缓存概述:

Since version 3.1, the Spring Framework provides support for transparently adding caching to an existing Spring application. Similar to the transaction support, the caching abstraction allows consistent use of various caching solutions with minimal impact on the code.

As from Spring 4.1, the cache abstraction has been significantly improved with the support of JSR-107 annotations and more customization options.

翻译：

自 3.1 版本以来，Spring 框架提供了对向现有 Spring 应用程序透明地添加缓存的支持。与事务支持类似，缓存抽象允许在对代码影响最小的情况下一致使用各种缓存解决方案。

从 Spring 4.1 开始，在 JSR-107 注释和更多定制选项的支持下，缓存抽象得到了显著改进。

At its core, the cache abstraction applies caching to Java methods, thus reducing the number of executions based on the information available in the cache. That is, each time a targeted method is invoked, the abstraction applies a caching behavior that checks whether the method has been already executed for the given arguments. If it has been executed, the cached result is returned without having to execute the actual method. If the method has not been executed, then it is executed, and the result is cached and returned to the user so that, the next time the method is invoked, the cached result is returned. This way, expensive methods (whether CPU- or IO-bound) can be executed only once for a given set of parameters and the result reused without having to actually execute the method again. The caching logic is applied transparently without any interference to the invoker.

缓存抽象的核心是将缓存应用于 Java 方法，从而根据缓存中可用的信息减少执行次数。也就是说，每次调用目标方法时，抽象都会应用缓存行为来检查是否已经为给定参数执行了该方法。如果已经执行，则返回缓存的结果，而不需要执行实际的方法。如果还没有执行该方法，那么就会执行该方法，并将结果缓存并返回给用户，以便在下一次调用该方法时返回缓存的结果。这样，对于给定的一组参数和重用的结果，昂贵的方法(无论是 CPU 绑定的还是 io 绑定的)只能执行一

次，而不必再次实际执行该方法。缓存逻辑透明地应用，不会对调用程序造成任何干扰。

总体上来说呢，Spring Cache 并不是一些缓存方案的具体实现，而是一个对缓存使用的抽象，通过在既有代码中添加少量它定义的各种 annotation，即能够达到缓存方法的返回对象的效果。Spring Cache 还支持使用 SPEL（Spring Expression Language）来定义缓存的 key 和各种 condition,在实际开发中具有很高的灵活性。

基于注解的缓存:

Spring 缓存提供了以下注解来实现缓存:

@Cacheable:触发器缓存人口。

@CacheEvict: 触发缓存回收。

@CachePut:在不影响方法执行的情况下更新缓存。触发缓存回收。

@Caching: 重新组合要应用于方法的多个缓存操作。

@CacheConfig:在类级别上共享一些常见的缓存相关设置。

本次呢，主要集中讲解@Cacheable,@CacheEvict,@CachePut 这三个注解的使用

配置 Spring Cache

首先呢，我们先声明一个配置类 CacheConfig,并在配置类上加上@EnableCaching 和@Configuration 注解。

@EnableCaching 注解是 spring framework 中的注解驱动的缓存管理功能。自 spring 版本 3.1 起加入了该注解。如果你使用了这个注解，那么你就不需要在 XML 文件中配置 cache manager 了。

@Configuration：声明一个 java 配置类

```java
@Configuration
@EnableCaching
public class CacheConfig {

    @Bean
    public CacheManager simpleCacheManager() {
        // 实例化一个缓存Manager
        SimpleCacheManager simpleCacheManager = new SimpleCacheManager();

        ConcurrentMapCache cache = new ConcurrentMapCache( name: "cache");

        simpleCacheManager.setCaches(Collections.singleton(cache));

        return simpleCacheManager;

    }


}
```

这里我们定义一个缓存管理器 simpleCacheManager，通过名字呢大致可以猜测到这个缓存管理器实现了 SpringCache 基本的功能，至于后续更加详细的解读，后续文章会提到。

然后构造一个缓存的对象 ConcurrentMapCache，其中构造器参数即是我们为缓存定义的名称，其中，一个 Spring 应用中允许存在多个缓存。

通过 SimpleCacheManager.的 setCaches 方法把我们的缓存对象和缓存管理器进行绑定。

缓存的简单使用:

//注入 SimpleCacheManager 对象

```
@Autowired
private CacheManager simpleCacheManager;
```

放入缓存:

```
Cache cache = simpleCacheManager.getCache("cache");
cache.put(key,value);
```

从缓存中拿数据:

```
Cache cache = simpleCacheManager.getCache("cache");
String value = cache.get(key,String.class);
```

环境准备:

1. 声明一个实体类 Person

```
public class Person implements Serializable {
    private String id;
    private String name;
    private int age;
```

注意:记得要实现 java 序列化接口 Serializable

声明一个 PersonRepository 接口类，用来声明缓存操作的一些方法。

```
@NoRepositoryBean
public interface PersonRepository {
```

@NoRepositoryBean:

一般用作父类的 repository，有这个注解，spring 不会去实例化该 repository

声明一个 PersonRepositoryImpl 类实现 PersonRepository，提供方法的具体实现。

```
@Repository
public class PersonRepositoryImpl implements PersonRepository {

    private  final Map<String, Person> repository = new HashMap<>();
```

定义一个 Map repository 用来模仿数据库操作。

@Repository：使用@Repository 注解可以确保 DAO 或者 repositories 提供异常转译，这个注解修饰的 DAO 或者 repositories 类会被 ComponetScan 发现并配置，同时也不需要为它们提供 XML 配置项

@Cacheable 注解:

As the name implies, you can use @Cacheable to demarcate methods that are cacheable — that is, methods for which the result is stored in the cache so that, on subsequent invocations (with the same arguments), the value in the cache is returned without having to actually execute the method. In its simplest form, the annotation declaration requires the name of the cache associated with the annotated method。

顾名思义，您可以使用@Cacheable 来划分可缓存的方法——也就是说，将结果存储在缓存中的方法，以便在后续调用时(参数相同)返回缓存中的值，而无需实际执行该方法。注释声明最简单的形式是需要与带注释的方法相关联的缓存名。

也就是说呢，@Cacheable 是作用在方法上的缓存注解,因为@Cacheable 最后会把该方法执行之后返回的结果放入到缓存之中，所以用@Cacheable 注释的方法是一定要有返回值的，不可以是 void 修饰的方法。

部分参数:

key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如: @Cacheable(value="testcache",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存	例如: @Cacheable(value="testcache",condition="#userName.length()>2")

因为 value 这个属性在之后的版本中已经不太建议使用，Spring 引入了新的 cacheNames 来实现相同的操作。

如果没有额外的配置，只需要声明缓存，可以不写 cacheNames 属性，类似于官方网站这样用:

```
@Cacheable("books")
public Book findBook(ISBN isbn) {...}
```

虽然大多数情况下使用一个缓存，但是 Sprig 支持使用多个缓存，在这种情况下，在执行方法之前检查每个缓存——如果至少命中一个缓存，则返回相关的值。

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

同时呢，Spring 支持自定义 key，如果不指定 key，则按照默认的生成器生成 Key

实战 ：

```
@NoRepositoryBean
public interface PersonRepository {

    @Cacheable(cacheNames = "cache")
    Person findPerson(String id);

    void savePerson(Person person);
```

声明 findPerson 和 savePerson 方法。
并在 findPerson 上添加@Cacheable 注解，指明 cacheNames 为我们上方配置的缓存名称 cache

编写具体实现:

```
@Repository
public class PersonRepositoryImpl implements PersonRepository {

    private  final Map<String, Person> repository = new HashMap<>();

    @Override
    public Person findPerson(String id) { return repository.get(id); }

    @Override
    public void savePerson(Person person) { repository.put(person.getId(),person); }
```

编写测试 Controller

```java
@RestController
@RequestMapping("cache")
public class CacheController {

    @Autowired
    private CacheManager simpleCacheManager;

    @Autowired
    private PersonRepository repository;


    @RequestMapping("save")
    public Object save(){

        Person person = new Person();
        person.setId("1");
        person.setAge(20);
        person.setName("张三");
        repository.savePerson(person);
        return person;
    }

    @RequestMapping("get")
    public Object get(){

        Person p = repository.findPerson( id: "1");

        return p;
    }
```

因为主要供测试使用，所以，就不写请求参数了，好吧，这点我犯懒了。

OUT :

算了，我就不 Debug 了，我们在 get 方法的实现中加入一个 println(),这样就可以直观通过控制台的输出来看缓存是否生效了。

通过 save 请求存入数据，然后通过 get 请求查询数据。最后发现，不论执行多少次 get 请求，只有第一次访问是通过真实的方法的。之后的请求都是直接在缓存中获取数据并且返回的。

```
2018-12-15 14:41:27.807  INFO 13856 --- [on(5)-127.0.0.1] io.lettuce.core.KqueueProvider           : St
get方法执行了..
```

有条件的缓存:

声明方法：

```
@Cacheable(value = "cache",condition = "#id.length()>5")
Person findPersonByCriteria(String id);
```

编写具体实现:

```
@Override
public Person findPersonByCriteria(String id) { return repository.get(id); }
```

OUT：

只有符合条件的 id 才会被缓存，而不符合条件的则会调用真实方法查询数据。

@CacheEvict 注解。

使用详见官方文档：

```
@CacheEvict(cacheNames="books", allEntries=true) (1)
public void loadBooks(InputStream batch)
```

使用 allEntries 属性从缓存中清除所有条目。

不过@CacheEvict 有一些需要注意的地方：

@CacheEvict 注解的方法可以使用 void 关键字修饰

当清除缓存的过程中出现异常时，往往会导致缓存无法正确清除掉，这个时候需要设置

beforeInvocation 属性的值为 true，在方法执行前清除缓存。

其他参数：

key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @CachEvict(value=”testcache”,key=”#userName”)
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才清空缓存	例如： @CachEvict(value=”testcache”, condition=”#userName.length()>2”)
allEntries	是否清空所有缓存内容，缺省为 false，如果指定为 true，则方法调用后将立即清空所有缓存	例如： @CachEvict(value=”testcache”,allEntries=true)
beforeInvocation	是否在方法执行前就清空，缺省为 false，如果指定为 true，则在方法还没有执行的时候就清空缓存，缺省情况下，如果方法执行抛出异常，则不会清空缓存	例如： @CachEvict(value=”testcache”，beforeInvocation=true)

@CachePut 注解。

When the cache needs to be updated without interfering with the method execution, you can use the @CachePut annotation. That is, the method is always executed and its result is placed into the cache (according to the @CachePut options). It supports the same options as @Cacheable and should be used for cache population rather than method flow optimization.

当需要在不影响方法执行的情况下更新缓存时，可以使用@CachePut 注释。也就是说，总是执行该方法，并将其结果放入缓存(根据@CachePut 选项)。它支持与@Cacheable 相同的选项，应该用于缓存填充，而不是用于方法流优化。

实战：

声明方法:

```
@CachePut(cacheNames = "cache")
void updataPerson(Person person);
```

编写实现：

```
@Override
public void updataPerson(Person person) {

    System.out.println("this mothed was executed....");

    repository.put(person.getId(),person);
}
```

编写 Controller

```
@RequestMapping("updata")
public Object updataPerson(){

    Person person = repository.findPerson( id: "1");
    person.setName("李四");
    repository.updataPerson(person);
    return person;
}
```

OUT：

当执行了 updataPerson 方法之后，执行查询操作的时候依然不会调用真实的方法，会从缓存中取更新过的新的 value 返回。

主要参数：

key	缓存的 key，可以为空，如果指定要按照 SpEL 表达式编写，如果不指定，则缺省按照方法的所有参数进行组合	例如： @Cacheable(value="testcache",key="#userName")
condition	缓存的条件，可以为空，使用 SpEL 编写，返回 true 或者 false，只有为 true 才进行缓存	例如： @Cacheable(value="testcache",condition="#userName.length()>2")

Spring Cache 的基本原理:

Spring cache 利用了 Spring AOP 的动态代理技术，调用使用缓存的 class 时，实际客户端拥有的是一个代理的引用，那么在调用目标方法的时候，会首先调用 proxy 的目标方法，这个时候 proxy 可以整体控制实际的 目标方法的入参和返回值，比如缓存结果，比如直接略过执行实际的目标方法等，都是可以轻松做到的。

总结：

总之，注释驱动的 spring cache 能够极大的减少我们编写常见缓存的代码量，通过少量的注释标签和配置文件，即可达到使代码具备缓存的能力。且具备很好的灵活性和扩展性。但是我们也应该看到，spring cache 由于急于 spring AOP 技术，尤其是动态的 proxy 技术，导致其不能很好的支持方法的内部调用或者非 public 方法的缓存设置，当然这都是可以解决的问题，通过学习这个技术，我们能够认识到，AOP 技术的应用还是很广泛的，如果有兴趣，我相信你也能基于 AOP 实现自己的缓存方案。--这句话是查资料的时候看到的，很适合做今天的总结，嘻嘻。

本篇文档只是较为简略的阐述了 Spring 一些缓存注解的配置及简单实用，如果想要更加详细的了解 Spring 缓存机制，那么建议大家去看一下 Spring 官方文档关于缓存的部分，整体官方文档写的还是非常不错的，同时建议大家读一下相关的 jsr 规范，剩下的，就要在漫长的踩坑中不断学习了！

代码环境:Springboot2.0.7.RELEASE
IDE:IDEA2018
版本:1.0
时间:2018-12-15 日
作者:韩数(代号)
邮箱:1758504262@qq.com
Github ： https://github.com/hanshuaikang

参考资料:

Spring 官方文档:

https://docs.spring.io/spring/docs/5.1.3.RELEASE/spring-framework-reference/integration.html#cache

JSR-107

小马哥微服务实战视频 Springboot 部分第十章

最后，给一个星星吧，我已经好几天没吃星星了。