## 一、通用缓存接口

### 1、缓存基础算法

**FIFO（First In First Out）**，先进先出，和OS里的FIFO思路相同，如果一个数据最先进入缓存中，当缓存满的时候，应当把最先进入缓存的数据给移除掉。

**LFU（Least Frequently Used）**，最不经常使用，如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小。

**LRU（Least Recently Used）**，最近最少使用，如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据移除。

### 2、接口定义

简单定义缓存接口，大致可以抽象如下：

package com.power.demo.cache.contract;  
  
import java.util.function.Function;  
  
/\*\*  
 \* 缓存提供者接口  
 \*\*/  
public interface CacheProviderService {  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    <T extends Object> T get(String key);  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \*\*/  
    <T extends Object> T get(String key, Function<String, T> function);  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm function函数的调用参数  
     \*\*/  
    <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm);  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    <T extends Object> T get(String key, Function<String, T> function, Long expireTime);  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm   function函数的调用参数  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm, Long expireTime);  
  
    /\*\*  
     \* 设置缓存键值  
     \*  
     \* @param key 缓存键 不可为空  
     \* @param obj 缓存值 不可为空  
     \*\*/  
    <T extends Object> void set(String key, T obj);  
  
    /\*\*  
     \* 设置缓存键值  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param obj        缓存值 不可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    <T extends Object> void set(String key, T obj, Long expireTime);  
  
    /\*\*  
     \* 移除缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    void remove(String key);  
  
    /\*\*  
     \* 是否存在缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    boolean contains(String key);  
}

*注意，这里列出的只是常见缓存功能接口，一些在特殊场景下用到的统计类的接口、分布式锁、自增（减）等功能不在讨论范围之内。*

Get相关方法，注意多个参数的情况，缓存接口里面传人的Function，这是Java8提供的函数式接口，虽然支持的入参个数有限（这里你会非常怀念.NET下的Func委托），但是仅对Java这个语言来说，这真是一个重大的进步^\_^。

接口定义好了，下面就要实现缓存提供者程序了。按照存储类型的不同，本文简单实现最常用的两种缓存提供者：**本地缓存和分布式缓存**。

## 二、本地缓存

本地缓存，也就是JVM级别的缓存(本地缓存可以认为是直接在进程内通信调用，而分布式缓存则需要通过网络进行跨进程通信调用)，一般有很多种实现方式，比如直接使用Hashtable、ConcurrentHashMap等天生线程安全的集合作为缓存容器，或者使用一些成熟的开源组件，如EhCache、Guava Cache等。本文选择上手简单的Guava缓存。

### 1、什么是Guava

Guava，简单来说就是一个开发类库，且是一个非常丰富强大的开发工具包，号称可以让使用Java语言更令人愉悦，主要包括基本工具类库和接口、缓存、发布订阅风格的事件总线等。在实际开发中，我用的最多的是集合、缓存和常用类型帮助类，很多人都对这个类库称赞有加。

### 2、添加依赖

  <**dependency**>  
            <**groupId**>com.google.guava</**groupId**>  
            <**artifactId**>guava</**artifactId**>  
        </**dependency**>

### 3、实现接口

package com.power.demo.cache.impl;  
  
import com.google.common.cache.Cache;  
import com.google.common.cache.CacheBuilder;  
import com.google.common.collect.Maps;  
import com.power.demo.cache.contract.CacheProviderService;  
import com.power.demo.common.AppConst;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.util.StringUtils;  
  
import java.util.Map;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
import java.util.function.Function;  
  
/\*  
 \* 本地缓存提供者服务 (Guava Cache)  
 \* \*/  
@Configuration  
@ComponentScan(basePackages = AppConst.BASE\_PACKAGE\_NAME)  
@Qualifier("localCacheService")  
public class LocalCacheProviderImpl implements CacheProviderService {  
  
    private static Map<String, Cache<String, Object>> \_cacheMap = Maps.newConcurrentMap();  
  
    static {  
  
        Cache<String, Object> cacheContainer = CacheBuilder.newBuilder()  
                .maximumSize(AppConst.CACHE\_MAXIMUM\_SIZE)  
                .expireAfterWrite(AppConst.CACHE\_MINUTE, TimeUnit.MILLISECONDS)//最后一次写入后的一段时间移出  
                //.expireAfterAccess(AppConst.CACHE\_MINUTE, TimeUnit.MILLISECONDS) //最后一次访问后的一段时间移出  
                .recordStats()//开启统计功能  
                .build();  
  
        \_cacheMap.put(String.valueOf(AppConst.CACHE\_MINUTE), cacheContainer);  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public <T extends Object> T get(String key) {  
        T obj = get(key, null, null, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function) {  
        T obj = get(key, function, key, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm function函数的调用参数  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm) {  
        T obj = get(key, function, funcParm, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function, Long expireTime) {  
        T obj = get(key, function, key, expireTime);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm   function函数的调用参数  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm, Long expireTime) {  
        T obj = null;  
        if (StringUtils.isEmpty(key) == true) {  
            return obj;  
        }  
  
        expireTime = getExpireTime(expireTime);  
  
        Cache<String, Object> cacheContainer = getCacheContainer(expireTime);  
  
        try {  
            if (function == null) {  
                obj = (T) cacheContainer.getIfPresent(key);  
            } else {  
                final Long cachedTime = expireTime;  
                obj = (T) cacheContainer.get(key, () -> {  
                    T retObj = function.apply(funcParm);  
                    return retObj;  
                });  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值  
     \*  
     \* @param key 缓存键 不可为空  
     \* @param obj 缓存值 不可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj) {  
  
        set(key, obj, AppConst.CACHE\_MINUTE);  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param obj        缓存值 不可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj, Long expireTime) {  
        if (StringUtils.isEmpty(key) == true) {  
            return;  
        }  
  
        if (obj == null) {  
            return;  
        }  
  
        expireTime = getExpireTime(expireTime);  
  
        Cache<String, Object> cacheContainer = getCacheContainer(expireTime);  
  
        cacheContainer.put(key, obj);  
    }  
  
    /\*\*  
     \* 移除缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public void remove(String key) {  
        if (StringUtils.isEmpty(key) == true) {  
            return;  
        }  
  
        long expireTime = getExpireTime(AppConst.CACHE\_MINUTE);  
  
        Cache<String, Object> cacheContainer = getCacheContainer(expireTime);  
  
        cacheContainer.invalidate(key);  
    }  
  
    /\*\*  
     \* 是否存在缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public boolean contains(String key) {  
        boolean exists = false;  
        if (StringUtils.isEmpty(key) == true) {  
            return exists;  
        }  
  
        Object obj = get(key);  
  
        if (obj != null) {  
            exists = true;  
        }  
  
        return exists;  
    }  
  
    private static Lock lock = new ReentrantLock();  
  
    private Cache<String, Object> getCacheContainer(Long expireTime) {  
  
        Cache<String, Object> cacheContainer = null;  
        if (expireTime == null) {  
            return cacheContainer;  
        }  
  
        String mapKey = String.valueOf(expireTime);  
  
        if (\_cacheMap.containsKey(mapKey) == true) {  
            cacheContainer = \_cacheMap.get(mapKey);  
            return cacheContainer;  
        }  
  
        try {  
            lock.lock();  
            cacheContainer = CacheBuilder.newBuilder()  
                    .maximumSize(AppConst.CACHE\_MAXIMUM\_SIZE)  
                    .expireAfterWrite(expireTime, TimeUnit.MILLISECONDS)//最后一次写入后的一段时间移出  
                    //.expireAfterAccess(AppConst.CACHE\_MINUTE, TimeUnit.MILLISECONDS) //最后一次访问后的一段时间移出  
                    .recordStats()//开启统计功能  
                    .build();  
  
            \_cacheMap.put(mapKey, cacheContainer);  
  
        } finally {  
            lock.unlock();  
        }  
  
        return cacheContainer;  
    }  
  
    /\*\*  
     \* 获取过期时间 单位：毫秒  
     \*  
     \* @param expireTime 传人的过期时间 单位毫秒 如小于1分钟，默认为10分钟  
     \*\*/  
    private Long getExpireTime(Long expireTime) {  
        Long result = expireTime;  
        if (expireTime == null || expireTime < AppConst.CACHE\_MINUTE / 10) {  
            result = AppConst.CACHE\_MINUTE;  
        }  
  
        return result;  
    }  
}

### 4、注意事项

Guava Cache初始化容器时，支持缓存过期策略，类似FIFO、LRU和LFU等算法。

expireAfterWrite：最后一次写入后的一段时间移出。

expireAfterAccess：最后一次访问后的一段时间移出。

Guava Cache对缓存过期时间的设置实在不够友好。常见的应用场景，比如，有些几乎不变的基础数据缓存1天，有些热点数据缓存2小时，有些会话数据缓存5分钟等等。

通常我们认为设置缓存的时候带上缓存的过期时间是非常容易的，而且只要一个缓存容器实例即可，比如.NET下的ObjectCache、System.Runtime.Cache等等。

但是Guava Cache不是这个实现思路，如果缓存的过期时间不同，Guava的CacheBuilder要初始化多份Cache实例。

好在我在实现的时候注意到了这个问题，并且提供了解决方案，可以看到getCacheContainer这个函数，根据过期时长做缓存实例判断，就算不同过期时间的多实例缓存也是完全没有问题的。

## 三、分布式缓存

分布式缓存产品非常多，本文使用应用普遍的Redis，在Spring Boot应用中使用Redis非常简单。

### 1、什么是Redis

Redis是一款开源（BSD许可）的、用C语言写成的高性能的键-值存储（key-value store）。它常被称作是一款数据结构服务器（data structure server）。它可以被用作缓存、消息中间件和数据库，在很多应用中，经常看到有人选择使用Redis做缓存，实现分布式锁和分布式Session等。作为缓存系统时，和经典的KV结构的Memcached非常相似，但又有很多不同。

Redis支持丰富的数据类型。Redis的键值可以包括字符串（strings）类型，同时它还包括哈希（hashes）、列表（lists）、集合（sets）和有序集合（sorted sets）等数据类型。对于这些数据类型，你可以执行原子操作。例如：对字符串进行附加操作（append）；递增哈希中的值；向列表中增加元素；计算集合的交集、并集与差集等。

**Redis的数据类型：**

Keys：非二进制安全的字符类型（ not binary-safe strings ），由于key不是binary safe的字符串，所以像“my key”和“mykey\n”这样包含空格和换行的key是不允许的。  
Values：Strings、Hash、Lists、 Sets、 Sorted sets。考虑到Redis单线程操作模式，Value的粒度不应该过大，缓存的值越大，越容易造成阻塞和排队。

为了获得优异的性能，Redis采用了内存中（in-memory）数据集（dataset）的方式。同时，Redis支持数据的持久化，你可以每隔一段时间将数据集转存到磁盘上（snapshot），或者在日志尾部追加每一条操作命令（append only file,aof）。

Redis同样支持主从复制（master-slave replication），并且具有非常快速的非阻塞首次同步（ non-blocking first synchronization）、网络断开自动重连等功能。

同时Redis还具有其它一些特性，其中包括简单的事物支持、发布订阅 （ pub/sub）、管道（pipeline）和虚拟内存（vm）等 。[后端开发都应该掌握的Redis基础](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247489241&idx=1&sn=16f86580a1fcab7d4375e2581bb15595&chksm=ebd629f5dca1a0e3dbd8405c4a3b6da49d134579742abb7276b63d4602db983770ca40e6c0a8&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/_blank)

### 2、添加依赖

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

### 3、配置Redis

在application.properties配置文件中，配置Redis常用参数：

## Redis缓存相关配置  
#Redis数据库索引（默认为0）  
spring.redis.database=0  
#Redis服务器地址  
spring.redis.host=127.0.0.1  
#Redis服务器端口  
spring.redis.port=6379    
#Redis服务器密码（默认为空）  
spring.redis.password=123321  
#Redis连接超时时间 默认：5分钟（单位：毫秒）  
spring.redis.timeout=300000ms  
#Redis连接池最大连接数（使用负值表示没有限制）  
spring.redis.jedis.pool.max-active=512  
#Redis连接池中的最小空闲连接  
spring.redis.jedis.pool.min-idle=0  
#Redis连接池中的最大空闲连接  
spring.redis.jedis.pool.max-idle=8  
#Redis连接池最大阻塞等待时间（使用负值表示没有限制）  
spring.redis.jedis.pool.max-wait=-1ms

常见的需要注意的是最大连接数（spring.redis.jedis.pool.max-active ）和超时时间（spring.redis.jedis.pool.max-wait）。Redis在生产环境中出现故障的频率经常和这两个参数息息相关。

接着定义一个继承自CachingConfigurerSupport（请注意cacheManager和keyGenerator这两个方法在子类的实现）的RedisConfig类：

package com.power.demo.cache.config;  
  
import org.springframework.cache.CacheManager;  
import org.springframework.cache.annotation.CachingConfigurerSupport;  
import org.springframework.cache.annotation.EnableCaching;  
import org.springframework.context.annotation.Bean;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.redis.cache.RedisCacheManager;  
import org.springframework.data.redis.connection.RedisConnectionFactory;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.data.redis.serializer.StringRedisSerializer;  
  
/\*\*  
 \* Redis缓存配置类  
 \*/  
@Configuration  
@EnableCaching  
public class RedisConfig extends CachingConfigurerSupport {  
  
    @Bean  
    public CacheManager cacheManager(RedisConnectionFactory connectionFactory) {  
        return RedisCacheManager.create(connectionFactory);  
    }  
  
    @Bean  
    public RedisTemplate<String, Object> redisTemplate(RedisConnectionFactory factory) {  
        RedisTemplate<String, Object> template = new RedisTemplate<>();  
  
        //Jedis的Key和Value的序列化器默认值是JdkSerializationRedisSerializer  
        //经实验，JdkSerializationRedisSerializer通过RedisDesktopManager看到的键值对不能正常解析  
  
        //设置key的序列化器  
        template.setKeySerializer(new StringRedisSerializer());  
  
        ////设置value的序列化器  默认值是JdkSerializationRedisSerializer  
        //使用Jackson序列化器的问题是，复杂对象可能序列化失败，比如JodaTime的DateTime类型  
  
        //        //使用Jackson2，将对象序列化为JSON  
        //        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer(Object.class);  
        //        //json转对象类，不设置默认的会将json转成hashmap  
        //        ObjectMapper om = new ObjectMapper();  
        //        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);  
        //        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON\_FINAL);  
        //        jackson2JsonRedisSerializer.setObjectMapper(om);  
        //        template.setValueSerializer(jackson2JsonRedisSerializer);  
  
        //将redis连接工厂设置到模板类中  
        template.setConnectionFactory(factory);  
  
        return template;  
    }  
  
//    //自定义缓存key生成策略  
//    @Bean  
//    public KeyGenerator keyGenerator() {  
//        return new KeyGenerator() {  
//            @Override  
//            public Object generate(Object target, java.lang.reflect.Method method, Object... params) {  
//                StringBuffer sb = new StringBuffer();  
//                sb.append(target.getClass().getName());  
//                sb.append(method.getName());  
//                for (Object obj : params) {  
//                    if (obj == null) {  
//                        continue;  
//                    }  
//                    sb.append(obj.toString());  
//                }  
//                return sb.toString();  
//            }  
//        };  
//    }  
}

在RedisConfig这个类上加上@EnableCaching这个注解，这个注解会被Spring发现，并且会创建一个切面（aspect） 并触发Spring缓存注解的切点（pointcut）。据所使用的注解以及缓存的状态，这个切面会从缓存中获取数据，将数据添加到缓存之中或者从缓存中移除某个值。

cacheManager方法，申明一个缓存管理器（CacheManager）的bean，作用就是@EnableCaching这个切面在新增缓存或者删除缓存的时候会调用这个缓存管理器的方法。keyGenerator方法，可以根据需求自定义缓存key生成策略。

而redisTemplate方法，则主要是设置Redis模板类，比如键和值的序列化器（从这里可以看出，Redis的键值对必须可序列化）、redis连接工厂等。[RedisTemplate实现轻量级消息队列](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247490938&idx=1&sn=75f8aa6a89e9aa8152575eabe00dd94a&chksm=ebd62256dca1ab40ac5e03b1fa2ee3a7b0c82a8980ef1bd90b58acb2692047fcd3a8f38347f7&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/_blank)

RedisTemplate支持的序列化器主要有如下几种：

**JdkSerializationRedisSerializer：**使用Java序列化；

**StringRedisSerializer：**序列化String类型的key和value；

**GenericToStringSerializer：**使用Spring转换服务进行序列化；

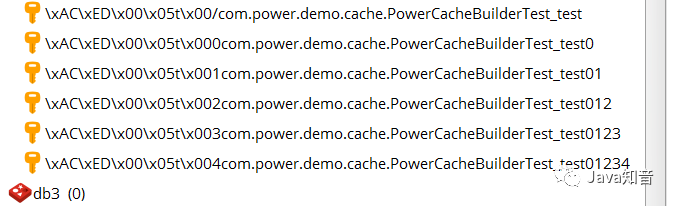
**JacksonJsonRedisSerializer：**使用Jackson 1，将对象序列化为JSON；

**Jackson2JsonRedisSerializer：**使用Jackson 2，将对象序列化为JSON；

**OxmSerializer：**使用Spring O/X映射的编排器和解排器（marshaler和unmarshaler）实现序列化，用于XML序列化；

*注意：RedisTemplate的键和值序列化器，默认情况下都是JdkSerializationRedisSerializer，它们都可以自定义设置序列化器。*

推荐将字符串键使用StringRedisSerializer序列化器，因为运维的时候好排查问题，JDK序列化器的也能识别，但是可读性稍差(是因为缓存服务器没有JRE吗？)，见如下效果：



而值序列化器则要复杂的多，很多人推荐使用Jackson2JsonRedisSerializer序列化器，但是实际开发过程中，经常有人碰到反序列化错误，经过排查多数都和Jackson2JsonRedisSerializer这个序列化器有关。[序列化框架的选型和比对](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247489315&idx=2&sn=74dca4b265a58c14caf68c3ec2f62bba&chksm=ebd6280fdca1a1191f2f76e9810add183a7b7c8c9283ce652b941bc06568c1118bb3c8798526&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/_blank)

### 4、实现接口

使用RedisTemplate，在Spring Boot中调用Redis接口比直接调用Jedis简单多了。

package com.power.demo.cache.impl;  
  
import com.power.demo.cache.contract.CacheProviderService;  
import com.power.demo.common.AppConst;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.data.redis.core.RedisTemplate;  
import org.springframework.data.redis.core.ValueOperations;  
import org.springframework.util.StringUtils;  
  
import javax.annotation.Resource;  
import java.io.Serializable;  
import java.util.concurrent.TimeUnit;  
import java.util.function.Function;  
  
@Configuration  
@ComponentScan(basePackages = AppConst.BASE\_PACKAGE\_NAME)  
@Qualifier("redisCacheService")  
public class RedisCacheProviderImpl implements CacheProviderService {  
  
    @Resource  
    private RedisTemplate<Serializable, Object> redisTemplate;  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public <T extends Object> T get(String key) {  
        T obj = get(key, null, null, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function) {  
        T obj = get(key, function, key, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm function函数的调用参数  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm) {  
        T obj = get(key, function, funcParm, AppConst.CACHE\_MINUTE);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function, Long expireTime) {  
        T obj = get(key, function, key, expireTime);  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm   function函数的调用参数  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm, Long expireTime) {  
        T obj = null;  
        if (StringUtils.isEmpty(key) == true) {  
            return obj;  
        }  
  
        expireTime = getExpireTime(expireTime);  
  
        try {  
  
            ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();  
            obj = (T) operations.get(key);  
            if (function != null && obj == null) {  
                obj = function.apply(funcParm);  
                if (obj != null) {  
                    set(key, obj, expireTime);//设置缓存信息  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值  
     \*  
     \* @param key 缓存键 不可为空  
     \* @param obj 缓存值 不可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj) {  
  
        set(key, obj, AppConst.CACHE\_MINUTE);  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入值，这会直接覆盖掉给定键之前映射的值  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param obj        缓存值 不可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj, Long expireTime) {  
        if (StringUtils.isEmpty(key) == true) {  
            return;  
        }  
  
        if (obj == null) {  
            return;  
        }  
  
        expireTime = getExpireTime(expireTime);  
  
        ValueOperations<Serializable, Object> operations = redisTemplate.opsForValue();  
  
        operations.set(key, obj);  
  
        redisTemplate.expire(key, expireTime, TimeUnit.MILLISECONDS);  
    }  
  
    /\*\*  
     \* 移除缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public void remove(String key) {  
        if (StringUtils.isEmpty(key) == true) {  
            return;  
        }  
  
        redisTemplate.delete(key);  
    }  
  
    /\*\*  
     \* 是否存在缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public boolean contains(String key) {  
        boolean exists = false;  
        if (StringUtils.isEmpty(key) == true) {  
            return exists;  
        }  
  
        Object obj = get(key);  
  
        if (obj != null) {  
            exists = true;  
        }  
  
        return exists;  
    }  
  
    /\*\*  
     \* 获取过期时间 单位：毫秒  
     \*  
     \* @param expireTime 传人的过期时间 单位毫秒 如小于1分钟，默认为10分钟  
     \*\*/  
    private Long getExpireTime(Long expireTime) {  
        Long result = expireTime;  
        if (expireTime == null || expireTime < AppConst.CACHE\_MINUTE / 10) {  
            result = AppConst.CACHE\_MINUTE;  
        }  
  
        return result;  
    }  
}

注意：很多教程里都讲到通过注解的方式（@Cacheable，@CachePut、@CacheEvict和@Caching）实现数据缓存，根据实践，我个人是不推崇这种使用方式的。

## 四、缓存“及时”过期问题

这个也是开发和运维过程中非常经典的问题。

有些公司写缓存客户端的时候，会给每个团队分别定义一个Area，但是这个只能做到缓存键的分布区分，不能保证缓存“实时”有效的过期。

多年以前我写过一篇结合实际情况的文章，也就是加上缓存版本，请猛击这里 ，算是提供了一种相对有效的方案，不过高并发站点要慎重，防止发生雪崩效应。

Redis还有一些其他常见问题，比如：Redis的字符串类型Key和Value都有限制，且都是不能超过512M，请猛击这里。还有最大连接数和超时时间设置等问题，本文就不再一一列举了。[Redis面试连环问，快看看你能走到哪一步！](http://mp.weixin.qq.com/s?__biz=MzI4Njc5NjM1NQ==&mid=2247490823&idx=2&sn=87a8b17f33d2826d00bf2aa413699f8f&chksm=ebd6222bdca1ab3da619b5c846ce7e0083a383d8614b3610c0be9a692f501d269daf0603a1ca&scene=21" \l "wechat_redirect" \t "https://mp.weixin.qq.com/_blank)

## 五、二级缓存

在配置文件中，加上缓存提供者开关：

##是否启用本地缓存  
spring.power.isuselocalcache=1  
##是否启用Redis缓存  
spring.power.isuserediscache=1

缓存提供者程序都实现好了，我们会再包装一个调用外观类PowerCacheBuilder，加上缓存版本控制，可以轻松自如地控制和切换缓存，code talks:

package com.power.demo.cache;  
  
import com.google.common.collect.Lists;  
import com.power.demo.cache.contract.CacheProviderService;  
import com.power.demo.common.AppConst;  
import com.power.demo.common.AppField;  
import com.power.demo.util.ConfigUtil;  
import com.power.demo.util.PowerLogger;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
import org.springframework.context.annotation.ComponentScan;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.util.StringUtils;  
  
import java.util.List;  
import java.util.UUID;  
import java.util.concurrent.TimeUnit;  
import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
import java.util.function.Function;  
  
/\*  
 \* 支持多缓存提供程序多级缓存的缓存帮助类  
 \* \*/  
@Configuration  
@ComponentScan(basePackages = AppConst.BASE\_PACKAGE\_NAME)  
public class PowerCacheBuilder {  
  
    @Autowired  
    @Qualifier("localCacheService")  
    private CacheProviderService localCacheService;  
  
    @Autowired  
    @Qualifier("redisCacheService")  
    private CacheProviderService redisCacheService;  
  
    private static List<CacheProviderService> \_listCacheProvider = Lists.newArrayList();  
  
    private static final Lock providerLock = new ReentrantLock();  
  
    /\*\*  
     \* 初始化缓存提供者 默认优先级：先本地缓存，后分布式缓存  
     \*\*/  
    private List<CacheProviderService> getCacheProviders() {  
  
        if (\_listCacheProvider.size() > 0) {  
            return \_listCacheProvider;  
        }  
  
        //线程安全  
        try {  
            providerLock.tryLock(1000, TimeUnit.MILLISECONDS);  
  
            if (\_listCacheProvider.size() > 0) {  
                return \_listCacheProvider;  
            }  
  
            String isUseCache = ConfigUtil.getConfigVal(AppField.IS\_USE\_LOCAL\_CACHE);  
  
            CacheProviderService cacheProviderService = null;  
  
            //启用本地缓存  
            if ("1".equalsIgnoreCase(isUseCache)) {  
                \_listCacheProvider.add(localCacheService);  
            }  
  
            isUseCache = ConfigUtil.getConfigVal(AppField.IS\_USE\_REDIS\_CACHE);  
  
            //启用Redis缓存  
            if ("1".equalsIgnoreCase(isUseCache)) {  
                \_listCacheProvider.add(redisCacheService);  
  
                resetCacheVersion();//设置分布式缓存版本号  
            }  
  
            PowerLogger.info("初始化缓存提供者成功，共有" + \_listCacheProvider.size() + "个");  
        } catch (Exception e) {  
            e.printStackTrace();  
  
            \_listCacheProvider = Lists.newArrayList();  
  
            PowerLogger.error("初始化缓存提供者发生异常：{}", e);  
        } finally {  
            providerLock.unlock();  
        }  
  
        return \_listCacheProvider;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public <T extends Object> T get(String key) {  
        T obj = null;  
  
        //key = generateVerKey(key);//构造带版本的缓存键  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            obj = provider.get(key);  
  
            if (obj != null) {  
                return obj;  
            }  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function) {  
        T obj = null;  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            if (obj == null) {  
                obj = provider.get(key, function);  
            } else if (function != null && obj != null) {//查询并设置其他缓存提供者程序缓存  
                provider.get(key, function);  
            }  
  
            //如果callable函数为空 而缓存对象不为空 及时跳出循环并返回  
            if (function == null && obj != null) {  
                return obj;  
            }  
  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key      缓存键 不可为空  
     \* @param function 如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm function函数的调用参数  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm) {  
        T obj = null;  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            if (obj == null) {  
                obj = provider.get(key, function, funcParm);  
            } else if (function != null && obj != null) {//查询并设置其他缓存提供者程序缓存  
                provider.get(key, function, funcParm);  
            }  
  
            //如果callable函数为空 而缓存对象不为空 及时跳出循环并返回  
            if (function == null && obj != null) {  
                return obj;  
            }  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> T get(String key, Function<String, T> function, long expireTime) {  
        T obj = null;  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            if (obj == null) {  
                obj = provider.get(key, function, expireTime);  
            } else if (function != null && obj != null) {//查询并设置其他缓存提供者程序缓存  
                provider.get(key, function, expireTime);  
            }  
  
            //如果callable函数为空 而缓存对象不为空 及时跳出循环并返回  
            if (function == null && obj != null) {  
                return obj;  
            }  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 查询缓存  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param function   如没有缓存，调用该callable函数返回对象 可为空  
     \* @param funcParm   function函数的调用参数  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object, M extends Object> T get(String key, Function<M, T> function, M funcParm, long expireTime) {  
        T obj = null;  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            if (obj == null) {  
                obj = provider.get(key, function, funcParm, expireTime);  
            } else if (function != null && obj != null) {//查询并设置其他缓存提供者程序缓存  
                provider.get(key, function, funcParm, expireTime);  
            }  
  
            //如果callable函数为空 而缓存对象不为空 及时跳出循环并返回  
            if (function == null && obj != null) {  
                return obj;  
            }  
        }  
  
        return obj;  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入或覆盖值  
     \*  
     \* @param key 缓存键 不可为空  
     \* @param obj 缓存值 不可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj) {  
  
        //key = generateVerKey(key);//构造带版本的缓存键  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            provider.set(key, obj);  
  
        }  
    }  
  
    /\*\*  
     \* 设置缓存键值  直接向缓存中插入或覆盖值  
     \*  
     \* @param key        缓存键 不可为空  
     \* @param obj        缓存值 不可为空  
     \* @param expireTime 过期时间（单位：毫秒） 可为空  
     \*\*/  
    public <T extends Object> void set(String key, T obj, Long expireTime) {  
  
        //key = generateVerKey(key);//构造带版本的缓存键  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            provider.set(key, obj, expireTime);  
  
        }  
    }  
  
    /\*\*  
     \* 移除缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public void remove(String key) {  
  
        //key = generateVerKey(key);//构造带版本的缓存键  
  
        if (StringUtils.isEmpty(key) == true) {  
            return;  
        }  
  
        for (CacheProviderService provider : getCacheProviders()) {  
  
            provider.remove(key);  
  
        }  
    }  
  
    /\*\*  
     \* 是否存在缓存  
     \*  
     \* @param key 缓存键 不可为空  
     \*\*/  
    public boolean contains(String key) {  
        boolean exists = false;  
  
        //key = generateVerKey(key);//构造带版本的缓存键  
  
        if (StringUtils.isEmpty(key) == true) {  
            return exists;  
        }  
  
        Object obj = get(key);  
  
        if (obj != null) {  
            exists = true;  
        }  
  
        return exists;  
    }  
  
    /\*\*  
     \* 获取分布式缓存版本号  
     \*\*/  
    public String getCacheVersion() {  
        String version = "";  
        boolean isUseCache = checkUseRedisCache();  
  
        //未启用Redis缓存  
        if (isUseCache == false) {  
            return version;  
        }  
  
        version = redisCacheService.get(AppConst.CACHE\_VERSION\_KEY);  
  
        return version;  
    }  
  
    /\*\*  
     \* 重置分布式缓存版本  如果启用分布式缓存，设置缓存版本  
     \*\*/  
    public String resetCacheVersion() {  
        String version = "";  
        boolean isUseCache = checkUseRedisCache();  
  
        //未启用Redis缓存  
        if (isUseCache == false) {  
            return version;  
        }  
  
        //设置缓存版本  
        version = String.valueOf(Math.abs(UUID.randomUUID().hashCode()));  
        redisCacheService.set(AppConst.CACHE\_VERSION\_KEY, version);  
  
        return version;  
    }  
  
    /\*\*  
     \* 如果启用分布式缓存，获取缓存版本，重置查询的缓存key，可以实现相对实时的缓存过期控制  
     \* <p>  
     \* 如没有启用分布式缓存，缓存key不做修改，直接返回  
     \*\*/  
    public String generateVerKey(String key) {  
  
        String result = key;  
        if (StringUtils.isEmpty(key) == true) {  
            return result;  
        }  
  
        boolean isUseCache = checkUseRedisCache();  
  
        //没有启用分布式缓存，缓存key不做修改，直接返回  
        if (isUseCache == false) {  
            return result;  
        }  
  
        String version = redisCacheService.get(AppConst.CACHE\_VERSION\_KEY);  
        if (StringUtils.isEmpty(version) == true) {  
            return result;  
        }  
  
        result = String.format("%s\_%s", result, version);  
  
        return result;  
    }  
  
    /\*\*  
     \* 验证是否启用分布式缓存  
     \*\*/  
    private boolean checkUseRedisCache() {  
        boolean isUseCache = false;  
        String strIsUseCache = ConfigUtil.getConfigVal(AppField.IS\_USE\_REDIS\_CACHE);  
  
        isUseCache = "1".equalsIgnoreCase(strIsUseCache);  
  
        return isUseCache;  
    }  
}

单元测试如下：

@Test  
    public void testCacheVerson() throws Exception {  
  
        String version = cacheBuilder.getCacheVersion();  
        System.out.println(String.format("当前缓存版本：%s", version));  
  
        String cacheKey = cacheBuilder.generateVerKey("goods778899");  
  
        GoodsVO goodsVO = new GoodsVO();  
        goodsVO.setGoodsId(UUID.randomUUID().toString());  
        goodsVO.setCreateTime(new Date());  
        goodsVO.setCreateDate(new DateTime(new Date()));  
        goodsVO.setGoodsType(1024);  
        goodsVO.setGoodsCode("123456789");  
        goodsVO.setGoodsName("我的测试商品");  
  
        cacheBuilder.set(cacheKey, goodsVO);  
  
        GoodsVO goodsVO1 = cacheBuilder.get(cacheKey);  
  
        Assert.assertNotNull(goodsVO1);  
  
        version = cacheBuilder.resetCacheVersion();  
        System.out.println(String.format("重置后的缓存版本：%s", version));  
  
  
        cacheKey = cacheBuilder.generateVerKey("goods112233");  
  
        cacheBuilder.set(cacheKey, goodsVO);  
  
        GoodsVO goodsVO2 = cacheBuilder.get(cacheKey);  
  
        Assert.assertNotNull(goodsVO2);  
  
        Assert.assertTrue("两个缓存对象的主键相同", goodsVO1.getGoodsId().equals(goodsVO2.getGoodsId()));  
    }

一个满足基本功能的多级缓存系统就好了。

在Spring Boot应用中使用缓存则非常简洁，选择调用上面包装好的缓存接口即可。

String cacheKey = \_cacheBuilder.generateVerKey("com.power.demo.apiservice.impl.getgoodsbyid." + request.getGoodsId());  
  
GoodsVO goodsVO = \_cacheBuilder.get(cacheKey, \_goodsService::getGoodsByGoodsId, request.getGoodsId());

到这里Spring Boot业务系统开发中最常用到的ORM，缓存和队列三板斧就介绍完了。

在开发的过程中你会发现，Java真的是非常非常中规中矩的语言，你需要不断折腾并熟悉常见的开源中间件和工具，开源的轮子实在是太丰富，多尝试几个，实践出真知。

预告一下，后面的文章我将继续学习分享介绍常用中间件和工具，如定时任务，MongoDB，ES，分布式文件系统以及各种实用工具。总之Java下工具链非常完备，我们要给自己更多动力。

## 参考：

http://ifeve.com/google-guava/  
http://www.cnblogs.com/luochengqiuse/p/4640932.html