```
author: Dawn1iu
date: 2015-09-07
layout: post
title: Spring Redis相关整理
categories:
- nosql
tags:
- redis
- nosql
```

Cache框架相关整理

1. springCache

1.1 主要注解

@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

```
public @interface Cacheable {
    String[] value();              //请参考@CachePut
    String key() default "";       //请参考@CachePut
    String condition() default "";//请参考@CachePut
    String unless() default "";    //请参考@CachePut
```

@CacheEvict

主要针对方法配置，能够根据一定的条件对缓存进行清空

```
public @interface CacheEvict {
    String[] value();                          //请参考@CachePut
    String key() default "";                   //请参考@CachePut
    String condition() default "";             //请参考@CachePut
    boolean allEntries() default false;        //是否移除所有数据
    boolean beforeInvocation() default false;//是调用方法之前移除/还是调用之后移除
```

@CachePut

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，和 @Cacheable 不同的是，它每次都会触发真实方法的调用

```
public @interface CachePut {
    String[] value();              //缓存的名字，可以把数据写到多个缓存
    String key() default "";       //缓存key，如果不指定将使用默认的KeyGenerator生成，后边介绍
    String condition() default ""; //满足缓存条件的数据才会放入缓存，condition在调用方法之前和之后都会判断
    String unless() default "";    //用于否决缓存更新的，不像condition，该表达只在方法执行之后判断，此时可以拿到返回值result
进行判断了
}
```

@Caching

有时候我们可能组合多个Cache注解使用，多个操作可以以此组合成以自己的自定义注解。

```
public @interface Caching {

    Cacheable[] cacheable() default {};

    CachePut[] put() default {};

    CacheEvict[] evict() default {};

}
```

1.2 key生成器

Spring 4之后使用SimpleKeyGenerato

我们也可以自定义自己的key生成器，然后通过xml风格的或注解风格的CachingConfigurer中指定keyGenerator。

1.3 Code

1.3.1 如何集成

org.springframework.cache.annotation.ProxyCachingConfiguration

BeanFactoryCacheOperationSourceAdvisor AnnotationCacheOperationSource CacheInterceptor。

AnnotationCacheOperationSource的主要作用是获取定义在类和方法上的SpringCache相关的标注并将其转换为对应的CacheOperation属性。

BeanFactoryCacheOperationSourceAdvisor是一个PointcutAdvisor，是SpringCache使用Spring AOP机制的关键所在，该advisor会织入到需要执行缓存操作的bean的增强代理中形成一个切面。并在方法调用时在该切面上执行拦截器CacheInterceptor的业务逻辑。 CacheInterceptor是一个拦截器，当方法调用时碰到了BeanFactoryCacheOperationSourceAdvisor定义的切面，就会执行CacheInterceptor的业务逻辑，该业务逻辑就是缓存的核心业务逻辑。

1.3.2 AnnotationCacheOperationSource

AnnotationCacheOperationSource->AbstractFallbackCacheOperationSource->CacheOperationSource

其中CacheOperationSource接口定义了一个方法：
`Collection<CacheOperation> getCacheOperations(Method method, Class<?> targetClass);` 该方法用于根据指定类上的指定方法上打的SpringCache注释来得到对应的CacheOperation集合。

AbstractFallbackCacheOperationSource是CacheOperationSource的抽象实现类，采用模板模式将获取某类的某方法上的CacheOperation的业务流程固化。该固化的流程可将方法上的属性缓存，并实现了一个获取CacheOperation的fallback策略，执行的顺序为：1目标方法、2目标类、3声明方法、4声明类/接口。当方法被调用过一次之后，其上的属性就会被缓存。它提供了两个抽象模板方法，供具体的子类类来实现：

```
protected abstract Collection<CacheOperation> findCacheOperations(Method method);
protected abstract Collection<CacheOperation> findCacheOperations(Class<?> clazz);
```

具体实现使用回调模式，用Set中的每一个CacheAnnotaionParser去解析一个方法或类，然后将得到的List合并，最终返回。

AnnotationCacheOperationSource内部持有一个Set的集合，默认包含SpringCacheAnnotationParser，并使用SpringCacheAnnotationParser来实现AbstractFallbackCacheOperationSource定义的两个抽象方法。

CacheAnnotaionParser，该接口定义了两个方法。具体实现使用回调模式，用Set中的每一个CacheAnnotaionParser去解析一个方法或类，然后将得到的List合并，最终返回。

该方法业务逻辑很清晰：首先查找该类/方法上的Cacheable标注并进行合并。针对合并后的每个Cacheable创建对应的CacheableOperation；然后同样逻辑执行CacheEvict和CachePut。最后处理Caching，Caching表示的是若干组Cache标注的集合，将其解析成一组CacheOperation并添加到Collection ops中。

1.3.3 CacheInterceptor

CacheInterceptor继承了CacheAspectSupport并实现了MethodInterceptor接口，因此它本质上是一个Advice也就是可在切面上执行的增强逻辑

这个增强逻辑的核心功能是在CacheAspectSupport中实现的，其中首先调用 `AnnotationCacheOperationSource.getCacheOperations(method, targetClass)` 方法得到被调用方法的Collection，然后将这些CacheOperation以及被调用方法、调用参数、目标类、相应的Cache信息统统封装到CacheOperation上下文里，随后调用真正的核心方法：`private Object execute(final CacheOperationInvoker invoker, Method method, CacheOperationContexts contexts)` 该方法封装了SpringCache核心的处理逻辑，也就是使用Cache配合来完成用户的方法调用，并返回结果。

1.4 流程

```
1、首先执行@CacheEvict（如果beforeInvocation=true且condition 通过），如果allEntries=true，则清空所有
2、接着收集@Cacheable（如果condition 通过，且key对应的数据不在缓存），放入cachePutRequests（也就是说如果cachePutRequests为空，则数据在缓存中）
3、如果cachePutRequests为空且没有@CachePut操作，那么将查找@Cacheable的缓存，否则result=缓存数据（也就是说只要当没有cache put请求时才会查找缓存）
4、如果没有找到缓存，那么调用实际的API，把结果放入result
5、如果有@CachePut操作(如果condition 通过)，那么放入cachePutRequests
6、执行cachePutRequests，将数据写入缓存（unless为空或者unless解析结果为false）；
7、执行@CacheEvict（如果beforeInvocation=false 且 condition 通过），如果allEntries=true，则清空所有
```

对于打了Cache相关注释的类，在创建其bean的时候已经由Spring AOP为其创建代理增强，并将BeanFactoryCacheOperationSourceAdvisor加入其代理中。 当调用其方法的时候会通过代理执行到BeanFactoryCacheOperationSourceAdvisor定义的切面。该切面是一个PointcutAdvisor，在SpringAOP底层框架DefaultAdvisorChainFactory.getInterceptorsAndDynamicInterceptionAdvice方法中使用 `pointcutAdvisor.getPointcut().getMethodMatcher().matches(method, targetClass))` 来判断被调用方法是否匹配切点逻辑，如果匹配就执行其拦截器中的逻辑，BeanFactoryCacheOperationSourceAdvisor中注册的拦截器是CacheInterceptor，其执行逻辑为：
```
@Override
public Object invoke(final MethodInvocation invocation) throws Throwable {
Method method = invocation.getMethod();
```

```
CacheOperationInvoker aopAllianceInvoker = new CacheOperationInvoker() {
    @Override
    public Object invoke() {
        try {
            return invocation.proceed();
        }
        catch (Throwable ex) {
            throw new ThrowableWrapper(ex);
        }
    }
};


try {
    return execute(aopAllianceInvoker, invocation.getThis(), method, invocation.getArguments());
}
catch (CacheOperationInvoker.ThrowableWrapper th) {
    throw th.getOriginal();
}
```

```
}
```

其中Spring AOP底层调用该方法时传递来的参数MethodInvocation invocation是一个把调用方法method与其对应的切面拦截器interceptors组装成类似于ChainFilter一样的结构。 CacheOperationInvoker回调接口的意义是将决定切面逻辑与实际调用方法顺序的权利转交给CacheAspectSupport的execute方法。 该逻辑中调用了CacheAspectSupport的方法：

```
protected Object execute(CacheOperationInvoker invoker, Object target, Method method, Object[] args) {
    // Check whether aspect is enabled (to cope with cases where the AJ is pulled in automatically)
    if (this.initialized) {
        Class<?> targetClass = getTargetClass(target);
        Collection<CacheOperation> operations = getCacheOperationSource().getCacheOperations(method, targetClass
);
        if (!CollectionUtils.isEmpty(operations)) {
            return execute(invoker, method, new CacheOperationContexts(operations, method, args, target, targetC
lass));
        }
    }


    return invoker.invoke();
}
```

该方法首先通过前面描述的

```
AnnotationCacheOperationSource.getCacheOperations(method, targetClass)
```

来获得调用方法上的Collection，然后将其和调用方法method、方法参数args、目标对象target、目标类targetClass一起创建CacheOperationContexts。

```
public CacheOperationContexts(Collection<? extends CacheOperation> operations, Method method,
            Object[] args, Object target, Class<?> targetClass) {


        for (CacheOperation operation : operations) {
            this.contexts.add(operation.getClass(), getOperationContext(operation, method, args, target, targetC
lass));
        }
        this.sync = determineSyncFlag(method);
}
```

其中，为每个operation分别创建CacheOperationContext，获取CacheOperationMetadata metadata时的较重要的动作就是获取CacheOperation中用String名称定义的CacheResolver和KeyGenerator的bean。 然后在创建CacheOperationContext时使用CacheResolver bean获得cache的信息：

```
public CacheOperationContext(CacheOperationMetadata metadata, Object[] args, Object target) {
        this.metadata = metadata;
        this.args = extractArgs(metadata.method, args);
        this.target = target;
        this.caches = CacheAspectSupport.this.getCaches(this, metadata.cacheResolver);
        this.cacheNames = createCacheNames(this.caches);
        this.methodCacheKey = new AnnotatedElementKey(metadata.method, metadata.targetClass);
}
```

在创建完上下文CacheOperationContexts后，调用SpringCache真正的核心业务逻辑：
`private Object execute(final CacheOperationInvoker invoker, Method method, CacheOperationContexts contexts)`
该方法执行逻辑如下：

1.4.1 首先判断是否需要在方法调用前执行缓存清除：

`processCacheEvicts(contexts.get(CacheEvictOperation.class), true,CacheOperationExpressionEvaluator.NO_RESULT);`
其作用是判断是否需要在方法调用前执行缓存清除。判断是否存在beforeInvocation==true并且condition符合条件的@CacheEvict注释，如果存在则最终执行方法 `performCacheEvict` 对于注释中定义的每一个cache都根据allEntries是否为true执行其clear()方法或evict(key)方法来清除全部或部分缓存。

1.4.2 然后检查是否能得到一个符合条件的缓存值：

`Cache.ValueWrapper cacheHit = findCachedItem(contexts.get(CacheableOperation.class));` 其中调用了cache.get(key)

1.4.3 随后如果Cacheable miss（没有获取到缓存），就会创建一个对应的CachePutRequest并收集起来：

```
List<CachePutRequest> cachePutRequests = new LinkedList<CachePutRequest>();
    if (cacheHit == null) {
        collectPutRequests(contexts.get(CacheableOperation.class),
                CacheOperationExpressionEvaluator.NO_RESULT, cachePutRequests);
}
```

1.4.4 接下来判断返回缓存值还是实际调用方法的结果

如果得到了缓存，并且CachePutRequests为空并且不含有符合条件（condition match）的@CachePut注释，那么就将returnValue赋值为缓存值；否则实际执行方法，并将returnValue赋值为方法返回值。

```
Object cacheValue;
Object returnValue;


if (cacheHit != null && cachePutRequests.isEmpty() && !hasCachePut(contexts)) {
    // If there are no put requests, just use the cache hit
    cacheValue = cacheHit.get();
    if (method.getReturnType() == javaUtilOptionalClass &&
            (cacheValue == null || cacheValue.getClass() != javaUtilOptionalClass)) {
        returnValue = OptionalUnwrapper.wrap(cacheValue);
    }
    else {
        returnValue = cacheValue;
    }
}
else {
    // Invoke the method if we don't have a cache hit
    returnValue = invokeOperation(invoker);
    if (returnValue != null && returnValue.getClass() == javaUtilOptionalClass) {
        cacheValue = OptionalUnwrapper.unwrap(returnValue);
    }
    else {
        cacheValue = returnValue;
    }
}
```

1.4.5 方法调用后收集@CachePut明确定义的CachePutRequest

收集符合条件的@CachePut定义的CachePutRequest，并添加到上面的cachePutRequests中：
`collectPutRequests(contexts.get(CachePutOperation.class), cacheValue, cachePutRequests);` 注意，此时处于方法调用后，返回结果已经存在了，因此在condition定义中可以使用上下文#result。

1.4.6 执行CachePutRequest将符合条件的数据写入缓存

对于上面收集到的cachePutRequests，逐个调用其apply(cacheValue)方法：

```
for (CachePutRequest cachePutRequest : cachePutRequests) {
        cachePutRequest.apply(cacheValue);
}
```

其中，CachePutRequest.apply方法首先判断unless条件，unless不符合的时候才会对operationContext中的每一个cache执行put动作：

```
public void apply(Object result) {
        if (this.context.canPutToCache(result)) {
            for (Cache cache : this.context.getCaches()) {
                doPut(cache, this.key, result);
            }
        }
}

protected boolean canPutToCache(Object value)
```

1.4.7 最后判断是否需要在方法调用后执行缓存清除：

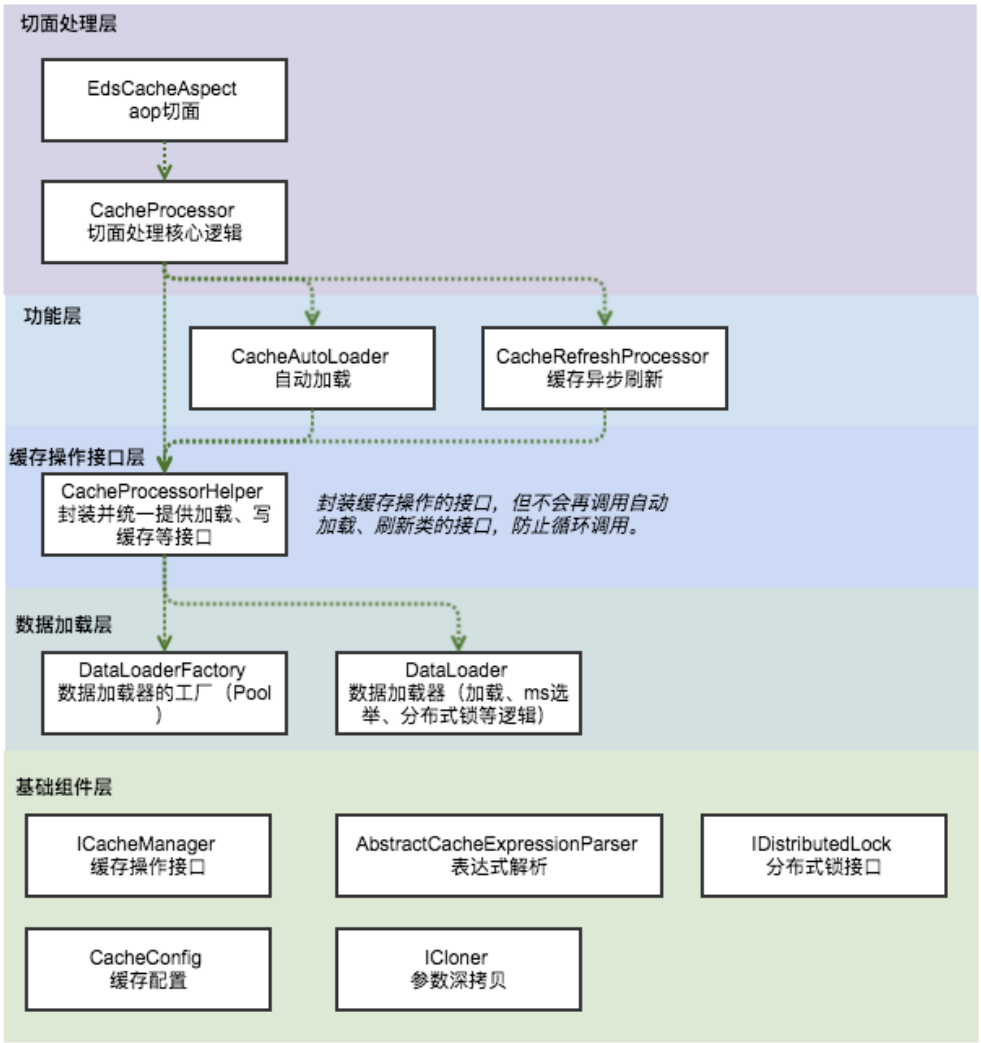
```
processCacheEvicts(contexts.get(CacheEvictOperation.class), false, cacheValue);
```

与步骤（1）相对应，其作用是判断是否需要在方法调用后执行缓存清除。判断是否存在beforeInvocation==false并且condition符合条件的@CacheEvict注释，注意此时上下文中结果#result是可用的。如果存在则最终执行缓存清除逻辑。

1.5 问题

无法设置时间、注意#result使用（调用时是否有result值），雪崩

2. EdsCache



2.1 基本

EdsCacheAspect： 缓存注解的切面类，仅简单调用CacheProcessor的接口。

CacheProcessor： 注解处理的核心逻辑。 负责注解表达式解析、调用缓存查询/删除接口、调用数据加载接口、刷新自动加载需要的数据、触发异步刷新等。

用户可以根据需求，通过构造方法或setter方法，传入分布式缓存的具体实现类、本地缓存具体实现类、分布式锁实现类等。 负责根据配置，初始化自动加载类、缓存异步刷新类等。

CacheAutoLoader： 负责自动加载的功能。实现细节看后面。

CacheRefreshProcessor： 负责异步刷新缓存。 当注解开启自动加载且未因超时等被驱逐时，缓存删除，会触发一次异步刷新任务。

CacheProcessorHelper： 由于正常的注解切面处理流程、自动加载、异步刷新，都最终需要执行“数据加载->写缓存”的逻辑，所以将逻辑抽出来提供统一接口， 但如果将CacheProcessor传入CacheAutoLoader、CacheRefreshProcessor等，会有循环调用的风险，所以提取出来放到CacheProcessorHelper中。 所有流程统一通过CacheProcessorHelper提供的接口来进行真正的数据加载、更细缓存。

DataLoaderFactory： 获取DataLoader的工厂方法。 由于每次进行数据加载都需要得到一个DataLoader，如果每次都new一个对象会影响性能，所以使用apache的GenericObjectPool缓存并复用dataLoader对象。 DataLoaderFactory自身提供静态的getInstance接口来产生工厂实例（单例、Lazy）。 提供borrow接口，从pool中借出并初始化dataLoader实例，提供return接口归回实例。

DataLoader： 负责数据加载的逻辑。除了调用真正的数据加载外，还负责“加载等待”功能，分布式锁功能等。

ICacheManager： 通用的缓存操作接口。 使用方都只调用该接口，而不是具体的实现类，目前实现了该接口的类有RedisCacheManager、EhCacheCacheManager，用户可以实现自己的缓存操作逻辑，并传入CacheProcessor即可。

AbstractCacheExpressionParser： 抽象的表达式解析类。 使用方都只调用该抽象类的接口，目前默认使用的是子类CacheSpelParser，实现了Spring Expression Language表达式。用户可以实现自己的表达式解析器并传入CacheProcessor。 由于表达式解析使用过的频率非常高，目前CacheSpelParser对Expression类进行了本地缓存。CacheSpelParser后续会提供注册自定义函数接口，方便表达式的使用。

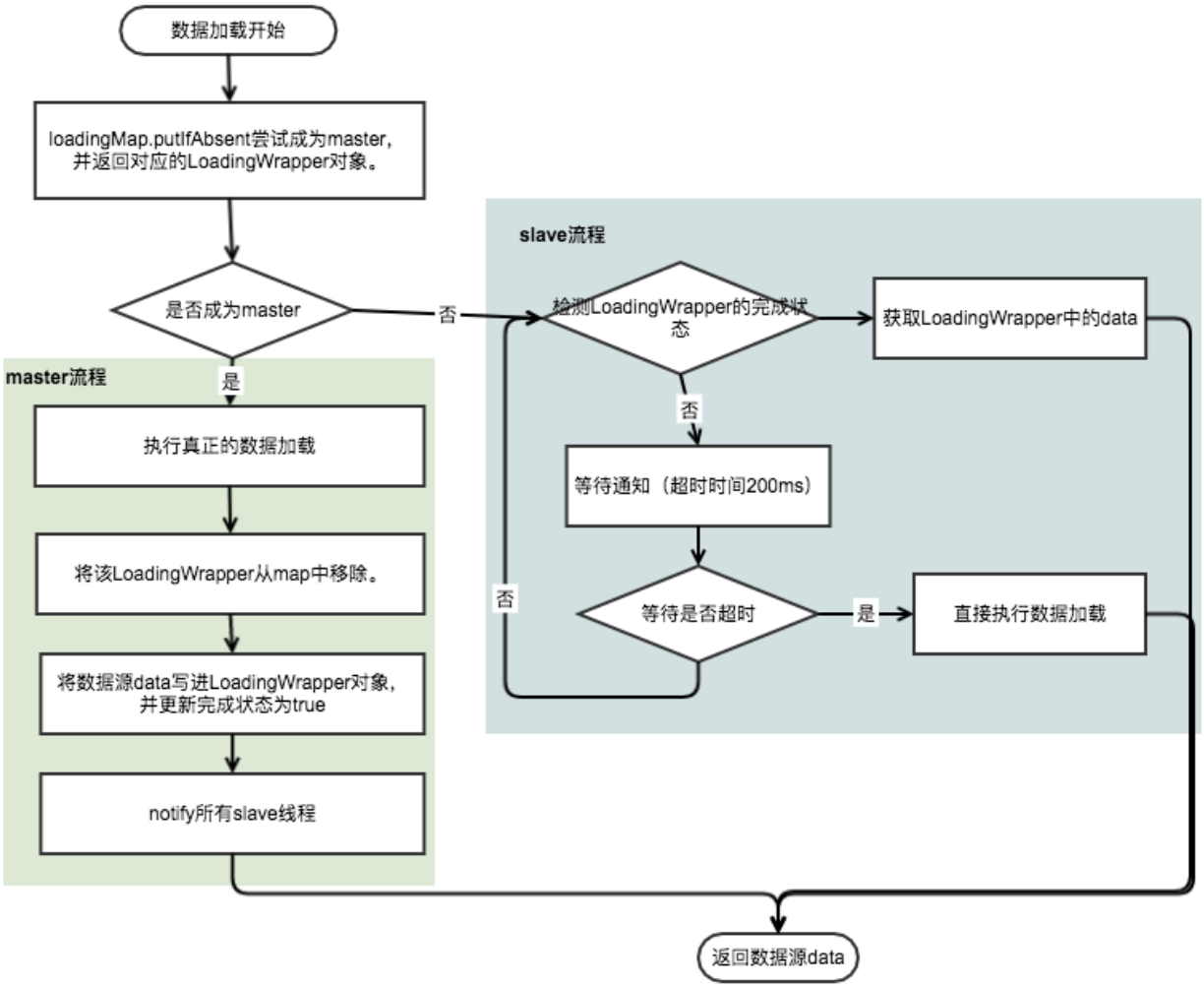
IDistributedLock： 分布式锁接口。 使用方都只调用该接口，而不是具体的实现类，目前默认使用的是RedisDistributedLock实现类（具体实现细节看后面）。

ICloner: 深拷贝接口。 使用方都只调用该接口，而不是具体的实现类，目前默认使用的是HessianCloner，使用Hessian序列化来实现深拷贝，如果是常见的不可变类，则直接引用。

CacheWrapper： 真正存在缓存中的，是CacheWrapper封装类，其中封装了data、超时时间等信息。 即使数据源返回的结果为null，也会在缓存中存一个CacheWrapper，避免了空值持续穿透到db的问题。

2.2 “加载等待”逻辑

ConcurrentHashMap loadingMap 记录当前哪些key正在进行数据加载，以及对应的状态信息。



LoadingWrapper

包含了从数据源加载出来的data、开始加载时间戳、是否加载完成这些信息。

2.3 自动加载逻辑

ConcurrentLinkedHashMap： ConcurrentLinkedHashMap autoLoadMap 存放所有需要自动加载的key与信息。 AutoLoading中包括最近刷新时间、最近请求时间、是否正在加载中、ProceedingJoinPoint、入参、注解信息、key等。 autoLoadMap使用google的ConcurrentLinkedHashMap结构，capacity是由配置指定的，达到capacity后会自动驱逐队列尾部（最长时间没有被访问过的元素）。

processingQueue： LinkedBlockingQueue processingQueue 阻塞队列中存放需要开始执行自动加载的AutoLoading。

生产者线程： 生产者线程会定时遍历该map，找出需要开始执行自动加载的AutoLoading，并添加到processingQueue阻塞队列中供消费者线程处理。 生产者线程也负责驱逐超时（超过requestTimeout没有被查询过的key）的AutoLoading，以停止自动加载。

消费者线： 消费者线程池是固定大小的线程池，线程数量由配置决定。 消费者线程阻塞式的从processingQueue take出需要处理的AutoLoading，并执行真正的数据加载、缓存更新等操作。 处理EdsCache切面的线程，每次都会判断是否启用了自动加载，是则尝试往autoLoadMap中添加自动加载项。

使用说明： http://doc.hz.netease.com/pages/viewpage.action?pageId=93270899#EdsCache%E5%8A%9F%E8%83%BD%E7%AE%80%E4%BB%8B-%E5%8F%AA%E5%B0%9D%E8%AF%95%E4%BB%8E%E7%BC%93%E5%AD%98%E8%AF%BB

扩展： 组概念