SpringCache整合Redis

文章分类：JavaDemo； 标签：JavaCodeSnippet； 作者：Hackyle；

更新时间：Thu Feb 09 16:37:22 CST 2023

本文主要内容

- SpringCache规范的基本用法：都有哪些技术组成，代码语法是什么，怎么用？
- SpringCache整合Redis的一般步骤：SpringCache与Redis是通过什么建立起联系的，具体步骤是什么？

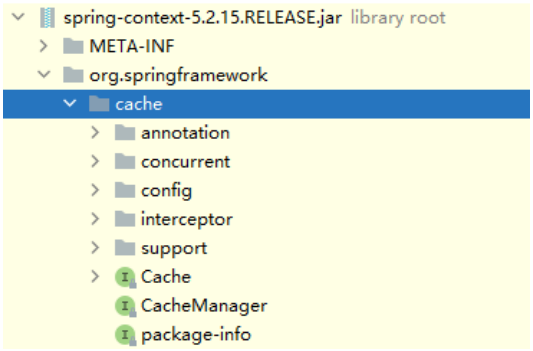
文章前置知识：SpringBoot、Redis

内容导览

SpringCache是一种规范

Spring 3.1开始，引入了Spring Cache，即Spring缓存抽象。

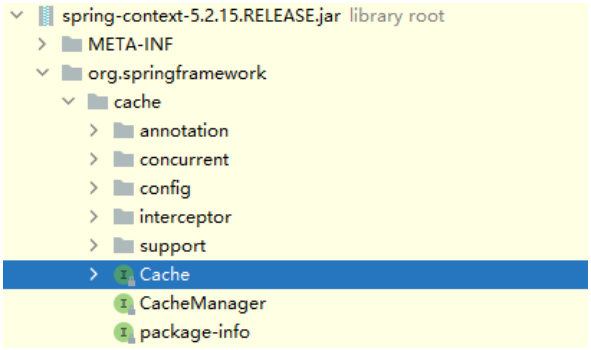
- 通过定义springframework.cache.Cache和org.springframework.cache.CacheManager接口来统一不同的缓存技术，并支持使用JCache注解简化开发过程。
- Cache接口：为缓存的组件规范定义，包含缓存的各种操作集合。
- CacheManager：指定缓存的底层实现。例如RedisCache，EhCacheCache，ConcurrentMapCache等。



源码所在位置：spring-context

Cache接口

Cache接口抽象了缓存的 get put evict 等相关操作。

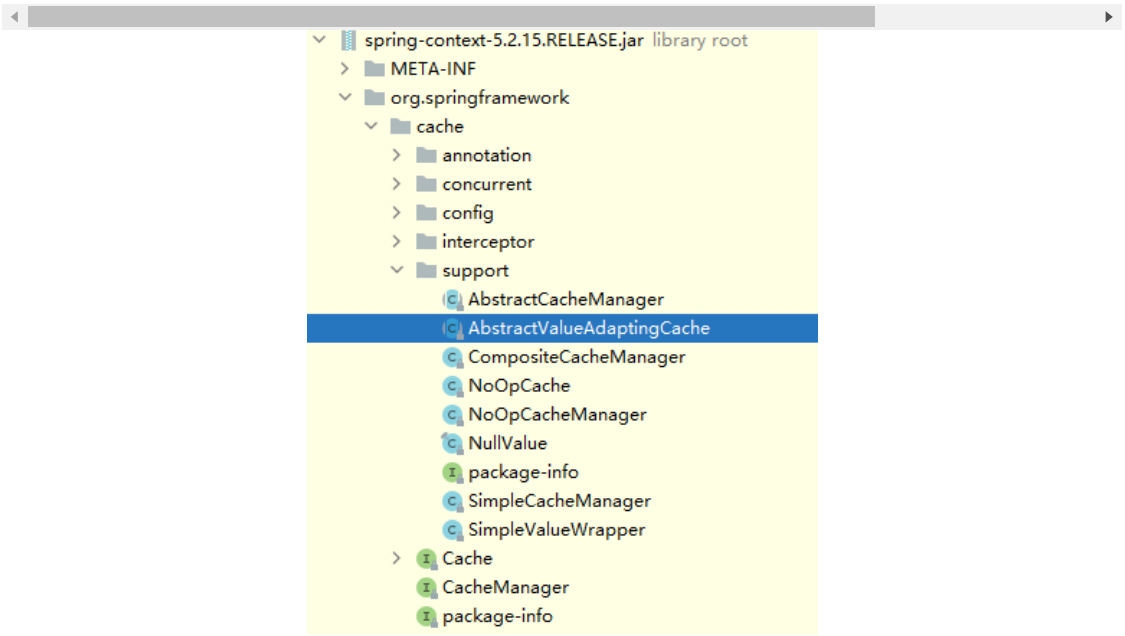


接口所在位置

接口规范的中文注释说明

```
public interface Cache {
    //Cache名称
    String getName();

    //Cache负责缓存的对象
    Object getNativeCache();

    /**
     * 获取key对应的ValueWrapper
     * 没有对应的key，则返回null
     * key对应的value是null，则返回null对应的ValueWrapper
     */
    @Nullable
    Cache.ValueWrapper get(Object key);

    //返回key对应type类型的value
    @Nullable
    <T> T get(Object key, @Nullable Class<T> type);

    //返回key对应的value，没有则缓存Callable::call，并返回
    @Nullable
    <T> T get(Object key, Callable<T> valueLoader);

    //缓存目标key-value（替换旧值），不保证实时性
    void put(Object key, @Nullable Object value);

    //插入缓存，并返回该key对应的value；先调用get，不存在则用put实现
    @Nullable
    default Cache.ValueWrapper putIfAbsent(Object key, @Nullable Object value) {
        Cache.ValueWrapper existingValue = this.get(key);
        if (existingValue == null) {
            this.put(key, value);
        }

        return existingValue;
    }

    //删除缓存，不保证实时性
    void evict(Object key);

    //立即删除缓存：返回false表示剔除前不存在制定key活不确定是否存在；返回true，表示该ke
    default boolean evictIfPresent(Object key) {
        this.evict(key);
        return false;
    }

```

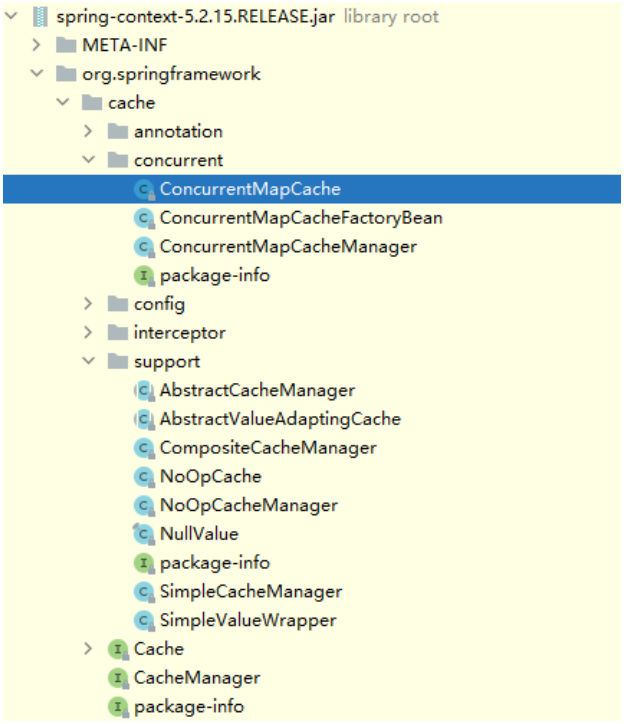
```
48      //清除所有缓存，不保证实时性
49      void clear();
50
51      //立即清楚所有缓存，返回false表示清除前没有缓存或不能确定是否有；返回true表示清除前有
52      default boolean invalidate() {
53          this.clear();
54          return false;
55      }
56
57      public static class ValueRetrievalException extends RuntimeException {
58          @Nullable
59          private final Object key;
60
61          public ValueRetrievalException(@Nullable Object key, Callable<?> loader, T
62              super(String.format("Value for key '%s' could not be loaded using '%s'
63              this.key = key;
64          }
65
66          @Nullable
67          public Object getKey() {
68              return this.key;
69          }
70      }
71
72      //缓存值的一个包装器接口，实现类为SimpleValueWrapper
73      @FunctionalInterface
74      public interface ValueWrapper {
75          @Nullable
76          Object get();
77      }
    }
```



抽象类AbstractValueAdaptingCache实现了Cache接口，主要抽象了对NULL值的处理逻辑。

- allowNullValues属性表示是否允许处理NULL值的缓存
- fromStoreValue方法处理NULL值的get操作，在属性allowNullValues为true的情况下，将NullValue处理为NULL
- toStoreValue方法处理NULL值得put操作，在属性allowNullValues为true的情况下，将NULL处理为NullValue，否则抛出异常
- toValueWrapper方法提供Cache接口中get方法的默认实现，从缓存中读取值，再通过fromStoreValue转化，最后包装为SimpleValueWrapper返回
- ValueWrapper get(Object key)和T get(Object key, @Nullable Classtype)方法基于上述方法实现
- ValueWrapper get(Object key)和@Nullable Classtype)方法基于上述方法实现

- lookup抽象方法用于给子类获取真正的缓存值

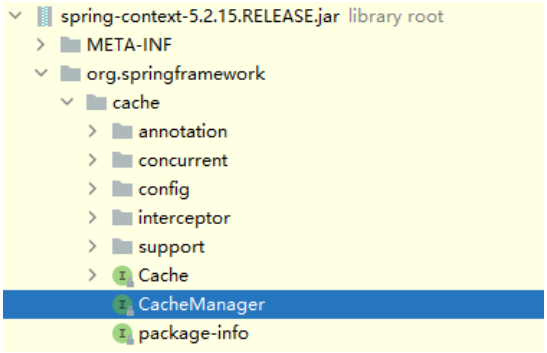


- ConcurrentMapCache继承了抽象类AbstractValueAdaptingCache，是Spring Cache的默认缓存实现。
- 它支持对缓存对象copy的缓存，由SerializationDelegate serialization 处理序列化，默认为 null 即基于引用的缓存。
- 缓存相关操作基于基类 AbstractValueAdaptingCache 的 null 值处理，默认允许为 null。

CacheManager接口

核心功能

- 指定底层的缓存技术（如ConcurrentHashMap、Redis等）
- 为换组分组，统一管理

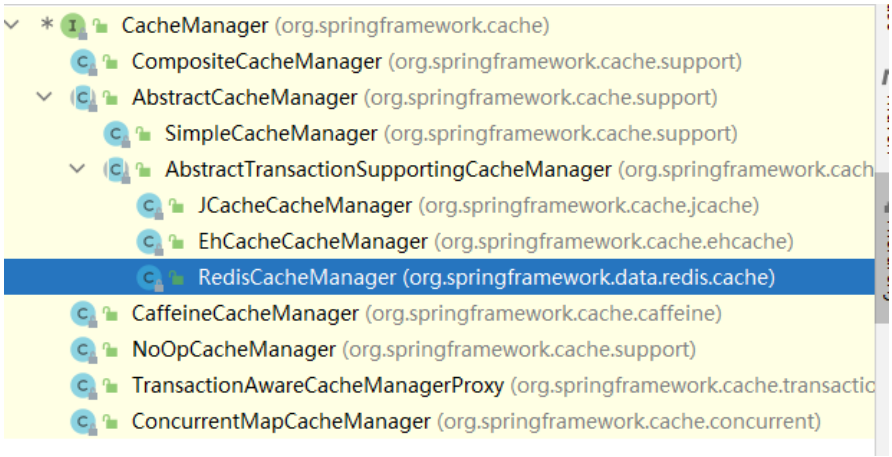


CacheManager接口所在包

- 功能点一：CacheManager 基于 name 管理一组 Cache。
- 功能点二：诸多缓存的底层实现
  - ConcurrentMapCacheManager：内置默认的实现，基于Map
  - AbstractCacheManager：可以定继承该抽象类，实现我们自己的底层实现。例如，RedisCacheManager是Redis依赖包提供的相关实现

缓存的实现

- EhCacheCacheManager 使用EhCache作为缓存的实现
- JCacheCacheManageer 使用JCache作为缓存的实现
- GuavaCacheManager 使用Google的GuavaCache作为缓存的实现
- RedisCacheManager 使用Redis作为缓存的实现



注解

@EnableCaching：开启SpringCache，一般在缓存核心配置类上使用

@CacheConfig

- 定义在类上，通常使用cacheNames。定义之后，该类下的所有含缓存注解的key之前都会拼接其属性值（附带两个：：）。
- 适用于某一个service的实现类或者mapper。
- 注意：如果service和mapper同时使用该注解并指定cacheNames，则以service上指定的cacheNames为准

@Caching注解中包含了@Cacheable、@CachePut和@CacheEvict注解，可以同时指定多个缓存规则。

Key的命名

- 缓存数据时，默认以类名+方法名+参数以键，以方法的返回值为value进行缓存
- 当然，key可以自定义

@Cacheable

功能:

- 用于方法上，待方法运行结束时，缓存该方法的返回值
- 每次执行该方法前，会先去缓存中查有没有相同条件下，缓存的数据，有的话直接拿缓存的数据，没有的话执行方法，并将执行结果返回。
- 默认以类名+方法名+参数为key，返回值为value
- 用于查询操作的方法上

实例

```
@Cacheable(cacheNames = {"emp"}, key = "#id", conditon = "mid>0", unless = "#resul
public Employee getEmp(Integer id) {
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}
```

参数释义

- cacheNames/value（指定缓存组件的名字，可以指定多个）
  - 会覆盖放在类上的@ConfigConfig{cacheNames}数据
- key：缓存数据时使用的key
  - 默认使用方法参数的值（类名+方法名+参数），也可以自定义
  - 可以为null
  - 可以通过SpEL进行自定义
- keyGenerator（key的生成器，可以自定义，key与keyGenerator二选一）
- cacheManager（指定缓存管理器，或者使用cacheResolver指定获取解析器）
- condition：符合条件才缓存
- unless（符合条件则不缓存，可以获取方法运行结果进行判断）
  - condition默认为true，unless默认为false。
  - condition为false时，unless为true。不被缓存
  - condition为false，unless为false。不被缓存

    - condition为true，unless为true。 不被缓存
    - conditon为true，unless为false。缓存
- sync（是否使用异步模式，不可与unless一起使用）
    - 在一个多线程的环境中，某些操作可能被相同的参数并发地调用，这样同一个 value 值可能被多次计算（或多次访问 db），这样就达不到缓存的目的。
    - 针对这些可能高并发的操作，我们可以使用 sync 参数来告诉底层的缓存提供者将缓存的入口锁住，这样就只能有一个线程计算操作的结果值，而其它线程需要等待，这样就避免了 n-1 次数据库访问。

@CachePut

@CachePut注解先调用目标方法，然后再缓存目标方法的结果。用于新增、更新操作的方法上。

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

@CacheEvict

功能

- 删除缓存，每次调用它注解的方法，就会执行删除指定的缓存
- 用于删除操作的方法上

```
@CacheEvict(value = "emp", key = "#id", allEntries = true)
public void deleteEmp(Integer id) {
    employeeMapper.deleteEmpById(id);
}
```

参数释义

- allEntries：默认为false，为true时，表示清空该cachename下的所有缓存
- beforeInvocation：默认为false，为true时，先删除缓存，再删除数据库。

SpringCache默认实现的示例

默认情况下，Spring使用ConcurrentMapCacheManager来实现缓存。

数据直接存储在内存中，当项目重启后，所有的缓存数据都消失了。对于中大型项目非常不合适。

POM

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

开启SpringCache

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.context.ApplicationPidFileWriter;
import org.springframework.cache.annotation.EnableCaching;

@EnableCaching //开启SpringCache
@SpringBootApplication
public class BootApp {
    public static void main(String[] args) {
        SpringApplication.run(BootApp.class, args);

```

```
    }
}
```

CacheService

```
import org.springframework.cache.annotation.CacheConfig;
import org.springframework.cache.annotation.CacheEvict;
import org.springframework.cache.annotation.CachePut;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

//该类下的所有含缓存注解的key之前都会拼接其属性值（附带两个：：）
@CacheConfig(cacheNames = "cacheService")
@Service
public class CacheService {

    /**
     * 每次请求，先看缓存中有没有，如果没有再执行方法
     * #p0：表示取方法入参的第一个参数
     */
    //@Cacheable(key = "'id:'+#p0",condition = "#id>8") 等价于下一行代码
    @Cacheable(key = "'id:'+#id",condition = "#id>8")
    public String findOne(Long id) {
        System.out.println("Cacheable 第二次访问，如果走了缓存，就不会显示这句");

        return "Cacheable-cacheKey=cacheService::id:" + id;
    }

    /**
     * 先走删除缓存，再执行方法
     * allEntries置为true表示删除所有缓存
     * beforeInvocation = true，将删除缓存行为在方法执行之前
     */
    @CacheEvict(key = "'id:'+#id",beforeInvocation =true)
    public String deleteOne(Long id) {
        System.out.println("模拟删除了一条记录");

        return "CacheEvict-cacheKey=cacheService::id:" + id;
    }

    /**
     * 先调用目标方法，然后再缓存目标方法的结果
     */
    @CachePut(key = "'id:'+#p0")
    public String updateById(Long id) {
        System.out.println("模拟进行了数据库更新操作");

        return "CachePut-cacheKey=cacheService::id:" + id;
    }
}
```

测试

```

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.cache.Cache;
import org.springframework.cache.concurrent.ConcurrentMapCacheManager;
```

```
7
8  import java.util.ArrayList;
9  import java.util.Collection;
10 import java.util.List;
11
12 @SpringBootTest
13 class CacheServiceTest {
14
15     @Autowired
16     private CacheService cacheService;
17     @Autowired
18     private ConcurrentMapCacheManager concurrentMapCacheManager;
19
20     public void getCacheData(List<String> keys) {
21         System.out.println("———获取ConcurrentMapCacheManager中的所有缓存数据———");
22         Collection<String> cacheNames = concurrentMapCacheManager.getCacheNames();
23         for (String cacheName : cacheNames) {
24             System.out.println(cacheName);
25             Cache cache = concurrentMapCacheManager.getCache(cacheName);
26             assert cache != null;
27             for (String key : keys) {
28                 String val = cache.get(key).get().toString();
29                 System.out.println(">>>key=" + key + " val=" + val);
30             }
31         }
32     }
33
34     @Test
35     public void testFindOne() {
36         for (int i = 0; i < 5; i++) {
37             String result = cacheService.findOne(111L);
38             System.out.println(result);
39         }
40
41         //显示缓存信息
42         List<String> keyList = new ArrayList<>();
43         keyList.add("id:111");
44         getCacheData(keyList);
45     }
46
47     @Test
48     public void testDeleteOne() {
49         String result = cacheService.deleteOne(111L);
50         System.out.println(result);
51
52         //显示缓存信息
53         List<String> keyList = new ArrayList<>();
54         keyList.add("id:111");
55         getCacheData(keyList);
56     }
57
58     @Test
59     public void testUpdateById() {
60         for (int i = 0; i < 5; i++) {
61             String result = cacheService.updateById(111L + i);
62             System.out.println(result);
63         }
64
65         //显示缓存信息
66         List<String> keyList = new ArrayList<>();
```

```
67          keyList.add("id:111");
68          keyList.add("id:112");
69          keyList.add("id:113");
70          keyList.add("id:114");
71          keyList.add("id:115");
72          getCacheData(keyList);
73      }
    }
```

```
Tests passed: 1 of 1 test – 224 ms

Cacheable 第二次访问，如果走了缓存，就不会显示这句
Cacheable-cacheKey=cacheService::id:111
Cacheable-cacheKey=cacheService::id:111
Cacheable-cacheKey=cacheService::id:111
Cacheable-cacheKey=cacheService::id:111
——获取ConcurrentMapCacheManager中的所有缓存数据——
cacheService
>>>key=id:111 val=Cacheable-cacheKey=cacheService::id:111
```

```
Tests failed: 1 of 1 test – 218 ms

模拟删除了一条记录
CacheEvict-cacheKey=cacheService::id:111
——获取ConcurrentMapCacheManager中的所有缓存数据——
cacheService

java.lang.NullPointerException Create breakpoint
    at com.ks.boot.service.CacheServiceTest.getCacheData(CacheServiceTest.java:29)
    at com.ks.boot.service.CacheServiceTest.testDeleteOne(CacheServiceTest.java:56) <31 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <9 internal calls>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1540) <21 internal calls>
```

因为删除了该个Key的缓存，所有发生了NPE

```
Tests passed: 1 of 1 test – 220 ms

模拟进行了数据库更新操作
CachePut-cacheKey=cacheService::id:111
模拟进行了数据库更新操作
CachePut-cacheKey=cacheService::id:112
模拟进行了数据库更新操作
CachePut-cacheKey=cacheService::id:113
模拟进行了数据库更新操作
CachePut-cacheKey=cacheService::id:114
模拟进行了数据库更新操作
CachePut-cacheKey=cacheService::id:115
——获取ConcurrentMapCacheManager中的所有缓存数据——
cacheService
>>>key=id:111 val=CachePut-cacheKey=cacheService::id:111
>>>key=id:112 val=CachePut-cacheKey=cacheService::id:112
>>>key=id:113 val=CachePut-cacheKey=cacheService::id:113
>>>key=id:114 val=CachePut-cacheKey=cacheService::id:114
>>>key=id:115 val=CachePut-cacheKey=cacheService::id:115
```

整合Redis的一般步骤

整合Redis

Redis与SpringCache的关系?

- SpringCache是Spring对缓存的一种规范
- Redis才是真正进行缓存的具体工具

- 可以类比： JDBC规范与实现该规范的MySQL驱动（mysql.cj.jdbc.Driver）一样

如何整合SpringCache与Redis?

1. 准备Redis环境：POM依赖、YML配置项、在配置类里注入RedisTemplate
2. 准备SpringCache环境：POM、YML配置项
3. 两者是如何建立起联系的？**指定CacheManager为RedisCacheManager**
    1. 写一个配置类继承于springframework.cache.annotation.CachingConfigurerSupport
    2. 重写cacheManager()或cacheResolver()方法：指定缓存的具体实现为RedisCacheManager
    3. 重写keyGenerator()方法：指定缓存Key的生成策略
    4. 重写errorHandler()方法：指定缓存出错的处理逻辑
4. 在需要进行缓存操作的方法上使用合适的注解

整合Redis环境

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

```yaml
spring:
  application:
    name: springcache-redis
  redis:
    host: 127.0.0.1
    port: 6379
    password:  #Redis服务器连接密码（默认为空）
    timeout: 30000 #连接超时时间（毫秒）
    jedis:
      pool:
        max-active: 20 # 连接池最大连接数（使用负值表示没有限制）
        max-wait: -1 # 连接池最大阻塞等待时间（使用负值表示没有限制）
        max-idle: 10 # 连接池中的最大空闲连接
        min-idle: 0 # 连接池中的最小空闲连接
```

写一个配置类，注入RedisTemplate

```java

@Configuration
public class RedisConfig {
    /**
     * RedisTemplate配置
     * 注意：对Value的序列化是以二进制的形式存储于内存，如果想要直接看到中文等非ASCII数据
     */
    @Bean
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory fact
        RedisTemplate<String, Object> template = new RedisTemplate<>();
        // 配置连接工厂
        template.setConnectionFactory(factory);

        //使用Jackson2JsonRedisSerializer来序列化和反序列化redis的value值（默认使用JD
        Jackson2JsonRedisSerializer<Object> jsonRedisSerializer = new Jackson2Json

        ObjectMapper om = new ObjectMapper();
        // 指定要序列化的域，field,get和set,以及修饰符范围，ANY是都有包括private和publ
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
        // 指定序列化输入的类型，类必须是非final修饰的，final修饰的类，比如String,Integ
        //om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
        om.activateDefaultTyping(LaissezFaireSubTypeValidator.instance, ObjectMapp
        jsonRedisSerializer.setObjectMapper(om);

```

```
        // 值采用json序列化
        template.setValueSerializer(jsonRedisSerializer);
        //使用StringRedisSerializer来序列化和反序列化redis的key值
        template.setKeySerializer(new StringRedisSerializer());

        // 设置hash的key和value序列化模式
        template.setHashKeySerializer(new StringRedisSerializer());
        template.setHashValueSerializer(jsonRedisSerializer);
        template.afterPropertiesSet();

        return template;
    }
}
```

整合SpringCache

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

```
spring:
  cache:
    type: redis
    redis:
      time-to-live: 3600000 #设置缓存过期时间
      key-prefix: CACHE_ #指定默认前缀，如果此处我们指定了前缀则使用我们指定的前缀，推荐
      cache-null-values: true #是否缓存空值，防止缓存穿透
```

SpringCache配置类（SpringCacheRedisConfig）

- 继承于springframework.cache.annotation.CachingConfigurerSupport
- 重写cacheManager()或cacheResolver()方法：指定缓存的具体实现，因为要整合Redis，所以返回的CacheManager为RedisCacheManager
- 重写keyGenerator()方法：指定缓存Key的生成策略
- 重写errorHandler()方法：指定缓存出错的处理逻辑

```
@Configuration
@EnableCaching // 开启springCache
public class SpringCacheRedisConfig extends CachingConfigurerSupport {
    private static final Logger LOGGER = LoggerFactory.getLogger(SpringCacheRedisC

    /**
     * 缓存管理器：指定使用那种缓存的具体实现，方式一
     *
     * 只有CacheManger才能扫描到cacheable注解
     * Value使用Jackson工具序列化
     */
    @Bean
    public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {
        RedisCacheManager cacheManager = RedisCacheManager.RedisCacheManagerBuilde
                .fromConnectionFactory(connectionFactory) //Redis链接工厂
                //缓存配置 通用配置  默认存储一小时
                .cacheDefaults(getCacheConfigurationWithTtl(Duration.ofHours(1)))
                //配置同步修改或删除  put/evict
                .transactionAware()
                //对于不同的cacheName我们可以设置不同的过期时间
```

```
                //.withCacheConfiguration("app:",getCacheConfigurationWithTtl(Dura
                //.withCacheConfiguration("user:",getCacheConfigurationWithTtl(Dur
                .build();
        return cacheManager;
    }
    private RedisCacheConfiguration getCacheConfigurationWithTtl(Duration duration
        return RedisCacheConfiguration
                .defaultCacheConfig()
                //设置key value的序列化方式
                // 设置key为String
                .serializeKeysWith(RedisSerializationContext.SerializationPair.fro
                // 设置value 为自动转Json的Object
                .serializeValuesWith(RedisSerializationContext.SerializationPair.f
                // 不缓存null
                .disableCachingNullValues()
                // 设置缓存的过期时间
                .entryTtl(duration);
    }

    /**
     * 缓存管理器：指定使用那种缓存的具体实现，方式二
     *
     * Value使用FastJSON序列化，需导入fastjson依赖包
     */
    //@Bean
    //public RedisCacheConfiguration redisCacheConfiguration() {
    //    FastJsonRedisSerializer<Object> fastJsonRedisSerializer = new FastJsonRe
    //    RedisCacheConfiguration config = RedisCacheConfiguration.defaultCacheCon
    //    config = config
    //            .serializeValuesWith(RedisSerializationContext.SerializationPair
    //            .entryTtl(Duration.ofHours(2)); //默认有效时间为2h
    //    return config;
    //}
}
```

缓存业务

```

@CacheConfig(cacheNames = "cacheDemo")
@Service
public class CacheDemoService {

    /**
     * 先调用目标方法，然后再缓存目标方法的结果
     * #p0：表示取方法入参的第一个参数，并取属性为uid的值
     */
    //@CachePut(key = "'uid:'+#p0.uid") 等价于下一行代码
    @CachePut(key = "'uid:'+#userinfo.uid")
    public String saveUser(UserinfoDto userinfo) {
        //模拟保存、更新操作操作
        System.out.println("模拟落库操作，第二次请求如果走缓存，则不会执行到这句。");

        return "保存、更新操作：cacheKey=uid:"+userinfo.getUid()+" ;userinfo="+user
    }

    @CacheEvict(key = "'uid:' + #p0")
    public String deleteUser(long uid) {
        System.out.println("模拟删除操作，第二次请求如果走缓存，则不会执行到这句。");

```

```
        return "删除操作：cacheKey=uid:" +uid +" ;uid="+uid;
    }

    /**
     * 用于方法上，可以将方法的运行结果进行缓存，之后就不用调用方法了，直接从缓存中取值即
     * #p0：表示取方法入参的第一个参数
     */
    //@Cacheable(key = "'uid:'+#p0") 等价于下一行代码
    @Cacheable(key = "'uid:'+#uid")
    //@Cacheable(cacheNames = {"uid:"}, key = "'uid:'+#uid") cacheNames为本个缓存指
    public String queryUserById(long uid) {
        //模拟查库
        UserinfoDto userinfo = new UserinfoDto();
        userinfo.setUid(uid);
        userinfo.setName("Kyle");
        userinfo.setGender("man");
        System.out.println("模拟查库操作，第二次请求如果走缓存，则不会执行到这句。");

        return "查询结果：cacheKey=uid:" +uid+ " ;uid="+uid+" ;userinfo="+userinfo;
    }
}
```

测试

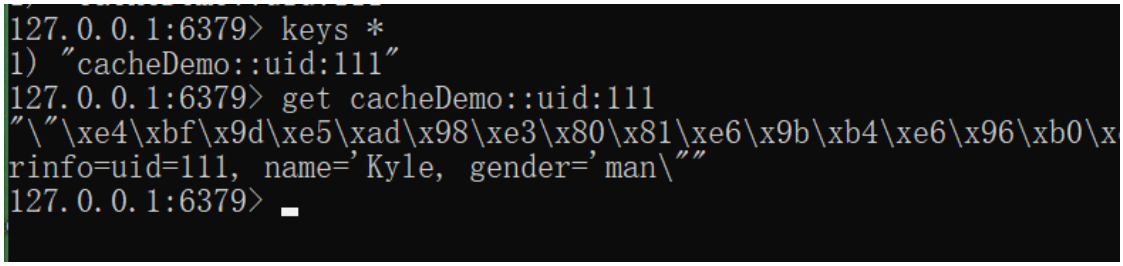
```
@RestController
public class CacheDemoController {
    @Autowired
    private CacheDemoService cacheDemoService;

    @GetMapping("/testCachePut")
    public String testCachePut() {
        UserinfoDto userinfo = new UserinfoDto();
        userinfo.setUid(111);
        userinfo.setName("Kyle");
        userinfo.setGender("man");

        return cacheDemoService.saveUser(userinfo);
    }

    @GetMapping("/testCacheEvict")
    public String testCacheEvict() {
        return cacheDemoService.deleteUser(111);
    }

    @GetMapping("/testCacheable")
    public String testCacheable() {
        return cacheDemoService.queryUserById(333);
    }
}
```

删除Redis中的所有数据

```
127.0.0.1:6379> flushall
OK
127.0.0.1:6379> keys *
(empty list or set)
127.0.0.1:6379>
```

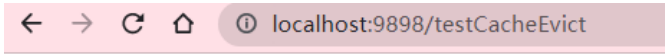
测试CachePut

保存、更新操作：cacheKey=uid:111 ;userinfo=uid=111, name='Kyle, gender='man

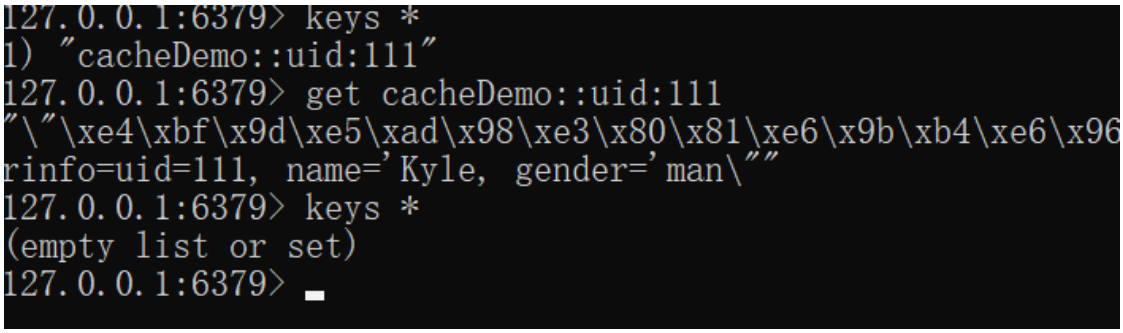


Redis中已存入相关数据

测试CacheEvict



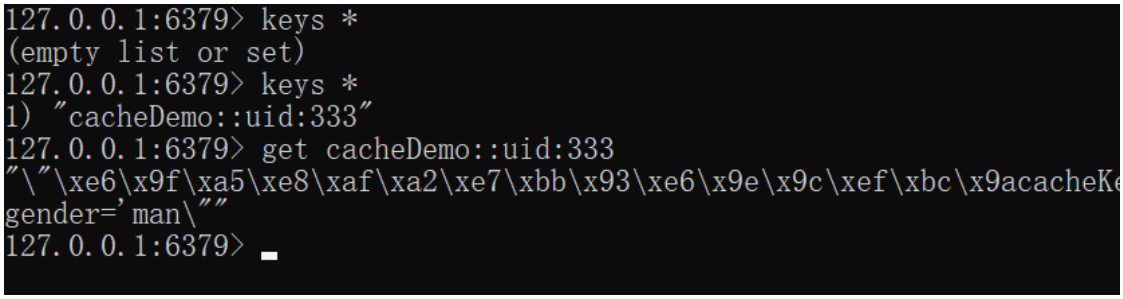
删除操作：cacheKey=uid:111 ;uid=111



缓存已删除

测试Cacheable

查询结果：cacheKey=uid:333 ;uid=333 ;userinfo=uid=333, name='Kyle, gender='man



缓存成功

留下你的评论

Name: Input your name, please

Email: Input your email, please

Link: Input your phone or website, please

File Edit View Format Tools Table Help

Input comment, please

p 0 words tiny

SUBMIT RESET