57_源码剖析：玩转 Spring Cache 中@Cacheable 注解的底层原理

1、开篇

上节课给大家简单介绍了教师模块的主要功能，同时带大家过了一遍教师模块的业务流程。这节课主要针对在高并发下教师列表的查询问题，提出解决思路和具体的解决方案。今天课程的内容包括以下几个部分：

- 一张大图了解@Cacheable 注解的类结构
- CacheOperationSource 介绍
- BeanFactoryCacheOperationSourceAdvisor 介绍
- CacheInterceptor 介绍

2、一张大图了解@Cacheable 注解的类结构

为了通过@Cacheable 进行缓存需要在容器中注入 3 个 bean：CacheOperationSource、BeanFactoryCacheOperationSourceAdvisor、CacheInterceptor。如图 1 所示，带颜色的部分就是我们需要关注的部分，其中 BeanFactoryCacheOperationSourceAdvisor 作为 bean 工厂是用来产生操作缓存源，并且对缓存@Cacheable 注解进行增强的，这一点从类的名字上可以看出来。CacheInterceptor 是缓存的拦截器，是整个过程中处理缓存信息的类，其主要实现的方法是 execute，这个方法是从 CacheAspectSupoort 类中继承过来的。最后就是 CacheOperationSource，它是处理缓存操作的源头，CacheInterceptor 也就是围绕这个缓存源头展开工作。

CacheOperationSource 有一个子类为 AnnotationCacheOperationSource，这个类是作为缓存操作的源头，它会关联 CacheAnnotationParser 接口，使用

SpringCacheAnnotationParser 对缓存信息进行解析。在图的最左边是缓存操作的几个类，所有的缓存操作 CacheOperation 是实现了 BasicOperation 接口，针对 @Cacheable 缓存注释是继承与 CacheOperation 类的。

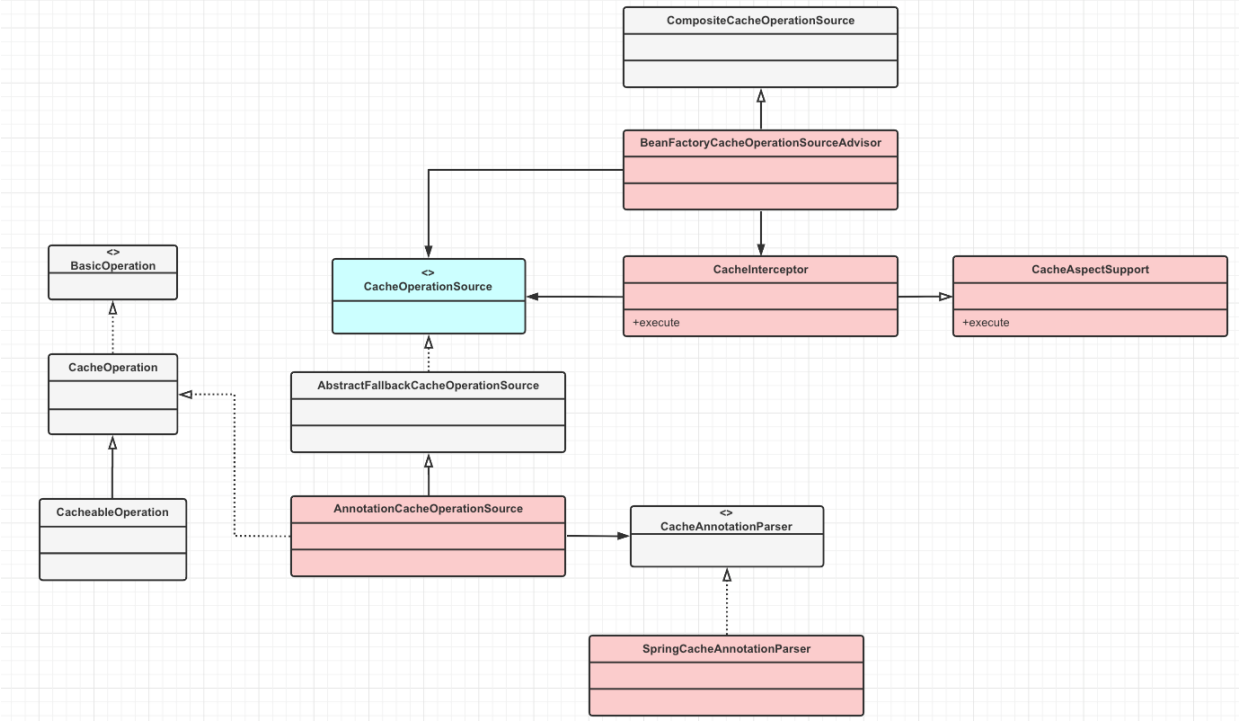

图 1 @Cacheable 注解的类结构

3、CacheOperationSource 介绍

介绍完了@Cacheable 的类/接口结构再来对其中涉及到的类文件进行表述。如图 2 所示，@Cacheable 作为接口定义，可以被用在方法和类型上面，在接口中还对其属性进行了描述。

```java
@Target({ElementType.METHOD, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Cacheable {
    // 缓存名称  可以写多个~
    @AliasFor("cacheNames")
    String[] value() default {};
    @AliasFor("value")
    String[] cacheNames() default {};
    // 支持写SpEL，切可以使用#root
    String key() default "";
    // Mutually exclusive: 它和key属性互相排斥。请只使用一个
    String keyGenerator() default "";
    String cacheManager() default "";
    String cacheResolver() default "";
    // SpEL，可以使用#root。  只有true时，才会作用在这个方法上
    String condition() default "";
    // 可以写SpEL #root，并且可以使用#result拿到方法返回值~~~
    String unless() default "";
    boolean sync() default false;
}
```

图 2 Cacheable

为了方便大家了解其属性结构，如图 3 所示，将属性通过的表格的方式展示出来。

属性名	解释
value	缓存的名称。可定义多个（至少需要定义一个）
cacheNames	同value属性
keyGenerator	key生成器。字符串为：beanName
key	缓存的 key。可使用SpEL。优先级大于 `keyGenerator`
cacheManager	缓存管理器。填写beanName
cacheResolver	缓存处理器。填写beanName
condition	缓存条件。若填写了，返回true才会执行此缓存。可使用SpEL
unless	否定缓存。false就生效。可以写SpEL
sync	true：所有相同key的同线程顺序执行。默认值是false
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
beforeInvocation	是否在方法执行前就清空，缺省为 false，如果指定为 true

图 3 Cacheable 属性

既然有了@Cacheable 可以在类型和方法上面标注缓存，就需要有一个对缓存进行操作的类。如图 4 所示，CacheableOperation 继承与 CacheOperation 类，在 CacheableOperation 通过 build 方法将缓存操作进行生成。

```
public class CacheableOperation extends CacheOperation {
    @Nullable
    private final String unless;
    private final boolean sync;

    public CacheableOperation(CacheableOperation.Builder b) {
        super(b);
        this.unless = b.unless;
        this.sync = b.sync;
    }
    ... // 省略get方法（无set方法哦）

    // @since 4.3
    public static class Builder extends CacheOperation.Builder {
        @Nullable
        private String unless;
        private boolean sync;
        ... // 生路set方法（没有get方法哦~）

        // Spring4.3抽象的这个技巧还是不错的，此处传this进去即可
        @Override
        public CacheableOperation build() {
            return new CacheableOperation(this);
        }

        @Override
        protected StringBuilder getOperationDescription() {
            StringBuilder sb = super.getOperationDescription();
            sb.append(" | unless='");
            sb.append(this.unless);
            sb.append("'");
            sb.append(" | sync='");
            sb.append(this.sync);
            sb.append("'");
            return sb;
        }

    }
}
```

图 4 CacheableOperation

有了 CacheableOperation 之后需要提供一个接口给拦截器使用，也就是说拦截器通过注释获取要缓存的对象以后需要调用缓存操作，此时就有了接口 CacheOperationSource，我们称之为缓存属性源。如图 5 所示，CacheOperationSource 通过 getCacheOperations 方法返回所有缓存操作的 CacheOperation 集合。

```
public interface CacheOperationSource {

    // 返回此Method方法上面所有的缓存操作CacheOperation 集合
    // 显然一个Method上可以对应有多个缓存操作
    @Nullable
    Collection<CacheOperation> getCacheOperations(Method method, @Nullable Class<?> targetClass)
}
```

图 5 CacheOperationSource

那么 CacheOperationSource 如何与 CacheOperation 产生关系的呢？如图 6 所示，AbstractFallbackCacheOperationSource 会实现 CacheOperationSource 接口，同时 AnnotationCacheOperationSource 继承与 AbstractFallbackCacheOperationSource，在 AnnotationCacheOperationSource 中会依赖 CacheOperation，调用其中对缓存操作的方法。

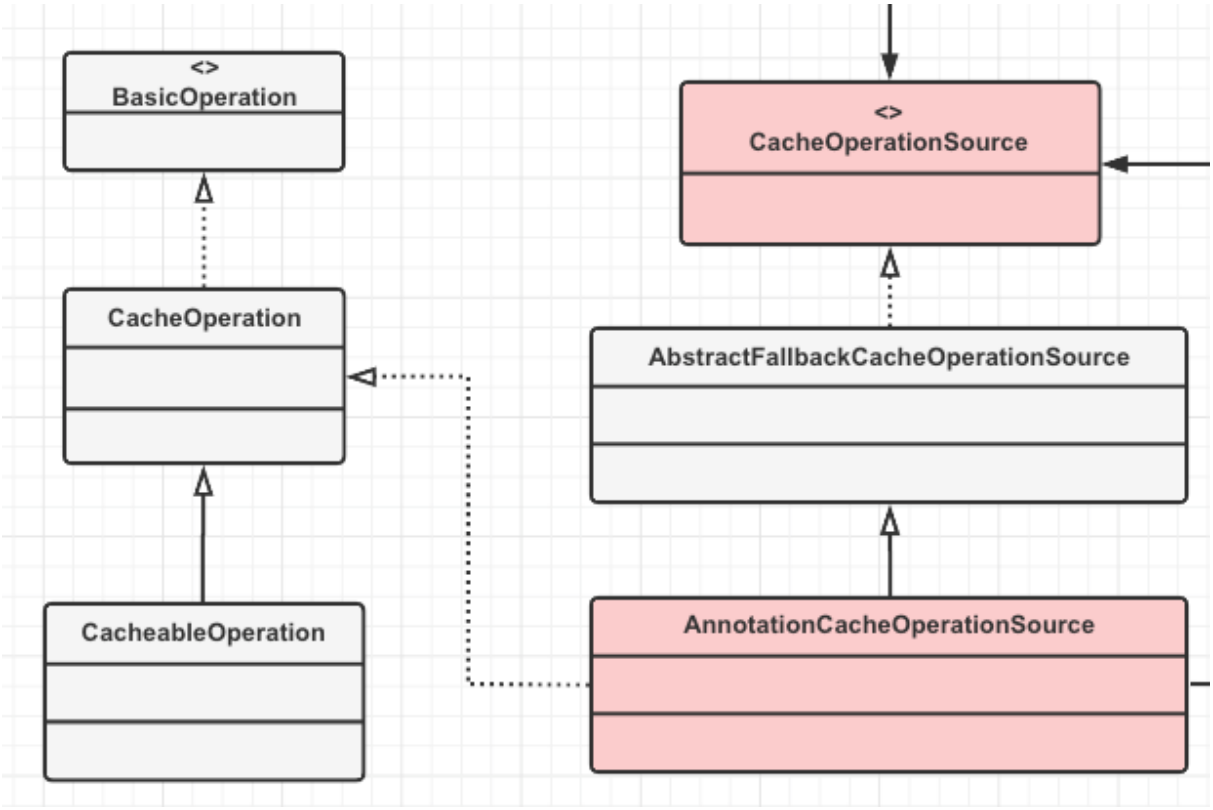

图 6 CacheOperationSource 与 CacheOperation 产生关系

4、BeanFactoryCacheOperationSourceAdvisor 介绍

说完了缓存操作源，在来看看它是如何被使用的。

BeanFactoryCacheOperationSourceAdvisor 就是来整合缓存和增强器的，根据 AOP 的思想有了目标，定义切面和连接点以后就需要通过增强器对方法进行增强操作。

如图 7 所示，cacheAdvisor 方法就是将缓存源和拦截器作为 BeanFactoryCacheOperationSourceAdvisor 的属性进行绑定。而 CacheInterceptor 也是处理缓存的主力军后面会给大家讲到。

```
@Bean(name = CacheManagementConfigUtils.CACHE_ADVISOR_BEAN_NAME)
@Role(BeanDefinition.ROLE_INFRASTRUCTURE)
public BeanFactoryCacheOperationSourceAdvisor cacheAdvisor() {
    BeanFactoryCacheOperationSourceAdvisor advisor = new BeanFactoryCacheOperationSourceAdvisor(
    advisor.setCacheOperationSource(cacheOperationSource());
    advisor.setAdvice(cacheInterceptor());
    if (this.enableCaching != null) {
        advisor.setOrder(this.enableCaching.<Integer>getNumber("order"));
    }
    return advisor;
}
```

图 7 BeanFactoryCacheOperationSourceAdvisor

如图 8 所示，BeanFactoryCacheOperationSourceAdvisor 继承与 AbstractBeanFactoryPointcutAdvisor 类，其中定义 pointcut 设置 cacheOperationSource。

```
public class BeanFactoryCacheOperationSourceAdvisor extends AbstractBeanFactoryPointcutAdvisor {
    @Nullable
    private CacheOperationSource cacheOperationSource;
    // 切面Pointcut
    private final CacheOperationSourcePointcut pointcut = new CacheOperationSourcePointcut() {
        @Override
        @Nullable
        protected CacheOperationSource getCacheOperationSource() {
            return cacheOperationSource;
        }
    };
    public void setCacheOperationSource(CacheOperationSource cacheOperationSource) {
        this.cacheOperationSource = cacheOperationSource;
    }
    // 注意：此处你可以自定义一个ClassFilter，过滤掉你想忽略的类
    public void setClassFilter(ClassFilter classFilter) {
        this.pointcut.setClassFilter(classFilter);
    }
    @Override
    public Pointcut getPointcut() {
        return this.pointcut;
    }
}
```

图 8 BeanFactoryCacheOperationSourceAdvisor

5、CacheInterceptor 介绍

说完缓存操作来源和生成缓存来源、增强器的工厂类以后，再来看看处理缓存的 CacheInterceptor，从字面意思可以看出它主要是做缓存拦截器的。如图 9 所示，它继承与 CacheAspectSupport 类，并且实现了 invoke 方法，在这个方法中有执行了 execute 方法，这个方法来自父类。

```
public class CacheInterceptor extends CacheAspectSupport implements MethodInterceptor, Serializable {

    @Override
    @Nullable
    public Object invoke(final MethodInvocation invocation) throws Throwable {
        Method method = invocation.getMethod();
        // 采用函数的形式，最终把此函数传交给父类的execute()去执行
        // 但是很显然，最终**执行目标方法**的是invocation.proceed();它
        //这里就是对执行方法调用的一次封装，主要是为了处理对异常的包装。
        CacheOperationInvoker aopAllianceInvoker = () -> {
            try {
                return invocation.proceed();
            }
            catch (Throwable ex) {
                throw new CacheOperationInvoker.ThrowableWrapper(ex);
            }
        };
        try {
            // //真正地去处理缓存操作的执行，很显然这是父类的方法，所以我们要到父类CacheAspectSupport中去看看。
            return execute(aopAllianceInvoker, invocation.getThis(), method, invocation.getArguments(
        } catch (CacheOperationInvoker.ThrowableWrapper th) {
            throw th.getOriginal();
        }
    }

}
```

图 9 invoke

接着看父类的 excute 方法，如图 10 所示，我们将 CacheAspectSupport 中的 execute 方法截取给大家讲解。其中红框的部分，首先获取 CacheOperationSource 也就是缓存操作源，然后通过 getCacheOperations 方法获取这个操作源中的所有对缓存的操作。保存到 operations 变量中，最后又调用了一个 execute 函数传入了 CacheOperationContexts 这个是缓存操作的上下文，所有的操作都会基于这个上下文进行，而且是多个上下文。

```
// 父类最为核心的方法，真正执行目标方法 + 缓存操作
@Nullable
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)
    // 如果已经表示初始化过了(有CacheManager，CacheResolver了)，执行这里
    if (this.initialized) {
        // getTargetClass拿到原始Class  解剖代理（N层都能解开）
        Class<?> targetClass = getTargetClass(target);
        CacheOperationSource cacheOperationSource = getCacheOperationSource();

        if (cacheOperationSource != null) {
            // 简单的说就是拿到该方法上所有的CacheOperation缓存操作，最终一个一个的执行~~~~
            Collection<CacheOperation> operations = cacheOperationSource.getCacheOperations(method, targetClass);
            if (!CollectionUtils.isEmpty(operations)) {
                // CacheOperationContexts是非常重要的一个私有内部类
                // 注意它是复数哦~不是CacheOperationContext单数  所以它就像持有多个注解上下文一样  一个个执行吧
                return execute(invoker, method, new CacheOperationContexts(operations, method, args, target, targetClass));
            }
        }
    }
    // 若还没初始化  直接执行目标方法即可
    return invoker.invoke();
}
```

图 10 execute

由于上面的代码会再调用一个 execute 方法，我们跟进去查看。如图 11 所示，红框的部分可以清晰地看到，对上下文 context 进行遍历，获取缓存对应的 key 和 cache，最后通过 wrapCacheValue 方法对其进行包装返回给调用者。

```
private Object execute(final CacheOperationInvoker invoker, Method method, CacheOperationContexts contexts) {
    // Special handling of synchronized invocation
    // 如果是需要同步执行的话，这块还是
    if (contexts.isSynchronized()) {
        CacheOperationContext context = contexts.get(CacheableOperation.class).iterator().next();
        if (isConditionPassing(context, CacheOperationExpressionEvaluator.NO_RESULT)) {
            Object key = generateKey(context, CacheOperationExpressionEvaluator.NO_RESULT);
            Cache cache = context.getCaches().iterator().next();
            try {
                return wrapCacheValue(method, cache.get(key, () -> unwrapReturnValue(invokeOperation(invoker))));
            }
            catch (Cache.ValueRetrievalException ex) {
                // The invoker wraps any Throwable in a ThrowableWrapper instance so we
                // can just make sure that one bubbles up the stack.
                throw (CacheOperationInvoker.ThrowableWrapper) ex.getCause();
            }
        }
        else {
            // No caching required, only call the underlying method
            return invokeOperation(invoker);
        }
    }
```

图 11 execute

6、总结

本节课对 Spring Cache 中的@Cacheable 注释的实现原理进行讲解，首先通过一张类结构的大图讲@Cacheable 注释涉及到的类和结构以及它们之间的关系都描述出来。然后把 CacheOperationSource 、BeanFactoryCacheOperationSourceAdvisor 、CacheInterceptor 做为切入点展开介绍几个重要的类和接口的源码。

下节课是代码实战，基于 Spring Cache 优化首页教师列表数据的热点查询功能。

下期见，拜拜。