使用Redis+AOP优化权限管理功能，这波操作贼爽！

原创 梦想de星空 macrozheng 2020-03-23 09:02

收录于合集
#mall学习教程（技术要点篇）
17个

> 之前有很多朋友提过，mall项目中的权限管理功能有性能问题，因为每次访问接口进行权限校验时都会从数据库中去查询用户信息。最近对这个问题进行了优化，通过Redis+AOP解决了该问题，下面来讲下我的优化思路。

前置知识

- 学习本文需要一些Spring Data Redis的知识，不了解的朋友可以看下《Spring Data Redis最佳实践！》。
- 还需要一些Spring AOP的知识，不了解的朋友可以看下《SpringBoot应用中使用AOP记录接口访问日志》。

问题重现

在 `mall-security` 模块中有一个过滤器，当用户登录后，请求会带着token经过这个过滤器。这个过滤器会根据用户携带的token进行类似免密登录的操作，其中有一步会从数据库中查询登录用户信息，下面是这个过滤器类的代码。

```
/**
 * JWT登录授权过滤器
 * Created by macro on 2018/4/26.
 */
publicclass JwtAuthenticationTokenFilter extends OncePerRequestFilter {
    privatestaticfinal Logger LOGGER = LoggerFactory.getLogger(JwtAuthenticationTokenFilter.class
    @Autowired
    private UserDetailsService userDetailsService;
    @Autowired
    private JwtTokenUtil jwtTokenUtil;
    @Value("${jwt.tokenHeader}")
    private String tokenHeader;
```

```
    @Value("${jwt.tokenHead}")
    private String tokenHead;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response,
                                    FilterChain chain) throws ServletException, IOException {
        String authHeader = request.getHeader(this.tokenHeader);
        if (authHeader != null && authHeader.startsWith(this.tokenHead)) {
            String authToken = authHeader.substring(this.tokenHead.length());// The part after "B
            String username = jwtTokenUtil.getUserNameFromToken(authToken);
            LOGGER.info("checking username:{}", username);
            if (username != null && SecurityContextHolder.getContext().getAuthentication() == nul
                //此处会从数据库中获取登录用户信息
                UserDetails userDetails = this.userDetailsService.loadUserByUsername(username);
                if (jwtTokenUtil.validateToken(authToken, userDetails)) {
                    UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuth
                    authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(r
                    LOGGER.info("authenticated user:{}", username);
                    SecurityContextHolder.getContext().setAuthentication(authentication);
                }
            }
        }
        chain.doFilter(request, response);
    }
}
```

当我们登录后访问任意接口时，控制台会打印如下日志，表示会从数据库中查询用户信息和用户所拥有的资源信息，每次访问接口都触发这种操作，有的时候会带来一定的性能问题。

```
2020-03-17 16:13:02.623 DEBUG 4544 --- [nio-8081-exec-2] c.m.m.m.UmsAdminMapper.selectByExample
2020-03-17 16:13:02.624 DEBUG 4544 --- [nio-8081-exec-2] c.m.m.m.UmsAdminMapper.selectByExample
2020-03-17 16:13:02.625 DEBUG 4544 --- [nio-8081-exec-2] c.m.m.m.UmsAdminMapper.selectByExample
2020-03-17 16:13:02.628 DEBUG 4544 --- [nio-8081-exec-2] c.macro.mall.dao.UmsRoleDao.getMenuList
2020-03-17 16:13:02.628 DEBUG 4544 --- [nio-8081-exec-2] c.macro.mall.dao.UmsRoleDao.getMenuList
2020-03-17 16:13:02.632 DEBUG 4544 --- [nio-8081-exec-2] c.macro.mall.dao.UmsRoleDao.getMenuList
```

使用Redis作为缓存

> 对于上面的问题，最容易想到的就是把用户信息和用户资源信息存入到Redis中去，避免频繁查询数据库，本文的优化思路大体也是这样的。

首先我们需要对Spring Security中获取用户信息的方法添加缓存，我们先来看下这个方法执行了哪些数据库查询操作。

```
/**
 * UmsAdminService实现类
 * Created by macro on 2018/4/26.
 */
@Service
publicclass UmsAdminServiceImpl implements UmsAdminService {
    @Override
    public UserDetails loadUserByUsername(String username){
        //获取用户信息
        UmsAdmin admin = getAdminByUsername(username);
        if (admin != null) {
            //获取用户的资源信息
            List<UmsResource> resourceList = getResourceList(admin.getId());
            returnnew AdminUserDetails(admin,resourceList);
        }
        thrownew UsernameNotFoundException("用户名或密码错误");
    }
}
```

主要是获取用户信息和获取用户的资源信息这两个操作，接下来我们需要给这两个操作添加缓存操作，这里使用的是RedisTemple的操作方式。当查询数据时，先去Redis缓存中查询，如果Redis中没有，再从数据库查询，查询到以后在把数据存储到Redis中去。

```
/**
 * UmsAdminService实现类
 * Created by macro on 2018/4/26.
 */
@Service
publicclass UmsAdminServiceImpl implements UmsAdminService {
    //专门用来操作Redis缓存的业务类
    @Autowired
    private UmsAdminCacheService adminCacheService;
    @Override
```

```
    public UmsAdmin getAdminByUsername(String username) {
        //先从缓存中获取数据
        UmsAdmin admin = adminCacheService.getAdmin(username);
        if(admin!=null) return  admin;
        //缓存中没有从数据库中获取
        UmsAdminExample example = new UmsAdminExample();
        example.createCriteria().andUsernameEqualTo(username);
        List<UmsAdmin> adminList = adminMapper.selectByExample(example);
        if (adminList != null && adminList.size() > 0) {
            admin = adminList.get(0);
            //将数据库中的数据存入缓存中
            adminCacheService.setAdmin(admin);
            return admin;
        }
        returnnull;
    }
    @Override
    public List<UmsResource> getResourceList(Long adminId) {
        //先从缓存中获取数据
        List<UmsResource> resourceList = adminCacheService.getResourceList(adminId);
        if(CollUtil.isNotEmpty(resourceList)){
            return  resourceList;
        }
        //缓存中没有从数据库中获取
        resourceList = adminRoleRelationDao.getResourceList(adminId);
        if(CollUtil.isNotEmpty(resourceList)){
            //将数据库中的数据存入缓存中
            adminCacheService.setResourceList(adminId,resourceList);
        }
        return resourceList;
    }
}
```

上面这种查询操作其实用Spring Cache来操作更简单，直接使用@Cacheable即可实现，为什么还要使用RedisTemplate来直接操作呢？因为作为缓存，我们所希望的是，如果Redis宕机了，我们的业务逻辑不会有影响，而使用Spring Cache来实现的话，当Redis宕机以后，用户的登录等种种操作就会都无法进行了。

由于我们把用户信息和用户资源信息都缓存到了Redis中，所以当我们修改用户信息和资源信息时都需要删除缓存中的数据，具体什么时候删除，查看缓存业务类的注释即可。

```
/**
 * 后台用户缓存操作类
```

```
 * Created by macro on 2020/3/13.
 */
publicinterface UmsAdminCacheService {
    /**
     * 删除后台用户缓存
     */
    void delAdmin(Long adminId);

    /**
     * 删除后台用户资源列表缓存
     */
    void delResourceList(Long adminId);

    /**
     * 当角色相关资源信息改变时删除相关后台用户缓存
     */
    void delResourceListByRole(Long roleId);

    /**
     * 当角色相关资源信息改变时删除相关后台用户缓存
     */
    void delResourceListByRoleIds(List<Long> roleIds);

    /**
     * 当资源信息改变时，删除资源项目后台用户缓存
     */
    void delResourceListByResource(Long resourceId);
}
```

经过上面的一系列优化之后，性能问题解决了。但是引入新的技术之后，新的问题也会产生，比如说当Redis宕机以后，我们直接就无法登录了，下面我们使用AOP来解决这个问题。

使用AOP处理缓存操作异常

> 为什么要用AOP来解决这个问题呢？因为我们的缓存业务类 `UmsAdminCacheService` 已经写好了，要保证缓存业务类中的方法执行不影响正常的业务逻辑，就需要在所有方法中添加 `try catch` 逻辑。使用AOP，我们可以在一个地方写上 `try catch` 逻辑，然后应用到所有方法上去。试想下，我们如果又多了几个缓存业务类，只要配置下切面即可，这波操作多方便！

首先我们先定义一个切面，在相关缓存业务类上面应用，在它的环绕通知中直接处理掉异常，保障后续操作能执行。

```
/**
 * Redis缓存切面，防止Redis宕机影响正常业务逻辑
 * Created by macro on 2020/3/17.
 */
@Aspect
@Component
@Order(2)
publicclass RedisCacheAspect {
    privatestatic Logger LOGGER = LoggerFactory.getLogger(RedisCacheAspect.class);

    @Pointcut("execution(public * com.macro.mall.portal.service.*CacheService.*(..)) || execution
    public void cacheAspect() {
    }

    @Around("cacheAspect()")
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {
        Object result = null;
        try {
            result = joinPoint.proceed();
        } catch (Throwable throwable) {
            LOGGER.error(throwable.getMessage());
        }
        return result;
    }

}
```

这样处理之后，就算我们的Redis宕机了，我们的业务逻辑也能正常执行。

不过并不是所有的方法都需要处理异常的，比如我们的验证码存储，如果我们的Redis宕机了，我们的验证码存储接口需要的是报错，而不是返回执行成功。

对于上面这种需求我们可以通过自定义注解来完成，首先我们自定义一个 `CacheException` 注解，如果方法上面有这个注解，发生异常则直接抛出。

```
/**
 * 自定义注解，有该注解的缓存方法会抛出异常
```

```
 */
@Documented
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public@interface CacheException {
}
```

之后需要改造下我们的切面类，对于有 `@CacheException` 注解的方法，如果发生异常直接抛出。

```
/**
 * Redis缓存切面，防止Redis宕机影响正常业务逻辑
 * Created by macro on 2020/3/17.
 */
@Aspect
@Component
@Order(2)
publicclass RedisCacheAspect {
    privatestatic Logger LOGGER = LoggerFactory.getLogger(RedisCacheAspect.class);

    @Pointcut("execution(public * com.macro.mall.portal.service.*CacheService.*(..)) || execution
    public void cacheAspect() {
    }

    @Around("cacheAspect()")
    public Object doAround(ProceedingJoinPoint joinPoint) throws Throwable {
        Signature signature = joinPoint.getSignature();
        MethodSignature methodSignature = (MethodSignature) signature;
        Method method = methodSignature.getMethod();
        Object result = null;
        try {
            result = joinPoint.proceed();
        } catch (Throwable throwable) {
            //有CacheException注解的方法需要抛出异常
            if (method.isAnnotationPresent(CacheException.class)) {
                throw throwable;
            } else {
                LOGGER.error(throwable.getMessage());
            }
        }
        return result;
    }
```

```
}
```

接下来我们需要把 `@CacheException` 注解应用到存储和获取验证码的方法上去，这里需要注意的是要应用在实现类上而不是接口上，因为 `isAnnotationPresent` 方法只能获取到当前方法上的注解，而不能获取到它实现接口方法上的注解。

```
/**
 * UmsMemberCacheService实现类
 * Created by macro on 2020/3/14.
 */
@Service
publicclass UmsMemberCacheServiceImpl implements UmsMemberCacheService {
    @Autowired
    private RedisService redisService;

    @CacheException
    @Override
    public void setAuthCode(String telephone, String authCode) {
        String key = REDIS_DATABASE + ":" + REDIS_KEY_AUTH_CODE + ":" + telephone;
        redisService.set(key,authCode,REDIS_EXPIRE_AUTH_CODE);
    }

    @CacheException
    @Override
    public String getAuthCode(String telephone) {
        String key = REDIS_DATABASE + ":" + REDIS_KEY_AUTH_CODE + ":" + telephone;
        return (String) redisService.get(key);
    }
}
```

总结

对于影响性能的，频繁查询数据库的操作，我们可以通过Redis作为缓存来优化。缓存操作不该影响正常业务逻辑，我们可以使用AOP来统一处理缓存操作中的异常。

项目源码地址

https://github.com/macrozheng/mall

推荐阅读

- What？纯Java居然能实现Xshell！
- 秒杀系统是如何防止超卖的？
- Spring Data Redis 最佳实践！
- 优化if-else代码的八种方案！
- 一个不容错过的Spring Cloud实战项目！
- 127.0.0.1和0.0.0.0地址的区别！
- SpringBoot中处理校验逻辑的两种方式，真的很机智！
- 盘点下我用的顺手的那些工具！
- Github标星25K+Star，SpringBoot实战电商项目mall出SpringCloud版本啦！
- 我的Github开源项目，从0到20000 Star！

欢迎关注，点个在看

收录于合集 #mall学习教程（技术要点篇） 17

上一篇
SpringBoot中处理校验逻辑的两种方式，真的很机智！

下一篇
Elasticsearch项目实战，商品搜索功能设计与实现！

阅读原文

喜欢此内容的人还喜欢

项目中到底该不该用Lombok？

macrozheng