicationTests {

    // redis 的自动配置类RedisAutoConfiguration
    @Autowired
    StringRedisTemplate stringRedisTemplate;//主要是用来操作字符串的，key和value都是字符串
    @Autowired
    RedisTemplate<Object,Object> redisTemplate;//key和value可自己进行设置
    @Autowired
    EmployeeMapper employeeMapper;
    //自动注入我们自己自定义的RedisTemplate
    @Autowired
```

```
15      RedisTemplate<Object, Employee>
empRedisTemplate;
16      /**
17       * redis中常见的五大数据类型：String,List(列
表),Set(集合)，Hash(散列),Zset(有序集合)
18       *
19       */
20      @Test
21      public void test01() {
22  //      stringRedisTemplate分别用来操作五种数据
类型的方法
23  //      redisTemplate中也有对应的用来操作数据的
五种方法
24  //      stringRedisTemplate.opsForValue();
25  //      stringRedisTemplate.opsForList();
26  //      stringRedisTemplate.opsForSet();
27  //      stringRedisTemplate.opsForHash();
28  //      stringRedisTemplate.opsForZSet();
29  //
stringRedisTemplate.opsForValue().set("aa",
"aa");
30          String string =
stringRedisTemplate.opsForValue().get("aa");
31          System.out.println(string);
32      }
33      @Test
34      public void test02() {
```

```
35          Employee employee =
employeeMapper.getEmpById(1);
36          //使用自定义的RedisTemplate（修改其序列
化规则）操作对象
37          //
redisTemplate.opsForValue().set("emp01",
employee);
38
empRedisTemplate.opsForValue().set("emp01",
employee);
39      }
40  }
```

编写自定义的RedisTemplate，主要是修改其序列化规则，代码如下。

其代码和RedisAutoConfiguration为我们配置的StringRedisTemplate和RedisTemplate类似，我们只是修改了其序列化规则

```
1  @Configuration//表明这是一个配置类
2  public class MyRedisConfig {
3      //自定义RedisTemplate,但是改变默认的序列化器
4      @Bean//将我们自定义的RedisTemplate加在容器中
5      public RedisTemplate<Object, Employee>
empRedisTemplate(
6              RedisConnectionFactory
redisConnectionFactory)
7              throws UnknownHostException {
```

```
8           RedisTemplate<Object, Employee>
template = new RedisTemplate<Object, Employee>
();
9
template.setConnectionFactory(redisConnectionF
actory);
10          Jackson2JsonRedisSerializer<Employee>
serializer = new
Jackson2JsonRedisSerializer<Employee>
(Employee.class);
11
template.setDefaultSerializer(serializer);
12          return template;
13      }
14  }
```

四、redisCache

1.原理

1. 通过cacheManager来获取cache,实际缓存的数据是在cache中
2. 当我们引入redis之后，容器中保存的就是RedisCacheManager,别的CacheManager不会再起作用。
3. RedisCacheManager会帮我们创建RedisCache，之后数据实际缓存在redis中。

4. RedisTemplate<Object,Object>（推荐使用）默认使用jdk的序列化机制。我们也可以自定义自己的CacheManager（目前不太推荐）

```
@Configuration // 表明这是一个配置类
public class MyRedisConfig {
    // 自定义RedisTemplate,但是改变默认的序列化器
    //默认这个组件的ID为其名称，即empRedisTemplate
    @Bean // 将我们自定义的RedisTemplate加在容器中
    public RedisTemplate<Object, Employee>
empRedisTemplate(RedisConnectionFactory
redisConnectionFactory)
            throws UnknownHostException {
        RedisTemplate<Object, Employee> template
= new RedisTemplate<Object, Employee>();

template.setConnectionFactory(redisConnectionFac
tory);
        Jackson2JsonRedisSerializer<Employee>
serializer = new
Jackson2JsonRedisSerializer<Employee>
(Employee.class);

template.setDefaultSerializer(serializer);
        return template;
    }

    @Bean // 将我们自定义的RedisTemplate加在容器中
```

```
16      public RedisTemplate<Object, Department>
deptRedisTemplate(RedisConnectionFactory
redisConnectionFactory)
17              throws UnknownHostException {
18          RedisTemplate<Object, Department>
template = new RedisTemplate<Object, Department>
();
19
template.setConnectionFactory(redisConnectionFac
tory);
20          Jackson2JsonRedisSerializer<Department>
serializer = new
Jackson2JsonRedisSerializer<Department>(
21                  Department.class);
22
template.setDefaultSerializer(serializer);
23          return template;
24      }
25
26      // 自定义empLoyeeCacheManager
27      @Primary//当容器中有多个CacheManager的时候，使
用Primary设置默认的缓存管理器
28      @Bean
29      public RedisCacheManager
empLoyeeCacheManager(RedisTemplate<Object,
Employee> empRedisTemplate) {
30          RedisCacheManager cacheManager = new
RedisCacheManager(empRedisTemplate);
```

```
        //缓存在redis中的数据的key值会自动拼接一个前缀，默认是拼接cacheName作为key的前缀
        cacheManager.setUsePrefix(true);
        return cacheManager;
    }

    // 自定义deptCacheManager
    @Bean
    public RedisCacheManager deptCacheManager(RedisTemplate<Object, Department> deptRedisTemplate) {
        RedisCacheManager cacheManager = new RedisCacheManager(deptRedisTemplate);
        cacheManager.setUsePrefix(true);
        return cacheManager;
    }
}
```

自定义CacheManaer使用

```
//自动注入
@Qualifier("empLoyeeCacheManager")
@Autowired
RedisCacheManager empLoyeeCacheManager;

    @Test
    public void test02() {
```

```
8           Employee employee =
employeeMapper.getEmpById(1);
9           //获取缓存
10          Cache cache =
empLoyeeCacheManager.getCache("emp");
11          cache.put("emp02", employee);//存值
12      ValueWrapper valueWrapper =
cache.get("emp02");//取值
13
    System.out.println(valueWrapper.getClass());
14
    System.out.println(valueWrapper.toString());
15      }
16
```

五、核心概念总结

1.核心概念

Cache	**缓存接口，定义缓存操作。实现有：RedisCache、 EhCacheCache、ConcurrentMapCache等**
CacheManager	缓存管理器，管理各种缓存（Cache）组件
@Cacheable	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略

2.@Cacheable/@CachePut/@CacheEvict 主要的参数

value	**缓存的名称，在 spring 配置文件中定义，必须指定 至少一个**
key	缓存的 key，可以为空，如果指定要按照 SpEL 表达 式编写，如果不指定，则缺省按照方法的所有参数 进行组合
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存/清除缓存，在 调用方法之前之后都能判断
allEntries (@CacheEvict)	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存
beforeInvocation (@CacheEvict)	是否在方法执行前就清空，缺省为 false，如果指定 为 true，则在方法还没有执行的时候就清空缓存， 缺省情况下，如果方法执行抛出异常，则不会清空 缓存
unless (@CachePut) (@Cacheable)	用于否决缓存的，不像condition，该表达式只在方 法执行之后判断，此时可以拿到返回值result进行判 断。条件为true不会缓存， fasle才缓存

3.Cache SpEL可以使用的元素

名字	位置	描述	示例
methodName	root object	当前被调用的方法名	#root.methodName
method	root object	当前被调用的方法	#root.method.name
target	root object	当前被调用的目标对象	#root.target
targetClass	root object	当前被调用的目标对象类	#root.targetClass
args	root object	当前被调用的方法的参数列表	#root.args[0]
caches	root object	当前方法调用使用的缓存列表（如@Cacheable(value={"cache1", "cache2"})），则有两个cache	#root.caches[0].nam
argument name	evaluation context	方法参数的名字. 可以直接 #参数名 ，也可以使用 #p0或#a0 的 形式， 0代表参数的索引；	#iban 、 #a0 、 #p0
result	evaluation context	方法执行后的返回值（仅当方法执行之后的判断有效，如 ‘unless’ ， ’cache put’的表达式 ’cache evict’的表达式 beforeInvocation=false）	#result