## 分布式Session一致性问题

### 什么是Session

Session 是客户端与服务器通讯会话技术， 比如浏览器登陆、记录整个浏览会话信息

### Session实现原理

客户对向服务器端发送请求后，Session 创建在服务器端，返回Sessionid给客户端浏览器保存在本地，当下次发送请求的时候，在请求头中传递sessionId获取对应的从服务器上获取对应的Sesison

### Session常见问题

#### Session 保证在那里？

答案：存放在服务器上

#### 关闭浏览器Session会失效吗

答案：不会消失

#### 相关代码

|  |
| --- |
| @SpringBootApplication  @RestController  **public** **class** TestSessionController {  // 创建session 会话  @RequestMapping("/createSession")  **public** String createSession(HttpServletRequest request, String nameValue) {  HttpSession session = request.getSession();  System.***out***.println("存入Session sessionid:信息" + session.getId() + ",nameValue:" + nameValue);  session.setAttribute("name", nameValue);  **return** "success";  }  // 获取session 会话  @RequestMapping("/getSession")  **public** Object getSession(HttpServletRequest request) {  HttpSession session = request.getSession();  System.***out***.println("获取Session sessionid:信息" + session.getId());  Object value = session.getAttribute("name");  **return** value;  }  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(TestSessionController.**class**, args);  }  } |

### 服务集群会产生那些问题

如果服务器产生了集群后，因为session是存放在服务器上，客户端会使用同一个Sessionid在多个不同的服务器上获取对应的Session，从而会导致Session不一致问题。

#### Nginx配置负载均衡

Nginx负载均衡提供上游服务器(真实业务逻辑访问的服务器),负载均衡、故障转移、失败重试、容错、健康检查等。

当上游服务器(真实业务逻辑访问的服务器)发生故障时，可以转移到其他上游服务器(真实业务逻辑访问的服务器)。

#### Upstream Server配置

upstream 主要配置如下:

IP地址和端口号：配置上游服务器的IP地址和端口

|  |
| --- |
| ###定义上游服务器(需要被nginx真实代理访问的服务器) 默认是轮训机制  upstream backServer{  server 127.0.0.1:8080;  server 127.0.0.1:8081;  }    server {  listen 80;  server\_name www.itmayiedu.com;  location / {  ### 指定上游服务器负载均衡服务器  proxy\_pass http://backServer;  index index.html index.htm;  }  } |

#### 负载均衡算法

1、轮询（默认）

每个请求按时间顺序逐一分配到不同的后端服务，如果后端某台服务器死机，自动剔除故障系统，使用户访问不受影响。

2、weight（轮询权值）

weight的值越大分配到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下。或者仅仅为在主从的情况下设置不同的权值，达到合理有效的地利用主机资源。

3、ip\_hash

每个请求按访问IP的哈希结果分配，使来自同一个IP的访客固定访问一台后端服务器，并且可以有效解决动态网页存在的session共享问题。俗称IP绑定。

4、fair（第三方）

比 weight、ip\_hash更加智能的负载均衡算法，fair算法可以根据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间 来分配请求，响应时间短的优先分配。Nginx本身不支持fair，如果需要这种调度算法，则必须安装upstream\_fair模块。

5、url\_hash(第三方)

按访问的URL的哈希结果来分配请求，使每个URL定向到一台后端服务器，可以进一步提高后端缓存服务器的效率。Nginx本身不支持url\_hash，如果需要这种调度算法，则必须安装Nginx的hash软件包。

|  |
| --- |
| @SpringBootApplication  @RestController  **public** **class** TestSessionController {  @Value("${server.port}")  **private** String serverPort;  @RequestMapping("/")  **public** String index() {  **return** serverPort;  }  // 创建session 会话  @RequestMapping("/createSession")  **public** String createSession(HttpServletRequest request, String nameValue) {  HttpSession session = request.getSession();  System.***out***.println(  "存入Session sessionid:信息" + session.getId() + ",nameValue:" + nameValue + ",serverPort:" + serverPort);  session.setAttribute("name", nameValue);  **return** "success-" + serverPort;  }  // 获取session 会话  @RequestMapping("/getSession")  **public** Object getSession(HttpServletRequest request) {  HttpSession session = request.getSession(**false**);  **if** (session == **null**) {  **return** serverPort + "-" + "没有找到对应的session值";  }  System.***out***.println("获取Session sessionid:信息" + session.getId() + "serverPort:" + serverPort);  Object value = session.getAttribute("name");  **return** serverPort + "-" + value;  }  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(TestSessionController.**class**, args);  }  } |

### 分布式Session一致性解决方案

#### nginx或者haproxy实现IP绑定

用Nginx 做的负载均衡可以添加ip\_hash这个配置，

用haproxy做的负载均衡可以用 balance source这个配置。

从而使同一个ip的请求发到同一台服务器。

#### 利用数据库同步session

#### 使用Session集群存放Redis

使用spring-session框架，底层实现原理是重写httpsession

##### 引入maven依赖

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.0.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <weixin-java-mp.version>2.8.0</weixin-java-mp.version>  <maven.compiler.source>1.8</maven.compiler.source>  <maven.compiler.target>1.8</maven.compiler.target>  <maven.compiler.encoding>UTF-8</maven.compiler.encoding>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.build.locales>zh\_CN</project.build.locales>  </properties>  <dependencies>  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <!-- <exclusions> <exclusion> <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId> </exclusion> </exclusions> -->  </dependency>  <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.47</version>  </dependency>  <!-- Testing Dependencies -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-redis</artifactId>  </dependency>  <!--spring session 与redis应用基本环境配置,需要开启redis后才可以使用，不然启动Spring boot会报错 -->  <dependency>  <groupId>org.springframework.session</groupId>  <artifactId>spring-session-data-redis</artifactId>  </dependency>  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-pool2</artifactId>  </dependency>  <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  </dependency>  </dependencies>  <build>  <plugins>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  <configuration>  <maimClass>com.meiteedu.WxMpApplication</maimClass>  </configuration>  <executions>  <execution>  <goals>  <goal>repackage</goal>  </goals>  </execution>  </executions>  </plugin>  </plugins>  </build> |

##### YML配置信息

|  |
| --- |
| server:  port: 8080  redis:  hostname: 192.168.212.151  port: 6379  password: 123456 |

启动redis /usr/local/redis/bin/redis-server /usr/local/redis/etc/redis.conf

##### 创建SessionConfig

|  |
| --- |
| **import org.springframework.beans.factory.annotation.Value;**  **import org.springframework.context.annotation.Bean;**  **import org.springframework.data.redis.connection.jedis.JedisConnectionFactory;**  **import org.springframework.session.data.redis.config.annotation.web.http.EnableRedisHttpSession;**  **//这个类用配置redis服务器的连接**  **//maxInactiveIntervalInSeconds为SpringSession的过期时间（单位：秒）**  **@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 1800)**  **public class SessionConfig {**  **// 冒号后的值为没有配置文件时，制动装载的默认值**  **@Value("${redis.hostname:localhost}")**  **String HostName;**  **@Value("${redis.port:6379}")**  **int Port;**  **@Bean**  **public JedisConnectionFactory connectionFactory() {**  **JedisConnectionFactory connection = new JedisConnectionFactory();**  **connection.setPort(Port);**  **connection.setHostName(HostName);**  **return connection;**  **}**  **}** |

##### 初始化Session

|  |
| --- |
| **//初始化Session配置**  **public class SessionInitializer extends AbstractHttpSessionApplicationInitializer{**  **public SessionInitializer() {**  **super(SessionConfig.class);**  **}**  **}** |

#### 最靠谱的分布式Session解决方案

基于令牌（Token）方式实现Session解决方案，因为Session本身就是分布式共享连接。

|  |
| --- |
| @Service  **public** **class** TokenService {  @Autowired  **private** RedisService redisService;  // 新增 返回token  **public** String put(Object object) {  String token = getToken();  redisService.setString(token, object);  **return** token;  }  // 获取信息  **public** String get(String token) {  String reuslt = redisService.getString(token);  **return** reuslt;  }  **public** String getToken() {  **return** UUID.*randomUUID*().toString();  }  } |

TokenController

|  |
| --- |
| @RestController  **public** **class** TokenController {  @Autowired  **private** TokenService tokenService;  @Value("${server.port}")  **private** String serverPort;  @RequestMapping("/put")  **public** String put(String nameValue) {  String token = tokenService.put(nameValue);  **return** token + "-" + serverPort;  }  @RequestMapping("/get")  **public** String get(String token) {  String value = tokenService.get(token);  **return** value + "-" + serverPort;  }  } |

## 网站跨域解决方案

### 什么是网站跨域

跨域原因产生：在当前域名请求网站中，默认不允许通过ajax请求发送其他域名。

### 网站跨域报错案例

jquery-1.7.2.min.js?t=2017-07-27:4 Failed to load http://b.itmayiedu.com:8081/ajaxB: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://a.itmayiedu.com:8080' is therefore not allowed access.

### 五种网站跨域解决方案

1. 使用jsonp解决网站跨域   
    2.使用HttpClient内部转发  
    3.使用设置响应头允许跨域  
    4.基于Nginx搭建企业级API接口网关  
    5.使用Zuul搭建微服务API接口网关

### 跨域项目环境搭建

#### 使用JSONP解决网站跨域

##### 前端代码

|  |
| --- |
| <script type=*"text/javascript"*  src=*"http://www.itmayiedu.com/static/common/jquery-1.7.2.min.js?t=2017-07-27"*></script>  <script type=*"text/javascript"*>  $(document).ready(**function**() {  $.ajax({  type : "GET",  async : **false**,  url : "http://b.itmayiedu.com:8081/ajaxB",  dataType : "jsonp",  jsonp : "jsonpCallback",//服务端用于接收callback调用的function名的参数  success : **function**(data) {  alert(data["errorCode"]);  },  error : **function**() {  alert('fail');  }  });  });  </script> |

##### 后端代码

|  |
| --- |
| @RequestMapping(value = "/ajaxB", method = RequestMethod.***GET***)  **public** **void** ajaxB(HttpServletResponse response, String jsonpCallback) **throws** IOException {  JSONObject root = **new** JSONObject();  root.put("errorCode", "200");  root.put("errorMsg", "登陆成功");  response.setHeader("Content-type", "text/html;charset=UTF-8");  PrintWriter writer = response.getWriter();  writer.print(jsonpCallback + "(" + root.toString() + ")");  writer.close();  } |

缺点：不支持post请求，代码书写比较复杂

#### 使用设置响应头允许跨域

##### 前端代码

|  |
| --- |
| <script type=*"text/javascript"*  src=*"http://www.itmayiedu.com/static/common/jquery-1.7.2.min.js?t=2017-07-27"*></script>  <script type=*"text/javascript"*>  $(document).ready(**function**() {  $.ajax({  type : "GET",  async : **false**,  url : "http://b.itmayiedu.com:8081/ajaxB",  dataType : "json",  success : **function**(data) {  alert(data["errorCode"]);  },  error : **function**() {  alert('fail');  }  });  });  </script> |

##### 后端代码

|  |
| --- |
| @RequestMapping("/ajaxB")  **public** Map<String, Object> ajaxB(HttpServletResponse response) {  response.setHeader("Access-Control-Allow-Origin", "\*");  Map<String, Object> result = **new** HashMap<String, Object>();  result.put("errorCode", "200");  result.put("errorMsg", "登陆成功");  **return** result;  } |

response.setHeader("Access-Control-Allow-Origin", "\*"); 设置响应头允许跨域

如果在实际项目中，该代码建议放在过滤器中。

#### 使用HttpClient进行内部转发

##### 前端代码

|  |
| --- |
| <script type=*"text/javascript"*  src=*"http://www.itmayiedu.com/static/common/jquery-1.7.2.min.js?t=2017-07-27"*></script>  <script type=*"text/javascript"*>  $(document).ready(**function**() {  $.ajax({  type : "POST",  async : **false**,  url : "http://a.itmayiedu.com:8080/forwardB",  dataType : "json",  success : **function**(data) {  alert(data["errorCode"]);  },  error : **function**() {  alert('fail');  }  });  });  </script> |

##### 后端代码

A项目进行转发到B项目

|  |
| --- |
| @RequestMapping("/forwardB")  @ResponseBody  **public** JSONObject forwardB() {  JSONObject result = HttpClientUtils.*httpGet*("http://b.itmayiedu.com:8081/ajaxB");  System.***out***.println("result:" + result);  **return** result;  } |

B项目代码

|  |
| --- |
| @RequestMapping("/ajaxB")  **public** Map<String, Object> ajaxB(HttpServletResponse response) {  response.setHeader("Access-Control-Allow-Origin", "\*");  Map<String, Object> result = **new** HashMap<String, Object>();  result.put("errorCode", "200");  result.put("errorMsg", "登陆成功");  **return** result;  } |

#### 搭建企业级API接口网关

##### 使用Nginx搭建API接口网关

###### Nginx相关配置

|  |
| --- |
| server {  listen 80;  server\_name www.itmayiedu.com;  ###A项目  location /a {  proxy\_pass http://a.itmayiedu.com:8080/;  index index.html index.htm;  }  ###B项目  location /b {  proxy\_pass http://b.itmayiedu.com:8081/;  index index.html index.htm;  }  } |

###### 前端代码

|  |
| --- |
| <script type=*"text/javascript"*  src=*"http://code.jquery.com/jquery-1.8.0.min.js"*></script>  <script type=*"text/javascript"*>  $(document).ready(**function**() {  $.ajax({  type : "POST",  async : **false**,  url : "http://www.itmayiedu.com/b/ajaxB",  dataType : "json",  success : **function**(data) {  alert(data["errorCode"]);  },  error : **function**() {  alert('fail');  }  });  });  </script> |

###### 后端代码

|  |
| --- |
| @RequestMapping("/ajaxB")  **public** Map<String, Object> ajaxB(HttpServletResponse response) {  response.setHeader("Access-Control-Allow-Origin", "\*");  Map<String, Object> result = **new** HashMap<String, Object>();  result.put("errorCode", "200");  result.put("errorMsg", "登陆成功");  **return** result;  } |

##### SpringCloud搭建API接口网关

使用SpringCloud Zuul搭建API接口网关

#### Maven初始化依赖参数

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.0.RELEASE</version>  </parent>  <dependencies>  <!-- SpringBoot 对lombok 支持 -->  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  </dependency>  <!-- SpringBoot web 核心组件 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </dependency>  <!-- SpringBoot 外部tomcat支持 -->  <dependency>  <groupId>org.apache.tomcat.embed</groupId>  <artifactId>tomcat-embed-jasper</artifactId>  </dependency>  <!-- springboot-log4j -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-log4j</artifactId>  <version>1.3.8.RELEASE</version>  </dependency>  <!-- springboot-aop 技术 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <dependency>  <groupId>org.apache.httpcomponents</groupId>  <artifactId>httpclient</artifactId>  </dependency>  <!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.47</version>  </dependency>  </dependencies> |

#### application.yml

|  |
| --- |
| server:  port: 8080  spring:  mvc:  view:  prefix: /WEB-INF/jsp/  suffix: .jsp |

#### HttpClientUtils工具类

|  |
| --- |
| **public** **class** HttpClientUtils {  **private** **static** Logger *logger* = LoggerFactory.*getLogger*(HttpClientUtils.**class**); // 日志记录  **private** **static** RequestConfig *requestConfig* = **null**;  **static** {  // 设置请求和传输超时时间  *requestConfig* = RequestConfig.*custom*().setSocketTimeout(2000).setConnectTimeout(2000).build();  }  /\*\*  \* post请求传输json参数  \*  \* **@param** url  \* url地址  \* **@param** json  \* 参数  \* **@return**  \*/  **public** **static** JSONObject httpPost(String url, JSONObject jsonParam) {  // post请求返回结果  CloseableHttpClient httpClient = HttpClients.*createDefault*();  JSONObject jsonResult = **null**;  HttpPost httpPost = **new** HttpPost(url);  // 设置请求和传输超时时间  httpPost.setConfig(*requestConfig*);  **try** {  **if** (**null** != jsonParam) {  // 解决中文乱码问题  StringEntity entity = **new** StringEntity(jsonParam.toString(), "utf-8");  entity.setContentEncoding("UTF-8");  entity.setContentType("application/json");  httpPost.setEntity(entity);  }  CloseableHttpResponse result = httpClient.execute(httpPost);  // 请求发送成功，并得到响应  **if** (result.getStatusLine().getStatusCode() == HttpStatus.***SC\_OK***) {  String str = "";  **try** {  // 读取服务器返回过来的json字符串数据  str = EntityUtils.*toString*(result.getEntity(), "utf-8");  // 把json字符串转换成json对象  jsonResult = JSONObject.*parseObject*(str);  } **catch** (Exception e) {  *logger*.error("post请求提交失败:" + url, e);  }  }  } **catch** (IOException e) {  *logger*.error("post请求提交失败:" + url, e);  } **finally** {  httpPost.releaseConnection();  }  **return** jsonResult;  }  /\*\*  \* post请求传输String参数 例如：name=Jack&sex=1&type=2  \* Content-type:application/x-www-form-urlencoded  \*  \* **@param** url  \* url地址  \* **@param** strParam  \* 参数  \* **@return**  \*/  **public** **static** JSONObject httpPost(String url, String strParam) {  // post请求返回结果  CloseableHttpClient httpClient = HttpClients.*createDefault*();  JSONObject jsonResult = **null**;  HttpPost httpPost = **new** HttpPost(url);  httpPost.setConfig(*requestConfig*);  **try** {  **if** (**null** != strParam) {  // 解决中文乱码问题  StringEntity entity = **new** StringEntity(strParam, "utf-8");  entity.setContentEncoding("UTF-8");  entity.setContentType("application/x-www-form-urlencoded");  httpPost.setEntity(entity);  }  CloseableHttpResponse result = httpClient.execute(httpPost);  // 请求发送成功，并得到响应  **if** (result.getStatusLine().getStatusCode() == HttpStatus.***SC\_OK***) {  String str = "";  **try** {  // 读取服务器返回过来的json字符串数据  str = EntityUtils.*toString*(result.getEntity(), "utf-8");  // 把json字符串转换成json对象  jsonResult = JSONObject.*parseObject*(str);  } **catch** (Exception e) {  *logger*.error("post请求提交失败:" + url, e);  }  }  } **catch** (IOException e) {  *logger*.error("post请求提交失败:" + url, e);  } **finally** {  httpPost.releaseConnection();  }  **return** jsonResult;  }  /\*\*  \* 发送get请求  \*  \* **@param** url  \* 路径  \* **@return**  \*/  **public** **static** JSONObject httpGet(String url) {  // get请求返回结果  JSONObject jsonResult = **null**;  CloseableHttpClient client = HttpClients.*createDefault*();  // 发送get请求  HttpGet request = **new** HttpGet(url);  request.setConfig(*requestConfig*);  **try** {  CloseableHttpResponse response = client.execute(request);  // 请求发送成功，并得到响应  **if** (response.getStatusLine().getStatusCode() == HttpStatus.***SC\_OK***) {  // 读取服务器返回过来的json字符串数据  HttpEntity entity = response.getEntity();  String strResult = EntityUtils.*toString*(entity, "utf-8");  // 把json字符串转换成json对象  jsonResult = JSONObject.*parseObject*(strResult);  } **else** {  *logger*.error("get请求提交失败:" + url);  }  } **catch** (IOException e) {  *logger*.error("get请求提交失败:" + url, e);  } **finally** {  request.releaseConnection();  }  **return** jsonResult;  }  } |

#### Jsp

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*  pageEncoding=*"UTF-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>  <title>Insert title here</title>  <script type=*"text/javascript"*  src=*"http://code.jquery.com/jquery-1.8.0.min.js"*></script>  <script type=*"text/javascript"*>  $(document).ready(**function**() {  $.ajax({  type : "POST",  async : **false**,  url : "http://www.itmayiedu.com/b/ajaxB",  dataType : "json",  success : **function**(data) {  alert(data["errorCode"]);  },  error : **function**() {  alert('fail');  }  });  });  </script>  </head>  <body>显示 ....  </body>  </html> |

## 分布式锁解决方案

# 分布式任务调度平台

## 什么是定时任务

指定时间去执行任务

## Java实现定时任务方式

### Thread

|  |
| --- |
| **public class Demo01 {**  **static long *count* = 0;**  **public static void main(String[] args) {**  **Runnable runnable = new Runnable() {**  **@Override**  **public void run() {**  **while (true) {**  **try {**  **Thread.*sleep*(1000);**  ***count*++;**  **System.*out*.println(*count*);**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **}**  **}**  **};**  **Thread thread = new Thread(runnable);**  **thread.start();**  **}**  **}** |

### TimerTask

|  |
| --- |
| **/\*\***  **\* 使用TimerTask类实现定时任务**  **\*/**  **public class Demo02 {**  **static long *count* = 0;**  **public static void main(String[] args) {**  **TimerTask timerTask = new TimerTask() {**  **@Override**  **public void run() {**  ***count*++;**  **System.*out*.println(*count*);**  **}**  **};**  **Timer timer = new Timer();**  **// 天数**  **long delay = 0;**  **// 秒数**  **long period = 1000;**  **timer.scheduleAtFixedRate(timerTask, delay, period);**  **}**  **}** |

### ScheduledExecutorService

使用ScheduledExecutorService是从Java

JavaSE5的java.util.concurrent里，做为并发工具类被引进的，这是最理想的定时任务实现方式。

|  |
| --- |
| **public class Demo003 {**  **public static void main(String[] args) {**  **Runnable runnable = new Runnable() {**  **public void run() {**  **// task to run goes here**  **System.*out*.println("Hello !!");**  **}**  **};**  **ScheduledExecutorService service = Executors.*newSingleThreadScheduledExecutor*();**  **// 第二个参数为首次执行的延时时间，第三个参数为定时执行的间隔时间**  **service.scheduleAtFixedRate(runnable, 1, 1, TimeUnit.*SECONDS*);**  **}**  **}** |

### Quartz

#### 创建一个quartz\_demo项目

#### 引入maven依赖

|  |
| --- |
| **<dependencies>**  **<!-- quartz -->**  **<dependency>**  **<groupId>org.quartz-scheduler</groupId>**  **<artifactId>quartz</artifactId>**  **<version>2.2.1</version>**  **</dependency>**  **<dependency>**  **<groupId>org.quartz-scheduler</groupId>**  **<artifactId>quartz-jobs</artifactId>**  **<version>2.2.1</version>**  **</dependency>**  **</dependencies>** |

#### 任务调度类

|  |
| --- |
| **public class MyJob implements Job {**  **public void execute(JobExecutionContext context) throws JobExecutionException {**  **System.*out*.println("quartz MyJob date:" + new Date().getTime());**  **}**  **}** |

#### 启动类

|  |
| --- |
| **//1.创建Scheduler的工厂**  **SchedulerFactory sf = new StdSchedulerFactory();**  **//2.从工厂中获取调度器实例**  **Scheduler scheduler = sf.getScheduler();**  **//3.创建JobDetail**  **JobDetail jb = JobBuilder.*newJob*(MyJob.class)**  **.withDescription("this is a ram job") //job的描述**  **.withIdentity("ramJob", "ramGroup") //job 的name和group**  **.build();**  **//任务运行的时间，SimpleSchedle类型触发器有效**  **long time= System.*currentTimeMillis*() + 3\*1000L; //3秒后启动任务**  **Date statTime = new Date(time);**  **//4.创建Trigger**  **//使用SimpleScheduleBuilder或者CronScheduleBuilder**  **Trigger t = TriggerBuilder.*newTrigger*()**  **.withDescription("")**  **.withIdentity("ramTrigger", "ramTriggerGroup")**  **//.withSchedule(SimpleScheduleBuilder.simpleSchedule())**  **.startAt(statTime) //默认当前时间启动**  **.withSchedule(CronScheduleBuilder.*cronSchedule*("0/2 \* \* \* \* ?")) //两秒执行一次**  **.build();**  **//5.注册任务和定时器**  **scheduler.scheduleJob(jb, t);**  **//6.启动 调度器**  **scheduler.start();** |

#### Quartz表达式

<http://cron.qqe2.com/>

## 分布式情况下定时任务会出现哪些问题？

分布式集群的情况下，怎么保证定时任务不被重复执行

## 分布式定时任务解决方案

①使用zookeeper实现分布式锁 缺点(需要创建临时节点、和事件通知不易于扩展)

②使用配置文件做一个开关 缺点发布后，需要重启

③数据库唯一约束，缺点效率低

④使用分布式任务调度平台XXLJOB、Elastric-Job、TBSchedule

## XXLJOB介绍

1、简单：支持通过Web页面对任务进行CRUD操作，操作简单，一分钟上手；

2、动态：支持动态修改任务状态、暂停/恢复任务，以及终止运行中任务，即时生效；

3、调度中心HA（中心式）：调度采用中心式设计，“调度中心”基于集群Quartz实现，可保证调度中心HA；

4、执行器HA（分布式）：任务分布式执行，任务"执行器"支持集群部署，可保证任务执行HA；

5、任务Failover：执行器集群部署时，任务路由策略选择"故障转移"情况下调度失败时将会平滑切换执行器进行Failover；

6、一致性：“调度中心”通过DB锁保证集群分布式调度的一致性, 一次任务调度只会触发一次执行；

7、自定义任务参数：支持在线配置调度任务入参，即时生效；

8、调度线程池：调度系统多线程触发调度运行，确保调度精确执行，不被堵塞；

9、弹性扩容缩容：一旦有新执行器机器上线或者下线，下次调度时将会重新分配任务；

10、邮件报警：任务失败时支持邮件报警，支持配置多邮件地址群发报警邮件；

11、状态监控：支持实时监控任务进度；

12、Rolling执行日志：支持在线查看调度结果，并且支持以Rolling方式实时查看执行器输出的完整的执行日志；

13、GLUE：提供Web IDE，支持在线开发任务逻辑代码，动态发布，实时编译生效，省略部署上线的过程。支持30个版本的历史版本回溯。

14、数据加密：调度中心和执行器之间的通讯进行数据加密，提升调度信息安全性；

15、任务依赖：支持配置子任务依赖，当父任务执行结束且执行成功后将会主动触发一次子任务的执行, 多个子任务用逗号分隔；

16、推送maven中央仓库: 将会把最新稳定版推送到maven中央仓库, 方便用户接入和使用;

17、任务注册: 执行器会周期性自动注册任务, 调度中心将会自动发现注册的任务并触发执行。同时，也支持手动录入执行器地址；

18、路由策略：执行器集群部署时提供丰富的路由策略，包括：第一个、最后一个、轮询、随机、一致性HASH、最不经常使用、最近最久未使用、故障转移、忙碌转移等；

19、运行报表：支持实时查看运行数据，如任务数量、调度次数、执行器数量等；以及调度报表，如调度日期分布图，调度成功分布图等；

20、脚本任务：支持以GLUE模式开发和运行脚本任务，包括Shell、Python等类型脚本;

21、阻塞处理策略：调度过于密集执行器来不及处理时的处理策略，策略包括：单机串行（默认）、丢弃后续调度、覆盖之前调度；

22、失败处理策略；调度失败时的处理策略，策略包括：失败告警（默认）、失败重试；

23、分片广播任务：执行器集群部署时，任务路由策略选择"分片广播"情况下，一次任务调度将会广播触发对应集群中所有执行器执行一次任务，同时传递分片参数；可根据分片参数开发分片任务；

24、动态分片：分片广播任务以执行器为维度进行分片，支持动态扩容执行器集群从而动态增加分片数量，协同进行业务处理；在进行大数据量业务操作时可显著提升任务处理能力和速度。

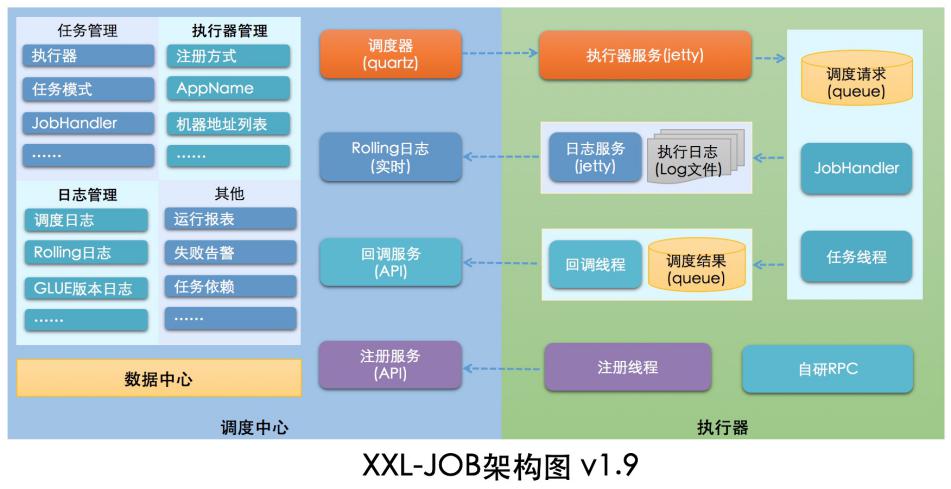
25、事件触发：除了"Cron方式"和"任务依赖方式"触发任务执行之外，支持基于事件的触发任务方式。调度中心提供触发任务单次执行的API服务，可根据业务事件灵活触发。

### XXLJOBGitHub地址

https://github.com/xuxueli/xxl-job

### 文档

[http://www.xuxueli.com/xxl-job/#/?id=\_21-%e5%88%9d%e5%a7%8b%e5%8c%96%e8%b0%83%e5%ba%a6%e6%95%b0%e6%8d%ae%e5%ba%93](http://www.xuxueli.com/xxl-job/" \l "/?id=_21-%e5%88%9d%e5%a7%8b%e5%8c%96%e8%b0%83%e5%ba%a6%e6%95%b0%e6%8d%ae%e5%ba%93)



### 步骤:

1. 部署: xxl-job-admin 作为注册中心
2. 创建执行器（具体调度地址） 可以支持集群
3. 配置文件需要填写xxl-job注册中心地址
4. 每个具体执行job服务器需要创建一个netty连接端口号
5. 需要执行job的任务类，集成IJobHandler抽象类注册到job容器中
6. Execute方法中编写具体job任务

### SpringBoot整合XXLJob

#### Maven依赖信息

|  |
| --- |
| <properties>  <javax.servlet-api.version>3.0.1</javax.servlet-api.version>  <jsp-api.version>2.2</jsp-api.version>  <spring.version>4.3.14.RELEASE</spring.version>  <jackson.version>2.9.4</jackson.version>  <aspectjweaver.version>1.8.13</aspectjweaver.version>  <slf4j-api.version>1.7.25</slf4j-api.version>  <freemarker.version>2.3.23</freemarker.version>  <junit.version>4.12</junit.version>  <jetty-server.version>9.2.24.v20180105</jetty-server.version>  <hessian.version>4.0.51</hessian.version>  <httpclient.version>4.5.5</httpclient.version>  <commons-exec.version>1.3</commons-exec.version>  <commons-collections4.version>4.1</commons-collections4.version>  <commons-lang3.version>3.7</commons-lang3.version>  <commons-email.version>1.5</commons-email.version>  <c3p0.version>0.9.5.2</c3p0.version>  <mysql-connector-java.version>5.1.45</mysql-connector-java.version>  <mybatis-spring.version>1.3.1</mybatis-spring.version>  <mybatis.version>3.4.5</mybatis.version>  <groovy-all.version>2.4.13</groovy-all.version>  <quartz.version>2.3.0</quartz.version>  <spring-boot.version>1.5.10.RELEASE</spring-boot.version>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <project.parent.version>1.9.2-SNAPSHOT</project.parent.version>  </properties>  <dependencyManagement>  <dependencies>  <dependency>  <!-- Import dependency management from Spring Boot (依赖管理：继承一些默认的依赖，工程需要依赖的jar包的管理，申明其他dependency的时候就不需要version) -->  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>${spring-boot.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  <!-- jetty -->  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artifactId>jetty-server</artifactId>  <version>${jetty-server.version}</version>  </dependency>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artifactId>jetty-util</artifactId>  <version>${jetty-server.version}</version>  </dependency>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artifactId>jetty-http</artifactId>  <version>${jetty-server.version}</version>  </dependency>  <dependency>  <groupId>org.eclipse.jetty</groupId>  <artifactId>jetty-io</artifactId>  <version>${jetty-server.version}</version>  </dependency>  </dependencies>  </dependencyManagement>  <dependencies>  <!-- spring-boot-starter-web (spring-webmvc + tomcat) -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  <!-- xxl-job-core -->  <dependency>  <groupId>com.xuxueli</groupId>  <artifactId>xxl-job-core</artifactId>  <version>${project.parent.version}</version>  </dependency>  </dependencies> |

#### 配置文件信息

application.properties

|  |
| --- |
| # web port  server.port=8081  # log config  logging.config=classpath:logback.xml  ### xxl-job admin address list, such as "http://address" or "http://address01,http://address02"  xxl.job.admin.addresses=http://127.0.0.1:8080/xxl-job-admin  ### xxl-job executor address  xxl.job.executor.appname=app-itmayiedu  xxl.job.executor.ip=  xxl.job.executor.port=9999  ### xxl-job, access token  xxl.job.accessToken=  ### xxl-job log path  xxl.job.executor.logpath=/data/applogs/xxl-job/jobhandler  ### xxl-job log retention days  xxl.job.executor.logretentiondays=-1 |

#### 配置XxlJobConfig

com.itmayiedu.job.executor.core.config

|  |
| --- |
| @Configuration  @ComponentScan(basePackages = "com.xxl.job.executor.service.jobhandler")  **public** **class** XxlJobConfig {  **private** Logger logger = LoggerFactory.*getLogger*(XxlJobConfig.**class**);  @Value("${xxl.job.admin.addresses}")  **private** String adminAddresses;  @Value("${xxl.job.executor.appname}")  **private** String appName;  @Value("${xxl.job.executor.ip}")  **private** String ip;  @Value("${xxl.job.executor.port}")  **private** **int** port;  @Value("${xxl.job.accessToken}")  **private** String accessToken;  @Value("${xxl.job.executor.logpath}")  **private** String logPath;  @Value("${xxl.job.executor.logretentiondays}")  **private** **int** logRetentionDays;  @Bean(initMethod = "start", destroyMethod = "destroy")  **public** XxlJobExecutor xxlJobExecutor() {  logger.info(">>>>>>>>>>> xxl-job config init.");  XxlJobExecutor xxlJobExecutor = **new** XxlJobExecutor();  xxlJobExecutor.setAdminAddresses(adminAddresses);  xxlJobExecutor.setAppName(appName);  xxlJobExecutor.setIp(ip);  xxlJobExecutor.setPort(port);  xxlJobExecutor.setAccessToken(accessToken);  xxlJobExecutor.setLogPath(logPath);  xxlJobExecutor.setLogRetentionDays(logRetentionDays);  **return** xxlJobExecutor;  }  } |

#### 创建handler接口

com.itmayiedu.job.executor.service.jobhandler

|  |
| --- |
| @JobHandler("demoJobHandler")  @Component  **public** **class** DemoHandler **extends** IJobHandler {  @Value("${server.port}")  **private** String serverPort;  @Override  **public** ReturnT<String> execute(String param) **throws** Exception {  System.***out***.println("######端口号：serverPort" + serverPort + "###定时Job开始执行啦!!!!######");  **return** ***SUCCESS***;  }  } |

### 调度中心集群

调度中心支持集群部署，提升调度系统容灾和可用性。

调度中心集群部署时，几点要求和建议：

DB配置保持一致；

登陆账号配置保持一致；

群机器时钟保持一致（单机集群忽视）；

建议：推荐通过nginx为调度中心集群做负载均衡，分配域名。调度中心访问、执行器回调配置、调用API服务等操作均通过该域名进行。

Nginx配置信息

|  |
| --- |
| upstream backServer{  server 127.0.0.1:8080 weight=1;  server 127.0.0.1:8081 weight=1;  }  server {  listen 80;  server\_name www.itmayiedu.com;  #charset koi8-r;  #access\_log logs/host.access.log main;  location / {  proxy\_pass http://backServer;  index index.html index.htm;  }  error\_page 500 502 503 504 /50x.html;  location = /50x.html {  root html;  }  } |

# 分布式配置中心

## 什么是分布式配置中心

项目中配置文件比较繁杂，而且不同环境的不同配置修改相对频繁，每次发布都需要对应修改配置，如果配置出现错误，需要重新打包发布，时间成本较高，因此需要做统一的分布式注册中心，能做到自动更新配置文件信息，解决以上问题

## 常用分布式配置中心框架

Disconf（依赖于zookpeer）、Zookpeer（）、diamond、携程（阿波罗）、Redis、xxl-conf

Zookpeer保证配置文件信息实时更新 （事件通知）

大型互联网公司自己内部都有自己独立分布式配置中心

独立RPC、独立分布式各种解决方案

注册中心解决 rpc服务治理

分布式配置中心 解决分布式配置文件管理

## Apollo阿波罗简介

Apollo（阿波罗）是携程框架部门研发的分布式配置中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性，适用于微服务配置管理场景。

## Apollo阿波罗特点

用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最新的配置，并通知到应用程序。

### 统一管理不同环境、不同集群的配置

所有的配置发布都有版本概念，从而可以方便的支持配置的回滚。

### 配置修改实时生效（热发布）

用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最新的配置，并通知到应用程序

### 版本发布管理

所有的配置发布都有版本概念，从而可以方便的支持配置的回滚

### 灰度发布

支持配置的灰度发布，比如点了发布后，只对部分应用实例生效，等观察一段时间没问题后再推给所有应用实例。

### 权限管理、发布审核、操作审计

应用和配置的管理都有完善的权限管理机制，对配置的管理还分为了编辑和发布两个环节，从而减少人为的错误。

所有的操作都有审计日志，可以方便的追踪问题。

### 客户端配置信息监控

可以方便的看到配置在被哪些实例使用

### 提供Java和.Net原生客户端

提供了Java和.Net的原生客户端，方便应用集成

支持Spring Placeholder, Annotation和Spring Boot的ConfigurationProperties，方便应用使用（需要Spring 3.1.1+）

同时提供了Http接口，非Java和.Net应用也可以方便的使用

### 提供开放平台API

Apollo自身提供了比较完善的统一配置管理界面，支持多环境、多数据中心配置管理、权限、流程治理等特性。

不过Apollo出于通用性考虑，对配置的修改不会做过多限制，只要符合基本的格式就能够保存。

在我们的调研中发现，对于有些使用方，它们的配置可能会有比较复杂的格式，如xml, json，需要对格式做校验。

还有一些使用方如DAL，不仅有特定的格式，而且对输入的值也需要进行校验后方可保存，如检查数据库、用户名和密码是否匹配。

对于这类应用，Apollo支持应用方通过开放接口在Apollo进行配置的修改和发布，并且具备完善的授权和权限控制

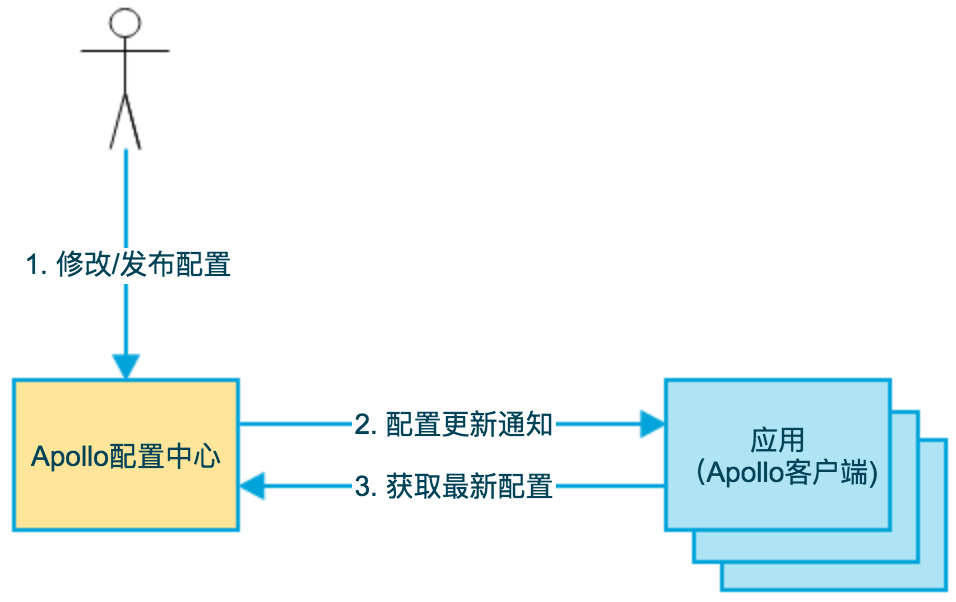
### 部署简单

配置中心作为基础服务，可用性要求非常高，这就要求Apollo对外部依赖尽可能地少

目前唯一的外部依赖是MySQL，所以部署非常简单，只要安装好Java和MySQL就可以让Apollo跑起来

Apollo还提供了打包脚本，一键就可以生成所有需要的安装包，并且支持自定义运行时参数

## Apollo整体架构原理

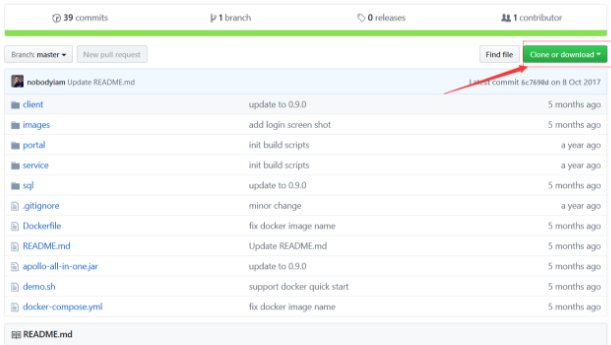


## Apollo(阿波罗)环境搭建

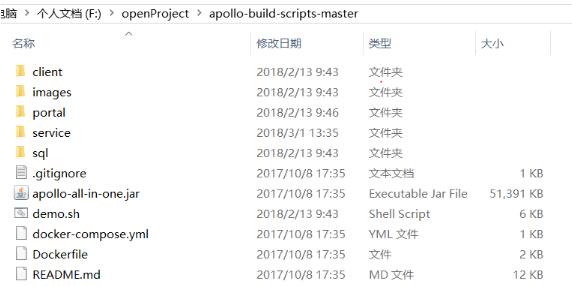
1. 下载apollo-client 打入到本地Maven仓库中
2. 部署

### 搭建Apollo配置中心

下载aploll配置中心 https://github.com/nobodyiam/apollo-build-scripts



下载好文件图



### 配置数据库参数配置

### 创建数据库

Apollo服务端共需要两个数据库：ApolloPortalDB和ApolloConfigDB，我们把数据库、表的创建和样例数据都分别准备了sql文件，只需要导入数据库即可。

**注意：如果你本地已经创建过Apollo数据库，请注意备份数据。我们准备的sql文件会清空Apollo相关的表。**

### 配置数据库连接信息

Apollo服务端需要知道如何连接到你前面创建的数据库，所以需要编辑demo.sh，修改ApolloPortalDB和ApolloConfigDB相关的数据库连接串信息。

注意：填入的用户需要具备对ApolloPortalDB和ApolloConfigDB数据的读写权限

|  |
| --- |
| #apollo config db info  apollo\_config\_db\_url=jdbc:mysql://localhost:3306/ApolloConfigDB?characterEncoding=utf8  apollo\_config\_db\_username=用户名  apollo\_config\_db\_password=密码（如果没有密码，留空即可）  # apollo portal db info  apollo\_portal\_db\_url=jdbc:mysql://localhost:3306/ApolloPortalDB?characterEncoding=utf8  apollo\_portal\_db\_username=用户名  apollo\_portal\_db\_password=密码（如果没有密码，留空即可） |

### 启动Apollo配置中心

#### 确保端口未被占用

Quick Start脚本会在本地启动3个服务，分别使用8070, 8080, 8090端口，请确保这3个端口当前没有被使用。

例如，在Linux/Mac下，可以通过如下命令检查：

#### 执行启动脚本

./demo.sh start

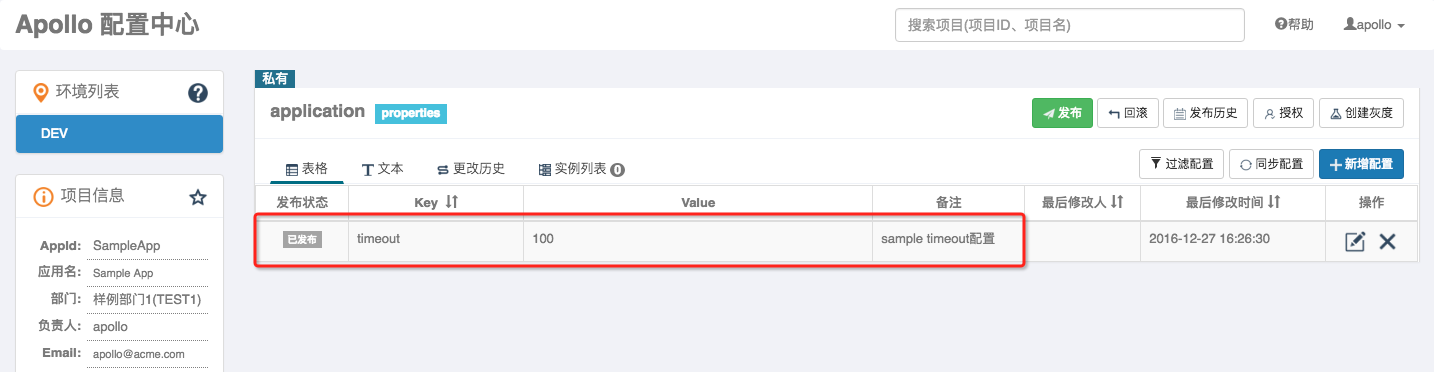
当看到如下输出后，就说明启动成功了！

### 使用Apollo配置中心

访问网址:打开http://localhost:8070



输入用户名apollo，密码admin后登录



### Apollo配置中心客户端集成

将apollo-master 打入到本地Maven仓库中

### Maven依赖信息

|  |
| --- |
| <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  <relativePath /> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  <spring-cloud.version>Finchley.RC1</spring-cloud.version>  </properties>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <!-- <exclusions> <exclusion> <groupId>com.fasterxml.jackson.core</groupId>  <artifactId>jackson-databind</artifactId> </exclusion> </exclusions> -->  </dependency>  <!-- apollo 携程apollo配置中心框架 -->  <dependency>  <groupId>com.ctrip.framework.apollo</groupId>  <artifactId>apollo-client</artifactId>  <version>1.0.0</version>  </dependency>  <dependency>  <groupId>com.ctrip.framework.apollo</groupId>  <artifactId>apollo-core</artifactId>  <version>1.0.0</version>  </dependency>  <dependency>  <groupId>org.projectlombok</groupId>  <artifactId>lombok</artifactId>  <optional>true</optional>  </dependency>  <dependency>  <groupId>com.alibaba</groupId>  <artifactId>fastjson</artifactId>  <version>1.2.3</version>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <dependencyManagement>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-dependencies</artifactId>  <version>${spring-cloud.version}</version>  <type>pom</type>  <scope>import</scope>  </dependency>  </dependencies>  </dependencyManagement>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.8</source>  <target>1.8</target>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-resources-plugin</artifactId>  <version>3.0.1</version>  <executions>  <execution>  <id>copy-conf</id>  <phase>package</phase>  <goals>  <goal>copy-resources</goal>  </goals>  <configuration>  <encoding>UTF-8</encoding>  <outputDirectory>${project.build.directory}/ext/conf</outputDirectory>  <resources>  <resource>  <directory>ext/conf</directory>  <includes>  <include>logback.xml</include>  </includes>  <filtering>true</filtering>  </resource>  </resources>  </configuration>  </execution>  </executions>  </plugin>  <plugin>  <groupId>org.jacoco</groupId>  <artifactId>jacoco-maven-plugin</artifactId>  <version>0.7.5.201505241946</version>  <executions>  <execution>  <id>default-prepare-agent</id>  <goals>  <goal>prepare-agent</goal>  </goals>  </execution>  <execution>  <id>default-prepare-agent-integration</id>  <goals>  <goal>prepare-agent-integration</goal>  </goals>  </execution>  </executions>  </plugin>  <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.3</version>  <configuration>  <imageName>hy\_uav\_gateway</imageName>  <dockerDirectory>src/main/docker</dockerDirectory>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}</directory>  <include>${project.build.finalName}.jar</include>  <include>ext/conf/logback.xml</include>  </resource>  </resources>  </configuration>  </plugin>  </plugins>  </build>  <repositories>  <repository>  <id>spring-milestones</id>  <name>Spring Milestones</name>  <url>https://repo.spring.io/milestone</url>  <snapshots>  <enabled>false</enabled>  </snapshots>  </repository>  </repositories> |

### application.yml配置文件信息

|  |
| --- |
| server:  port: 8001  spring:  application:  name: itmayiedu\_a  eureka:  client:  service-url:  defaultZone: http://192.168.212.162:8080/eureka |

### 修改环境

修改/opt/settings/server.properties（Mac/Linux）或C:\opt\settings\server.properties（Windows）文件，设置env为DEV：

env=DEV

### 创建apollo-env配置文件信息

apollo-env.properties

|  |
| --- |
| local.meta=http://192.168.212.162:8080  dev.meta=http://192.168.212.162:8080  fat.meta=${fat\_meta}  uat.meta=${uat\_meta}  lpt.meta=${lpt\_meta}  pro.meta=${pro\_meta} |

### App信息

在META-INF文件夹创建app.properties 指定appid

|  |
| --- |
| app.id=appid\_0064 |

### 项目启动

|  |
| --- |
| @Configuration  @EnableApolloConfig  @SpringBootApplication  @EnableAutoConfiguration  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

### 监听服务器通知

**使用以下方法可以监听配置文件是否被修改**

|  |
| --- |
| @Component  **public** **class** TestJavaConfigBean {  // config change listener for namespace application  @ApolloConfigChangeListener("application")  **private** **void** anotherOnChange(ConfigChangeEvent changeEvent) {  ConfigChange change = changeEvent.getChange("zhangsan\_conf");  System.***out***.println(String.*format*("Found change - key: %s, oldValue: %s," + "newValue: %s, changeType: %s",  change.getPropertyName(), change.getOldValue(), change.getNewValue(), change.getChangeType()));  }  } |

## 数据库账号密码

数据库账号连接cdb-8y8qmojr.gz.tencentcdb.com:10010

root Ww861642669+

https://github.com/ctripcorp/apollo/wiki/Apollo%E9%85%8D%E7%BD%AE%E4%B8%AD%E5%BF%83%E8%AE%BE%E8%AE%A1

## Apollo原理分析

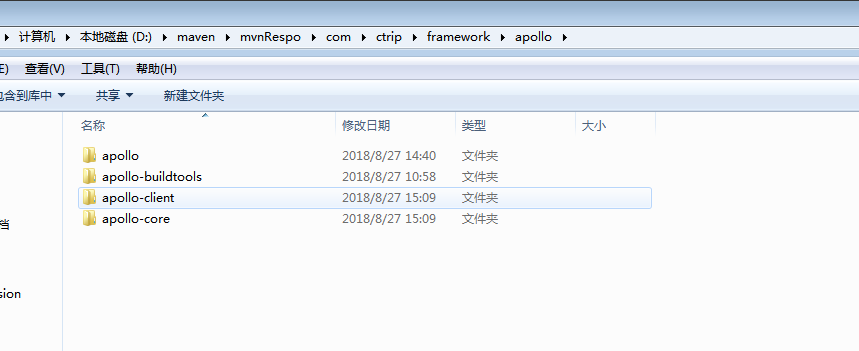
参考地址：

https://github.com/ctripcorp/apollo/wiki/Apollo%E9%85%8D%E7%BD%AE%E4%B8%AD%E5%BF%83%E8%AE%BE%E8%AE%A1

### Apollo客户端原理

## 分布式日志收集系统

## 分布式事务解决方案



# 分布式锁解决方案

分布式锁一般有三种实现方式：

1. 数据库乐观锁；
2. 基于Redis的分布式锁；
3. 基于ZooKeeper的分布式锁

## 数据库乐观锁；

## 基于Redis的分布式锁；

## 基于ZooKeeper的分布式锁

基于zookeeper瞬时有序节点实现的分布式锁，其主要逻辑如下（该图来自于IBM网站）。大致思想即为：每个客户端对某个功能加锁时，在zookeeper上的与该功能对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。同时，其可以避免服务宕机导致的锁无法释放，而产生的死锁问题。

锁安全性高，zk可持久化

## 基于Redis的分布式锁

### 使用常用命令

#### SETNX

SETNX key val当且仅当key不存在时，set一个key为val的字符串，返回1；若key存在，则什么都不做，返回0。

#### Expire

expire key timeout

为key设置一个超时时间，单位为second，超过这个时间锁会自动释放，避免死锁。

#### Delete

delete key

删除key

在使用Redis实现分布式锁的时候，主要就会使用到这三个命令。

### 实现原理

使用的是jedis来连接Redis。

#### 实现思路

1.获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。

2.获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

3.释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

#### 核心代码

##### Maven依赖信息

|  |
| --- |
| <dependency>  <groupId>redis.clients</groupId>  <artifactId>jedis</artifactId>  <version>2.9.0</version>  </dependency> |

##### LockRedis

|  |
| --- |
| **public** **class** LockRedis {  **private** JedisPool jedisPool;  **public** LockRedis(JedisPool jedisPool) {  **this**.jedisPool = jedisPool;  }  /\*\*  \* redis 上锁方法 作者：余胜军  \*  \* **@param** lockKey  \* 锁的key<br>  \* **@param** acquireTimeout  \* 在没有上锁之前,获取锁的超时时间<br>  \* **@param** timeOut  \* 上锁成功后,锁的超时时间<br>  \* **@return**  \*/  **public** String lockWithTimeout(String lockKey, Long acquireTimeout, Long timeOut) {  Jedis conn = **null**;  String retIdentifierValue = **null**;  **try** {  // 1.建立redis连接  conn = jedisPool.getResource();  // 2.随机生成一个value  String identifierValue = UUID.*randomUUID*().toString();  // 3.定义锁的名称  String lockName = "redis\_lock" + lockKey;  // 4.定义上锁成功之后,锁的超时时间  **int** expireLock = (**int**) (timeOut / 1000);  // 5.定义在没有获取锁之前,锁的超时时间  Long endTime = System.*currentTimeMillis*() + acquireTimeout;  **while** (System.*currentTimeMillis*() < endTime) {  // 6.使用setnx方法设置锁值  **if** (conn.setnx(lockName, identifierValue) == 1) {  // 7.判断返回结果如果为1,则可以成功获取锁,并且设置锁的超时时间  conn.expire(lockName, expireLock);  retIdentifierValue = identifierValue;  **return** retIdentifierValue;  }  // 8.否则情况下继续循环等待  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (conn != **null**) {  conn.close();  }  }  **return** retIdentifierValue;  }  /\*\*  \* 释放锁  \*  \* **@return**  \*/  **public** **boolean** releaseLock(String lockKey, String identifier) {  Jedis conn = **null**;  **boolean** flag = **false**;  **try** {  // 1.建立redis连接  conn = jedisPool.getResource();  // 2.定义锁的名称  String lockName = "redis\_lock" + lockKey;  // 3.如果value与redis中一直直接删除，否则等待超时  **if** (identifier.equals(conn.get(lockName))) {  conn.del(lockName);  System.***out***.println(identifier + "解锁成功......");  }  } **catch** (Exception e) {  e.printStackTrace();  } **finally** {  **if** (conn != **null**) {  conn.close();  }  }  **return** flag;  }  } |

|  |
| --- |
| **public** **class** LockService {  **private** **static** JedisPool *pool* = **null**;  **static** {  JedisPoolConfig config = **new** JedisPoolConfig();  // 设置最大连接数  config.setMaxTotal(200);  // 设置最大空闲数  config.setMaxIdle(8);  // 设置最大等待时间  config.setMaxWaitMillis(1000 \* 100);  // 在borrow一个jedis实例时，是否需要验证，若为true，则所有jedis实例均是可用的  config.setTestOnBorrow(**true**);  *pool* = **new** JedisPool(config, "39.107.69.43", 6379, 3000);  }  LockRedis lockRedis = **new** LockRedis(*pool*);  **public** **void** seckill() {  String identifier = lockRedis.lockWithTimeout("itmayiedu", 5000l, 5000l);  **if** (StringUtils.*isEmpty*(identifier)) {  // 获取锁失败  System.***out***.println(Thread.*currentThread*().getName() + ",获取锁失败，原因时间超时!!!");  **return**;  }  System.***out***.println(Thread.*currentThread*().getName() + "获取锁成功,锁id identifier:" + identifier + "，执行业务逻辑");  **try** {  Thread.*sleep*(30);  } **catch** (Exception e) {  }  // 释放锁  **boolean** releaseLock = lockRedis.releaseLock("itmayiedu", identifier);  **if** (releaseLock) {  System.***out***.println(Thread.*currentThread*().getName() + "释放锁成功,锁id identifier:" + identifier);  }  }  } |

|  |
| --- |
| **class** ThreadRedis **extends** Thread {  **private** LockService lockService;  **public** ThreadRedis(LockService lockService) {  **this**.lockService = lockService;  }  @Override  **public** **void** run() {  lockService.seckill();  }  }  **public** **class** Test001 {  **public** **static** **void** main(String[] args) {  LockService lockService = **new** LockService();  **for** (**int** i = 0; i < 50; i++) {  ThreadRedis threadRedis = **new** ThreadRedis(lockService);  threadRedis.start();  }  }  } |

在分布式环境中，对资源进行上锁有时候是很重要的，比如抢购某一资源，这时候使用分布式锁就可以很好地控制资源。

当然，在具体使用中，还需要考虑很多因素，比如超时时间的选取，获取锁时间的选取对并发量都有很大的影响，上述实现的分布式锁也只是一种简单的实现，主要是一种思想。

## 三种分布式对比

上面几种方式，哪种方式都无法做到完美。就像CAP一样，在复杂性、可靠性、性能等方面无法同时满足，所以，根据不同的应用场景选择最适合自己的才是王道。

从理解的难易程度角度（从低到高）

数据库 > 缓存 > Zookeeper

从实现的复杂性角度（从低到高）

Zookeeper >= 缓存 > 数据库

从性能角度（从高到低）

缓存 > Zookeeper >= 数据库

从可靠性角度（从高到低）

Zookeeper > 缓存 > 数据库

## Redis实现分布式锁与Zookeeper实现分布式锁区别

使用redis实现分布式锁

redis中的set nx 命令，当key不存在时，才能在redis中将key添加成功，利用该属性可以实现分布式锁，并且redis对于key有失效时间，可以控制当某个客户端加锁成功之后挂掉，导致阻塞的问题。

使用Zookeeper实现分布式锁

多个客户端在Zookeeper上创建一个相同的临时节点，因为临时节点只能允许一个客户端创建成功，那么只要任意一个客户端创建节点成功，谁就成功的获取到锁，当释放锁后，其他客户端同样道理在Zookeeper节点。