<http://www.jianshu.com/p/49fc4065201a>

<https://docs.spring.io/spring/docs/current/spring-framework-reference/integration.html#cache>

<https://www.cnblogs.com/cjsblog/p/9150482.html>

<http://www.cnblogs.com/hujunzheng/p/9660681.html>

# 36．Cache Abstraction：缓存抽象

## 36.1 介绍

从版本3.1开始，Spring Framework支持对现有Spring应用程序透明地添加缓存。与事务支持相似，缓存抽象允许一致地使用各种缓存解决方案，而对代码的影响最小。

从Spring 4.1开始，通过JSR-107注解和更多定制选项的支持，缓存抽象得到了显着改善。

## 36.2 理解缓存抽象

缓存与缓冲区

术语“缓冲”和“高速缓存”一般情况下可以交替使用; 但请注意，它们代表不同的事物。

缓冲区传统上被用作快速实体和慢速实体之间数据的中间临时存储。

由于一方需要等待另一方操作完毕，而造成性能影响，缓冲区允许将整个数据块进行移动而不是移动一小块来减轻这种影响。 数据只能从缓冲区写入和读取一次。 而且，缓冲区对于使用方来说是可见的。

另一方面，根据定义，高速缓存对使用方是不可见的。通过允许相同数据以快速方式被多次读取，以提高性能。

更多内容请参看下面网址：

<https://en.wikipedia.org/wiki/Cache_(computing)#The_difference_between_buffer_and_cache>

抽象缓存的核心是将抽象应用于Java方法，从而基于缓存中的可用信息可以减少方法的执行次数。 也就是说，每次调用目标方法时，抽象将应用缓存行为，检查是否已经为给定参数执行了方法。 如果有，则返回缓存的结果而不必执行目标方法; 如果没有，则执行方法，将结果缓存并返回给用户，以便下次调用方法时，返回缓存的结果。 这样，对于一组给定的参数，耗费资源的方法只需执行一次，并且结果被重用，而不必再次执行该方法。缓存逻辑对应用而言是透明的，对调用者没有任何干扰。

Important：这种方法只适用于给定输入返回相同输出的方法。

抽象也提供与缓存相关的其他操作，例如更新缓存、删除缓存。 因为缓存的数据在应用程序的生命周期是会变化的。

与Spring框架中的其他服务一样，缓存服务只是一个抽象（而不是具体实现），需要使用实际的存储来存储缓存数据，也就是说，抽象使开发人员不必写缓存逻辑 也不提供实际的存储。

缓存抽象是由org.springframework.cache.Cache和org.springframework.cache.CacheManager组成的。

有几个开箱即用的实现：基于JDK java.util.concurrent.ConcurrentMap的缓存，Ehcache 2.x，Gemfire缓存，Caffeine，Guava缓存和JSR-107兼容缓存（例如Ehcache 3.x）。

Important：缓存抽象没有多线程和多进程环境的特殊处理，因为这些特性是由具体实现处理的。

如果您有多进程环境（即在多个节点上部署应用程序），则需要相应地配置缓存提供程序。 根据您的使用情况，在多个节点上复制相同的数据可能就足够了，但如果在应用程序过程中更改数据，则可能需要启用其他传播机制。

缓存一个特定的项目等价于：get-if-not-found-then-proceed-and-put，不应用锁定，并且多个线程可能会尝试同时加载相同的项目。 移除数据也是如此：如果有几个线程试图同时更新或移除数据，则可能会使用到旧数据。

使用缓存抽象，需要关注下面两个方面：

1. 缓存声明：确定需要缓存的方法及其策略

2. 缓存配置：数据存储和读取的地方

## 36.3基于声明式注解的缓存

缓存抽象提供了一组Java注解：

@Cacheable：触发缓存

@CacheEvict:触发缓存移除

@CachePut: 更新缓存而不干扰方法执行

@Caching:在方法上组合多个缓存操作

@CacheConfig:在类基本上共享一些缓存相关配置

下面详细介绍每个注解。

### @Cacheable注解

顾名思义，@Cacheable用于标识将方法的结果进行缓存，以便在随后的调用中（具有相同的参数），直接从缓存中获取结果而不必执行该方法。 最简单的使用形式是，注解声明需要与关联一个缓存的名称(缓存名称是必须的)：

@Cacheable("books")  
**public** Book findBook(ISBN isbn)

{ }

在上面的代码片段中，findBook方法与名为books的缓存相关联。 每次调用该方法时，都会检查缓存以查看调用是否已经执行而不必重复。 在大多数情况下，只声明一个缓存，注释允许指定多个名称，以便使用多个缓存。 在这种情况下，将在执行方法之前检查每个缓存：如果至少有一个缓存被命中，则将返回相关的值：

@Cacheable({"books", "isbns"})  
**public** Book findBook(ISBN isbn)

{}

#### 默认key生成器

由于缓存本质上是键值存储，所以缓存方法的每次调用都需要被转换为适合缓存访问的Key。缓存抽象使用基于以下算法的简单KeyGenerator：

1. 如果没有给出参数，则返回SimpleKey.EMPTY
2. 如果只给出一个参数，则返回该实例。
3. 如果给出多于一个参数，返回一个包含所有参数的SimpleKey

这种方法适用于大多数情况; 只要参数具有自然键并实现有效的hashCode（）和equals（）方法。 如果情况并非如此，那么战略就需要改变。

要提供一个不同的默认密钥生成器，需要实现

org.springframework.cache.interceptor.KeyGenerator接口。

Note：默认的密钥生成策略随着Spring 4.0的发布而改变。 Spring的早期版本使用了一个Key生成策略，对于多个Key参数，只考虑参数的hashCode（）而不考虑equals（）; 这可能会导致意外的Key冲突。 新的“SimpleKeyGenerator”使用复合键来实现这种情况。

如果要继续使用以前的关键策略，则可以配置弃用的策略

org.springframework.cache.interceptor.DefaultKeyGenerator类或创建一个定制基于散列的“KeyGenerator”实现。

#### 自定义Key生成器

由于缓存是通用的，所以目标方法很可能具有不能简单映射到缓存结构的各种签名。 当目标方法有多个参数，可能只有其中一些适用于缓存。 例如：

@Cacheable("books")  
**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

乍一看，虽然两个布尔参数影响了数据查找，但它们对缓存没有用处。 更进一步，如果只有一个是有影响而另一个没影响呢？

对于这种情况，@Cacheable注解允许用户指定如何通过其关键属性来生成Key。 开发人员可以使用SpEL来选择感兴趣的参数，执行操作，甚至调用任意方法，而不必编写任何代码或实现任何接口。 这是默认生成器的推荐方法。

下面是各种SpEL声明的一些例子， 如果你不熟悉它，可以阅读第10章，Spring Expression Language（SpEL）：

@Cacheable(cacheNames=***"books"***, **key="#isbn"**)  
**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

@Cacheable(cacheNames=***"books"***, **key="#isbn.rawNumber"**)  
**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

@Cacheable(cacheNames=***"books"***, **key="T(someType).hash(#isbn)"**)  
**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

上面的代码片断展示了选择某个参数，参数的一个属性，甚至是任意一个静态）法。

如果负责生成Key的算法过于具体，或者需要共享，则可以在操作中定义一个自定义密钥生成器。 为此，请指定要使用的KeyGenerator bean实现的名称：

@Cacheable(cacheNames=***"books"***,**keyGenerator="myKeyGenerator"**)  
**public** Book findBook(ISBN isbn, **boolean** checkWarehouse, **boolean** includeUsed)

Note：key和keyGenerator参数是互斥的，同时指定两者将导致异常。

#### 默认的缓存方案

开箱即用，缓存抽象使用一个简单的CacheResolver，它使用配置的CacheManager检索在操作级别定义的缓存。

为了提供一个不同的默认缓存解析器，需要实现一个

org.springframework.cache.interceptor.CacheResolver接口。

#### 自定义缓存方案

默认缓存方案非常适合使用单个CacheManager的应用程序，而且不需要复杂的缓存方案要求。

对于使用多个缓存管理器的应用程序，可以为每个操作设置cacheManager：

@Cacheable(cacheNames=***"books"***,**cacheManager="anotherCacheManager"**)  
**public** Book findBook(ISBN isbn) { }

也可以以与替换Key生成器的方式来替换CacheResolver。对于每个缓存操作，基于请求参数进行缓存解析：

@Cacheable(**cacheResolver="runtimeCacheResolver"**)  
**public** Book findBook(ISBN isbn) { }

Note：自Spring 4.1以来，缓存注解的属性值不再是必须性的，因为无论注解的内容如何，CacheResolver都可以提供此特定信息。

与key和keyGenerator类似，cacheManager和cacheResolver参数是互斥的，同时指定两者的操作将导致异常，因为CacheResolver实现将忽略定制的CacheManager。

#### 同步缓存

在多线程环境中，某些操作可能会同时调用相同的参数（通常在启动时）。 默认情况下，缓存抽象不会锁定任何内容，所以可能会多次计算相同的值，从而违背缓存的目的。

对于这些特定情况，可以使用sync属性指示基础缓存提供者在计算值时锁定缓存条目。 结果，只有一个线程忙于计算值，而其他线程被阻塞，直到在缓存中更新条目。

@Cacheable(cacheNames=***"foos"***, **sync=true**)  
**public** Foo executeExpensiveOperation(String id) { }

Note：这是一个可选功能，你选择的缓存库可能不支持它。 核心框架提供的所有CacheManager实现都支持它。

#### 条件缓存

有时候，一个方法可能并不适合于缓存（例如，它可能取决于给定的参数）。 缓存注解通过**condition**参数来支持这样的功能，该**condition**参数采用被评估为真或假的SpEL表达式。 如果为true，则该方法被缓存，如果不是，则执行方法一个简单的例子，只有参数名称长度小于32时才会缓存：

@Cacheable(cacheNames=***"book"***, **condition="#name.length() < 32"**)  
**public** Book findBook(String name)

除了**condition**之外，还可以**unless**参数，**unless**参数对应的表达式在方法执行后才会执行：

@Cacheable(cacheNames=***"book"***, condition=***"#name.length() < 32"***, **unless="#result.hardback"**)  
**public** Book findBook(String name)

缓存抽象支持java.util.Optional，只有当它存在时才将其内容用作缓存值。 #result始终引用业务实体，所以前面的示例可以重写为：

@Cacheable(cacheNames=***"book"***, condition=***"#name.length() < 32"***, **unless="#result?.hardback"**)  
**public** Optional<Book> findBook(String name)

注意到result仍旧引用 Book对象而不是Optional，因此可能为null。

#### 可用的缓存SPEL执行上下文

每个SpEL表达式都会在一个专用的上下文执行。 除了内置参数外，框架还提供了与参数名称相关的专用缓存元数据。 下表列出了可用于上下文的项，因此可以将它们用于Key和条件计算：

表格 1 可用的SpEL元数据

|  |  |  |  |
| --- | --- | --- | --- |
| **名字** | **位置** | **描述** | **例子** |
| methodName | root对象 | 被执行函数的名字 | #root.methodName |
| method | root对象 | 被执行的函数 | #root.method.name |
| target | root对象 | 被执行的目标对象 | #root.target |
| targetClass | root对象 | 被执行的目标类 | #root.targetClass |
| args | root对象 | 执行目标对象所需要的参数 | #root.args[0] |
| caches | root对象 | 当前执行方法的缓存集合 | #root.caches[0].name |
| argument name | 执行上下文 | 任何方法参数的名称，如果由于某些原因名称不可用，则参数名称也可在#a <#arg>下使用，其中#arg代表参数索引（从0开始）。 | #iban or #a0 |
| result | 执行上下文 | 方法的返回结果 |  |

### @CachePut注解

在不干扰方法执行的情况下更新缓存，可以使用@CachePut注解。 也就是说，方法总是被执行，并且它的结果被放入缓存（根据@CachePut选项）。 它支持与@Cacheable相同的选项：

@CachePut(cacheNames="book", key="#isbn")  
**public** Book updateBook(ISBN isbn, BookDescriptor descriptor)

Important：强烈反对在同一个方法上使用@CachePut和@Cacheable注解，因为它们有不同的行为。 后者导致使用缓存跳过方法执行，前者强制执行以更新缓存。 这会导致意想不到的行为。

### @CacheEvict注解

缓存抽象不仅允许缓存存储，而且还可以将缓存移除。从缓存中删除过时或未使用的数据通常是很有用的。 与@Cacheable相反，@CacheEvict注解标注的方法会执行缓存移除，即充当从缓存中移除数据的触发器。@CacheEvict需要指定一个（或多个）受操作影响的缓存，允许指定自定义的缓存和Key解析或移除条件，另外还有一个参数allEntries， 对满足条件的数据进行移除，而不是仅仅是一条数据（基于key）：

@CacheEvict(cacheNames=***"books"***, **allEntries=true**)  
**public void** loadBooks(InputStream batch)

当需要清除整个缓存区时，这个选项会派上用场（逐条清除会耗费很长的时间，效率很低），所有的条目在一个操作中被删除，如上所示。

可以配置缓存移除的操作是在方法执行之后（缺省）还是之前。 如果配置为之后执行，则一旦方法成功完成，就移除缓存中的数据。 如果该方法没有执行（因为它可能被缓存）或抛出异常，缓存移除不会发生。 设置（beforeInvocation = true）则缓存清楚始终发生在方法被调用之前，而不管方法是否没被执行或者执行过程发生异常。

Note： void方法可以和@CacheEvict一起使用，此时这些方法充当触发器，返回值被忽略（因为它们不与缓存交互），@Cacheable不能用于void方法，它将数据添加/更新到缓存中，因此需要返回结果。

### @Caching注解

在某些情况下，需要指定相同类型的多个注解（例如@CacheEvict或@CachePut），例如因为条件或key表达式在不同的缓存之间不同。 @Caching允许在同一个方法上使用多个嵌套的@Cacheable，@CachePut和@CacheEvict：

@Caching(evict = { @CacheEvict("primary"), @CacheEvict(cacheNames="secondary", key="#p0") })  
**public** Book importBooks(String deposit, Date date)

### @CacheConfig注解

到目前为止，我们已经看到，缓存操作提供了许多可定制选项，可以在操作上进行设置。 但是，如果某些自定义选项适用于该类的所有操作，则可能会进行繁琐的配置。 例如，指定要用于该类的每个缓存操作的缓存名称可以由单个类级别的定义替换。 这时需要@CacheConfig出场了。

**@CacheConfig("books")  
public class** BookRepositoryImpl **implements** BookRepository {

@Cacheable  
**public** Book findBook(ISBN isbn) { }

}

@CacheConfig是一个类级注解，允许共享缓存名称、自定义KeyGenerator、自定义CacheManager和自定义CacheResolver。 将该注释放在类上不会启用任何缓存操作。

操作级定制选项将始终覆盖@CacheConfig上的定制选项。 每个缓存操作选择有三个级别：

1. 全局配置，用于CacheManager、KeyGenerator
2. 在类上使用@CacheConfig
3. 每个操作方法上的配置

### 启用缓存注解

需要注意的是，即使声明缓存注释也不会自动触发它们的动作，就像Spring中的许多事情一样，这个功能必须被声明性地启用。

要启用缓存注释，需要将注释@EnableCaching添加到一个@Configuration类中：

@Configuration  
@EnableCaching  
**public class** AppConfig {  
}

对于XML配置，使用cache：annotation-driven元素：

**<beans xmlns**=**"http://www.springframework.org/schema/beans"**

**xmlns:xsi**=**"http://www.w3.org/2001/XMLSchema-instance"  
xmlns:cache**=**"http://www.springframework.org/schema/cache"  
xsi:schemaLocation**=**"**

**http://www.springframework.org/schema/beans**

**http://www.springframework.org/schema/beans/springbeans.xsd  
http://www.springframework.org/schema/cache** [**http://www.springframework.org/schema/cache/springcache.xsd**](http://www.springframework.org/schema/cache/springcache.xsd)**"**

**<cache:annotation-driven />**

**</beans>**

cache：annotation-driven元素和@EnableCaching注解都允许指定各种选项，以通过AOP方式影响缓存行为。这个配置与@Transactional的配置很相似：

Note：用于处理缓存注解的默认通知模式是“代理”，它允许通过代理拦截调用; 类内的相互调用不能被拦截。 对于更高级的拦截模式，考虑切换到“aspectj”模式结合编译时或加载时编织。

使用Java配置的高级定制需要实现CachingConfigurer。

表格 2 缓存注解设置

|  |  |  |  |
| --- | --- | --- | --- |
| **XML属性** | **注解属性** | **默认值** | **描述** |
| cache-manager | N/A | cacheManager | 缓存管理器的名称 |
| cache-resolver | N/A | SimpleCacheResolve | CacheResolver的Bean名称 |
| key-generator | N/A | SimpleKeyGenerator | 自定义Key生成器的名称 |
| error-handler | N/A | SimpleCacheErrorHandler | 自定义缓存错误处理程序的名称。默认情况下，在进行缓存相关操作过程抛出的任何异常都将抛给客户端。 |
| mode | mode | proxy | 默认使用Spring AOP进行代理，也可以使用aspectj |
| proxy-target-class | proxyTargetClass | false | 仅适用于代理模式。 控制@Cacheable或@CacheEvict注解进行注释的类创建哪种类型的缓存代理。 如果proxy-target-class属性设置为true，则创建基于类的代理。 如果proxy-target-class为false或者该属性被省略，则创建标准的基于JDK接口的代理。 |
| order | order | Ordered.LOWEST\_PRECEDENCE |  |

Note：<cache：annotation-driven />

只查找@Cacheable/@CachePut/@CacheEvict/ @Caching等注解标注的，在同一应用程序上下文中定义的bean。

这意味着，如果为DispatcherServlet的WebApplicationContext放置<cache：annotation-driven />，则它只会检查控制器中的bean。

方法可见性和缓存注释

使用代理时，应将缓存注解仅应用于public方法。 如果使用这些注释标注受保护的，私有的或包可见的方法，不会引发错误，但注释的方法不会显示已配置的缓存设置。如果你需要注解非公共方法，考虑使用AspectJ，因为它改变了字节码本身。

Tip：Spring建议您只使用@Cache\* 注解来标注具体类（以及具体类的方法），而不是标注接口。 当然可以将@Cache\*注解放在一个接口（或一个接口方法）上，但是这只能是在您使用基于接口的代理的情况下使用。 Java注解是不能从接口继承的，这意味着如果您使用基于类的代理（proxy-targetclass =“true”）或基于交织的aspect（mode =“aspectj”），则无法识别缓存设置，由代理和编织，对象不会被包装成一个缓存代理，这将是非常糟糕的。

Note：在代理模式下（这是默认模式），只拦截通过代理进入的外部方法调用。 这意味着，即使被调用的方法被标记为@Cacheable，实际上，调用目标对象是目标对象的另一个方法的自我调用，是不会导致缓存的，可以考虑使用aspectj模式。 此外，代理必须完全初始化，以提供预期的行为，所以不应该在初始化代码中依赖此功能，即@PostConstruct。

## 36.6配置缓存存储

### 基于JDK ConcurrentMap的缓存

基于JDK的缓存实现位于org.springframework.cache.concurrent包下。 它使用ConcurrentHashMap作为后端缓存存储。

*<!-- simple cache manager -->***<bean id**=**"cacheManager"**

**class**=**"org.springframework.cache.support.SimpleCacheManager">  
<property name**=**"caches">**

**<set>  
<bean class**=**"org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name**=**"default"/>  
<bean class**=**"org.springframework.cache.concurrent.ConcurrentMapCacheFactoryBean" p:name**=**"books"/>  
</set>**

**</property>  
</bean>**

上面的代码片断使用SimpleCacheManager创建一个CacheManager，并且嵌套了两个名为default和books的ConcurrentMapCache实例。 请注意，名称是针对每个缓存配置的。

由于缓存是由应用程序创建的，因此它被绑定到应用程序的生命周期。

### 基于Ehcache的缓存

基于Ehcache 2.x的缓存实现位于org.springframework.cache.ehcache包下。 同样，要使用它，只需要声明适当的CacheManager：

**<bean id**=**"cacheManager"  
class**=**"org.springframework.cache.ehcache.EhCacheCacheManager" p:cache-manager-ref**=**"ehcache"/>***<!-- EhCache library setup -->***<bean id**=**"ehcache"  
class**=**"org.springframework.cache.ehcache.EhCacheManagerFactoryBean" p:configlocation**=**"ehcache.xml"/>**

这个设置引导了Spring IoC中的ehcache库（通过ehcache bean），然后将其连接到专用的CacheManager实现中。 请注意，整个ehcache特定的配置是从ehcache.xml中读取的。

### 基于Redis的缓存

基于Redis的缓存需要引用外部jar包，常用的有Spring-data-redis包的org.springframework.data.redis.cache. RedisCacheManager或redisson包下的org.redisson.spring.cache.RedissonSpringCacheManager

@Configuration  
@EnableCaching  
**public class** CacheConfig {  
 @Bean  
 **public** RedisCacheManager redisCacheManager(RedisTemplate redisTemplate) {  
 RedisCacheWriter redisCacheWriter = RedisCacheWriter.*nonLockingRedisCacheWriter*(redisTemplate.getConnectionFactory());  
 RedisCacheConfiguration redisCacheConfiguration = RedisCacheConfiguration.*defaultCacheConfig*()  
 .serializeValuesWith(RedisSerializationContext.SerializationPair.*fromSerializer*(redisTemplate.getValueSerializer()))  
 .computePrefixWith(name -> name + **":"**);  
 **return new** RedisCacheManager(redisCacheWriter, redisCacheConfiguration);  
 }  
  
 @Bean  
 **public** GenericJackson2JsonRedisSerializer genericJackson2JsonRedisSerializer() {  
 **return new** GenericJackson2JsonRedisSerializer();  
 }  
  
 @Bean  
 **public** RedisTemplate redisTemplate(GenericJackson2JsonRedisSerializer genericJackson2JsonRedisSerializer,  
 @Qualifier(**"commonJedisConnectionFactory"**) RedisConnectionFactory commonJedisConnectionFactory) {  
 RedisTemplate redisTemplate = **new** RedisTemplate();  
 redisTemplate.setHashValueSerializer(genericJackson2JsonRedisSerializer);  
 redisTemplate.setValueSerializer(genericJackson2JsonRedisSerializer);  
 redisTemplate.setConnectionFactory(commonJedisConnectionFactory);  
  
 **return** redisTemplate;  
 }  
  
  
}

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-cache</**artifactId**>  
 *<!--<version>2.0.3.RELEASE</version>-->*</**dependency**>

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

SpringApplicationRunListener

**org.springframework.boot.SpringApplicationRunListener**=**\**