# 项目模板说明

模板地址：<http://xxxxx/chinaunicom-origin.git>

a.介绍

本日志框架并不是对日志的具体实现，而是基于门面模式的一套对日志的集成接口，支持SLF4J标准，支持logback、log4j2、、log4j、控制台、基于kibana的rabbitmq和kafka、以及自定义等多种日志组件组合等各种方式，如需要可自行扩展。

注：虽然也可使用log4j，但springboot 1.3.x后默认不再支持log4j，不建议使用，建议用springboot 2.x默认的Logback，也可使用log4j2代替。

b.日志组件的实例化

@Component  
public class AppLogHubFactory extends LogHubFactory *{*

private AppLogHubFactory*() {  
 }* @Override  
 public LogHub build*(*String className*) {*

*//* 也可自定义一种日志类型，见后文“自定义日志类型“LogHub logger = LogHub.*build(* RabbitMQLog.*build(*className*)*,

// KafkaLog.build(className),  
// ConsoleLog.build(className),  
 Slf4JLog.*build(*className*))*;  
 return logger;  
 *}  
 }*

public class Demo {

private static LogHub logger = AppLogHubFactory.*staticBuild()*;

public void log() {

Book book = new Book();

log.info("测试日志", book);

}

}

c.日志的使用

日志有多种重载方法，实体类直接传入即可，不需要单独进行序列化，以下只列出部分，其他可自行查看。

private void foo() {  
 log.i("info日志分类", "这里是待记录的日志内容");  
 log.info("info日志分类", "这里是待记录的日志内容，info与i方法两者相同");  
 log.d("debug日志分类", "这里是待记录的日志内容");  
 log.debug("debug日志分类", "这里是待记录的日志内容");  
 log.t("trace日志分类", "这里是待记录的日志内容");  
 log.trace("trace日志分类", "这里是待记录的日志内容");  
 List<Integer> list = new ArrayList<>();  
 try {  
 list.add(0);  
 list.add(1);  
 int i = list.get(0);  
 int result = i / i;  
 } catch (Exception e) {  
 log.e("error日志分类", list , e);  
 log.error("error日志分类", list , e);  
 log.w("warn日志分类", list , e);  
 log.warn("warn日志分类", list , e);  
 log.f("fatal日志分类", list , e);  
 log.fatal("fatal日志分类", list , e);  
 }  
}

d.基于rabbitmq的elk日志

application.yml配置如下：

**spring**:  
 **log**:  
 **rabbitmq**:  
 **host**: 10.4.3.75  
 **port**: 5672  
 **username**: uuuuuu  
 **password**: pppppp  
 **virtual-host**: vvvvvv  
 **routing-key**: application-log-queue  
 **queue**: application-log-queue  
 **exchange**: application-log-exchange  
 **es-index**: dev-myprojectname-log  
 **publisher-confirms**: true *# 支持发布确认* **publisher-returns**: true *# 支持发布返回*

e.基于kafka的elk日志

application.yml配置如下：

spring:  
 log:  
 kafka:  
 enable: false  
 esIndex: dev-myprojectname-log  
 topic: basic\_application\_log  
 template:  
 defaultTopic: basic\_application\_log  
 producer:  
 bootstrap-servers: 192.168.177.211:9092  
 properties:  
 linger-ms: 100  
 max-block-ms: 3000  
 request-timeout-ms: 1000  
 transaction-timeout-ms: 1000  
 compression-type: gzip  
 acks: 1

f.自定义日志的构建方式

用户也可以方便的自定义一种新的日志输出类型，只要实现LogSingle类即可，如自带的Slf4JLog类型内容定义如下：

public class Slf4JLog extends LogSingle *{*  
 private org.slf4j.Logger logger;  
  
 public Slf4JLog*(*String name*) {* super*(*name*)*;  
 logger = org.slf4j.LoggerFactory.*getLogger(*name*)*;  
 *}* public final static Slf4JLog build*() {* String name = StackTraceSugar.Previous.*getClassName()*;  
 return *build(*name*)*;  
 *}* public final static Slf4JLog build*(*Class clazz*) {* String name = clazz.toString*()*;  
 return *build(*name*)*;  
 *}* public final static Slf4JLog build*(*String name*) {* Slf4JLog slf4jLogger = new Slf4JLog*(*name*)*;  
 return slf4jLogger;  
 *}* @Override  
 protected void initialPrinter*() {* String name = this.getClass*()*.getName*()*;  
  
 printer = *(*text, level*)* -> *{* switch *(*level*)  
 {* case *VERBOSE*: *{* Global.*unsupportedOperationException(*name + ": verbose"*)*;  
 *}* break;  
 case *TRACE*: *{* logger.trace*(*text*)*;  
 *}* break;  
 case *INFORMATION*: *{* logger.info*(*text*)*;  
 *}* break;  
 case *DEBUG*: *{* logger.debug*(*text*)*;  
 *}* break;  
 case *WARNING*: *{* logger.warn*(*text*)*;  
 *}* break;  
 case *ERROR*: *{* logger.error*(*text*)*;  
 *}* break;  
 case *FATAL*: *{* Global.*unsupportedOperationException(*name + ": fatal"*)*;  
 *}* break;  
 default:*{* Global.*unsupportedOperationException(*name + ": unknown level"*)*;  
 *}* break;  
 *}  
 }*;  
 *}  
  
}*

g.aop集成

模板中已集成对接口请求的切面记录：

1. 会记录每一次http请求内容、返回内容的完整信息
2. 会自行拦截发生的异常，并记录日志，并返回统一的错误请求
3. 切面异常日志

定义了业务异常类BizException，以区别于系统异常。它包含一个错误码和错误消息，其定义如下：

public class BizException extends RuntimeException {  
 private int code;  
 private String message;  
  
 public BizException(int code, String message){  
 this.code = code;  
 this.message = message;  
 }  
}

在service业务逻辑模块中，当发生业务失败行为时，我们可以在代码中直接抛出BizException类型异常，而无需担心系统的一些调试信息泄露到外部。在抛出此类异常后，会统一被之前定义好的全局异常拦截器拦截，同时记录日志，并返回给调用方一个已定义好的错误内容，而不再需要逐个定义返回给调用方的消息体。

1. 切面出入参日志

所有接口调用的入参、出参信息均可快速记录，此部分日志无需手动记录，只需在控制器Controller类或内部方法上加@LogMarker注解即可，也可通过@LogMarkerExcept注解排除某一个接口的日志输出。

@LogMarker  
@RestController  
@RequestMapping*(*"demo"*)*public class DemoController *{* // extends InjectedBaseLogger {  
 @ApiOperation*(*value = "日志记录用例1（带参数）"*)* @ApiImplicitParams*({* @ApiImplicitParam*(*name = "summary", required = true, paramType = "path", value = "日志摘要", dataType = "string"*)*,  
 @ApiImplicitParam*(*name = "msg", required = true, paramType = "path", value = "日志内容", dataType = "string"*)  
 })* @GetMapping*(*"log1/{summary}/{msg}"*)* @LogMarker  
 public String log1*(*@PathVariable String summary, @PathVariable String msg*) {* return msg;  
 *}  
  
}*

h.消息队列篇

1. application.yml配置如下：

**spring**:  
 **rabbitmq**:  
 **main**:  
 **host**: 10.4.8.7  
 **port**: 5672  
 **username**: uuuuuu  
 **password**: pppppp  
 **binding-list**:  
 - **type**: direct  
 **routing-key**: routingKey1  
 **queue**: queue1  
 **exchange**: exchange1

- **type**: topic  
 **routing-key**: xxx  
 **queue**: xxx  
 **exchange**: xxx

**second**:

…

1. 配置绑定

@Configuration  
public class RabbitMQConfig {  
 @ConditionalOnMissingBean  
 @ConfigurationProperties(prefix = "spring.rabbitmq.main")  
 @Bean  
 public ConnectionFactory cachingConnectionFactory() {  
 CachingConnectionFactory connectionFactory = new CachingConnectionFactory();  
 return connectionFactory;  
 }  
  
 @ConfigurationProperties(prefix = "spring.rabbitmq.main")  
 @Bean("main.bindingList")  
 public RabbitMQBindingListProperty rabbitMQBindingListProperty() {  
 RabbitMQBindingListProperty rabbitMQBindingListProperty = new RabbitMQBindingListProperty();  
 return rabbitMQBindingListProperty;  
 }  
  
 @Autowired  
 @Qualifier(value = "main.bindingList")  
 public void bindingMain(RabbitMQBindingListProperty rabbitMQBindingListProperty) {  
 RabbitMQUtil.*binding*(rabbitMQBindingListProperty);  
 }  
}

以上已集成进模板，如只使有一套RabbitMQ，不再需另行修改。

1. 消息队列的使用：与springboot原生消息队列的发送、接收方式相同，发送即自定义监听器即可，此处不再举例。

@Autowired  
private AmqpTemplate rabbitTemplate;  
  
public void send(String msg) {  
 rabbitTemplate.convertAndSend("application-log-queue", msg);  
}

i.缓存篇

1. application.yml配置如下：

**spring**:  
 **redis**:  
 **host**: 10.4.9.4  
 **port**: 6379  
 **password**: Yixin4redis

b.使用：与springboot原生形式完全相同。

@Autowired  
private StringRedisTemplate stringRedisTemplate;  
  
public void write() {  
 ValueOperations<String, String> cache = stringRedisTemplate.opsForValue();  
 cache.set("cache1", "1");  
 cache.set("cache1", "1", 7200, TimeUnit.*SECONDS*);  
 String val = cache.get("cache1");  
}

1. http请求返回消息格式篇

请求生成器

@GetMapping("foo")  
public ReturnResult foo() {  
 List<String> list = null;  
 ReturnResult returnResult;

returnResult = ReturnResult.*build*();  
 returnResult = ReturnResult.*build*(BizStatusEnum.*SUCCEED*.getCode(), list);  
 returnResult = ReturnResult.*build*(BizStatusEnum.*EXCEPTION*.getCode(), list);  
 returnResult = ReturnResult.*build*(BizStatusEnum.*EXCEPTION*.getCode(), "系统出现异常，请联系管理员");  
 return returnResult;  
}

其中ReturnResult的结构如下：

public class ReturnResult<T> {

private ReturnResult() {  
 this. traceId = UUID.*randomUUID*().toString().replace("-", StringUtils.*EMPTY*);  
 this.timeStamp = System.*currentTimeMillis*();  
 }

// @ApiModelProperty("返回内容id")  
 private String traceId;  
// @ApiModelProperty("返回状态码")  
 private int statusCode;  
// @ApiModelProperty("返回消息")  
 private String message;  
// @ApiModelProperty("数据内容")  
 private T data;  
// @ApiModelProperty("时间戳")  
 private Long timeStamp;  
}

其中：

（1）traceId 此次调用请求的唯一id，返回给调用方，不需手动设置。当与请求方联调时，可用此作为某一次请求的唯一标识，方便联调期的调试，与运营期的问题定位。

（2）statusCode 此次请求的状态码，在以下枚举中定义，可自行添加

com.taoche.crm.usedcar.common.model. BizStatusEnum

（3）message此次请求的返回消息

（4）data此次请求的返回数据

（5）timeStamp此次请求的时间，不需手动设置

j.接口文档篇

接口文档使用swagger实现，修改配置信息即可实现。

1. application.yml配置如下：

不设置默认关闭，版本号可根据实际情况自定义。

swagger:  
 enable: true  
 version: 1.0  
 api-info:  
 title: 接口在线文档  
 description: 中国联通软件研究院  
 terms-of-service-url: www.chinaunicom.cn  
 license: Apache 2.0  
 licenseUrl: http://www.apache.org/licenses/LICENSE-2.0.html  
 version: 1.1  
 contact:  
 name: 联通  
 url: www.chinacom.com  
 email: chinacom@chinacom.com

如果请求或返回内容是一个实体，对实体内参数的注释方法如下，其他更详细内容可自行查阅swagger文档：

@ApiModel(value = "主品牌")  
public class CarMasterbrandDTO implements Serializable {  
 @ApiModelProperty(value = "主品牌主键")  
 private Long unifiedMasterBrandId;  
 @ApiModelProperty(value = "主品牌Id")  
 private Long masterBrandId;  
 @ApiModelProperty(value = "主品牌名称")  
 private String name;  
 @ApiModelProperty(value = "主品牌别名")  
 private String otherName;  
 @ApiModelProperty(value = "主品牌英文")  
 private String englishName;  
 @ApiModelProperty(value = "主品牌seo名称")  
 private String seoName;  
}

在前期对接接口定义时，可先利用这种方式将接口定义出来，通过swagger-ui或生成离线文档释放给调用方，之后再实现接口的具体逻辑。

优点：在定义接口的同时，通过注释的方式，即生成了接口文档，不需额外工作，

误区：接口文档可减少“接口定义方”与“接口调用方”对定义的理解误差，使双方达成统一认识，并不是为了徒增额外的工作量，且使用swagger的方式，在写注释的同时，也顺便编写出了接口文档，并不会增加额外的工作量，这也是编程标准化的要求之一。

k.定时任务篇

@Autowired  
ThreadPoolTaskScheduler threadPoolTaskScheduler;

@Override  
public void run*(*String... args*)* throws Exception *{* LauncherConfiig launcherConfiig = config.getLauncher*()*;  
 Launcher launcher = new Launcher*(*new AbstractTask*() {* @Override  
 public void run*() {* Config appConfig = ConfigService.*getAppConfig()*;  
 System.*out*.println*(*"hello"*)*;  
 *}  
 }*, "\*/10 \* \* \* \* ?", threadPoolTaskScheduler, false*)*;  
 launcher.start*()*;  
*}*

可使用 com.aio.portable.swiss.schedule.launcher.Launcher 对象，构造方法的三个参数分别为：

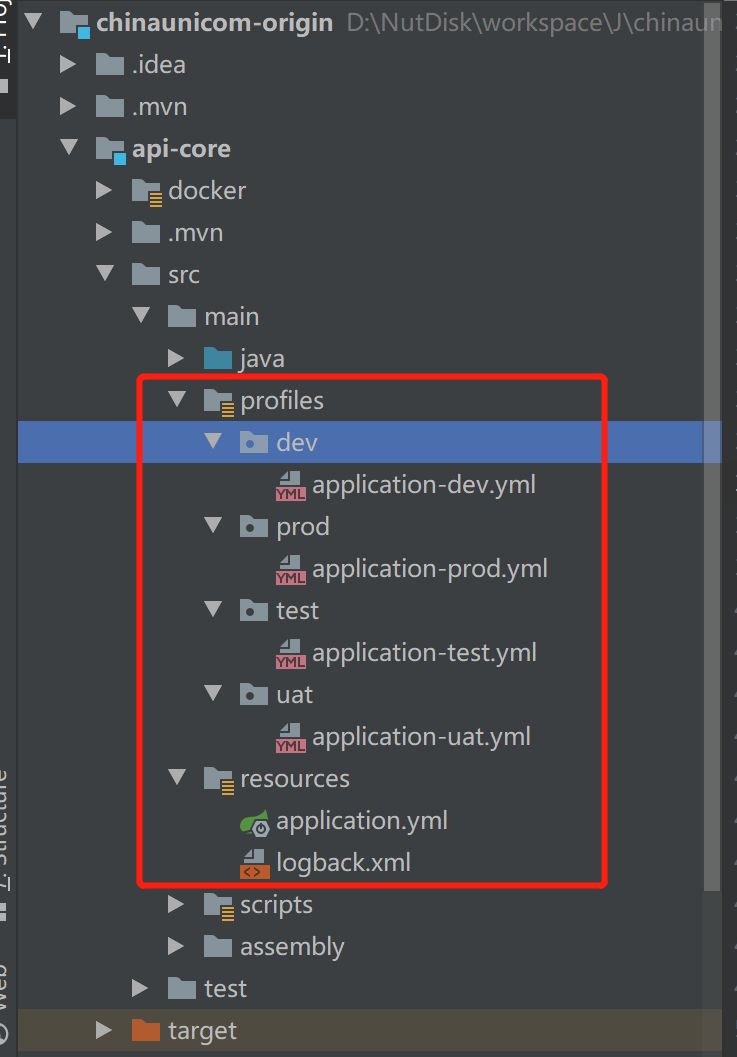
第一参数，传入自定义一个抽象类AbstractTask的实现类，内容为定时任务的具体内容，可随意构造该类型。

第二参数，传入cron表达式，定义定时器执行的周期。

第三参数，传入布尔类型，执行start()方法后，是否需要先立刻执行一次任务。传true后，立刻执行一次任务，而后按cron表达式定义周期执行；传false，仅按cron表达式定义周期执行。

l.配置篇

再来看一下配置文件，当前项目配置文件的结构如下图所示：



通用配置可在resources/application.yml中进行配置，不同环境下配置，可在profiles下相应环境目录中的application-{active}.yml中进行配置。同时支持apollo配置中心。

m.多数据源篇

请注意，这里使用最新的mysql8驱动，驱动名称使用com.mysql.cj.jdbc.Driver代替了mysql5中的com.mysql.jdbc.Driver。

第一数据源配置为mybatis，第二配置源为jpa：

spring:  
 datasource:  
 master:  
 type: com.alibaba.druid.pool.DruidDataSource  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 jdbc-url: jdbc:mysql://mecs.com:3306/tdatabase?characterEncoding=utf8&useSSL=false&serverTimezone=UTC&autoReconnect=true&allowMultiQueries=true&useCursorFetch=true  
 username: root  
 password: 1111  
 mybatis:  
 mapper-locations: classpath:config/mapper/\*.xml  
  
 slave:  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://mecs.com:3306/tdatabase?characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai&autoReconnect=true&allowMultiQueries=true&useCursorFetch=true  
 username: root  
 password: 1111  
 jpa:  
 database: mysql  
 show-sql: true  
 open-in-view: false  
 hibernate:  
 naming:  
 physical\_strategy: org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy  
 properties:  
 hibernate:  
 dialect: org.hibernate.dialect.MySQL5InnoDBDialect  
 physical\_naming\_strategy: org.springframework.boot.orm.jpa.hibernate.SpringPhysicalNamingStrategy

n.eureka篇

对eureka的配置如下：

**eureka**:  
 **instance**:  
 **prefer-ip-address**: true  
 **hostname**: ${spring.cloud.client.ipAddress}:${server.port}  
 **lease-expiration-duration-in-seconds**: 60 *# default value* **lease-renewal-interval-in-seconds**: 30 *# default value* **client**:  
 **service-url**:  
 **defaultZone**: http://10.4.3.48:7777/eureka, http://10.4.3.45:7777/eureka, http://10.4.3.39:7777/eureka  
 **register-with-eureka**: true

o.部署篇

待补充。