Spring Boot + Redis 缓存方案深度解读

本文主要涉及到的几个类

RedisCache是Cache接口的实现类，在Cache接口中定义了缓存的基本增删改查操作。

- CacheAspectSupport是spring缓存操作的aop切面，缓存产生作用的入口主要在这里
- RedisCacheManager是redis cache的主要配置类

一个重要的设计原则

面向接口编程，所有的设计都是基于接口的，这样的代码更加优雅、具有很强的扩展性。

> 注意事项：
> 使用缓存注解时根本原理是AOP切面，要让切面起作用，方法的调用必须发生在从外部调用内部，如果是同一个类两个方法的调用切面是不会起作用的，此时缓存也不会起作用。

以一个问题进行开篇

问题：使用**spring data redis**的缓存方案时,是如何关联删除掉**books**下面的所有缓存的？

结论：**spring data redis**事先在**redis**中维护一个**sorted set**用来存储所有已知的**keys,**当删除指定**allEntries=true**参数的时候，直接从**sorted set**中所有维护的**key**，然后删除**sorted set**本身

> 备注：
> 分析过程中使用的代码为spring cache官方demo，经过少许改造，可以从github地址获取：
> https://github.com/pluone/gs-caching/tree/master/complete

```
//当我们使用@CacheEvict注解来清除缓存时,当参数allEntries=true的时候会关联清除books缓存下所有的key,那么redis是如何知道books下面有哪些需要删除的缓存的呢?
@CacheEvict(cacheNames = "books", allEntries = true)
public void clearCache() {
    //do nothing
}
```

分析过程如下：
我们可以先推测spring维护了一个类似list的东西，存储了所有已知的key，那么在第一次设置缓存时一定会将已知key存入这样的一个list

如下：

```
//redis的@Cacheable和@CachePut注解最终都将转化为这个操作
//具体在redis中的操作又分为三步
//第一步：调用set命令设置缓存
//第二部：设置缓存的过期时间
//第三部：maintainKnownKeys维护已知key,这一步其实是将所有已知的key存进一个sorted set中，具体分析见下一个代码片段
static class RedisCachePutCallback extends AbstractRedisCacheCallback<Void> {
    public RedisCachePutCallback(BinaryRedisCacheElement element, RedisCacheMetadata metadata) {

        super(element, metadata);
    }

    /*
        * (non-Javadoc)
        * @see org.springframework.data.redis.cache.RedisCache.AbstractRedisPutCallback#doInRedis(org.springframework.data.redis.cache.RedisCache.RedisCacheElement, org.springframework.data.redis.connection.RedisConnection)
        */
    @Override
    public Void doInRedis(BinaryRedisCacheElement element, RedisConnection connection) throws DataAccessException {

        if (!isClusterConnection(connection)) {
            connection.multi();
        }

        if (element.get().length == 0) {
            connection.del(element.getKeyBytes());
        } else {
            connection.set(element.getKeyBytes(), element.get());

            processKeyExpiration(element, connection);
            maintainKnownKeys(element, connection);
```

```
        }

        if (!isClusterConnection(connection)) {
            connection.exec();
        }
        return null;
    }
}


//这一步是将已知key加入sorted set的具体操作
protected void maintainKnownKeys(RedisCacheElement element,
RedisConnection connection) {

    if (!element.hasKeyPrefix()) {

        connection.zAdd(cacheMetadata.getSetOfKnownKeysKey(), 0,
element.getKeyBytes());

        if (!element.isEternal()) {
            connection.expire(cacheMetadata.getSetOfKnownKeysKey(),
element.getTimeToLive());
        }
    }
}
```



```
//spring的缓存注解最终都是通过CacheAspectSupport类中的这个方法来执行的，可以
看到倒数第二行代码是处理缓存删除逻辑的
private Object execute(final CacheOperationInvoker invoker, Method
method, CacheOperationContexts contexts) {
    // Special handling of synchronized invocation
    //这里的判断是当设置Cacheable(sync=true)时执行的操作，确保了程序并发更新
缓存的安全性
    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, new
Callable<Object>() {
                    @Override
                    public Object call() throws Exception {
                        return
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);
        }
    }


    // Process any early evictions
    //处理缓存的清除操作，具体为@CacheEvict(beforeInvocation=true)时会在已
进入方法就执行删除操作，而不会等待方法内的具体逻辑执行完成
    processCacheEvicts(contexts.get(CacheEvictOperation.class), true,
            CacheOperationExpressionEvaluator.NO_RESULT);

    // Check if we have a cached item matching the conditions
    Cache.ValueWrapper cacheHit =
findCachedItem(contexts.get(CacheableOperation.class));

    // Collect puts from any @Cacheable miss, if no cached item is
found
    List<CachePutRequest> cachePutRequests = new
LinkedList<CachePutRequest>();
    if (cacheHit == null) {
        collectPutRequests(contexts.get(CacheableOperation.class),
                CacheOperationExpressionEvaluator.NO_RESULT,
cachePutRequests);
    }

    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();
        returnValue = wrapCacheValue(method, cacheValue);
    }
    else {
        // Invoke the method if we don't have a cache hit
        returnValue = invokeOperation(invoker);
        cacheValue = unwrapReturnValue(returnValue);
    }

    // Collect any explicit @CachePuts
```

```
    collectPutRequests(contexts.get(CachePutOperation.class),
cacheValue, cachePutRequests);

    // Process any collected put requests, either from @CachePut or a
@Cacheable miss
    for (CachePutRequest cachePutRequest : cachePutRequests) {
        cachePutRequest.apply(cacheValue);
    }

    // Process any late evictions
    processCacheEvicts(contexts.get(CacheEvictOperation.class), false,
cacheValue);

    return returnValue;
}
```

```
//经过一步步跟踪我们能看到执行了doClear方法来清除所有缓存,else条件中的doEvit
只清除指定缓存
//而doClear又调用了cache.clear()方法，cache是一个接口，具体的实现类在
RedisCache中能看到
private void performCacheEvict(CacheOperationContext context,
CacheEvictOperation operation, Object result) {
    Object key = null;
    for (Cache cache : context.getCaches()) {
        if (operation.isCacheWide()) {
            logInvalidating(context, operation, null);
            doClear(cache);
        }
        else {
            if (key == null) {
                key = context.generateKey(result);
            }
            logInvalidating(context, operation, key);
            doEvict(cache, key);
        }
    }
}
```



```
//具体调用了RedisCacheCleanByKeysCallback类
public void clear() {
    redisOperations.execute(cacheMetadata.usesKeyPrefix() ? new
RedisCacheCleanByPrefixCallback(cacheMetadata)
            : new RedisCacheCleanByKeysCallback(cacheMetadata));
}
```

```
//最终的操作在doInLock方法中实现，可以看到调用了redis的zRange方法从sorted
set中取出了所有的keys,然后使用del批量删除方法，先删除了所有的缓存，然后删除掉
```

```
了sorted set
static class RedisCacheCleanByKeysCallback extends
LockingRedisCacheCallback<Void> {

    private static final int PAGE_SIZE = 128;
    private final RedisCacheMetadata metadata;

    RedisCacheCleanByKeysCallback(RedisCacheMetadata metadata) {
        super(metadata);
        this.metadata = metadata;
    }

    /*
        * (non-Javadoc)
        * @see
org.springframework.data.redis.cache.RedisCache.LockingRedisCacheCallb
ack#doInLock(org.springframework.data.redis.connection.RedisConnection
)
        */
    @Override
    public Void doInLock(RedisConnection connection) {

        int offset = 0;
        boolean finished = false;

        do {
            // need to paginate the keys
            Set<byte[]> keys =
connection.zRange(metadata.getSetOfKnownKeysKey(), (offset) *
PAGE_SIZE,
                    (offset + 1) * PAGE_SIZE - 1);
            finished = keys.size() < PAGE_SIZE;
            offset++;
            if (!keys.isEmpty()) {
                connection.del(keys.toArray(new byte[keys.size()][]));
            }
        } while (!finished);

        connection.del(metadata.getSetOfKnownKeysKey());
        return null;
    }
}
```


为什么不推荐使用key前缀

从RedisCacheManager Bean的定义说起，一般的定义如下，代码第5行有一个设置是否使用keyPrefix的选项，这个选项设置为true和false有很大的区别，这是官方的文档里没有提到的地方，也是可能有坑的地方。

```
@Bean
public RedisCacheManager redisCacheManager(RedisOperations
redisOperations) {
    RedisCacheManager redisCacheManager = new
RedisCacheManager(redisOperations);
    redisCacheManager.setLoadRemoteCachesOnStartup(true);
    redisCacheManager.setUsePrefix(true);
    redisCacheManager.setCachePrefix(cacheName -> ("APP_CACHE:" +
cacheName + ":").getBytes());
    redisCacheManager.setExpires(CacheConstants.getExpirationMap());
    return redisCacheManager;
}
```

为什么需要使用keyPrefix?

考虑下面Service层的伪代码。

```
@Cacheable(cacheNames="books")
public Book getBook(Long bookId){
    return bookRepo.getById(bookId);
}

@Cacheable(cacheNames="customers")
public Customer getCustomer(Long customerId){
    return customerRepo.getById(customerId);
}
```


Controller层调用的伪代码。

```
@Autowired
BookService bookService;
@Autowired
CustomerService customerService;

public Book foo(){
    return bookService.getBook(123456L);
}

public Customer bar(){
    return customerService.getCustomer(123456L);
}
```

当我们使用@Cacheable注解时，如果没有指定key参数，也没有自定义KeyGenerator，此时会使用spring提供的SimpleKeyGenerator来生成缓存的key。调用时两个方法传入的参数都是 123456 ,产生的key也一样都是 123456 。此时两个key在redis中会互相覆盖，导致getBook方法取到的值有可能是Customer对象，从而产生ClassCastException。

为了避免这种情况要使用keyPrefix，即为key加一个前缀（或者称为namespace）来区分。例如 books:123456 和 customers:123456 。

RedisCachePrefix接口是这样定义的：

```
public interface RedisCachePrefix {
    byte[] prefix(String cacheName);
}
```

而它的默认实现是这样的,在cacheName后面添加分隔符（默认为冒号）作为keyPrefix。

```
public class DefaultRedisCachePrefix implements RedisCachePrefix {

    private final RedisSerializer serializer = new
StringRedisSerializer();
    private final String delimiter;

    public DefaultRedisCachePrefix() {
        this(":");
    }

    public DefaultRedisCachePrefix(String delimiter) {
        this.delimiter = delimiter;
    }

    public byte[] prefix(String cacheName) {
        return serializer.serialize((delimiter != null ?
cacheName.concat(delimiter) : cacheName.concat(":")));
    }
}
```

除了使用为key加前缀的方式来避免产生相同的key外，还有一种方式：可以自定义KeyGenerator,保证产生的key不会重复，下面提供一个简单的实现，通过类名+方法名+方法参数来保证key的唯一性。

```
@Bean
public KeyGenerator myKeyGenerator() {
    return (target, method, params) -> {
        StringBuilder sb = new StringBuilder();
        sb.append(target.getClass().getName());
        sb.append("_");
        sb.append(method.getName());
```

```
            sb.append("_");
            for (int i = 0; i < params.length; i++) {
                sb.append(params[i].toString());
                if (i != params.length - 1) {
                    sb.append("_");
                }
            }
            return sb.toString();
        };
    }
```

使用keyPrefix后需要注意的地方

@CacheEvict操作在使用keyPrefix后会有很大的不同，如下操作：

```
    @CacheEvict(cacheName="books",allEntries=true){
        //do nothing
    }
```

清除会调用RedisCache的clear方法，在不使用keyPrefix时，将 books~keys 这个集合中的所有key取出来进行删除，最后删除 books~keys 本身。在使用keyPrefix时，使用 keys cachePrefix* 命令来取出所有前缀相同的key，进行删除。


```
    //从代码中我们看到使用前缀和不使用时执行了不同的处理逻辑
    public void clear() {
        redisOperations.execute(cacheMetadata.usesKeyPrefix() ? new
    RedisCacheCleanByPrefixCallback(cacheMetadata)
                : new RedisCacheCleanByKeysCallback(cacheMetadata));
    }
```

```
        //使用key前缀时调用的清除缓存代码如下，分析可以看到使用了keys *的方式来删除所有的key
        static class RedisCacheCleanByPrefixCallback extends
    LockingRedisCacheCallback<Void> {

            private static final byte[] REMOVE_KEYS_BY_PATTERN_LUA = new
    StringRedisSerializer().serialize(
                    "local keys = redis.call('KEYS', ARGV[1]); local
    keysCount = table.getn(keys); if(keysCount > 0) then for _, key in
    ipairs(keys) do redis.call('del', key); end; end; return keysCount;");
            private static final byte[] WILD_CARD = new
    StringRedisSerializer().serialize("*");
            private final RedisCacheMetadata metadata;
```

```
        public RedisCacheCleanByPrefixCallback(RedisCacheMetadata
metadata) {
            super(metadata);
            this.metadata = metadata;
        }

        /*
         * (non-Javadoc)
         * @see
org.springframework.data.redis.cache.RedisCache.LockingRedisCacheCallb
ack#doInLock(org.springframework.data.redis.connection.RedisConnection
)
         */
        @Override
        public Void doInLock(RedisConnection connection) throws
DataAccessException {

            byte[] prefixToUse =
Arrays.copyOf(metadata.getKeyPrefix(), metadata.getKeyPrefix().length
+ WILD_CARD.length);
            System.arraycopy(WILD_CARD, 0, prefixToUse,
metadata.getKeyPrefix().length, WILD_CARD.length);
            //这里判断了redis是否是集群模式，因为集群模式下不能使用lua脚本，
所以直接通过循环进行删除
            if (isClusterConnection(connection)) {

                // load keys to the client because currently Redis
Cluster connections do not allow eval of lua scripts.
                //使用缓存时通过keys加前缀的方式类匹配出所有的key，然后通过
循环进行删除操作
                Set<byte[]> keys = connection.keys(prefixToUse);
                if (!keys.isEmpty()) {
                    connection.del(keys.toArray(new byte[keys.size()]
[]));
                }
            } else {
                //非集群模式下，通过上面定义的lua脚本进行删除
                connection.eval(REMOVE_KEYS_BY_PATTERN_LUA,
ReturnType.INTEGER, 0, prefixToUse);
            }

            return null;
        }
    }
```

这里非常不推荐使用 keys * 的方式在生产环境中使用，因为改命令可能会阻塞redis的其它命令，具体参考官方文档。

对分页的支持不够完美

在应用中经常使用分页，大多数时候要对整页的数据进行缓存，以提高读取速度。但是当插入一条分页数据时，整个页面都发生了变化，此时我们只能将所有分页的缓存清除，操作的粒度比较粗。

对事务的支持

对事务的支持主要在RedisCacheManager中，该类继承了抽象类。AbstractTransactionSupportingCacheManager，这个抽象类使用装饰器模式增加了事务属性，事务的支持主要应用在put,evict,clear三个操作上。只有事务提交时，才会进行对应的缓存操作。