缓存

缓存运行原理:

```
*   1、自动配置类：CacheAutoConfiguration
*   2、缓存的配置类
*   org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.GuavaCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration
*   org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration
*   3、哪个配置类默认生效：SimpleCacheConfiguration；
*
*   4、给容器中注册了一个CacheManager：ConcurrentMapCacheManager
*   5、可以获取和创建ConcurrentMapCache类型的缓存组件；他的作用将数据保存在ConcurrentMap中；
*
```

缓存运行流程:

```
*   运行流程：
*   @Cacheable：
*   1、方法运行之前，先去查询Cache（缓存组件），按照cacheNames指定的名字获取；
*       （CacheManager先获取相应的缓存），第一次获取缓存如果没有Cache组件会自动创建。
*   2、去Cache中查找缓存的内容，使用一个key，默认就是方法的参数；
*       key是按照某种策略生成的；默认是使用keyGenerator生成的，默认使用SimpleKeyGenerator生成key；
*           SimpleKeyGenerator生成key的默认策略；
*                   如果没有参数：key=new SimpleKey()；
*                   如果有一个参数：key=参数的值
*                   如果有多个参数：key=new SimpleKey(params)；
*   3、没有查到缓存就调用目标方法；
*   4、将目标方法返回的结果，放进缓存中
*
*   @Cacheable标注的方法执行之前先来检查缓存中有没有这个数据，默认按照参数的值作为key去查询缓存，
*   如果没有就运行方法并将结果放入缓存；以后再来调用就可以直接使用缓存中的数据；
*
*   核心：
*       1）、使用CacheManager【ConcurrentMapCacheManager】按照名字得到Cache【ConcurrentMapCache】组件
*       2）、key使用keyGenerator生成的，默认是SimpleKeyGenerator
*
*
```

几个重要的概念及缓存注解

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

缓存的cachemanager 管理各种缓存组件,每一个缓存逐渐都有一个唯一的名字

Cacheable

属性:

1. cacheName/value :指定缓存组件的名字

```
@Cacheable(value = {"emp"})
public Employee getEmp(Integer id){
    System.out.println("查询"+id+"号员工");
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}
```

2. key: 缓存数据要使用的key,默认使用方法的参数值, 比如使用 SqEL表达式

编写SpEL,#id 表示参数的值

```
@Cacheable(value = {"emp"},key = "#root.methodName+'['+#id+']'")
public Employee getEmp(Integer id){
    System.out.println("查询"+id+"号员工");
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}
```

Cache SpEL available metadata

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].name
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

3. keyGenerator: key的生成器,可以自己指定key的生成器的组件id. key/keyGenerator二选一
4. cacheManager:指定使用哪一个缓存管理器, 或者cacheResolver指定获取解析器
5. condition: 指定符合条件的情况下才会去缓存

```java
@Cacheable(cacheNames = {"emp"},condition = "#id>0")
public Employee getEmp(Integer id){
    System.out.println("查询"+id+"号员工");
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}
```

6. unless:否定缓存,单unless指定的条件为true,方法的返回值就不会被缓存,可以获取到结果之后进行判断. 比如:

```java
@Cacheable(cacheNames = {"emp"},condition = "#id>0",unless = "#result == null")
public Employee getEmp(Integer id){
    System.out.println("查询"+id+"号员工");
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}
```

7. sync: 是否使用异步模式

CachePut

既调用方法,又更新缓存数据(修改数据库的某个数据,同时更新缓存)

运行时机

1. 先调用目标方法
2. 将目标方法的结果缓存起来

```java
@CachePut(value = "emp")
public Employee updateEmp(Employee employee){
    System.out.println("updateEmp:"+employee);
    employeeMapper.updateEmp(employee);
    return employee;
}
```

测试步骤

```
* 测试步骤:
*  1、查询1号员工;查到的结果会放在缓存中;
*          key:1  value:lastName:张三
*  2、以后查询还是之前的结果
*  3、更新1号员工;【lastName:zhangsan;gender:0】
*          将方法的返回值也放进缓存了;
*          key:传入的employee对象  值:返回的employee对象;
*  4、查询1号员工?
*      应该是更新后的员工;
*          key = "#employee.id":使用传入的参数的员工id;
*          key = "#result.id":使用返回后的id
*              @Cacheable的key是不能用#result
*      为什么是没更新前的?【1号员工没有在缓存中更新】
*
```

```java
@CachePut(value = "emp",key = "#result.id")
public Employee updateEmp(Employee employee){
    System.out.println("updateEmp:"+employee);
    employeeMapper.updateEmp(employee);
    return employee;
}
```

CacheEvict

缓存清除

属性

1. key :可以通过key指定要清除缓存组件中的某一个特定的值

```
@CacheEvict(value="emp",key = "#id")
public void deleteEmp(Integer id){
    System.out.println("deleteEmp:"+id);
    //employeeMapper.deleteEmpById(id);

}
```

2. allEntries= true/false :来定义是否删除全部数据

```
@CacheEvict(value="emp",/*key = "#id",*/allEntries = true)
public void deleteEmp(Integer id){
    System.out.println("deleteEmp:"+id);
    //employeeMapper.deleteEmpById(id);

}
```

3. beforeInvocation = false:缓存的清除是否在方法执行之前 (默认在方法执行之后执行清空缓存)

```
@CacheEvict(value="emp",beforeInvocation = true/*key = "#id",*/)
public void deleteEmp(Integer id){
    System.out.println("deleteEmp:"+id);
    //employeeMapper.deleteEmpById(id);
    int i = 10/0;
}
```

Caching

可以在一个方法上 指定多个缓存规则

例子:

```
@Caching(
    cacheable = {
        @Cacheable(value="emp",key = "#lastName")
    },
    put = {
        @CachePut(value="emp",key = "#result.id"),
        @CachePut(value="emp",key = "#result.email")
    }
)
```

CacheConfig

可以使用CacheConfig指定了该类的全局配置; 比如当配置了value="emp"在方法上就可以不用在指定具体的缓存组件名称

```
import org.springframework.cache.annotati
import org.springframework.stereotype.Ser

@CacheConfig(value="emp")
@Service
public class EmployeeService {
```

Redis

Redis可以作为数据库缓存和消息中间件

Redis常见的五大数据类型

String（字符串）、List（列表）、Set（集合）、Hash（散列）、ZSet（有序集合）

1. stringRedisTemplate.opsForValue() [String（字符串）]
2. stringRedisTemplate.opsForList() [List（列表）]
3. stringRedisTemplate.opsForSet() [Set（集合）]
4. stringRedisTemplate.opsForHash() [Hash（散列）]
5. stringRedisTemplate.opsForZSet() [ZSet（有序集合）]

StringRedisTemplate 方法

```
 @Bean
@ConditionalOnMissingBean({StringRedisTemplate.class})
public StringRedisTemplate stringRedisTemplate(RedisConnectionFactory
 redisConnectionFactory) throws UnknownHostException {
            StringRedisTemplate template = new StringRedisTemplate();
            template.setConnectionFactory(redisConnectionFactory);
            return template;
}
```

stringRedisTemplate的操作空间在与K-V为对象类型的

RedisTemplate 方法

```
 @Bean
@ConditionalOnMissingBean( name = {"redisTemplate"} )
public RedisTemplate<Object, Object> redisTemplate(RedisConnectionFactory
 redisConnectionFactory) throws UnknownHostException {
            RedisTemplate<Object, Object> template = new RedisTemplate();
            template.setConnectionFactory(redisConnectionFactory);
            return template;
}
```

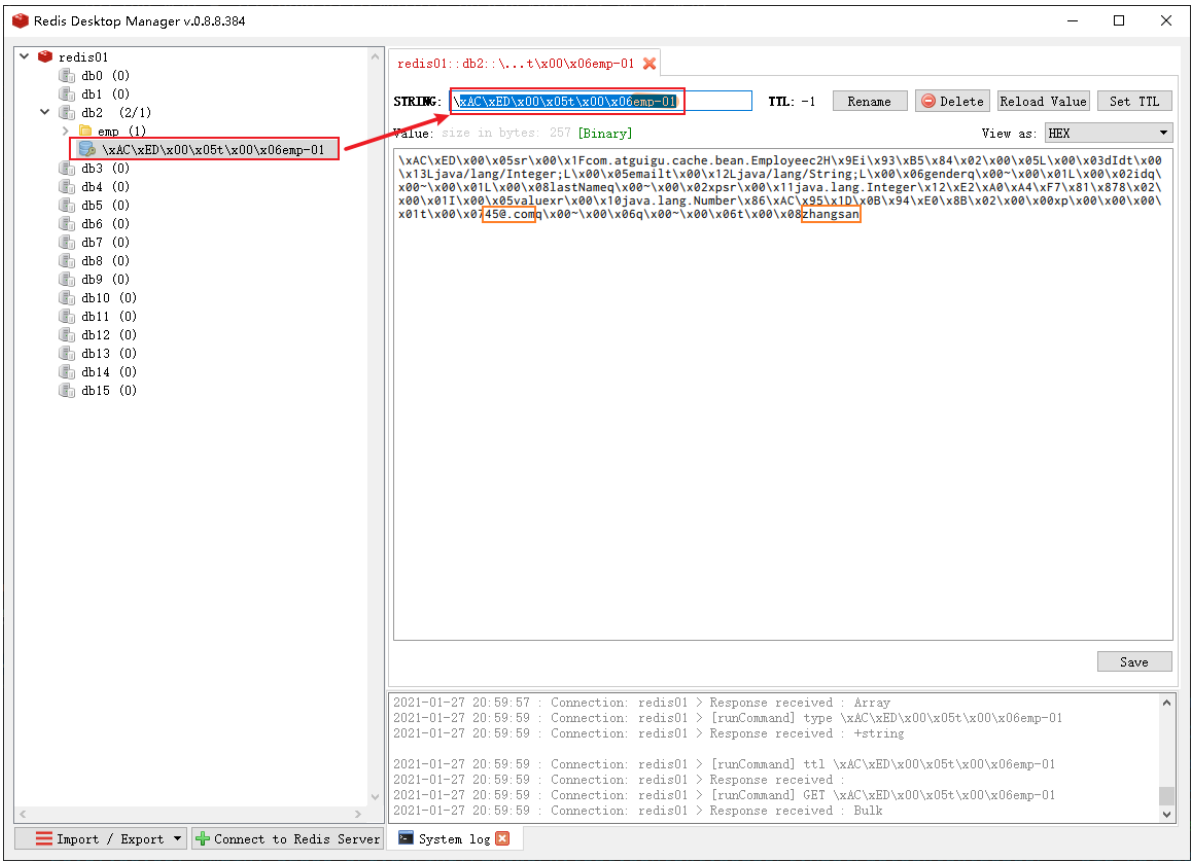
redisTemplate的操作空间在于K-V都是字符串类型的 >>> RedisTemplate<Object, Object>

Redis的序列化

在RedisTemplate中直接存入对象值时会出现序列化的问题, 这个时候就需要在Employee中implements Serializable实现 实体类序列化.

问题:

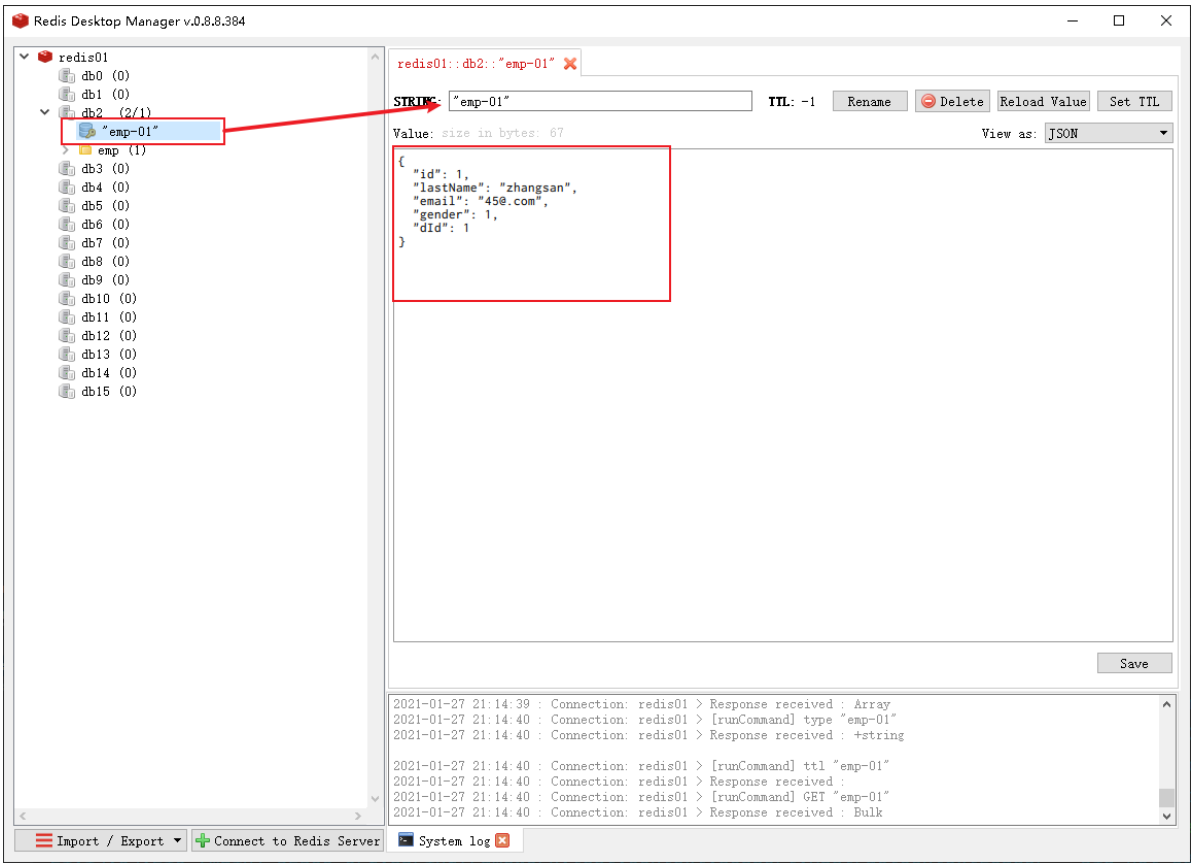
```java
@Test
public void test02(){
    Employee empById = employeeMapper.getEmpById(1);
    //默认如果保存对象，使用jdk序列化机制，序列化后的数据保存到redis中
    redisTemplate.opsForValue().set("emp-01",empById);
    //1、将数据以json的方式保存
     //(1)自己将对象转为json
     //(2)redisTemplate默认的序列化规则；改变默认的序列化规则；
    // empRedisTemplate.opsForValue().set("emp-01",empById);
}
```



解决

如果要避免出现上面这种情况则需要配置一个有关Employee实体类的RedisTemplate方法

```
 @Bean
public RedisTemplate<Object, Employee> empRedisTemplate( RedisConnectionFactory
redisConnectionFactory) throws UnknownHostException {
        RedisTemplate<Object, Employee> template = new RedisTemplate<Object,
Employee>();
        template.setConnectionFactory(redisConnectionFactory);
        Jackson2JsonRedisSerializer<Employee> ser = new
Jackson2JsonRedisSerializer<Employee>(Employee.class);
        template.setDefaultSerializer(ser);
        return template;
}
```



定制Redis缓存规则(缓存管理器)

```
//CacheManagerCustomizers可以来定制缓存的一些规则
@Primary  //将某个缓存管理器作为默认的
@Bean
public RedisCacheManager employeeCacheManager(RedisTemplate<Object, Employee>
empRedisTemplate){
        RedisCacheManager cacheManager = new RedisCacheManager(empRedisTemplate);
        //key多了一个前缀
        //使用前缀，默认会将CacheName作为key的前缀
        cacheManager.setUsePrefix(true);
        return cacheManager;
}
```

添加缓存前缀

```
//使用前缀，默认会将CacheName作为key的前缀
cacheManager.setUsePrefix(true);
```

通过编码方式缓存

可以在service层进行添加一个缓存管理器RedisCacheManager使用定义的缓存管理器来使用编码缓存替换注解缓存

```
    //ualifier的意思是合格者，通过这个标示，表明了哪个实现类才是我们所需要的，
    //添加@Qualifier注解，需要注意的是@Qualifier的参数名称为我们之前定义@Service注解的名称之一。
    @Qualifier("employeeCacheManager")
    @Autowired
    RedisCacheManager employeeCacheManager;
```

创建一个方法在方法内进行缓存的替换

消息

消息有两大重要的概念: 消息代理 和 目的地

当消息发送者发送消息以后,将由消息代理接管,消息代理保证消息传递到指定目的地

消息队列

消息队列主要有两种形式的目的地:

1. 队列: 点对点消息通信 >>>
2. 主题: 发布/订阅消息通信

消息代理规范

JMS 和 AMQP

	JMS	AMQP
定义	Java api	网络线级协议
跨语言	否	是
跨平台	否	是
Model	提供两种消息模型： （1）、Peer-2-Peer （2）、Pub/sub	提供了五种消息模型： （1）、direct exchange （2）、fanout exchange （3）、topic change （4）、headers exchange （5）、system exchange 本质来讲，后四种和JMS的pub/sub模型没有太大差别，仅是在路由机制上做了更详细的划分；
支持消息类型	多种消息类型： TextMessage MapMessage BytesMessage StreamMessage ObjectMessage Message （只有消息头和属性）	byte[] 当实际应用时，有复杂的消息，可以将消息序列化后发送。
综合评价	JMS 定义了JAVA API层面的标准；在java体系中，多个client均可以通过JMS进行交互，不需要应用修改代码，但是其对跨平台的支持较差；	AMQP定义了wire-level层的协议标准；天然具有跨平台、跨语言特性。

8. Spring支持
   - **spring-jms提供了对JMS的支持**
   - **spring-rabbit提供了对AMQP的支持**
   - **需要ConnectionFactory的实现来连接消息代理**
   - **提供JmsTemplate、RabbitTemplate来发送消息**
   - **@JmsListener（JMS）、@RabbitListener（AMQP）注解在方法上监听消息代理发布的消息**
   - **@EnableJms、@EnableRabbit开启支持**

9. Spring Boot自动配置
   - **JmsAutoConfiguration**
   - **RabbitAutoConfiguration**

RabbitMQ核心概念

Message

消息它有消息头和消息体组成. 消息体是不透明的,消息头是由一系列的可选属性组成

Publisher

消息的生产者,也是一个向交换器发布消息的客户端应用程序

Exchange

交换器,用来接收生产者发送的消息并且将消息路由给服务器中的队列. Exchange有四种类型:direct(默认), fanout,topic和header,不同类型的Exchange转发消息的策略有所区别

direct	**可以实现消息点对点发送的模式**
fanout,topic和header	**可以实现发布订阅模式**

Queue

消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。

Binding

绑定，用于消息队列和交换器之间的关联。一个绑定就是基于路由键将交换器和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。
Exchange 和Queue的绑定可以是多对多的关系。

Connection

网络连接，比如一个TCP连接。

Channel

信道，多路复用连接中的一条独立的双向数据流通道。信道是建立在真实的TCP连接内的虚拟连接，AMQP 命令都是通过信道发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说建立和销毁 TCP 都是非常昂贵的开销，所以引入了信道的概念，以复用一条 TCP 连接。

Consumer

消息的消费者，表示一个从消息队列中取得消息的客户端应用程序。

Virtual Host

虚拟主机，表示一批交换器、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个 vhost 本质上就是一个 mini 版的 RabbitMQ 服务器，拥有自己的队列、交换器、绑定和权限机制。vhost 是 AMQP 概念的基础，必须在连接时指定，RabbitMQ 默认的 vhost 是 / 。

Broker

表示消息队列服务器实体