Day8 3月7日

软件51 庞建业 2151601012

Spring Boot学习

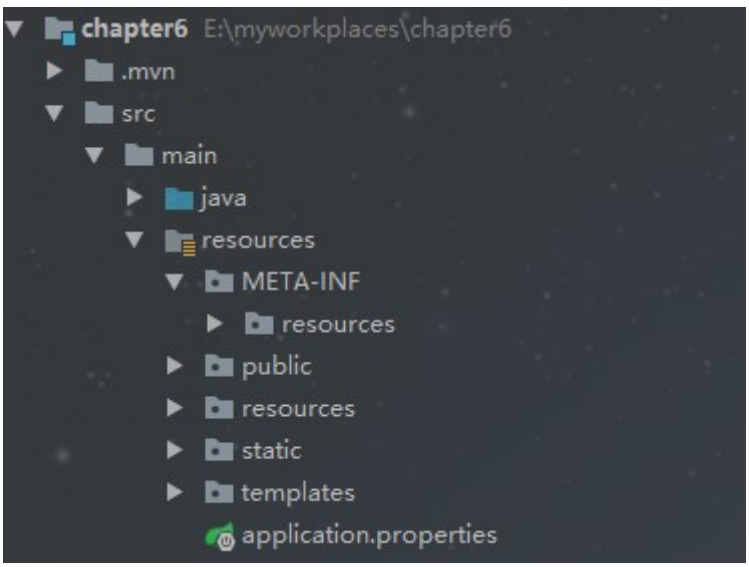
静态资源和拦截器处理

SpringBoot对静态资源的支持以及很重要的一个类WebMvcConfigurerAdapter

默认资源映射

Spring Boot 默认为我们提供了静态资源处理，使用 WebMvcAutoConfiguration 中的配置各种属性。 使用Spring Boot的默认配置方式，提供的静态资源映射如下:

- classpath:/META-INF/resources
- classpath:/resources
- classpath:/static
- classpath:/public



这几个都是静态资源的映射路径，优先级顺序为：`META-INF/resources > resources > static > public`
大家可以自己在上面4个路径下都放一张同名的图片，访问一下即可验证。
还有，你可以随机在上面一个路径下面放上`index.html`，当我们访问应用根目录`http://lcoalhost:8080` 时，会直接映射到`index.html`页面。

配置文件

```
# 默认值为 /**
spring.mvc.static-path-pattern=
# 默认值为 classpath:/META-
INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/
spring.resources.static-locations=这里设置要指向的路径，多个使用英文逗号隔开
```

自定义资源映射addResourceHandlers

只需重写addResourceHandlers方法

```
@Configuration
public class MyWebMvcConfigurerAdapter extends WebMvcConfigurerAdapter {
    /**
     * 配置静态访问资源
     * @param registry
     */
    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/my/**").addResourceLocations("classpath:/my/");
        super.addResourceHandlers(registry);
    }
}
```

通过addResourceHandler添加映射路径，然后通过addResourceLocations来指定路径

指定外部目录：

```
@Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/my/**").addResourceLocations("file:E:/my/");
        super.addResourceHandlers(registry);
    }
```

addResourceLocations指的是文件放置的目录，addResoureHandler指的是对外暴露的访问路径

页面跳转addViewControllers

> 以前写SpringMVC的时候，如果需要访问一个页面，必须要写Controller类，然后再写一个方法跳转到页面，感觉好麻烦，其实重写WebMvcConfigurerAdapter中的addViewControllers方法即可达到效果了

```
    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/toLogin").setViewName("login");
        super.addViewControllers(registry);
    }
```

在这里重写addViewControllers方法，并不会覆盖WebMvcAutoConfiguration中的addViewControllers（在此方法中，Spring Boot将“/”映射至index.html）

拦截器addInterceptors

要实现拦截器功能需要完成以下2个步骤：

- 创建我们自己的拦截器类并实现 HandlerInterceptor 接口
- 其实重写WebMvcConfigurerAdapter中的addInterceptors方法把自定义的拦截器类添加进来即可

  自定义拦截器代码：

```

public class MyInterceptor implements HandlerInterceptor {
```

```
    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {
        boolean flag =true;
        User user=(User)request.getSession().getAttribute("user");
        if(null==user){
            response.sendRedirect("toLogin");
            flag = false;
        }else{
            flag = true;
        }
        return flag;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object
handler, ModelAndView modelAndView) throws Exception {
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {
    }
}
```

重写WebMvcConfigurerAdapter中的addInterceptors方法如下：

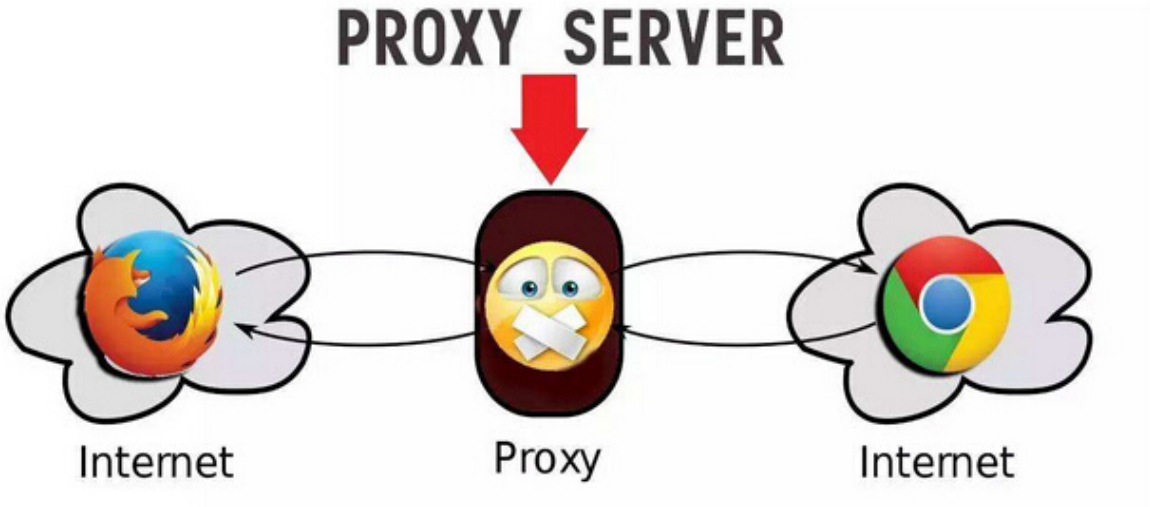
```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    // addPathPatterns 用于添加拦截规则
    // excludePathPatterns 用户排除拦截
    registry.addInterceptor(new
MyInterceptor()).addPathPatterns("/**").excludePathPatterns("/toLogin","/login");
    super.addInterceptors(registry);
}
```

`addPathPatterns("/**")`对所有请求都拦截，但是排除了`/toLogin`和`/login`请求的拦截。

爬虫复习

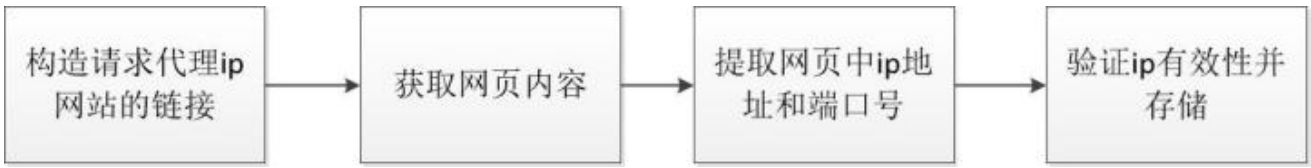
```
# # import requests
# # from lxml import etree
# # import time
# # # for a in range(10):
# # #     url = 'https://book.douban.com/top250?start={}'.format(a*25)
# # #     data = requests.get(url).text
# # #     data = requests.get(url).text
# # #     s=etree.HTML(data)
# # #     file=s.xpath('//*[@id="content"]/div/div[1]/div/table')
```

```
# # #     #print (file)
# # #     for div in file :
# # #         title = div.xpath("./tr/td[2]/div[1]/a/@title")[0]
# # #         score=div.xpath("./tr/td[2]/div[2]/span[2]/text()")[0]
# # #         print("{} {}".format(title,score))
# # url= "http://cd.xiaozhu.com/"
# # data=requests.get(url).text
# # s=etree.HTML(data)
# # file=s.xpath('//*[@id="page_list"]/ul/li/div[2]/div/a/span/text()')
# # time.sleep(20)
# # for title in file:
# #     print (title)
# #-*- coding:utf-8 -*-
# from lxml import etree
# import requests
# import time

# with open('D:\\crawler\\xzzf.txt','w',encoding='utf-8') as f:
#     for a in range(1,6):
#         url = 'http://cd.xiaozhu.com/search-duanzufang-p{}-0/'.format(a)
#         data = requests.get(url).text

#         s=etree.HTML(data)
#         file=s.xpath('//*[@id="page_list"]/ul/li')
#         time.sleep(3)

#         for div in file:
#             title=div.xpath("./div[2]/div/a/span/text()")[0]
#             price=div.xpath("./div[2]/span[1]/i/text()")[0]
#             scrible=div.xpath("./div[2]/div/em/text()")[0].strip()
#             pic=div.xpath("./a/img/@lazy_src")[0]
#             print ("{},{},{},{}\n".format(title,price,scrible,pic))
#             f.write("{},{},{},{}\n".format(title,price,scrible,pic))
import requests
import json
import time

with open('D:\\crawler\\xzzf.txt','w',encoding='utf-8') as f:
    for a in range(3):
        url_visit = 'https://movie.douban.com/j/new_search_subjects?
sort=T&range=0,10&tags=&start={}'.format(a*20)
        file = requests.get(url_visit).json()   #这里跟之前的不一样，因为返回的是 json 文件
        time.sleep(2)

        for i in range(20):
            dict=file['data'][i]   #取出字典中 'data' 下第 [i] 部电影的信息
            urlname=dict['url']
            title=dict['title']
            rate=dict['rate']
            cast=dict['casts']

            print('{}  {}  {}  {}\n'.format(title,rate,'  '.join(cast),urlname))

            f.write('{}  {}  {}  {}\n'.format(title,rate,'  '.join(cast),urlname))
```



建立爬虫代理ip池

> 在爬取网站信息的过程中，有些网站为了防止爬虫，可能会限制每个ip的访问速度或访问次数。对于限制访问速度的情况，我们可以通过time.sleep进行短暂休眠后再次爬取。对于限制ip访问次数的时候我们需要通过代理ip轮换去访问目标网址。所以建立并维护好一个有效的代理ip池也是爬虫的一个准备工作。



第一步：构造请求代理ip网站链接

```
def get_url(url):     # 国内高匿代理的链接
    url_list = []
    for i in range(1,100):
        url_new = url + str(i)
        url_list.append(url_new)
    return url_list
```

get_url： 生成要爬取目标网址的链接

第二步：获取网页内容

```
def get_content(url):     # 获取网页内容
    user_agent = 'Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/49.0.2623.22 Safari/537.36 SE 2.X MetaSr 1.0'
    headers = {'User-Agent': user_agent}
    req = urllib.request.Request(url=url, headers=headers)
    res = urllib.request.urlopen(req)
    content = res.read()
    return content.decode('utf-8')
```

get_content：接受的参数是传入的目标网站链接

第三步：提取网页中ip地址和端口号信息

```
def get_info(content):      # 提取网页信息 / ip 端口
    datas_ip = etree.HTML(content).xpath('//table[contains(@id,"ip_list")]/tr/td[2]/text()')
    datas_port = etree.HTML(content).xpath('//table[contains(@id,"ip_list")]/tr/td[3]/text()')
    with open("data.txt", "w") as fd:
        for i in range(0,len(datas_ip)):
            out = u""
            out += u"" + datas_ip[i]
            out += u":" + datas_port[i]
            fd.write(out + u"\n")     # 所有ip和端口号写入data文件
```

get_info：接收从get_content函数传来的网页内容，并使用etree解析出ip和端口号，将端口号和ip写入data.

第四步：验证代理ip的有效性

```
def verif_ip(ip,port):    # 验证ip有效性
    user_agent ='Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/49.0.2623.22 Safari/537.36 SE 2.X MetaSr 1.0'
    headers = {'User-Agent':user_agent}
    proxy = {'http':'http://%s:%s'%(ip,port)}
    print(proxy)

    proxy_handler = urllib.request.ProxyHandler(proxy)
    opener = urllib.request.build_opener(proxy_handler)
    urllib.request.install_opener(opener)

    test_url = "https://www.baidu.com/"
    req = urllib.request.Request(url=test_url,headers=headers)
    time.sleep(6)
    try:
        res = urllib.request.urlopen(req)
        time.sleep(3)
        content = res.read()
        if content:
            print('that is ok')
            with open("data2.txt", "a") as fd:       # 有效ip保存到data2文件夹
                fd.write(ip + u":" + port)
                fd.write("\n")
        else:
            print('its not ok')
    except urllib.request.URLError as e:
        print(e.reason)
```

verif_ip：使用ProxyHandler建立代理，使用代理ip访问某网址，查看是否得到响应。如数据有效，则保存到data2.txt文件

最后：调用各个函数

```
if __name__ == '__main__':
    url = 'http://www.xicidaili.com/nn/'

    url_list = get_url(url)
```

```
    for i in url_list:
        print(i)
        content = get_content(i)
        time.sleep(3)
        get_info(content)

    with open("dali.txt", "r") as fd:
        datas = fd.readlines()
        for data in datas:
            print(data.split(u":")[0])
        # print('%d : %d'%(out[0],out[1]))
            verif_ip(data.split(u":")[0],data.split(u":")[1])
```

Hadoop序列化

> 简单来说，序列化就是将对象（实例）转换为字符流（字符数组）的过程，转换后的字符流可用于网络传输或写入磁盘；相对的，反序列化就是将字符流转换成对象的过程。Hadoop有自己的序列化实现，并已提取为Avro子项目。序列化要求具有字符流紧凑，处理快速，可扩展，多语言支持特性。

Hadoop并没有采用Java的序列化，而是引入了它自己的系统。

Hadoop中定义了两个序列化相关的接口：Writable接口和Comparable接口，这两个接口可以合成一个接口WritableComparable.

Writable接口，所有实现了Writable接口的类都可以被序列化和反序列化

```
import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  /**
  * 将对象转换为字节流并写入到输出流out中
  */
  void write(DataOutput out) throws IOException;
  /**
  * 从输入流in中读取字节流反序列化为对象
  */
  void readFields(DataInput in) throws IOException;
}
```

Comparable接口

所有实现了Comparable的对象都可以和自身相同类型的对象比较大小。该接口定义为：

```
public interface Comparable<T> {
  /**
  * 将this对象和对象o进行比较，约定：返回负数为小于，零为大于，整数为大于
  */
  public int compareTo(T o);
}
```

Hadoop自带的序列化接口

实现了WritableComparable接口的类：

- 基础：BooleanWritable | ByteWritable
- 数字：IntWritable | VIntWritable | FloatWritable | LongWritable | VLongWritable | DoubleWritable
- 高级：NullWritable | Text | BytesWritable | MDSHash | ObjectWritable | GenericWritable

仅实现了Writable接口的类：

- 数组：ArrayWritable | TwoDArrayWritable
- 映射：AbstractMapWritable | MapWritable | SortedMapWritable

首先写3个实用的静态方法，serialize方法將Writable对象转换成字节流，deserialize方法将字节流转换为对象，printBytesHex方法打印byte[]数组对象。并通过最简单的IntWritable类，将int变量序列化，再反序列化。

```
public class Serialize {

    /**
     * 将Writable对象转换成字节流
     */
    public static byte[] serialize(Writable writable) throws IOException {
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        DataOutputStream dataOut = new DataOutputStream(out);
        writable.write(dataOut);
        dataOut.close();
        return out.toByteArray();
    }

    /**
     * 将字节流转换成Writable对象
     */
    public static void deserialize(Writable writable, byte[] bytes)
            throws IOException {
        ByteArrayInputStream in = new ByteArrayInputStream(bytes);
        DataInputStream dataIn = new DataInputStream(in);
        writable.readFields(dataIn);
        dataIn.close();
    }

    /**
     * 打印字节流
     */
    public static void printBytesHex(byte[] bytes) {
        for (int i = 0; i < bytes.length; i++) {
            System.out.print(StringUtils.byteToHexString(bytes, i, i + 1)
                    .toUpperCase());
            if (i % 16 == 15)
                System.out.print('\n');
            else if (i % 1 == 0)
                System.out.print(' ');
        }
    }

    public static void main(String[] args) throws IOException {
        IntWritable intWritable = new IntWritable(99999);
```

```
        // 序列化
        byte[] bytes = serialize(intWritable);
        printBytesHex(bytes);

        IntWritable intWritable2 = new IntWritable();
        // 反序列化
        deserialize(intWritable2, bytes);
        System.out.println(intWritable2);
    }
}
```

ObjectWritable相对于其他对象，它有不同的地位。当我们讨论Hadoop的RPC时，我们会提到RPC上交换的信息，必须是Java的基本类型，String和Writable接口的实现类，以及元素为以上类型的数组。ObjectWritable对象保存了一个可以在RPC上传输的对象和对象的类型信息。这样，我们就有了一个万能的，可以用于客户端/服务器间传输的Writable对象。例如，我们要把上面例子中的对象作为RPC请求，需要根据MyWritable创建一个ObjectWritable，ObjectWritable往流里会写如下信息

（对象类名长度，对象类名，对象自己的串行化结果）

这样，到了对端，ObjectWritable可以根据对象类名创建对应的对象，并解串行。应该注意到，ObjectWritable依赖于WritableFactories，那存储了Writable子类对应的工厂。我们需要把MyWritable的工厂，保存在WritableFactories中（通过WritableFactories.setFactory）