# Redis缓存实现

## 利用注解实现缓存

### 实现思路

业务说明:利用AOP中采用注解的形式实现业务缓存处理.

通知选择:环绕通知. 利用注解定义切入点表达式.

业务思路:当执行方法时如果该方法添加了@Cache()则进行AOP业务执行.

首先根据key(????)查询redis缓存.

null: 应该**执行业务方法**.,获取返回值之后添加缓存.

!null:将缓存数据转化为返回值对象.之后返回.

通用做法:

key:按照方法名称+第一个参数 拼接**动态**的Key

findAll();

### 自定义注解

//定义一个查询的注解

@Retention(RetentionPolicy.***RUNTIME***)

@Target({ElementType.***METHOD***}) //注解的作用范围

**public** **@interface** Cache\_Find {

String key() **default** "";//接收用户key值

KEY\_ENUM keyType() **default** KEY\_ENUM.***AUTO***;//定义key类型

**int** secondes() **default** 0; //永不失效

}

### 编辑AOP

@Component //将对象交给spring容器管理

@Aspect //标识这是一个切面 切面=切入点表达式+通知

@Slf4j

**public** **class** RedisAspect {

//表示当spring容器启动时不会立即注入对象???

@Autowired(required = **false**)

**private** JedisCluster jedisCluster;

/\*\*

\* 规定:

\* 1.环绕通知必须使用ProceedingJoinPoint

\* 2.如果通知中有参数joinPoint.必须位于第一位

\*

\* **@param** joinPoint

\* **@return**

\*/

@SuppressWarnings("unchecked")

//问题1:如何获取注解中的属性?

//该切入点表达式 规定只能获取注解类型 用法名称必须匹配

@Around(value="@annotation(cache\_Find)")

**public** Object around(ProceedingJoinPoint joinPoint,Cache\_Find cache\_Find) {

//1.动态获取key

String key = getKey(joinPoint,cache\_Find);

//2.从redis中获取数据

String resultJSON = jedisCluster.get(key);

Object resultData = **null**;

**try** {

//3.判断数据是否有值

**if**(StringUtils.*isEmpty*(resultJSON)) {

//3.1表示缓存中没有数据,则查询数据库(调用业务方法)

resultData = joinPoint.proceed();

//3.2将数据保存到缓存中

String json = ObjectMapperUtil.*toJSON*(resultData);

//3.3判断当前数据是否有失效时间

**if**(cache\_Find.secondes()==0) {

jedisCluster.set(key, json);

}**else** {

jedisCluster.setex(key,cache\_Find.secondes(), json);

}

System.***out***.println("AOP查询数据库成功!!!");

}**else** {

//4.表示redis缓存中有数据

Class returnType = getClass(joinPoint);

resultData = ObjectMapperUtil.*toObject*(resultJSON,returnType);

System.***out***.println("AOP查询缓存!!!!!!");

}

} **catch** (Throwable e) {

e.printStackTrace();

***log***.error(e.getMessage());

**throw** **new** RuntimeException(e);

}

**return** resultData;

}

/\*\*

\* key的定义规则如下:

\* 1.如果用户使用AUTO.则自动生成KEY 方法名\_第一个参数

\* 2.如果用户使用EMPTY,使用用户自己的key

\* **@param** joinPoint

\* **@param** cache\_Find

\* **@return**

\*/

**private** String getKey(ProceedingJoinPoint joinPoint, Cache\_Find cache\_Find) {

//1.判断用户选择类型

**if**(KEY\_ENUM.***EMPTY***.equals(cache\_Find.keyType())) {

**return** cache\_Find.key();

}

//2.表示用户动态生成key findITemCat::0

String methodName = joinPoint.getSignature().getName();

String arg0 = String.*valueOf*(joinPoint.getArgs()[0]);

**return** methodName+"::"+arg0;

}

/\*\*

\* 表示获取方法对象的返回值类型

\* **@param** joinPoint

\* **@return**

\*/

**private** Class getClass(ProceedingJoinPoint joinPoint) {

MethodSignature signature = (MethodSignature)joinPoint.getSignature();

**return** signature.getReturnType();

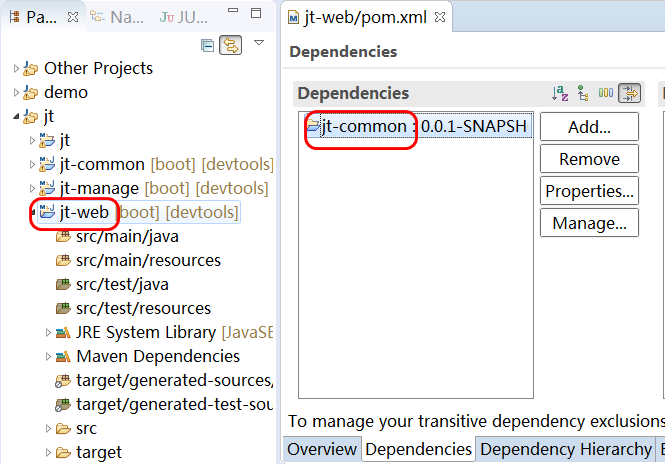
}

}

# 京淘项目前台构建

## 构建项目

### 创建项目/添加继承和依赖



### 添加bulid标签

<build>

<plugins>

<!--跳过测试类打包 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-surefire-plugin</artifactId>

<configuration>

<skip>true</skip>

</configuration>

</plugin>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

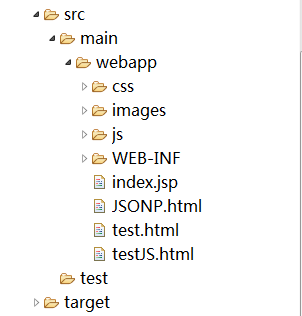
</plugin>

</plugins>

</build>

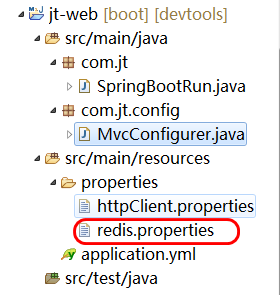
### 导入src文件

1.将课前资料中的src文件导入新项目



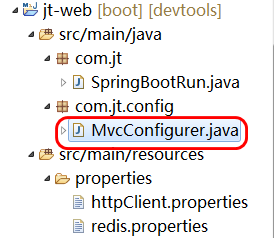
2.拷贝redis.properties文件

从jt-manage中导入redis.properties文件

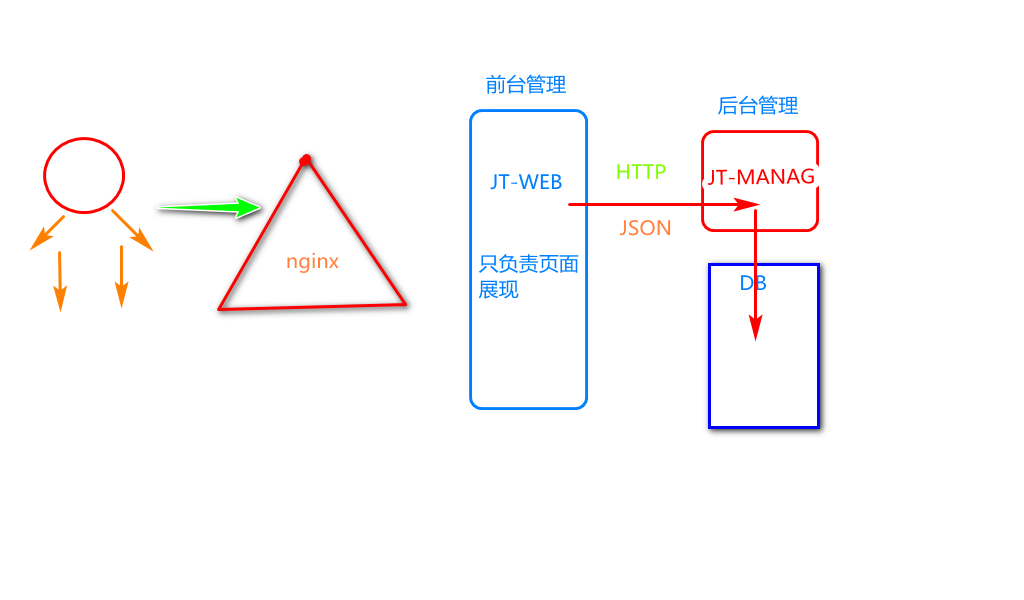


3.导入配置类信息

将课前资料中的配置类文件导入jt.com.config目录中



### 京淘web架构图设计



### 关于主启动类配置说明

问题描述:

当程序启动时报错如下

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Action:

Consider the following:

If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.

If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

问题说明:上边报错表示无法加载数据源.原因是因为springboot自动加载了数据源配置项之后报错.

解决办法:

//容器启动时不要加载数据源相关配置

@SpringBootApplication(exclude=DataSourceAutoConfiguration.**class** )

**public** **class** SpringBootRun {

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

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

}

}

### 关于伪静态描述

HTML特点:容易被**搜索引擎收录/加载速度快**

JSP 特点: 交互性强/加载速度慢

需求:又想被搜索引擎收录.交互性强.

解决方案:

伪静态是相对真实静态来讲的，通常我们为了增强[搜索引擎](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E/104812)的友好面，都将文章内容生成静态页面，但是有的朋友为了实时的显示一些信息。或者还想运用动态脚本解决一些问题。不能用静态的方式来展示网站内容。但是这就损失了对搜索引擎的友好面。怎么样在两者之间找个中间方法呢，这就产生了伪静态技术。就是展示出来的是以html一类的静态页面形式，但其实是用ASP一类的动态脚本来处理的。

总结:以html结尾的动态页面.

#### 编辑配置类

@Configuration

**public** **class** MvcConfigurer **implements** WebMvcConfigurer{

//开启匹配后缀型配置

@Override

**public** **void** configurePathMatch(PathMatchConfigurer configurer) {

configurer.setUseSuffixPatternMatch(**true**);

}

}

### 修改Nginx

[www.jt.com跳转localhost:8092](http://www.jt.com跳转localhost:8092)服务器.

#配置前台管理系统

server {

listen 80;

server\_name www.jt.com;

location / {

proxy\_pass http://localhost:8092;

}

}

#配置单点登录管理系统

server {

listen 80;

server\_name sso.jt.com;

location / {

proxy\_pass http://localhost:8093;

}

}

## HTTPClient

### 介绍

HTTP 协议可能是现在 Internet 上使用得最多、最重要的协议了，越来越多的 Java 应用程序需要直接通过 HTTP 协议来访问网络资源。虽然在 JDK 的 java net包中已经提供了访问 HTTP 协议的基本功能，但是对于大部分应用程序来说，JDK 库本身提供的功能还不够丰富和灵活。HttpClient 是 Apache Jakarta Common 下的子项目，用来**提供高效的、最新的、功能丰富的支持 HTTP 协议的客户端编程工具包**，并且它支持 HTTP 协议最新的版本和建议。HttpClient 已经应用在很多的项目中，比如 Apache Jakarta 上很著名的另外两个开源项目 Cactus 和 [HTMLUnit](https://baike.baidu.com/item/HTMLUnit) 都使用了 HttpClient。现在HttpClient最新版本为 HttpClient 4.5 .6（2015-09-11）

总结:在java代码中模拟发起http请求.

### HttpClient入门案例

**public** **class** TestHttpClient {

/\*\*

\* 编码思路:

\* 1.创建工具API对象

\* 2.定义远程url地址

\* 3.定义请求类型对象

\* 4.发起http请求,获取响应结果.

\* 5.从响应对象中获取数据.

\* **@throws** IOException

\* **@throws** ClientProtocolException

\*/

@Test

**public** **void** testGet() **throws** ClientProtocolException, IOException {

CloseableHttpClient client =

HttpClients.*createDefault*();

String url = "https://item.jd.com/8790543.html";

HttpGet get = **new** HttpGet(url);

CloseableHttpResponse response =

client.execute(get);

//判断响应是否正确

**if**(response.getStatusLine().getStatusCode()==200) {

//从中获取响应数据 JSON

String result =

EntityUtils.*toString*(response.getEntity());

System.***out***.println(result);

}**else** {

System.***out***.println("请求失败请稍后重试");

}

}

}

## SpringBoot整合HttpClient

### 编辑properties文件

#最大连接数

http.maxTotal = 1000

#并发数

http.defaultMaxPerRoute = 20

#创建连接的最长时间

http.connectTimeout=5000

#从连接池中获取到连接的最长时间

http.connectionRequestTimeout=500

#数据传输的最长时间

http.socketTimeout=5000

#提交请求前测试连接是否可用

http.staleConnectionCheckEnabled=true

### 编辑配置类

//目的:获取几个有用对象?????

@Configuration

@PropertySource(value="classpath:/properties/httpClient.properties")

**public** **class** HttpClientConfig {

@Value("${http.maxTotal}")

**private** Integer maxTotal; //最大连接数

@Value("${http.defaultMaxPerRoute}")

**private** Integer defaultMaxPerRoute; //最大并发链接数

@Value("${http.connectTimeout}")

**private** Integer connectTimeout; //创建链接的最大时间

@Value("${http.connectionRequestTimeout}")

**private** Integer connectionRequestTimeout; //链接获取超时时间

@Value("${http.socketTimeout}")

**private** Integer socketTimeout; //数据传输最长时间

@Value("${http.staleConnectionCheckEnabled}")

**private** **boolean** staleConnectionCheckEnabled; //提交时检查链接是否可用

//定义httpClient链接池

@Bean(name="httpClientConnectionManager")

**public** PoolingHttpClientConnectionManager getPoolingHttpClientConnectionManager() {

PoolingHttpClientConnectionManager manager = **new** PoolingHttpClientConnectionManager();

manager.setMaxTotal(maxTotal); //设定最大链接数

manager.setDefaultMaxPerRoute(defaultMaxPerRoute); //设定并发链接数

**return** manager;

}

//定义HttpClient

/\*\*

\* 实例化连接池，设置连接池管理器。

\* 这里需要以参数形式注入上面实例化的连接池管理器

**@Qualifier** 指定bean标签进行注入

为什么使用保护模式:

如果代码在后期有可能会变化.则一般限定方法的

使用.对外暴露一个公共的统一方法,

如果后期对方法进行了修改,.则可以通过修改暴露的方法

实现修改代码的目的.而使用者无需做任何操作.

\*/

@Bean(name = "httpClientBuilder")

**public** HttpClientBuilder getHttpClientBuilder(@Qualifier("httpClientConnectionManager")PoolingHttpClientConnectionManager httpClientConnectionManager){

//HttpClientBuilder中的构造方法被protected修饰，所以这里不能直接使用new来实例化一个HttpClientBuilder,可以使用HttpClientBuilder提供的静态方法create()来获取HttpClientBuilder对象

HttpClientBuilder httpClientBuilder = HttpClientBuilder.*create*();

httpClientBuilder.setConnectionManager(httpClientConnectionManager);

**return** httpClientBuilder;

}

/\*\*

\* 注入连接池，用于获取httpClient

\* **@param** httpClientBuilder

\* **@return**

\*/

@Bean

@Scope("prototype")

**public** CloseableHttpClient getCloseableHttpClient(@Qualifier("httpClientBuilder") HttpClientBuilder httpClientBuilder){

**return** httpClientBuilder.build();

}

/\*\*

\* Builder是RequestConfig的一个内部类

\* 通过RequestConfig的custom方法来获取到一个Builder对象

\* 设置builder的连接信息

\* **@return**

\*/

@Bean(name = "builder")

**public** RequestConfig.Builder getBuilder(){

RequestConfig.Builder builder = RequestConfig.*custom*();

**return** builder.setConnectTimeout(connectTimeout)

.setConnectionRequestTimeout(connectionRequestTimeout)

.setSocketTimeout(socketTimeout)

.~~setStaleConnectionCheckEnabled~~(staleConnectionCheckEnabled);

}

/\*\*

\* 使用builder构建一个RequestConfig对象

\*

\* **@param** builder

\* **@return**

\*/

@Bean

**public** RequestConfig getRequestConfig(@Qualifier("builder") RequestConfig.Builder builder){

**return** builder.build();

}

}

## 封装工具API

### 业务分析

目的:用户调用工具方法发起http请求.最终获取结果.

/\*\*

\*url://www.jt.com?id=1&name=tomcat

\* 目的:发起请求获取服务器数据

\* 参数说明:

\* 1.url地址

\* 2.用户提交的参数使用Map封装

\* 3.指定编码格式

\*

\*步骤:

\* 1.校验字符集. 如果字符集为null 设定默认值

\* 2.校验params是否为null

\* null:表示用户get请求无需传参.

\* !null:需要传参, get请求规则 url?key=value&key2=value2...

\* 3.发起http的get请求获取返回值结果

\*/

### 编辑配置类

@Service

**public** **class** HttpClientService {

@Autowired

**private** CloseableHttpClient httpClient;

@Autowired

**private** RequestConfig requestConfig;

/\*\*

\*url://www.jt.com?id=1&name=tomcat

\* 目的:发起请求获取服务器数据

\* 参数说明:

\* 1.url地址

\* 2.用户提交的参数使用Map封装

\* 3.指定编码格式

\*

\*步骤:

\* 1.校验字符集. 如果字符集为null 设定默认值

\* 2.校验params是否为null

\* null:表示用户get请求无需传参.

\* !null:需要传参, get请求规则 url?key=value&key2=value2...

\* 3.发起http的get请求获取返回值结果

\*/

**public** String doGet(String url,Map<String,String> params,String charshet) {

//1.校验字符集

**if**(StringUtils.*isEmpty*(charshet)) {

charshet = "UTF-8";

}

/\*\*

\* 2.校验参数是否为null url如何拼接????

\* url:www.jt.com?id=1&name=tomcat&....

\* Map<entry<k,v>>

\*/

**if**(params!=**null**) {

url +="?";

//2.1遍历map集合 迭代器 fore循环

**for** (Map.Entry<String,String> entry : params.entrySet()) {

String key = entry.getKey();

String value = entry.getValue();

url = url+key+"="+value+"&";

}

//2.2经过循环遍历最终url多个&

url = url.substring(0,url.length()-1);

}

//3.发起get请求

HttpGet get = **new** HttpGet(url);

get.setConfig(requestConfig);//定义请求超时时间

String result = **null**;

**try** {

CloseableHttpResponse response =

httpClient.execute(get);

**if**(response.getStatusLine().getStatusCode()==200) {

result = EntityUtils.*toString*(response.getEntity(),charshet);

}

} **catch** (Exception e) {

e.printStackTrace();

**throw** **new** RuntimeException(e);

}

**return** result;

}

//重载方法.对方法进行扩充方便使用者调用

**public** String doGet(String url) {

**return** doGet(url, **null**, **null**);

}

**public** String doGet(String url,Map<String,String> params) {

**return** doGet(url, params, **null**);

}

}

### 编辑测试类

@SpringBootTest

@RunWith(SpringRunner.**class**)

**public** **class** TestSpringBootHttpClient {

@Autowired

**private** HttpClientService httpClient;

@Test

**public** **void** testGet() {

String url = "https://item.jd.com/45703246467.html";

String result = httpClient.doGet(url);

System.***out***.println(result);

}

}