# 控制层访问拦截实现

## 背景

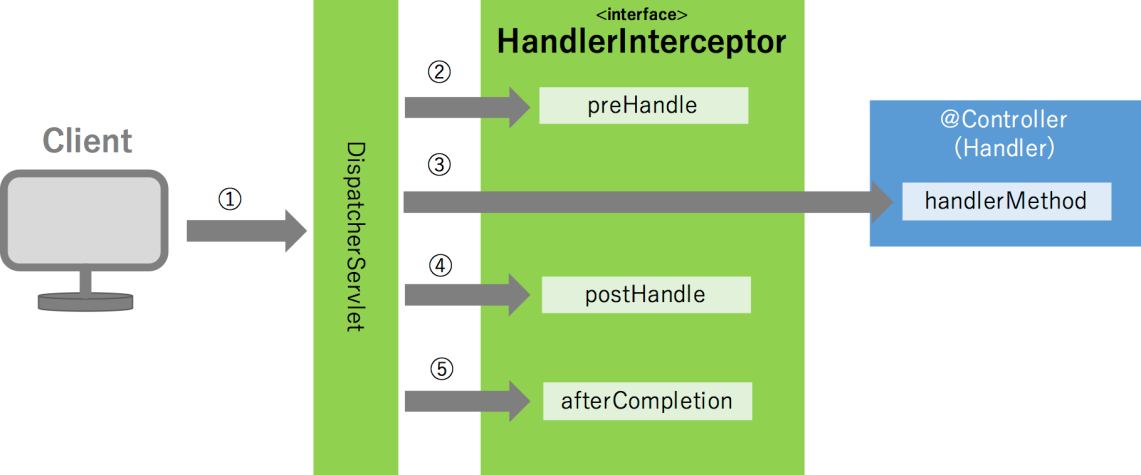
最近项目业务上有新的需求，要求系统登陆操作要有时间限制。

## 初步分析

对于类似需求的实现，可采用过滤器，SpringMVC拦截器，AOP等进行实现。对于过滤器而言一般主要应用在项目中共性的过滤，AOP需要依托于动态代理以及切面对象性能方面相对较差，所以最终选择使用Spring MVC拦截器进行实现。

## 原理应用分析

Spring MVC中的拦截器基于回调机制，可以在目标方法执行之前，先进行业务检测，满足条件则放行，不满足条件则进行拦截，拦截器原理分析如下图所示：



## 最终解决方案实现

第一步：拦截器定义，关键代码如下：

**package** com.cy.pj.common.web;

/\*\*

\* Spring MVC中拦截器

\* **@author** Administrator

\*/

**public** **class** TimeAccessInterceptor

**implements** HandlerInterceptor {

/\*\*

\* preHandle在控制层目标方法执行之前执行

\*/

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)

**throws** Exception {

System.***out***.println("preHandler()");

//获取java中的日历对象

Calendar c=Calendar.*getInstance*();

c.set(Calendar.***HOUR\_OF\_DAY***, 6);

c.set(Calendar.***MINUTE***, 0);

c.set(Calendar.***SECOND***, 0);

**long** start=c.getTimeInMillis();

c.set(Calendar.***HOUR\_OF\_DAY***,24);

**long** end=c.getTimeInMillis();

**long** cTime=System.*currentTimeMillis*();

**if**(cTime<start||cTime>end)

**throw** **new** ServiceException("不在访问时间之内");

**return** **true**;

}

}

第二步：拦截器配置，关键代码如下

**package** com.cy.pj.common.config;

@Configuration

**public** **class** SpringWebConfig **implements** WebMvcConfigurer{//web.xml

//配置spring mvc 拦截器

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(**new** TimeAccessInterceptor())

.addPathPatterns("/user/doLogin");

}

}

# 异步写用户行为日志

## 背景

最近项目上在测试人员压测过程中发现了OOM问题，项目使用springboot搭建项目工程，通过查看日志中包含信息：unable to create new native thread。

## 初步分析

初步怀疑是线程创建太多导致，使用jstack 线程号 > /tmp/oom.log将应用的线程信息打印出来。查看oom.log，发现大量线程处于Runnable状态，基本可以确认是线程创建太多了。

## 原理应用分析

Spring异步任务应用时，底层基于AOP方式为目标对象创建代理对象，在执行目标方法时，将目标方法运行在一个异步线程中。

项目中采用@Async默认异步配置获取异步线程。@Async默认异步配置可能会产生大量的线程，假如会有大量写库请求(例如将日志写入数据库)，这时就会不断创建大量线程，极有可能压爆服务器内存。

## 最终解决方案实现

我们最后选择自定义线程池，使用LinkedBlockingQueue阻塞队列来限定线程池的上限。使用自定义的拒绝策略，如果队列满了，则拒绝处理该任务，打印日志。

### 方案一

第一步：启动异步，在项目启动类上添加@EnableAsync注解。

**package** com.cy;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.scheduling.annotation.EnableAsync;

@EnableAsync //spring容器启动时会创建线程池

@SpringBootApplication

**public** **class** Application {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Application.**class**, args);

}

}

第二步：定义异步池，关键代码实现如下：

**package** com.cy.pj.common.config;

@Slf4j

@Setter

@Configuration

@ConfigurationProperties("async-thread-pool")

**public** **class** SpringAsyncConfig **implements** AsyncConfigurer{

/\*\*核心线程数\*/

**private** **int** corePoolSize=3;

/\*\*最大线程数\*/

**private** **int** maxPoolSize=5;

/\*\*线程空闲时间\*/

**private** **int** keepAliveTime=30;

/\*\*队列容量\*/

**private** **int** queueCapacity=100;

@Override

**public** Executor getAsyncExecutor() {

ThreadPoolTaskExecutor executor = **new** ThreadPoolTaskExecutor();

executor.setCorePoolSize(corePoolSize);

executor.setMaxPoolSize(maxPoolSize);

executor.setKeepAliveSeconds(keepAliveTime);

executor.setQueueCapacity(queueCapacity);

executor.setThreadNamePrefix("db-log-thread-");

executor.setRejectedExecutionHandler((Runnable r, ThreadPoolExecutor exe) -> {

***log***.warn("当前任务线程池队列已满.");

});

executor.initialize();

**return** executor;

}

@Override

**public** AsyncUncaughtExceptionHandler getAsyncUncaughtExceptionHandler() {

**return** **new** AsyncUncaughtExceptionHandler() {

@Override

**public** **void** handleUncaughtException(Throwable ex , Method method , Object... params) {

***log***.error("线程池执行任务发生未知异常.", ex);

}

};

}

}

其中，@ConfigurationProperties("async-thread-pool")配置是在spring boot的配置文件中，参考代码如下：

async-thread-pool:

corePoolSize: 5

maxPoolSize: 10

keepAliveTime: 30

queueCapacity: 50

第三步：在写日志的业务方法上使用异步，关键代码如下：

@Async

@Override

**public** **void** saveObject(SysLog entity) {

sysLogDao.insertObject(entity);

}

### 方案二

自定义池方式实现，关键代码如下

**package** com.cy.pj.common.config;

@Slf4j

@Setter

@Configuration

@ConfigurationProperties("async-thread-pool")

**public** **class** SpringAsyncConfig {

/\*\*核心线程数\*/

**private** **int** corePoolSize=3;

/\*\*最大线程数\*/

**private** **int** maxPoolSize=5;

/\*\*线程空闲时间\*/

**private** **int** keepAliveTime=30;

/\*\*队列容量\*/

**private** **int** queueCapacity=100;

/\*\*构建线程工厂\*/

**private** ThreadFactory threadFactory=**new** ThreadFactory() {

//CAS算法

**private** AtomicInteger at=**new** AtomicInteger(1000);

@Override

**public** Thread newThread(Runnable r) {

**return** **new** Thread(r, "db-async-thread-"+at.getAndIncrement());

}

};

/\*\*

\*/

@Bean("asyncPoolExecutor")

**public** ThreadPoolExecutor newPoolExecutor() {

System.***out***.println("corePoolSize="+corePoolSize);

//创建阻塞式对象:基于数组存储结构,FIFO算法实现的一个阻塞式队列

BlockingQueue<Runnable> workQueue=

**new** ArrayBlockingQueue<>(queueCapacity);

//创建池对象

ThreadPoolExecutor threadPoolExecutor=

**new** ThreadPoolExecutor(

corePoolSize,

maxPoolSize,

keepAliveTime,

TimeUnit.***SECONDS***,

workQueue,

threadFactory);

**return** threadPoolExecutor;

}

}

自定义池对象(基于java中的ThreadPoolExecutor类)应用分析说明:

* 当池中线程数没有达到corePoolSize指定的值,每次请求都会创建新的线程并存储到池中
* 当池中线程数已经达到corePoolSize指定的值,并且线程都在忙,新的请求会进入阻塞队列
* 当池中线程数已经达到corePoolSize指定的值,并且线程都在忙,并且队列已满,此时会创建新的线程,直到达到maximumPoolSize.

其死亡。这样的设计思想非常适合网络这种不稳定的应用场景。

池对象应用，关键代码如下：

//@Async

@Async("asyncPoolExecutor")

@Transactional(propagation = Propagation.***REQUIRES\_NEW***)

@Override

**public** **void** saveObject(SysLog entity) {

sysLogDao.insertObject(entity);

}

# 多数据源配置及实现

自己尝试实现,例如将日志表放到一个其它的数据库中，此时该如何实现日志数据操作。

# 第三方缓存配置及实现

自己尝试实现，例如将业务层数据进行缓存。

# 项目中任务调度实现

自己尝试实现，例如将业务层数据进行缓存。

# 项目打包策略及实现

## jar包方式实现（推荐）

假如使用的是sts，可直接对项目进行maven install操作，然后在项目target目录中找到对应的jar包，在当前目录下执行java -jar project-a.jar即可（假如project-a.jar为打包好的jar文件）

## war包方式实现（了解）

第一步：修改pom.xml,将打包方式修改为war

<packaging>war</packaging>

第二步：添加依赖

<dependency>

    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-starter-tomcat</artifactId>

    <scope>provided</scope>

</dependency>

修改启动类

@SpringBootApplication

**public** **class** Application **extends** SpringBootServletInitializer{

@Override

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(Application.**class**);

}

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(Application.**class**, args);

}

}

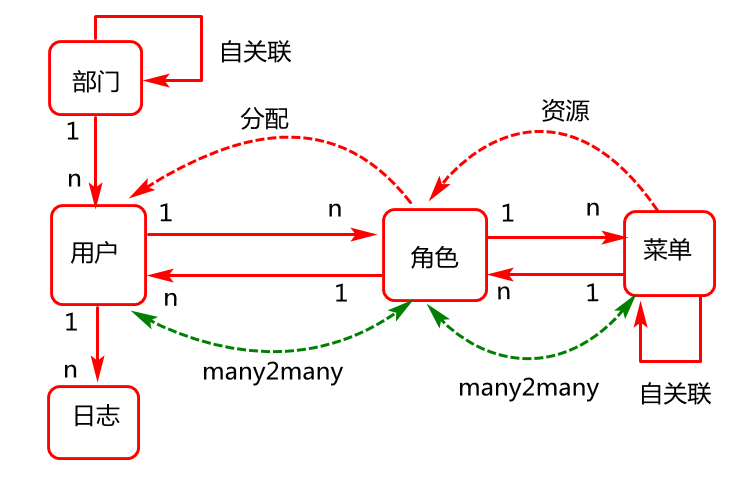
第三步：对项目进行maven install操作

第四步：将项目target目录下生成的war包拷贝到指定tomcat的webapps目录，启动tomcat测试运行,例如

http://localhost:8080/CGB-DB-SYS-V1.04-0.0.1-SNAPSHOT/doIndexUI

# 项目总结

## 业务总结



## 代码实现总结

