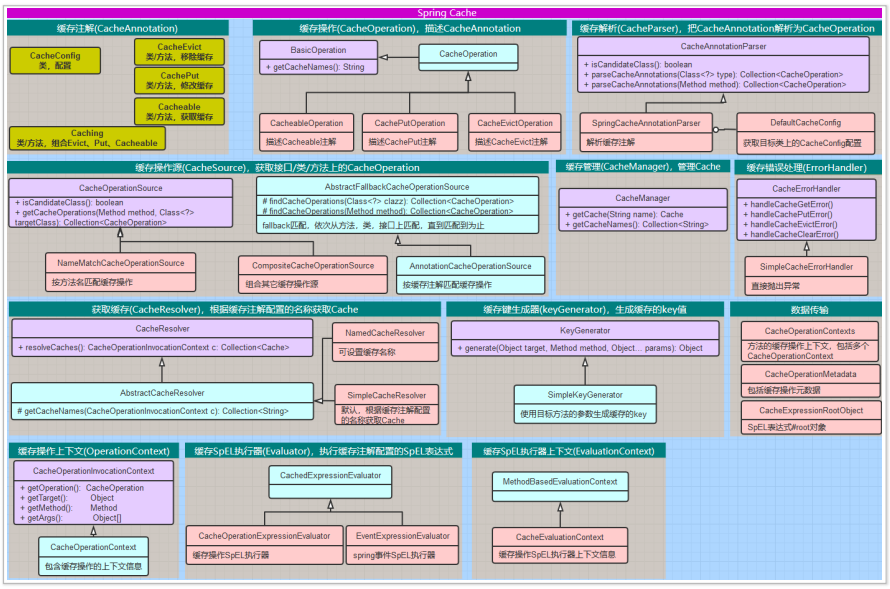
(44 条消息) 7.springboot cache 基础类 zhouping118 的博客 - CSDN 博客

1.spring cache 基础类简介

1.1. 结构



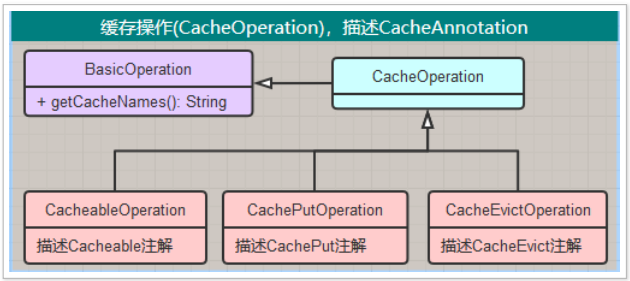
1.2. 运行过程简要说明

- 使用：在 User 类上编写方法 String getName()，标注注解 @Cacheable;
- 运行过程
  - 拦截：
    - 拦截器拦截标注有 @Cacheable 的 getName 方法，调用父类 CacheAspectSupport 的 execute 方法执行；
  - 解析缓存注解：
    - 使用 AnnotationCacheOperationSource，AnnotationCacheOperationSource 委托 SpringCacheAnnotationParser 把 @Cacheable 解析为 CacheableOperation，返回 CacheOperation 集合（一个方法上可以有多个缓存注解，所以返回集合）；
    - 使用 DefaultCacheConfig 读取类上的 CacheConfig 参数;
  - 拼装配置：
    - 把 getName 上的所有缓存注解封装为 CacheOperationContexts。里面包含多个 CacheOperationContext，CacheableOperation 封装为其中的一个 CacheOperationContext（CacheOperationContext 包含 CacheOperationMetadata）；
  - 如果配置了 sync=true（只有解 @Cacheable 可配置 sync）
    - 获取缓存的 key 值
      - 如果 @Cacheable 没有配置 key，则使用 SimpleKeyGenerator 生成 key;
      - 如果配置了 key，使用 CacheOperationExpressionEvaluator 创建 EvaluationContext，然后计算 key 中的 SpEL 表达式的值，做为 key 返回;
    - 获取 Cache
      - 使用 SimpleCacheResolver，SimpleCacheResolver 根据 @Cacheable 配置的 cacheNames，调用 CacheManager 获取 Cache;
      - 然后调用 Cache 获取缓存，如果没有缓存，则调用目标方法，并返回目标方法的返回值;
  - 如果没有配置 sync=true
    - 执行移除缓存 (beforeInvocation=true)
      - 如果指定了 key，执行上面获取缓存的 key 的逻辑。移除指定 key 的缓存 / 移除所有缓存;

- 执行获取缓存
  - 执行上面获取缓存的 key 的逻辑，然后从配置的所有缓存中获取缓存值；
  - 如果缓存没有命中，则执行目标方法，并把返回值保存在缓存中；
- 执行修改缓存
- 执行移除缓存 (beforeInvocation=false)

2.spring cache 基础类源码解读

2.1. 缓存操作 (CacheOperation)



作用

- 描述获取 / 修改 / 移除三个缓存注解，包含了缓存注解中的配置属性；
- 实现 BasicOperation 接口，Set getCacheNames() 方法，返回值会作为 CacheManager 中管理的 CaChe 的名称；

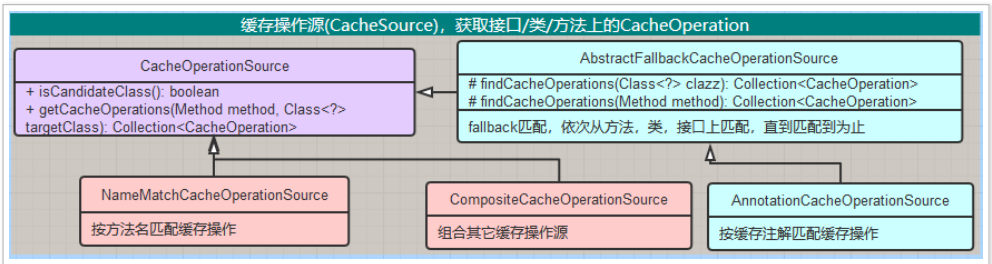
缓存操作 CacheOperation

```
public abstract class CacheOperation implements BasicOperation {
    //**三个缓存注解的公共配置
        private final String name;
        private final Set<String> cacheNames;
        private final String key;
        private final String keyGenerator;
        private final String cacheManager;
        private final String cacheResolver;
        private final String condition;
        private final String toString;

        protected CacheOperation(Builder b) {
                this.name = b.name;
                ...
                this.toString = b.getOperationDescription().toString();
        }

        ...

    //**缓存操作构建器
        public abstract static class Builder {
            ...
        }
}
```

修改 CachePutOperation

```
public class CachePutOperation extends CacheOperation {
    //**@CachePut额外的配置
        private final String unless;

        public CachePutOperation(CachePutOperation.Builder b) {
                super(b);
                this.unless = b.unless;
        }

    //**CachePutOperation构建器
        public static class Builder extends CacheOperation.Builder {
        ...
        }
}
```

2.2. 缓存操作源 (CacheSource)



作用

获取接口 / 类 / 方法上所有的 CacheOperation，一个接口 / 类 / 方法上可以有多个缓存注解。先获取缓存注解，然后调用缓存解析 (CacheParser)，把 CacheAnnotation 解析为 CacheOperation；

缓存操作源 CacheOperationSource

```
public interface CacheOperationSource {
    //**5.2版本新增，当前缓存操作源是否支持处理目标类
        default boolean isCandidateClass(Class<?> targetClass) {
                return true;
        }

    //**获取方法上所有的CacheOperation
        Collection<CacheOperation> getCacheOperations(Method method, @Nullable Class<?> targetClass);

}
```

按方法名匹配 NameMatchCacheOperationSource

作用：按方法匹配获取方法上所有的 CacheOperation。匹配不到返回 null；

```
public class NameMatchCacheOperationSource implements CacheOperationSource, Serializable {
    //**缓存所有方法的缓存操作，key为方法名，value为缓存操作集合
        private Map<String, Collection<CacheOperation>> nameMap = new LinkedHashMap<>();

    //**设置方法的缓存操作
        public void setNameMap(Map<String, Collection<CacheOperation>> nameMap) {
                nameMap.forEach(this::addCacheMethod);
        }

    //**添加指定方法的缓存操作
        public void addCacheMethod(String methodName, Collection<CacheOperation> ops) {
                this.nameMap.put(methodName, ops);
        }

    //**从nameMap中获取方法上所有的CacheOperation
        public Collection<CacheOperation> getCacheOperations(Method method, @Nullable Class<?> targetClass) {
            //**以方法名称作为key从nameMap中获取CacheOperation
                String methodName = method.getName();
                Collection<CacheOperation> ops = this.nameMap.get(methodName);

        //**如果nameMap中不存在，根据下面的isMatch方法匹配规则获取匹配度最高的一个
                if (ops == null) {
                        String bestNameMatch = null;
                        for (String mappedName : this.nameMap.keySet()) {
                                if (isMatch(methodName, mappedName) && (bestNameMatch == null || bestNameMatch.lengt
                                        ops = this.nameMap.get(mappedName);
                                        bestNameMatch = mappedName;
                                }
                        }
                }

                return ops;
        }

    //**匹配规则："equals, xxx*, *xxx, *xxx*, xxx*yyy"
        protected boolean isMatch(String methodName, String mappedName) {
                return PatternMatchUtils.simpleMatch(mappedName, methodName);
        }
}
```

组合缓存操作 CompositeCacheOperationSource

作用：组合其它的缓存操作源，可同时使用多种缓存操作源。CacheManager 也有类似的组合类 CompositeCacheManager；

```
public class CompositeCacheOperationSource implements CacheOperationSource, Serializable {
    //**持有其它缓存操作源集合
        private final CacheOperationSource[] cacheOperationSources;
```

```
        //**遍历持有的缓存操作源，只要其中一个支持处理目标类，则返回true
        public boolean isCandidateClass(Class<?> targetClass) {
                for (CacheOperationSource source : this.cacheOperationSources) {
                        if (source.isCandidateClass(targetClass)) {
                                return true;
                        }
                }
                return false;
        }

        //**遍历持有的缓存操作源，返回最后一个获取到的缓存操作集合(如果有多个缓存操作源获取到了缓存操作集合，只返回
        public Collection<CacheOperation> getCacheOperations(Method method, @Nullable Class<?> targetClass) {
                Collection<CacheOperation> ops = null;
                for (CacheOperationSource source : this.cacheOperationSources) {
                        Collection<CacheOperation> cacheOperations = source.getCacheOperations(method, targetClass);
                        if (cacheOperations != null) {
                                if (ops == null) {
                                        ops = new ArrayList<>();
                                }
                                ops.addAll(cacheOperations);
                        }
                }
                return ops;
        }
}
```

fallback 匹配 AbstractFallbackCacheOperationSource

作用：依次从方法，类，接口上匹配，直到匹配到为止。如果匹配不到，则会计算缓存操作；

```
public abstract class AbstractFallbackCacheOperationSource implements CacheOperationSource {
    //**匹配不到缓存操作集合时，默认的空集合
        private static final Collection<CacheOperation> NULL_CACHING_ATTRIBUTE = Collections.emptyList();
    //**缓存所有方法的缓存操作
        private final Map<Object, Collection<CacheOperation>> attributeCache = new ConcurrentHashMap<>(1024);

    //**从attributeCache获取缓存操作源，如果attributeCache没有，则计算缓存操作源
        public Collection<CacheOperation> getCacheOperations(Method method, @Nullable Class<?> targetClass) {
            //**继承Object的方法，直接返回null
                if (method.getDeclaringClass() == Object.class) {
                        return null;
                }
        //**根据方法和类获取key，key为MethodClassKey对象
                Object cacheKey = getCacheKey(method, targetClass);
                //**从attributeCache中获取缓存操作集合
                Collection<CacheOperation> cached = this.attributeCache.get(cacheKey);

        //**获取到缓存操作集合，如果缓存操作集合为NULL_CACHING_ATTRIBUTE，则返回null
                if (cached != null) {
                        return (cached != NULL_CACHING_ATTRIBUTE ? cached : null);
                }
                else { //**计算缓存操作源
                        Collection<CacheOperation> cacheOps = computeCacheOperations(method, targetClass);
                        if (cacheOps != null) {
                                ...
                                //**计算成功则保存到attributeCache中
                                this.attributeCache.put(cacheKey, cacheOps);
                        }
                        else { //**计算失败则保存NULL_CACHING_ATTRIBUTE
                                this.attributeCache.put(cacheKey, NULL_CACHING_ATTRIBUTE);
                        }
                        return cacheOps;
                }
        }

    //**根据方法和类获取key，key为MethodClassKey对象
        protected Object getCacheKey(Method method, @Nullable Class<?> targetClass) {
                return new MethodClassKey(method, targetClass);
        }

    //**计算缓存操作源，具体从方法/类上获取缓存操作源的方法由子类实现
        private Collection<CacheOperation> computeCacheOperations(Method method, @Nullable Class<?> targetClass) {
                //**如果只解析public方法 && 目标方法非public，直接返回null
                if (allowPublicMethodsOnly() && !Modifier.isPublic(method.getModifiers())) {
                        return null;
                }

                //**实际执行的方法
                Method specificMethod = AopUtils.getMostSpecificMethod(method, targetClass);

                //**从实际执行方法上找缓存操作，如果找到就返回
                Collection<CacheOperation> opDef = findCacheOperations(specificMethod);
                if (opDef != null) {
                        return opDef;
                }

                //**从实际执行的类上找缓存操作，如果找到 && 方法是用户声明的，则返回
                opDef = findCacheOperations(specificMethod.getDeclaringClass());
```

```
                if (opDef != null && ClassUtils.isUserLevelMethod(method)) {
                        return opDef;
                }

        //**如果实际执行的方法 != 标注缓存注解的方法，则缓存注解标注了接口/抽象类的在抽象方法上
        //**从缓存接口/抽象类上查找缓存操作
                if (specificMethod != method) {
                        opDef = findCacheOperations(method);
                        if (opDef != null) {
                                return opDef;
                        }
                        opDef = findCacheOperations(method.getDeclaringClass());
                        if (opDef != null && ClassUtils.isUserLevelMethod(method)) {
                                return opDef;
                        }
                }

                return null;
        }

    //**从类上获取缓存集合
        protected abstract Collection<CacheOperation> findCacheOperations(Class<?> clazz);
    //**方法上获取缓存集合
        protected abstract Collection<CacheOperation> findCacheOperations(Method method);
    //**是否只解析public方法
        protected boolean allowPublicMethodsOnly() {
                return false;
        }
}
```

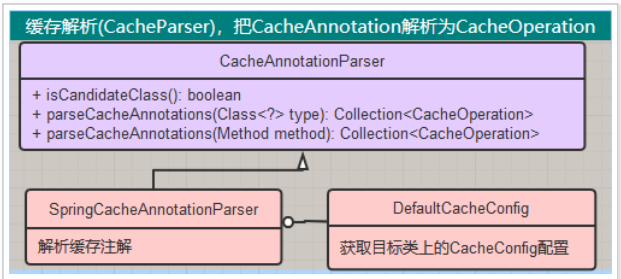
按缓存注解匹配 AnnotationCacheOperationSource

作用：按缓存注解获取方法上所有的 CacheOperation，如果匹配不到，则会把缓存注解解析成缓存操作；

```
public class AnnotationCacheOperationSource extends AbstractFallbackCacheOperationSource implements Serializable {
    //**是否只解析public方法
        private final boolean publicMethodsOnly;
        //**缓存注解解析器
        private final Set<CacheAnnotationParser> annotationParsers;

    //**默认只解析public方法，默认使用SpringCacheAnnotationParser缓存注解解析器
    //**可设置单个/多个缓存注解解析器(spring默认只有SpringCacheAnnotationParser一个实现)
        ...

        //**遍历所有的缓存注解解析器，只要有一个支持解析目标类，则返回true
        //**SpringCacheAnnotationParser支持解析Cacheable、CacheEvict、CachePut和Caching四个缓存注解
        public boolean isCandidateClass(Class<?> targetClass) {
                for (CacheAnnotationParser parser : this.annotationParsers) {
                        if (parser.isCandidateClass(targetClass)) {
                                return true;
                        }
                }
                return false;
        }

        //**调用缓存注解解析器的parseCacheAnnotations，解析类上的缓存注解
        protected Collection<CacheOperation> findCacheOperations(Class<?> clazz) {
                return determineCacheOperations(parser -> parser.parseCacheAnnotations(clazz));
        }

        //**调用缓存注解解析器的parseCacheAnnotations，解析方法上的缓存注解
        protected Collection<CacheOperation> findCacheOperations(Method method) {
                return determineCacheOperations(parser -> parser.parseCacheAnnotations(method));
        }

    //**使用每个缓存注解解析器解析目标方法/类上的缓存注解，并把解析后的缓存操作合并后返回
        protected Collection<CacheOperation> determineCacheOperations(CacheOperationProvider provider) {
                Collection<CacheOperation> ops = null;
                for (CacheAnnotationParser parser : this.annotationParsers) {
                        Collection<CacheOperation> annOps = provider.getCacheOperations(parser);
                        if (annOps != null) {
                                if (ops == null) {
                                        ops = annOps;
                                }
                                else {
                                        Collection<CacheOperation> combined = new ArrayList<>(ops.size() + annOps.si
                                        combined.addAll(ops);
                                        combined.addAll(annOps);
                                        ops = combined;
                                }
                        }
                }
                return ops;
        }

    //**函数式接口，方便调用，等同于Function<CacheAnnotationParser, Collection<CacheOperation>>
        @FunctionalInterface
        protected interface CacheOperationProvider {
```

```
            Collection<CacheOperation> getCacheOperations(CacheAnnotationParser parser);
        }
}
```

2.3. 缓存注解解析器 (CacheParser)



作用

获取类 / 方法上的缓存注解，并解析为 CacheOperation；

缓存注解解析器 CacheAnnotationParser

```
public interface CacheAnnotationParser {
    //**缓存注解解析器是否支持解析目标类
        default boolean isCandidateClass(Class<?> targetClass) {
                return true;
        }

        //**解析类上的缓存注解
        Collection<CacheOperation> parseCacheAnnotations(Class<?> type);

        //**解析方法上的缓存注解
        Collection<CacheOperation> parseCacheAnnotations(Method method);

}
```

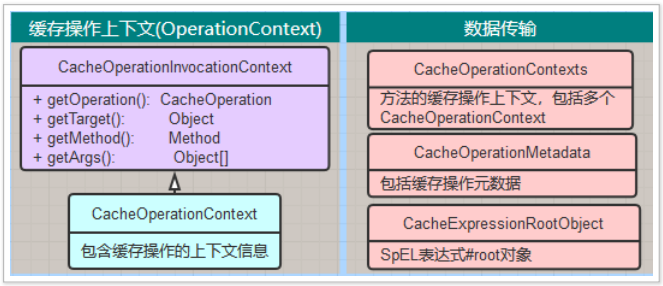
Spring 缓存注解解析器 SpringCacheAnnotationParser

作用：把 Cacheable、CacheEvict、CachePut 和 Caching 四个缓存注解解析成对应的缓存操作，并读取目标类的 @CacheConfig 配置信息；

```
public class SpringCacheAnnotationParser implements CacheAnnotationParser, Serializable {
    //**支持解析的缓存注解
        private static final Set<Class<? extends Annotation>> CACHE_OPERATION_ANNOTATIONS = new LinkedHashSet<>(8);
        static {
                CACHE_OPERATION_ANNOTATIONS.add(Cacheable.class);
                CACHE_OPERATION_ANNOTATIONS.add(CacheEvict.class);
                CACHE_OPERATION_ANNOTATIONS.add(CachePut.class);
                CACHE_OPERATION_ANNOTATIONS.add(Caching.class);
        }

    //**支持解析Cacheable、CacheEvict、CachePut和Caching四个缓存注解
        public boolean isCandidateClass(Class<?> targetClass) {
                return AnnotationUtils.isCandidateClass(targetClass, CACHE_OPERATION_ANNOTATIONS);
        }

        //**解析类上的缓存注解，读取类上的@CacheConfig配置信息
        public Collection<CacheOperation> parseCacheAnnotations(Class<?> type) {
                DefaultCacheConfig defaultConfig = new DefaultCacheConfig(type);
                return parseCacheAnnotations(defaultConfig, type);
        }

        //**解析方法上的缓存注解，读取方法所属类上的@CacheConfig配置信息
        public Collection<CacheOperation> parseCacheAnnotations(Method method) {
                DefaultCacheConfig defaultConfig = new DefaultCacheConfig(method.getDeclaringClass());
                return parseCacheAnnotations(defaultConfig, method);
        }

        //**解析类/方法上的缓存注解
        private Collection<CacheOperation> parseCacheAnnotations(DefaultCacheConfig cachingConfig, AnnotatedElement
                Collection<CacheOperation> ops = parseCacheAnnotations(cachingConfig, ae, false);
                if (ops != null && ops.size() > 1) {
                        Collection<CacheOperation> localOps = parseCacheAnnotations(cachingConfig, ae, true);
                        if (localOps != null) {
                                return localOps;
                        }
                }
                return ops;
        }

        //**解析类/方法上的缓存注解
```

```
    private Collection<CacheOperation> parseCacheAnnotations(
                    DefaultCacheConfig cachingConfig, AnnotatedElement ae, boolean localOnly) {

    /**获取类的注解：getAllMergedAnnotations会获取当前类的指定注解，findAllMergedAnnotations会获取当前类及其接口
       获取方法的注解：getAllMergedAnnotations会获取当前方法的指定注解，findAllMergedAnnotations会获取当前方法及
            Collection<? extends Annotation> anns = (localOnly ?
                    AnnotatedElementUtils.getAllMergedAnnotations(ae, CACHE_OPERATION_ANNOTATIONS) :
                    AnnotatedElementUtils.findAllMergedAnnotations(ae, CACHE_OPERATION_ANNOTATIONS));
            if (anns.isEmpty()) {
                    return null;
            }

    //**分别解析不同类型的缓存注解
            final Collection<CacheOperation> ops = new ArrayList<>(1);
            anns.stream().filter(ann -> ann instanceof Cacheable).forEach(
                    ann -> ops.add(parseCacheableAnnotation(ae, cachingConfig, (Cacheable) ann)));
            anns.stream().filter(ann -> ann instanceof CacheEvict).forEach(
                    ann -> ops.add(parseEvictAnnotation(ae, cachingConfig, (CacheEvict) ann)));
            anns.stream().filter(ann -> ann instanceof CachePut).forEach(
                    ann -> ops.add(parsePutAnnotation(ae, cachingConfig, (CachePut) ann)));
            anns.stream().filter(ann -> ann instanceof Caching).forEach(
                    ann -> parseCachingAnnotation(ae, cachingConfig, (Caching) ann, ops));
            return ops;
    }

//**解析@Cacheable
    private CacheableOperation parseCacheableAnnotation(
                    AnnotatedElement ae, DefaultCacheConfig defaultConfig, Cacheable cacheable) {
    //**创建CacheableOperation构建器
            CacheableOperation.Builder builder = new CacheableOperation.Builder();

            builder.setName(ae.toString());
            builder.setCacheNames(cacheable.cacheNames());
            builder.setCondition(cacheable.condition());
            builder.setUnless(cacheable.unless());
            builder.setKey(cacheable.key());
            builder.setKeyGenerator(cacheable.keyGenerator());
            builder.setCacheManager(cacheable.cacheManager());
            builder.setCacheResolver(cacheable.cacheResolver());
            builder.setSync(cacheable.sync());
    //**合并类上的@CacheConfig配置
            defaultConfig.applyDefault(builder);
            CacheableOperation op = builder.build();
            //**验证不能同时配置'key' and 'keyGenerator'
            //**验证不能同时配置'cacheManager' and 'cacheResolver'
            validateCacheOperation(ae, op);

            return op;
    }

//**解析@CacheEvict和@CachePut
...

//**解析@caching,@caching是其它三个缓存注解的组合
    private void parseCachingAnnotation(
                    AnnotatedElement ae, DefaultCacheConfig defaultConfig, Caching caching, Collection<CacheOper

            Cacheable[] cacheables = caching.cacheable();
            for (Cacheable cacheable : cacheables) {
                    ops.add(parseCacheableAnnotation(ae, defaultConfig, cacheable));
            }
            ...
    }

    private static class DefaultCacheConfig {
    ...
    //**读取类上的CacheConfig配置，并与参数CacheOperation构造器的配置合并
            public void applyDefault(CacheOperation.Builder builder) {
                    if (!this.initialized) {
                            CacheConfig annotation = AnnotatedElementUtils.findMergedAnnotation(this.target, Ca
                            if (annotation != null) {
                                    this.cacheNames = annotation.cacheNames();
                                    this.keyGenerator = annotation.keyGenerator();
                                    this.cacheManager = annotation.cacheManager();
                                    this.cacheResolver = annotation.cacheResolver();
                            }
                            this.initialized = true;
                    }
        ...
            }
    }
}
```

2.4. 缓存操作上下文 (CacheOperationContext)



作用

保存缓存操作执行时的参数信息，包括目标类 / 方法，参数，缓存操作、缓存名称、Cache 等等；

方法级缓存操作上下文 CacheOperationContexts

作用：保存目标方法上所有的缓存操作上下文信息 (可包括多个缓存操作上下文)，并计算缓存操作是否同步执行；

```
private class CacheOperationContexts {
    //**每个种类的缓存操作及其上下文
        private final MultiValueMap<Class<? extends CacheOperation>, CacheOperationContext> contexts;
    //**是否同步
        private final boolean sync;

        public CacheOperationContexts(Collection<? extends CacheOperation> operations, Method method,
                        Object[] args, Object target, Class<?> targetClass) {
                this.contexts = new LinkedMultiValueMap<>(operations.size());
                //**遍历目标方法的所有缓存操作，把每个缓存操作封装为一个缓存操作上下文，并保存在contexts中；
                for (CacheOperation op : operations) {
                        this.contexts.add(op.getClass(), getOperationContext(op, method, args, target, targetClass));
                }
                //**是否同步执行，一个方法有多个缓存操作，但只有一个是否同步执行的开关
                this.sync = determineSyncFlag(method);
        }

        //**是否同步执行
        //**设置了@Cacheable(sync=true)，则不能设置其它任何缓存注解，且@Cacheable缓存名称只能设置一个，不能设置unle
        private boolean determineSyncFlag(Method method) {
            //**获取目标方法上的@Cacheable缓存操作，如果没有，直接返回false（因为只@Cacheable有sync参数）
                List<CacheOperationContext> cacheOperationContexts = this.contexts.get(CacheableOperation.class);
                if (cacheOperationContexts == null) {
                        return false;
                }

                //**目标方法上的所有@Cacheable，只要其中一个设置了sync=true，则syncEnabled=true
                boolean syncEnabled = false;
                for (CacheOperationContext cacheOperationContext : cacheOperationContexts) {
                        if (((CacheableOperation) cacheOperationContext.getOperation()).isSync()) {
                                syncEnabled = true;
                                break;
                        }
                }

                //**如果syncEnabled=true
                if (syncEnabled) {
                    //**设置了sync=true，则不能标注除@Cacheable外的其它类型的缓存注解
                        if (this.contexts.size() > 1) {
                                throw new IllegalStateException(
                                                "@Cacheable(sync=true) cannot be combined with other cache operatior
                        }

                        //**设置了sync=true，也不能标注其它的@Cacheable
                        if (cacheOperationContexts.size() > 1) {
                                throw new IllegalStateException(
                                                "Only one @Cacheable(sync=true) entry is allowed on '" + method + "
                        }

                        //**获取设置了sync=true的缓存操作
                        CacheOperationContext cacheOperationContext = cacheOperationContexts.iterator().next();
                        CacheableOperation operation = (CacheableOperation) cacheOperationContext.getOperation();

                        //**设置了sync=true，cacheNames中只能设置一个名称
                        if (cacheOperationContext.getCaches().size() > 1) {
                                throw new IllegalStateException(
                                                "@Cacheable(sync=true) only allows a single cache on '" + operation
                        }

                        //**设置了sync=true，不能设置unless
                        if (StringUtils.hasText(operation.getUnless())) {
                                throw new IllegalStateException(
                                                "@Cacheable(sync=true) does not support unless attribute on '" + ope
                        }
                        return true;
```

```
                }
                return false;
        }
}
```

缓存操作上下文 CacheOperationContext

作用：保存单个缓存操作执行时的参数信息，并计算 condition 和 unless 参数是否通过；

```
protected class CacheOperationContext implements CacheOperationInvocationContext<CacheOperation> {
    //**缓存操作元数据
        private final CacheOperationMetadata metadata;
    //**参数
        private final Object[] args;
    //**目标对象
        private final Object target;
    //**Cache集合
        private final Collection<? extends Cache> caches;
    //**Cache名称集合
        private final Collection<String> cacheNames;
    //**condition是否通过
        private Boolean conditionPassing;

        public CacheOperationContext(CacheOperationMetadata metadata, Object[] args, Object target) {
                this.metadata = metadata;
                this.args = extractArgs(metadata.method, args);
                this.target = target;
                //**调用CacheAspectSupport的getCaches方法获取Cache，getCaches方法使用SimpleCacheResolver获取Cache
                this.caches = CacheAspectSupport.this.getCaches(this, metadata.cacheResolver);
                this.cacheNames = createCacheNames(this.caches);
        }

    //**计算设置的condition是否通过
        protected boolean isConditionPassing(@Nullable Object result) {
            //**如果conditionPassing为null，则计算conditionPassing
                if (this.conditionPassing == null) {
                    //**如果设置了condition，则进行计算
                        if (StringUtils.hasText(this.metadata.operation.getCondition())) {
                            //**使用CacheOperationExpressionEvaluator创建SpEL执行上下文
                                EvaluationContext evaluationContext = createEvaluationContext(result);
                                //**使用CacheOperationExpressionEvaluator执行SpEL计算condition是否通过
                                this.conditionPassing = evaluator.condition(this.metadata.operation.getCondition(),
                                                this.metadata.methodKey, evaluationContext);
                        }
                        else {  //**如果没有设置condition，conditionPassing=true
                                this.conditionPassing = true;
                        }
                }
                return this.conditionPassing;
        }

    //**计算是否能修改缓存
        protected boolean canPutToCache(@Nullable Object value) {
                String unless = "";
                //**如果当前注解是@Cacheable，获取unless参数
                if (this.metadata.operation instanceof CacheableOperation) {
                        unless = ((CacheableOperation) this.metadata.operation).getUnless();
                }

                //**如果当前注解是@CachePut，获取unless参数
                else if (this.metadata.operation instanceof CachePutOperation) {
                        unless = ((CachePutOperation) this.metadata.operation).getUnless();
                }

                //**如果设置了unless参数，执行SpEL表达式计算unless是否通过
                if (StringUtils.hasText(unless)) {
                        EvaluationContext evaluationContext = createEvaluationContext(value);
                        return !evaluator.unless(unless, this.metadata.methodKey, evaluationContext);
                }
                return true;  //**没有设置unless，直接返回true
        }

        //**生成Cache的key值
        protected Object generateKey(@Nullable Object result) {
            //**如果设置了key参数，执行SpEL表达式计算key值
                if (StringUtils.hasText(this.metadata.operation.getKey())) {
                        EvaluationContext evaluationContext = createEvaluationContext(result);
                        return evaluator.key(this.metadata.operation.getKey(), this.metadata.methodKey, evaluationC
                }

                //**没有设置key参数，使用KeyGenerator生成key
                return this.metadata.keyGenerator.generate(this.target, this.metadata.method, this.args);
        }

    //**使用CacheOperationExpressionEvaluator创建SpEL执行上下文
        private EvaluationContext createEvaluationContext(@Nullable Object result) {
                return evaluator.createEvaluationContext(this.caches, this.metadata.method, this.args,
                                this.target, this.metadata.targetClass, this.metadata.targetMethod, result, beanFact
```

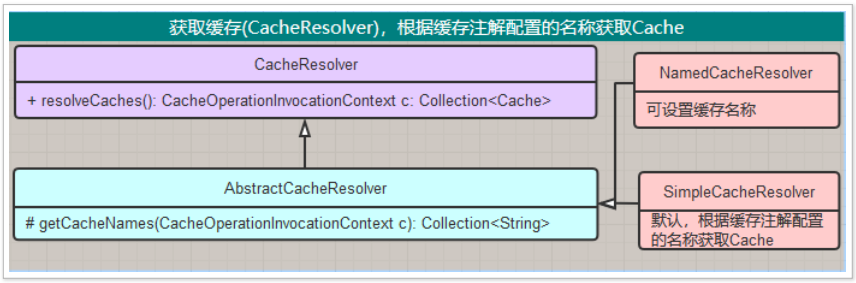
```
        }
}
```

#root 对象 CacheExpressionRootObject

作用：代表缓存注解中 #root 对象，CacheExpressionRootObject 的属性即 #root 可引用的参数；

```
class CacheExpressionRootObject {
        private final Collection<? extends Cache> caches;
        private final Method method;
        private final Object[] args;
        private final Object target;
        private final Class<?> targetClass;
}
```

2.5. 获取缓存 (CacheResolver)



作用

根据缓存注解配置的名称调用 CacheManager 获取 Cache；

抽象类 AbstractCacheResolver

```
public abstract class AbstractCacheResolver implements CacheResolver, InitializingBean {
    //**引用CacheManager，且CacheManager不能为空
        private CacheManager cacheManager;
        public void afterPropertiesSet()  {
                Assert.notNull(this.cacheManager, "CacheManager is required");
        }

    //**获取缓存名称，并根据名称从CacheManager获取Cache
        public Collection<? extends Cache> resolveCaches(CacheOperationInvocationContext<?> context) {
                Collection<String> cacheNames = getCacheNames(context);
                if (cacheNames == null) {
                        return Collections.emptyList();
                }
                Collection<Cache> result = new ArrayList<>(cacheNames.size());
                for (String cacheName : cacheNames) {
                        Cache cache = getCacheManager().getCache(cacheName);
                        if (cache == null) {
                                throw new IllegalArgumentException("Cannot find cache named '" +
                                                cacheName + "' for " + context.getOperation());
                        }
                        result.add(cache);
                }
                return result;
        }

        //**获取缓存名称
        protected abstract Collection<String> getCacheNames(CacheOperationInvocationContext<?> context);
}
```

简单的 SimpleCacheResolver

作用：从缓存注解获取配置的 cacheNames 作为缓存名称，获取 Cahce；

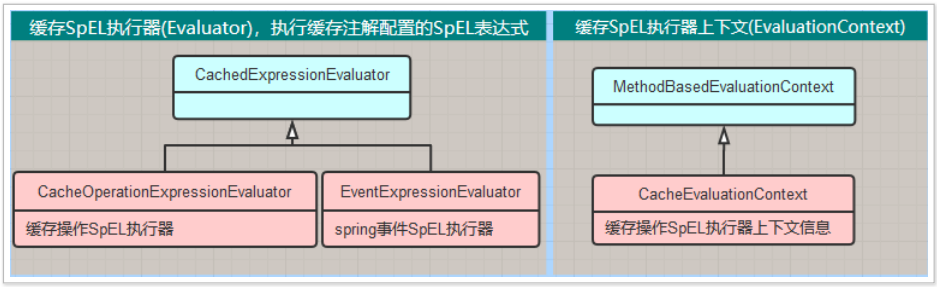
```
public class SimpleCacheResolver extends AbstractCacheResolver {
    //**从缓存注解获取配置的cacheNames作为缓存名称
        protected Collection<String> getCacheNames(CacheOperationInvocationContext<?> context) {
                return context.getOperation().getCacheNames();
        }
}
```

可设置名称 NamedCacheResolver

作用：可设置缓存的名称，根据设置的名称获取 Cahce；

```
public class NamedCacheResolver extends AbstractCacheResolver {
        private Collection<String> cacheNames;

    //**可设置缓存的名称
        public NamedCacheResolver(CacheManager cacheManager, String... cacheNames) {
                super(cacheManager);
                this.cacheNames = new ArrayList<>(Arrays.asList(cacheNames));
        }
        public void setCacheNames(Collection<String> cacheNames) {
                this.cacheNames = cacheNames;
        }

        @Override
        protected Collection<String> getCacheNames(CacheOperationInvocationContext<?> context) {
                return this.cacheNames;
        }
}
```

2.6.SpEL 执行器 (Evaluator)



作用

执行 SpEL 表达式，获取表达式的值；

抽象类 CachedExpressionEvaluator

```
public abstract class CachedExpressionEvaluator {
    //**SpEL解析器
        private final SpelExpressionParser parser;
    //**默认使用SpelExpressionParser解析器
        protected CachedExpressionEvaluator() {
                this(new SpelExpressionParser());
        }

    //**获取表达式对象
        protected Expression getExpression(Map<ExpressionKey, Expression> cache,
                        AnnotatedElementKey elementKey, String expression) {
                //**生成表过式的key,key为ExpressionKey对象
                ExpressionKey expressionKey = createKey(elementKey, expression);
                //**从cache中获取，如果没有，则调用SpelExpressionParser解析SpEL表达式，并保存在cache中
                Expression expr = cache.get(expressionKey);
                if (expr == null) {
                        expr = getParser().parseExpression(expression);
                        cache.put(expressionKey, expr);
                }
                return expr;
        }
}
```

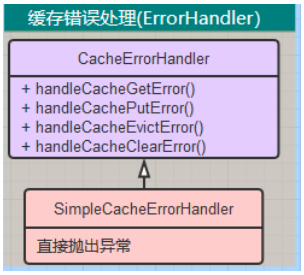
缓存 SpEL 执行器 CacheOperationExpressionEvaluator

作用：执行缓存注解配置 key、condition 和 unless 中的 SpEL 表达式，获取表达式的值；

```
class CacheOperationExpressionEvaluator extends CachedExpressionEvaluator {
    //**表达式没有结果时的占位符
        public static final Object NO_RESULT = new Object();
    //**表达式不能使用#result变量
        public static final Object RESULT_UNAVAILABLE = new Object();
    //**表达式#result变量
        public static final String RESULT_VARIABLE = "result";

    //**缓存注解key及其表达式对象
        private final Map<ExpressionKey, Expression> keyCache = new ConcurrentHashMap<>(64);
    //**缓存注解condition及其表达式对象
        private final Map<ExpressionKey, Expression> conditionCache = new ConcurrentHashMap<>(64);
    //**缓存注解unless及其表达式对象
        private final Map<ExpressionKey, Expression> unlessCache = new ConcurrentHashMap<>(64);

    //**创建SpEL表达工式执行器上下文
        public EvaluationContext createEvaluationContext(Collection<? extends Cache> caches,
                        Method method, Object[] args, Object target, Class<?> targetClass, Method targetMethod,
```

```
                        @Nullable Object result, @Nullable BeanFactory beanFactory) {
        //**创建#root对象
                CacheExpressionRootObject rootObject = new CacheExpressionRootObject(
                                caches, method, args, target, targetClass);
                //**创建SpEL表达工式执行器上下文对象
                CacheEvaluationContext evaluationContext = new CacheEvaluationContext(
                                rootObject, targetMethod, args, getParameterNameDiscoverer());
                //**如果目标方法的返回结值 = RESULT_UNAVAILABLE，#result不可用
                if (result == RESULT_UNAVAILABLE) {
                        evaluationContext.addUnavailableVariable(RESULT_VARIABLE);
                }
                //**如果目标方法的返回值 != NO_RESULT，把方法的返回值赋值给#result
                else if (result != NO_RESULT) {
                        evaluationContext.setVariable(RESULT_VARIABLE, result);
                }
                if (beanFactory != null) {
                        evaluationContext.setBeanResolver(new BeanFactoryResolver(beanFactory));
                }
                return evaluationContext;
        }

        //**获取key表达式的值
        public Object key(String keyExpression, AnnotatedElementKey methodKey, EvaluationContext evalContext) {
                return getExpression(this.keyCache, methodKey, keyExpression).getValue(evalContext);
        }

    //**执行condition表达式计算是否通过
        public boolean condition(String conditionExpression, AnnotatedElementKey methodKey, EvaluationContext evalC
            //**condition表达式返回true代表通过
                return (Boolean.TRUE.equals(getExpression(this.conditionCache, methodKey, conditionExpression).getV
                                evalContext, Boolean.class)));
        }

    //**执行unless表达式计算是否通过
        public boolean unless(String unlessExpression, AnnotatedElementKey methodKey, EvaluationContext evalContext
            //**unless表达式返回true代表通过
                return (Boolean.TRUE.equals(getExpression(this.unlessCache, methodKey, unlessExpression).getValue(
                                evalContext, Boolean.class)));
        }

        //**清除key,condition和unless及其表达式对象
        void clear() {
                this.keyCache.clear();
                this.conditionCache.clear();
                this.unlessCache.clear();
        }
}
```

2.7. 缓存错误处理 (ErrorHandler)

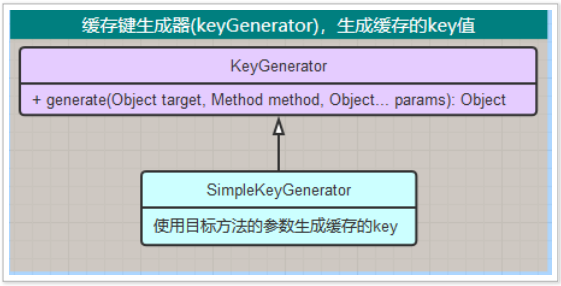


作用

在获取 / 修改 / 移除缓存时发生异常时，处理特定的逻辑；

```
public interface CacheErrorHandler {
    //**获取缓存异常时处理特定的逻辑
        void handleCacheGetError(RuntimeException e, Cache cache, Object key);
    //**修改缓存异常时处理特定的逻辑
        void handleCachePutError(RuntimeException e, Cache cache, Object key, @Nullable Object value);
    //**移除缓存异常时处理特定的逻辑
        void handleCacheEvictError(RuntimeException e, Cache cache, Object key);
    //**清除缓存异常时处理特定的逻辑
        void handleCacheClearError(RuntimeException e, Cache cache);
}

//**默认的处理逻辑，直接抛出异常
public class SimpleCacheErrorHandler implements CacheErrorHandler {
        public void handleCacheGetError(RuntimeException e, Cache cache, Object key) { throw exception; }
        public void handleCachePutError(RuntimeException e, Cache cache, Object key, Object value) { throw exceptior
        public void handleCacheEvictError(RuntimeException e, Cache cache, Object key) { throw
        public void handleCacheClearError(RuntimeException e, Cache cache)
}
```

2.8. 缓存键生成器 (keyGenerator)



作用

生成缓存的 key 值;

简单的 SimpleKeyGenerator

```
public class SimpleKeyGenerator implements KeyGenerator {

        @Override
        public Object generate(Object target, Method method, Object... params) {
                return generateKey(params);
        }

        //**根据方法的参数列表生成key，key为SimpleKey对象，保存到缓存中时会调用SimpleKey的toString方法转换为String
        public static Object generateKey(Object... params) {
                if (params.length == 0) {
                        return SimpleKey.EMPTY;
                }
                if (params.length == 1) {
                        Object param = params[0];
                        if (param != null && !param.getClass().isArray()) {
                                return param;
                        }
                }
                return new SimpleKey(params);
        }
}
```

2.9. 缓存管理 (CacheManager)

前面有专门介绍过 CacheManager，这里不在赘述。

全文完