# 尚马教育 JAVA 高级课程

# 远程接口调用与任务

文档编号：C10

创建日期： 2017-07-05

最后修改日期：2021-02-22

版 本 号：V3.5

电子版文件名：尚马教育-第三阶段-10.远程接口与定时任务.docx

**文档修改记录：**

|  |  |  |  |
| --- | --- | --- | --- |
| 更新日期 | 更新作者 | 更新说明 | 版本号 |
| 2017-07-30 | 张元林 | 初始版本 | V1.0 |
| 2018-08-01 | 王绍成 | Mybatis版本更新 | V2.0 |
| 2019-10-14 | 冯勇涛 | 课件格式以及课程深度加深 | V3.0 |
| 2019-10-21 | 冯勇涛 | 加入RestTemplate对象 | V3.1 |
| 2021-02-22 | 冯勇涛 | 对spring-task深化,加入串行并行，异步任务 | V3.5 |

目录

[尚马教育 JAVA 高级课程 1](#_Toc9308)

[远程接口调用与任务 1](#_Toc11321)

[1. httpClient介绍 3](#_Toc31624)

[1.1. 了解HttpUrlConnection 3](#_Toc20189)

[1.2. 使用httpclient 4](#_Toc28488)

[1.3. 模拟get请求 4](#_Toc25654)

[1.4. 模拟post请求 6](#_Toc21133)

[1.5. 模拟put 7](#_Toc28953)

[1.6. 模拟delete 9](#_Toc9713)

[2. 掌握RestTemplate 11](#_Toc11879)

[3. 定时任务 14](#_Toc22203)

[3.1. TimerTask 14](#_Toc13452)

[3.2. Spring-task 15](#_Toc735)

[3.3. Quartz 17](#_Toc23550)

[3.4. cronExpression表达式 20](#_Toc21002)

[4. 异步任务 21](#_Toc17051)

[4.1. 线程池ThreadPoolTaskExecutor对象 21](#_Toc13306)

[4.2. 调用ThreadPoolTaskExecutor执行异步任务 22](#_Toc9497)

知识点：

|  |
| --- |
| 1. 掌握RestTemplte与HttpClient 2. 掌握TimerTask 3. 掌握springTask定时任务与异步任务 4. 了解quartz |

## 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。

实现了所有 HTTP 的方法（GET,POST,PUT,HEAD 等）

支持自动转向

支持 HTTPS 协议

支持代理服务器等

### 了解HttpUrlConnection

|  |
| --- |
| String path ="http://t.weather.sojson.com/api/weather/city/101030100";  URL url = **new** URL(path);  //默认是get请求.参数追加到url地址就行。  HttpURLConnection urlConn = (HttpURLConnection)url.openConnection();  urlConn.connect();//发起请求  InputStream in = urlConn.getInputStream();//读取服务端的返回数据  Reader isr = **new** InputStreamReader(in);    BufferedReader br = **new** BufferedReader(isr);  String datas = br.readLine();  JSONObject obj = JSON.*parseObject*(datas);//就是个Map  JSONObject dataObj = obj.getJSONObject("data");    String pm25= dataObj.getString("pm25");  System.***out***.println(pm25);  //解析项目中需要的数据，封装到一个Map中，保存到本地redis，或mysql.  System.***out***.println(datas); |

### 使用httpclient

#### 添加依赖jar

添加jar包

|  |
| --- |
|  |

添加maven依赖

|  |
| --- |
| <**dependency**>  <**groupId**>org.apache.httpcomponents</**groupId**>  <**artifactId**>httpclient</**artifactId**>  <**version**>4.5.6</**version**> </**dependency**> |

#### 执行流程

* 创建 HttpClient 的实例.
* 创建某种连接方法的实例，如 : HttpGet/ HttpPost。在连接方法实例的构造函数中传入待请求的连接地址.
* 使用Execute方法执行连接方法实例.
* 读取response中的内容.
* 释放连接，无论执行方法是否成功，都必须释放连接并终止连接实例.

### 模拟get请求

#### 代码示例

|  |
| --- |
| **public** **static** String doget(String path) {  CloseableHttpClient hc = HttpClients.*createDefault*();  //HttpUriRequest接口的实现类,四个http请求对象:HttpGet,HttpPost,HttpPut,HttpDelete  HttpGet get = **new** HttpGet(path);  //防止TCP状态一直保持在 CLOSE\_WAIT 状态,在请求头中进行connection配置  get.setHeader("Connection", "close");  HttpResponse resp = **null**;  HttpEntity respEntity = **null**;  **try** {  resp = hc.execute(get);//是响应对象  //得到响应状态码  **int** statusCode = resp.getStatusLine().getStatusCode();  //根据response的StatusCode进行请求结果判断  **if**(statusCode==HttpStatus.***SC\_OK***) {  //得到响应数据  respEntity = resp.getEntity();//响应体对象  //响应内容，自己解析流  // InputStream content = respEntity.getContent();  //使用EntityUtils工具类解析响应体  String datas = EntityUtils.*toString*(respEntity);  **return** datas;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  //使用 EntityUtils 关闭 InputStream 流  **if**(respEntity!=**null**) EntityUtils.*consume*(respEntity);  //使用 httpGet 实例中的 abort 方法终止连接实例  **if**(get!=**null**)get.abort();  //使用 CloseableHttpClient 中的 close方法关闭连接  **if**(hc!=**null**)hc.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** **null**;  } |

### 模拟post请求

#### 代码示例

|  |
| --- |
| /\*\*  \* post请求,返回响应字符串  \* **@param** path：请求路径  \* **@param** args：post请求需要传递给服务端的参数，以map封装。  \* **@return**  \*/  **public** **static** String dopost(String path,Map<String, Object> args) {  CloseableHttpClient hc = HttpClients.*createDefault*();  HttpPost post = **new** HttpPost(path);  post.setHeader("Connection", "close");  post.setHeader("Content-type","application/json");  //请求体对象  HttpEntity reqEntity = **null**;  HttpResponse resp = **null**;  //响应体对象  HttpEntity respEntity = **null**;  **try** {  reqEntity = new StringEntity(JSON.toJSONString(args));//传输json字符串  // reqEntity = new UrlEncodedFormEntity();//传输kv字符串  post.setEntity(reqEntity);//把请求参数封装到HttpPost请求对象中  resp = hc.execute(post);  **int** statusCode = resp.getStatusLine().getStatusCode();  **if**(statusCode==HttpStatus.***SC\_OK***) {  respEntity = resp.getEntity();  String datas = EntityUtils.*toString*(respEntity);  **return** datas;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if**(respEntity!=**null**) EntityUtils.*consume*(respEntity);  **if**(post!=**null**)post.abort();  **if**(reqEntity!=**null**) EntityUtils.*consume*(reqEntity);  **if**(hc!=**null**)hc.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** **null**;  } |

### 模拟put

#### 代码示例

|  |
| --- |
| **public** **static** String doput(String path,Map<String, Object> args) {  CloseableHttpClient hc = HttpClients.*createDefault*();  HttpPut put = **new** HttpPut(path);//HttpUriRequest接口的实现类,请求对象  put.setHeader("Connection", "close");  put.setHeader("Content-type","application/json");  HttpEntity reqEntity = **null**;  HttpResponse resp = **null**;  HttpEntity respEntity = **null**;  **try** {  reqEntity = new StringEntity(JSON.toJSONString(args));//传输json字符串  put.setEntity(reqEntity);//把请求参数封装到HttpPost请求对象中  resp = hc.execute(put);  **int** statusCode = resp.getStatusLine().getStatusCode();  **if**(statusCode==HttpStatus.***SC\_OK***) {  respEntity = resp.getEntity();  String datas = EntityUtils.*toString*(respEntity);  **return** datas;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if**(respEntity!=**null**) EntityUtils.*consume*(respEntity);  **if**(put!=**null**)put.abort();  **if**(reqEntity!=**null**) EntityUtils.*consume*(reqEntity);  **if**(hc!=**null**)hc.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** **null**;  } |

### 模拟delete

|  |
| --- |
| **public** **static** String dodelete(String path) {  CloseableHttpClient hc = HttpClients.*createDefault*();  HttpDelete del = **new** HttpDelete(path);//HttpUriRequest接口的实现类,请求对象  del.setHeader("Connection", "close");  HttpEntity reqEntity = **null**;  HttpResponse resp = **null**;  HttpEntity respEntity = **null**;  **try** {  resp = hc.execute(del);  **int** statusCode = resp.getStatusLine().getStatusCode();  **if**(statusCode==HttpStatus.***SC\_OK***) {  respEntity = resp.getEntity();  String datas = EntityUtils.*toString*(respEntity);  **return** datas;  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **try** {  **if**(respEntity!=**null**) EntityUtils.*consume*(respEntity);  **if**(del!=**null**)del.abort();  **if**(reqEntity!=**null**) EntityUtils.*consume*(reqEntity);  **if**(hc!=**null**)hc.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **return** **null**;  } |

## 掌握RestTemplate

在spring-data框架中提供的RestTemplate类可用于在应用中调用rest服务，它简化了与http服务的通信方式，统一了RESTful的标准，封装了http链接，只需要传入url及返回值类型即可。相较于之前常用的HttpClient，RestTemplate是一种更优雅的调用RESTful服务的方式。

#### 代码示例

get请求

|  |
| --- |
| @Test  public void testGet(){  String path="http://api.tianapi.com/txapi/saylove/index?key=db3d22eebbff236d1b3b424ea12c3dd6";  RestTemplate t = new RestTemplate();  String result = t.getForObject(path, String.class);  System.out.println(result);  } |
| @Test  public void testRestUrlGet(){  String path="http://localhost:8080/user/{0}";  RestTemplate t = new RestTemplate();  //rest接口调用url传参  String result = t.getForObject(path, String.class,22);  System.out.println(result);  } |
| @Test  public void testRestUrlGet2(){  String path="http://localhost:8080/user/{uid}";  RestTemplate t = new RestTemplate();  //rest接口调用url传参  Map<String,Object> map = new HashMap<>();  map.put("uid",22);  String result = t.getForObject(path, String.class,map);  System.out.println(result);  } |

Post请求

|  |
| --- |
| @Test  public void testPost(){  String path = "http://localhost:8080/user/{0}";  RestTemplate t = new RestTemplate();  Sysuser u = new Sysuser();  u.setUname("mnb");  u.setUpwd("mmmm");  u.setUphone("11111");  String s = t.postForObject(path, u, String.class);//默认的请求头Content-type:application/json  System.out.println(s);  } |

Put请求

|  |
| --- |
| @Test  public void testPut(){  String path = "http://localhost:8080/user";  RestTemplate t = new RestTemplate();  Sysuser u = new Sysuser();  u.setUid(23);  u.setUname("mnb2");  u.setUpwd("mmmm2");  u.setUphone("22222");  t.put(path,u);//可以向请求体放数据，没有返回值  } |

Delete请求

|  |
| --- |
| @Test  public void testDelete(){  String path = "http://localhost:8080/user/{0}";  RestTemplate t = new RestTemplate();//无法向请求体放数据，没有返回值  } |

## 定时任务

### TimerTask

JDK中内置的定时任务实现方法。

#### 创建任务类

从TimerTask类派生任务子类。

|  |
| --- |
| **public** **class** GetWeatherTask **extends** TimerTask{  **public** **void** getDatas() {  String path ="http://api.tianapi.com/txapi/tianqi/index?key=db3d22eebbff236d1b3b424ea12c3dd6&city=郑州市";  String datas = HttpClientUtil.*doget*(path);  System.***out***.println(datas);  }  @Override  **public** **void** run() {  getDatas();  }  } |

#### 任务调度

|  |
| --- |
| **public** **static** **void** main(String[] args) {  Timer t = **new** Timer();  TimerTask tt = **new** GetWeatherTask();  t.schedule(tt, 30\*60\*1000);//延迟30分钟执行一次任务。最常用  //t.schedule(tt, 2000,24\*60\*60\*1000);//延迟2秒执行任务，并间隔24小时重复执行任务。  } |

### Spring-task

#### 开启注解识别

|  |
| --- |
| <!--开启spring-task注解识别,@Scheduled-->  <task:annotation-driven></task:annotation-driven> |

#### @Scheduled注解

|  |
| --- |
| @Component  **public** **class** GetWeatherTask2{  //任务方法,写cron表达式，日期表达式  //秒 分 时 日 月 周  @Scheduled(cron = "0 \* 15 ? \* \*")  public void getData(){  if(rs.get("weatherTask")!=null){  if(rs.setnx("weatherTask","running")){  String city="郑州市";  String url="http://api.tianapi.com/txapi/tianqi/index?key=db3d22eebbff236d1b3b424ea12c3dd6&city="+city;  RestTemplate t = new RestTemplate();  String json = t.getForObject(url, String.class);  JSONObject jsonObject = JSON.parseObject(json);  JSONArray newslist = jsonObject.getJSONArray("newslist");  for(int i=0;i<newslist.size();i++){  JSONObject dayJson = newslist.getJSONObject(i);  //存redis,String  String date = dayJson.getString("date");  String key = Keys.WEATHER+city+":"+date;  rs.set(key,JSON.toJSONString(dayJson));  }  rs.del("weatherTask");  }  }  }  } |

#### 多任务并行与串行

##### 串行任务

同时定义多个任务，springTask默认是串行执行，如下发现task1与task2在同一个线程中执行。

|  |
| --- |
| @Component **public class** GetWeatherTask {   @Scheduled(cron = **"0/5 \* \* ? \* \*"**)  **public void** task1(){  System.***out***.println(**"task1,当前线程名："**+Thread.*currentThread*().getName());  }   @Scheduled(cron = **"1/5 \* \* ? \* \*"**)  **public void** task2(){  System.***out***.println(**"task2,当前线程名："**+Thread.*currentThread*().getName());  } } |

##### 并行任务

|  |
| --- |
| *<!--识别Scheduled注解-->* <**task:annotation-driven** ></**task:annotation-driven**>  *<!--定义任务调度器线程池，两个任务，池子大小定义2-->* <**task:scheduler id="threadPoolTaskScheduler" pool-size="2"**></**task:scheduler**>  *<!--定义任务bean对象-->* <**bean id="getWeatherTask" class="com.javasm.commons.task.GetWeatherTask"**></**bean**>  *<!--定义任务方法，指定所使用的任务调度线程池，默认串行，使用同一个线程调度多个任务-->* <**task:scheduled-tasks scheduler="threadPoolTaskScheduler"**>  <**task:scheduled ref="getWeatherTask" method="task1" cron="0/5 \* \* ? \* \*"**/>  <**task:scheduled ref="getWeatherTask" method="task2" cron="1/5 \* \* ? \* \*"**/> </**task:scheduled-tasks**> |

### Quartz

Quartz 是一个完全由 Java 编写的开源作业调度框架，为在 Java 应用程序中进行作业调度提供了简单却强大的机制。

Quartz 可以与 JavaEE 和 JavaSE 应用程序相结合也可以单独使用。

Quartz 允许程序开发人员根据时间的间隔来调度作业。

Quartz 实现了作业和触发器的多对多的关系，还能把多个作业与不同的触发器关联。

#### Quartz概念

Job 表示一个工作，要执行的具体内容。

JobDetail 表示一个具体的可执行的调度程序，Job 是这个可执行程调度程序所要执行的内容，另外 JobDetail 还包含了这个任务调度的方案和策略。

Trigger 代表一个调度参数的配置，什么时候去调。

Scheduler 代表一个调度容器，一个调度容器中可以注册多个 JobDetail 和 Trigger。当 Trigger 与 JobDetail 组合，就可以被 Scheduler 容器调度了。

#### Quartz环境

|  |
| --- |
| 结合Spring使用时,需要额外引入的Jar  quartz-2.2.1.jar  quartz-jobs-2.2.1.jar  spring-context-support-5.0.8.RELEASE.jar |

或者maven

|  |
| --- |
| <**dependency**>  <**groupId**>org.quartz-scheduler</**groupId**>  <**artifactId**>quartz</**artifactId**>  <**version**>2.3.2</**version**> </**dependency**> <**dependency**>  <**groupId**>org.springframework</**groupId**>  <**artifactId**>spring-context-support</**artifactId**>  <**version**>${spring.version}</**version**> </**dependency**> |

#### quartz应用

##### 创建任务类

|  |
| --- |
| **public** **class** GetWeatherTask{  **public** **void** getDatas() {  System.out.println(“任务执行”);  }  } |

##### Xml配置

|  |
| --- |
| <!-- 任务bean -->  <bean id=*"wt"* class=*"com.javasm.task.GetWeatherTask"*></bean>    <!-- 任务bean中的任务方法 -->  <bean id=*"wtJobDetail"* class=*"org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean"*>  <property name=*"targetObject"* ref=*"wt"*></property>  <property name=*"targetMethod"* value=*"getDatas"*></property>  <property name=*"concurrent"* value=*"false"*></property><!-- 禁止任务同时执行 -->  </bean>  <!-- 任务方法的执行时间策略 -->  <bean id=*"wtCronTrigger"* class=*"org.springframework.scheduling.quartz.CronTriggerFactoryBean"*>  <property name=*"jobDetail"* ref=*"wtJobDetail"*></property>  <property name=*"cronExpression"* value=*"0 0 55 \* \* ?"*></property>  </bean>  <!-- 任务调度 -->  <bean class=*"org.springframework.scheduling.quartz.SchedulerFactoryBean"*>  <property name=*"triggers"*>  <list>  <ref bean=*"wtCronTrigger"*/>  </list>  </property>  </bean> |

### cronExpression表达式

Cron表达式包含以下字段,使用空格分隔。

|  |
| --- |
| 秒 分 小时 月内日期 月 周内日期 年（可选字段） |
| ***Cron表达式中的特殊字符***  反斜线（/）字符表示增量值。例如，在秒字段中“5/15”代表从第 5 秒开始，每 15 秒一次。  问号（?）字符和字母 (L)字符只有在月内日期和周内日期字段中可用。问号表示这个字段不包含具体值。所以，如果指定月内日期，可以在周内日期字段中插入“?”，表示周内日期值无关紧要。字母 L 字符是 last 的缩写。放在月内日期字段中，表示安排在当月最后一天执行。在周内日期字段中，如果“L”单独存在，就等于“7”，否则代表当月内周内日期的最后一个实例。所以“0L”表示安排在当月的最后一个星期一执行。  在月内日期字段中的字母（W）字符把执行安排在最靠近指定值的工作日。把“1W”放在月内日期字段中，表示把执行安排在当月的第一个工作日内。  井号（#）字符为给定月份指定具体的工作日实例。把“MON#2”放在周内日期字段中，表示把任务安排在当月的第二个星期一。  星号（\*）字符是通配字符，表示该字段可以接受任何可能的值。 |
| ***Cron表达式示例***  “0 0 12 \* \* ?" 每天中午12点触发  "0 15 10 ? \* \*" 每天上午10:15触发  "0 15 10 \* \* ?" 每天上午10:15触发  "0 15 10 \* \* ?" 每天上午10:15触发  "0 15 10 \* \* ? 2005" 2005年的每天上午10:15触发  "0 \* 14 \* \* ?" 在每天下午2点到下午2:59期间的每1分钟触发  "0 0/5 14 \* \* ?" 在每天下午2点到下午2:55期间的每5分钟触发  "0 0/5 14,18 \* \* ?" 在每天下午2点到2:55期间和下午6点到6:55期间的每5分钟触发  "0 0-5 14 \* \* ?" 在每天下午2点到下午2:05期间的每1分钟触发  "0 10,44 14 ? 3 WED" 每年三月的星期三的下午2:10和2:44触发  "0 15 10 ? \* 2-6" 周一至周五的上午10:15触发  "0 15 10 15 \* ?" 每月15日上午10:15触发  "0 15 10 L \* ?" 每月最后一日的上午10:15触发  "0 15 10 ? \* 6L" 每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6L 2002-2005" 2002年至2005年的每月的最后一个星期五上午10:15触发  "0 15 10 ? \* 6#3" 每月的第三个星期五上午10:15触发 |

## 异步任务

### 创建线程池ThreadPoolTaskExecutor对象

|  |
| --- |
| <**bean id="taskpoolExector" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor"**>  *<!-- 核心线程数 -->* <**property name="corePoolSize" value="5"**/>  *<!-- 最大线程数 -->* <**property name="maxPoolSize" value="20"**/>  *<!-- 队列最大长度 -->* <**property name="queueCapacity" value="10"**/>  *<!-- 线程池维护线程所允许的空闲时间 -->* <**property name="keepAliveSeconds" value="20"**/>  *<!-- 线程池对拒绝任务(无线程可用)的处理策略，CallerRunsPolicy表示在当前线程直接执行 -->* <**property name="rejectedExecutionHandler"** >  <**bean class="java.util.concurrent.ThreadPoolExecutor$CallerRunsPolicy"**/>  </**property**> </**bean**> |

### 调用ThreadPoolTaskExecutor执行异步任务

以下代码已模拟发短信为例，由于发短信属于网络请求，会阻塞程序，因此需要异步发短信，使用线程池异步发送。

|  |
| --- |
| @Resource **private** ThreadPoolTaskExecutor **executor**; @Override **public boolean** sendValiCode(String uphone) {  String valicode = StringUtils.*getSixNumCode*();  System.***out***.println(**"验证码："**+valicode+**",线程："**+Thread.*currentThread*().getName());  **executor**.execute(**new** Runnable() {  @Override  **public void** run() {  System.***out***.println(**"发短信实现，线程："**+Thread.*currentThread*().getName());  }  });   **rs**.setex(RedisKey.***phonecode***+uphone,5\*60,valicode);  **return true**; } |

注意点：4.1中配置TheadPoolTaskExector线程池可以简化配置，效果与4.1配置相同。配置如下：

|  |
| --- |
| <**task:executor id="taskExecutor" pool-size="5" keep-alive="20" queue-capacity="10" rejection-policy="CALLER\_RUNS"**></**task:executor**> |