使用JDK17+SpringBoot3+GRPC+Mybatis-plus全面重构系统微服务架构 (已上线)
==========

> 不断学习 持续新型技术 全面比较优劣 进行替换落地使用。
> 早点睡, 没人能懂你凌晨两三点的心事 。

1. 时间线/背景/历程/收获/展望
---

* **时间线:** 2023-02-24-2023-12-21 Q1/Q2/Q3/Q4 算了一下 花了一年时间(真okr指标)
* **背景/历程:**

+ 老的系统架构框架是2014年11月搭建起来, 当时的后端研发是大厂出来的(可能当时那家公司也是这个技术栈吧),将当时的系统架构设计都是围绕他想法来推动设计的, 主要是Jersey/Guice/Ormlite, Grpc/MySQL/Redis/Quartz等等框架,现在2024年了,10年了期间,用户量在不断上升, 在性能和高可靠/高可用性达到了瓶颈, 这不OKR机会来了,重构嘛都懂。
+ 现在都2024年了,相信大家基本都没有接触过前几个框架对吧,对新人学习成本太大,相关的文档和资料都是不维护/社区也是不活跃的,解决复杂问题的速度越来越慢,确实很难用,为了探索新技术，优化服务架构，提升研发效率, 开启重构模式。
+ 改造的过程中最累的是Controller层和Dao层, 开发花费的时间最多, 当时那会AI很火，就直接把对应ORM代码复制过去,告诉AI转化为MP代码改造，加快了改造速度，当然AI也不是100%可靠的，我们还需要写很多单元测试(静态/动态测试)去比较老的结果和新的结果是否一致。接口调用会出现很多异常/离谱的情况，为了防止400的异常变成500异常,全局异常处理转化,防止出现群告警通知。
+ 单元测试路径覆盖率要达到90%以上 (当时为了提升1% 一直写各种边界的样例/删除无用代码), 服务才能上线, 使用开源框架 [github.com/jacoco/jaco...](http://cxyroad.com/"https://github.com/jacoco/jacoco") 统计单元测试各种方面的覆盖率。整个的流程是 自测->测试环境->预发环境->线上环境->回归测试。
+ ![image.png](https://p3-juejin.byteimg.com/tos-cn-i-k3u1fbpfcp/f399d0d5a9fd4180bcfef3c42fddd15c~tplv-k3u1fbpfcp-jj-mark:3024:0:0:0:q75.awebp#?w=2712&h=1080&s=191619&e=png&a=1&b=ffffff)
* **收获:**

+ 负责6个服务(每个Q两个服务)模块老技术重构, 将老技术Jersey,Guice和Ormlite使用 SpringBoot3.0和Mybatis-Plus等技术进行升级, GRpc方法单元测试代码覆盖率95%以上, 技术升级节省日常业务开发30%时间和接口整体性能RT15%左右。
+ 技术栈全面扩展开花, JDK17+ SpringBoot3.0+Mybatis-Plus+GRPC+Webclient(非阻塞调用第三方接口) Java鸡架（微服务）+ MySQL+Redis+ElasticSearch数据库+ Nacos(2.0版本)+Apollo 服务注册/配置中心+ Mycat(分表) 一致性hash+RocketMQ4.7 + AmazonSQS 消息队列+ Redission分布式锁+日志存储 log4j+slf4j / 分布式日志存储 ELK+Flume+Logstash
* **未来展望:**

+ (**重点监控监控监控** 都半年来监控还没搭建起来)服务性能全面监控 Prometheus+Grafana, 服务健康状态/服务内部性能监控/线程数量/JVM参数/OOM场景现场保留等等。
+ 现在系统项目技术都更新为比较前沿的，对于我来说，很多东西只是会使用，不明白其中的原理,多看看掘金博客/官方文档/参与开源社区。
+ 分布式事务 是否需要去拓展（服务之间调用 失败直接500异常 暂时没有任何处理）
+ elasticsearch 现在还是使用原生的, 是否考虑使用 spring-boot-starter-data-elasticsearch
+ 本地缓存和Redis缓存配合使用 是否可以封装一套注解简单使用
+ Nacos和apollo 可以整合 是否可以把apollo舍弃
+ Api网关没有这一层,客户端调用接口 从域名转发到Nginx转发到对应的Grpc服务(Nacos上拉取健康节点),进行复杂均衡策略调用对应的GRpc服务。
+ 现在上线都是直接全量发布，没有那种灰度发布/金丝雀, 蓝绿部署，滚动发布的一些上线流程。

2. 技术点的改造讲解

2.0 JDK17升级 替换 JDK8

* 为什么要进行JDK17升级, Spring Boot 3.0**需要Java 17**版本, 在这一年半载里面, 使用的比较多的17特性还是var 自动类型推断(用习惯了python/golang), 对比垃圾回收器的改变我感知不到的,可能带来的提升是JVM层面的指标，其他的17特效没用过，日常还是使用的lambda操作比较多（JDK8这些都有）。

> brew 直接安装 mac真舒服呀

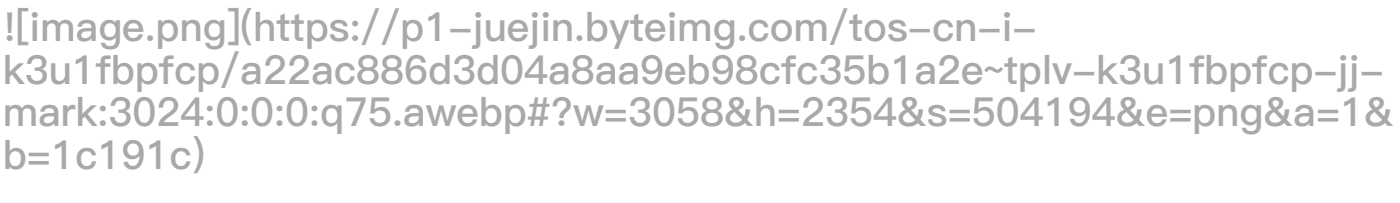
```
brew install openjdk@17
brew install openjdk@21

```

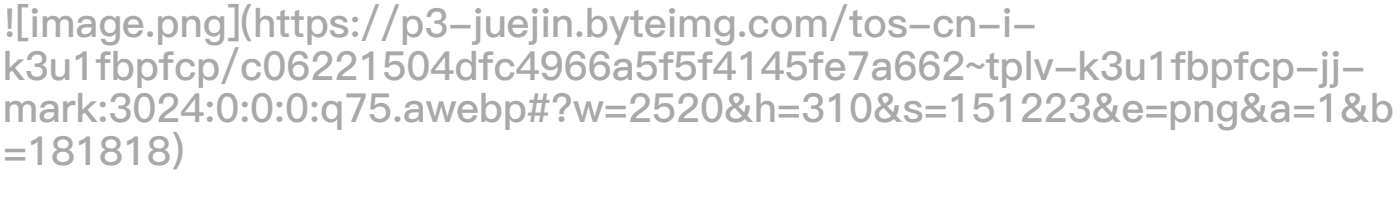
* 下载JDK17 [www.oracle.com/java/techno...](http://cxyroad.com/”https://www.oracle.com/java/technologies/downloads/#java17”)
* 修改环境变量
* IDEA修改编译版本

```
<properties>
  <java.version>17</java.version>
  <maven.compiler.source>17</maven.compiler.source>
  <maven.compiler.target>17</maven.compiler.target>
</properties>

```

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-plugin-plugin</artifactId>
  <version>3.7.0</version>
</plugin>

```

> `我这边电脑是m1 使用zulu-17的版本`





> `在终端alias 命令里面设置快捷指令 切换jdk版本`

```
jdk17='export JAVA_HOME=/Library/Java/JavaVirtualMachines/zulu-17.jdk/Contents/Home'
jdk21='export JAVA_HOME=/Users/yinpeng/javaversion/jdk-21.jdk/Contents/Home'
jdk8='export JAVA_HOME=/Library/Java/JavaVirtualMachines/zulu-8.jdk/Contents/Home'
```

2.1. SpringMVC 替换Jersey Restful框架/ jakarta 替换 javax

* 为什么要使用SpringMvc进行替换呢？主要是采用统一的Spring框架作为基层, SpringMVC作为它生态中的一员, 无缝衔接Spring IOC容器进行Bean注入,采用它改造Controller是最好的选择, 并且能够延续之前的RestFul风格GET、POST、PUT、DELETE操作，加上各种简单方便的注解使用,大大提高开发效率。

> 原生 Jersey Restful框架 废弃

* [github.com/javaee/jers...](http://cxyroad.com/"https://github.com/javaee/jersey")

```
import javax.validation.Valid;
import javax.validation.constraints.Max;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;
import javax.ws.rs.DefaultValue;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.QueryParam;
```

```
import javax.ws.rs.container.AsyncResponse;
import javax.ws.rs.container.Suspended;
import javax.ws.rs.core.Response;


@Path("/test_validation")
public class TestValidationResource extends BaseResource {


  @GET
  @Path("/type/{type: \d+}")
  public void getType(
      @Max(value = 100, message = "不能大于100")
      @PathParam("type") int type,
      @Suspended final AsyncResponse response) {
    submitTask(response, () -> Response.ok(type).build());
  }


  @POST
  @Path("/example3")
  public void postExample3(
      @NotNull Objects data,
      @Suspended final AsyncResponse response) {
    submitTask(response, () -> Response.ok("success3").build());
  }

}
```

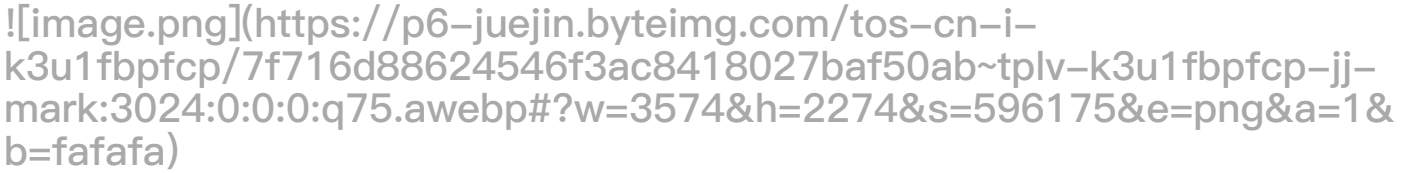

> SpringMVC RestFul 改造


```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-web</artifactId>
   <version>3.0.2</version>
</dependency>
```


```
<!-- jakarta -->
```

```
<dependency>
    <groupId>jakarta.ws.rs</groupId>
    <artifactId>jakarta.ws.rs-api</artifactId>
    <version>${jakarta.version}</version>
</dependency>

```

```
package cn.lollypop.www.lollypopv2controller.controller.smart;

import java.util.Objects;

import jakarta.validation.constraints.Max;
import jakarta.validation.constraints.NotNull;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import cn.lollypop.www.lollypopv2controller.controller.BaseController;


@Slf4j
@Validated
@RestController
@RequestMapping("/test_validation")
public class TestValidationController extends BaseController {

  @GetMapping("/type/{type}")
  public ResponseEntity<Integer> getType(
      @Max(value = 100, message = "不能大于100")
      @PathVariable("type") int type) {
    return ResponseEntity.ok(type);
  }


  @PostMapping("/example3")
  public ResponseEntity<Objects> postExample3(
      @NotNull Objects data) {
    return ResponseEntity.ok(data);
  }
}
```

```

### 2.2. MyBatis-plus 替换Ormlite框架

* Ormlite框架 废弃 其实这个框架也是通过不同的api拼接起来的sql, 和mybatis-plus很相似, 但是没有mp使用的简单, 为什么要用MP呢? 首先MP单表操作封装api直接方便使用,分页操作插件, 支持dao/mapper/service层等自定义模版，写一个MP代码自动生成器,只需要输入表名,直接产生对应的代码, 还有一个点是支持lambda操作(个人比较喜欢这种代码风格)，业务里面95%都是单表操作，剩下的多表操作都是写xml 原生sql进行执行，以上这些都是我们考虑MP的一些原因。
* [github.com/j256/ormlit...](http://cxyroad.com/”https://github.com/j256/ormlite-android”)
* [ormlite.com/](http://cxyroad.com/ ”https://ormlite.com/”)

#### 2.2.1 废弃的Ormlite db层框架



```
import java.sql.SQLException;

import com.google.inject.Inject;
import com.j256.ormlite.jdbc.JdbcConnectionSource;
import com.j256.ormlite.stmt.QueryBuilder;
import com.j256.ormlite.stmt.UpdateBuilder;
import com.j256.ormlite.stmt.Where;

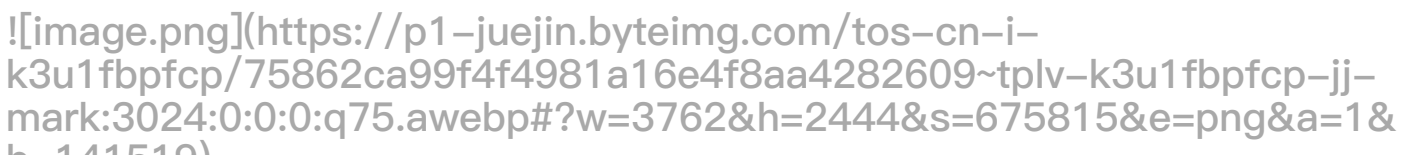
public class xxxxxxRdbmsStorageService extends
RdbmsBaseStorageService
    implements xxxxxStorageService {

  private final JdbcConnectionSource connectionSource;

  private final xxxxxDao xxxxxDao;

  @Inject
```
```

```
  public xxxxxRdbmsStorageService(JdbcConnectionSource
connectionSource) {
    try {
      this.connectionSource = connectionSource;
      this.xxxxxDao = new xxxxxDao(connectionSource,
          xxxxxModel.class);
    }
  }

  @Override
  public JdbcConnectionSource getConnectionSource() {
    return connectionSource;
  }

  @Override
  public xxxxx getxxxxxx(int userId, int app,int time){
    try {
      QueryBuilder<xxxxxModel, Integer> queryBuilder =
          xxxxxDao.queryBuilder();
      Where<SignInRecordModel, Integer> where =
          queryBuilder.where();
      where.eq(xxxxxModel.USER_ID, userId)
          .and()
          .eq(xxxxxModel.APP_FLAG, app)
          .and()
          .eq(SignInRecordModel.SIGN_IN_DATE, time);
      SignInRecordModel model = queryBuilder.queryForFirst();
      if (model != null) {
        return model.toxxxxx();
      }
      return null;
    }
  }

```



> MyBatis-plus 改造 这边只一个例子

2.2.2 **代码生成器自动生成toBean()、fromModel()、toString()方法**

> 代码生成器不生成controller包

* [github.com/baomidou/my...](http://cxyroad.com/”https://github.com/baomidou/mybatis-plus”)
* [baomidou.com/](http://cxyroad.com/ ”https://baomidou.com/”)

```
<!-- mybatis-plus -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>${mybatis-plus.version}</version>
</dependency>
<!-- mybatis-plus代码生成器 -->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>${mybatis-plus.version}</version>
</dependency>
```

```
import java.util.List;
import java.util.Objects;
import java.util.stream.Collectors;

import
com.baomidou.mybatisplus.core.conditions.query.LambdaQueryWrapper;
import
com.baomidou.mybatisplus.core.conditions.update.LambdaUpdateWrapp
er;
import com.baomidou.mybatisplus.core.toolkit.Wrappers;
import com.google.common.collect.Lists;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;


@Slf4j
@Repository
public class xxxxxStorageServiceImpl implements xxxxxStorageService {
```

```
  @Autowired
  xxxxxDao xxxxxDao;

  @Override
  public void getxxxxxx(Xxxxx xxxxx) {
    LambdaUpdateWrapper<SkinReportModel> lambdaUpdateWrapper =
        Wrappers.lambdaUpdate();
    lambdaUpdateWrapper.set(XxxxxModel::getDetail, xxxxx.getDetail())
        .set(XxxxxModel::getUserId, xxxxx.getUserId())
        .set(XxxxxModel::getCreateTime, xxxxx.getCreateTime())
        .eq(XxxxxModel::getId, xxxxx.getId());
    skinReportDao.update(lambdaUpdateWrapper);
  }
```

2.2.3 mybatis-plus 模版生成器自定义模版

> dao.java.vm

```
package ${package.Service};

import ${package.Entity}.${entity};
import ${superServiceClassPackage};

/**
 * $!{table.comment} 服务类.
 * Copyright (c) ${date},
 * All rights reserved
 * Author: ${author}
 */

public interface ${table.serviceName} extends
    ${superServiceClass}<${entity}> {

}
```

> daoImpl.java.vm

```
package ${package.ServiceImpl};

import ${package.Entity}.${entity};
import ${package.Mapper}.${table.mapperName};
#if(${table.serviceInterface})
import ${package.Service}.${table.serviceName};
#end
import ${superServiceImplClassPackage};
import com.baomidou.mybatisplus.core.toolkit.support.SFunction;
import
com.baomidou.mybatisplus.core.conditions.update.LambdaUpdateWrapp
er;
import org.springframework.stereotype.Service;

/**
 * $!{table.comment} 服务实现类.
 * Copyright (c) ${date},
 * All rights reserved
 * Author: ${author}
 */

@Service
public class ${table.serviceImplName} extends
    ${superServiceImplClass}<${table.mapperName}, ${entity}>
    #if(${table.serviceInterface})
implements ${table.serviceName}#end {

}

```


> entity.java.vm
> ..... 省略 代码太多

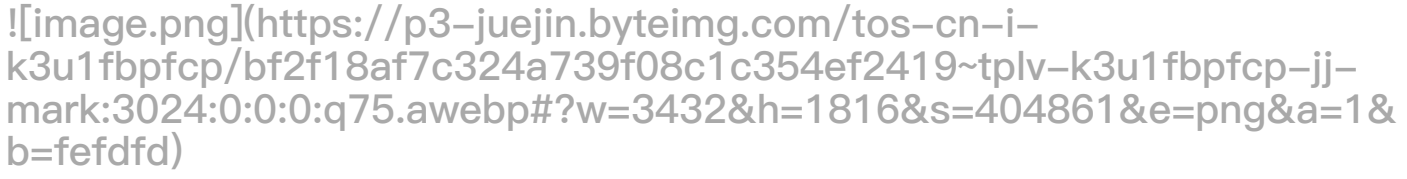
> mapper.java.vm


```
package ${package.Mapper};

import ${package.Entity}.${entity};
```

```
import ${superMapperClassPackage};
#if(${mapperAnnotationClass})
import ${mapperAnnotationClass.name};
#end

/**
 * $!{table.comment} Mapper 接口.
 * Copyright (c) ${date},
 * All rights reserved
 * Author: ${author}
 */
#if(${mapperAnnotationClass})
@${mapperAnnotationClass.simpleName}
#end
#if(${kotlin})
interface ${table.mapperName} : ${superMapperClass}<${entity}>
#else
public interface ${table.mapperName} extends
    ${superMapperClass}<${entity}> {

}
#end

```

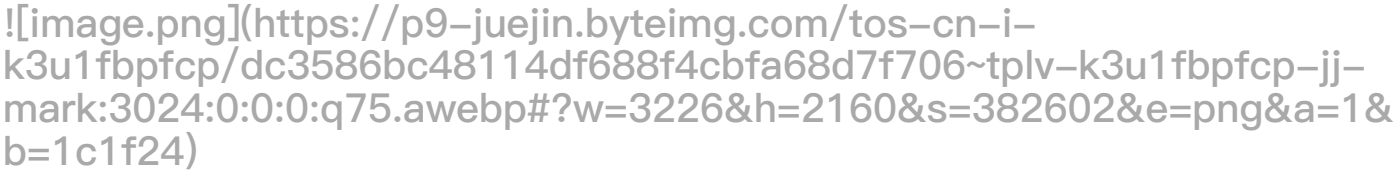
2.3. Spring 替换Guice框架 （依赖注入/IOC容器）

* Guice主要通过Java注解进行配置，如`@Inject`、`@Singleton`,是一个比较轻量级别的依赖注入框架。Guice 也是作为IOC容器的一个框架, 对比与Spring真的很难用, 学习资料少,异常问题不好排查, 替换它也是迟早的事。我们使用SpringBoot3 肯定是使用内置Spring5作为DI的框架，生态丰富，我意识到的缺点是Spring复杂的依赖管理导致服务启动时间长。
* [spring.io/projects/sp…](http://cxyroad.com/”https://spring.io/projects/spring-boot#learn”)
* 我们项目使用的是3.0.2版本 没想到迭代的这么快 都3.3.0了(最近也使用了一下对接spring-ai)



2.3.1 Guice 相关使用 服务Bean注册到容器中/使用容器中的Bean

* [github.com/google/guic…](http://cxyroad.com/”https://github.com/google/guice”)
* [github.com/google/guic…](http://cxyroad.com/”https://github.com/google/guice/wiki/Guice600”)




```
import com.google.inject.AbstractModule;
import com.google.inject.Singleton;
import com.google.inject.name.Names;

public class AuthServerModule extends AbstractModule {

  public void configure() {
    RpcBaseModule rpcBaseModule = new RpcBaseModule();
    install(rpcBaseModule);
    bind(xxxxxService.class).to(xxxxxStorageService.class);
    bind(xxxxxStorageService.class).in(Singleton.class);
  }
}
```

```
import com.google.inject.Inject;

@Inject
xxxxxService xxxxService;
```

2.3.1 使用Spring 内部的IOC容器 依赖注入

> 提供了大量的直接使用的注解进行Bean操作

* `@Component`、`@Repository`、`@Controller`和`@Service`
`@Configuration+@Bean`等注解

* `@Autowired` `@Resource`

2.4. WebClient替换JerseyClient 第三方url请求

* 在第三方url调用框架我们选择使用webClient，首先它也是Spring WebFlux生态中的一部分，我记得它有一个特性**非阻塞、响应式操作、异步操作**, 链式调用的代码风格/异常处理重试机制，采用事件处理IO的方式对性能上来说有所提高.

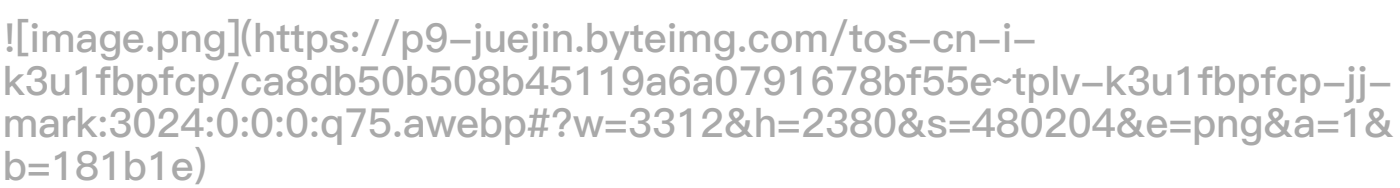
2.4.1 废弃的JerseyClient

```
import javax.ws.rs.client.Entity;
import javax.ws.rs.client.WebTarget;
import javax.ws.rs.core.MultivaluedHashMap;
import javax.ws.rs.core.Response;

import org.glassfish.jersey.apache.connector.ApacheConnectorProvider;
import org.glassfish.jersey.client.ClientConfig;
import org.glassfish.jersey.client.ClientProperties;
import org.glassfish.jersey.client.JerseyClient;
import org.glassfish.jersey.client.JerseyClientBuilder;
import
org.glassfish.jersey.client.authentication.HttpAuthenticationFeature;


public class JerseyClientUtil {

  private static final Log LOG =
LogFactory.getLog(JerseyClientUtil.class);

  private static final ClientConfig CLIENT_CONFIG =
      new ClientConfig()
          .property(ClientProperties.CONNECT_TIMEOUT, 30000)
          .property(ClientProperties.READ_TIMEOUT, 30000);

  public static WebTarget gsonTarget(String path) {
    JerseyClient client =
JerseyClientBuilder.createClient(CLIENT_CONFIG);
    return client.target(path).register(GsonJsonProvider.class);
  }

  private String getAccessToken() {
```

```
  Response response = JerseyClientUtil.gsonTarget(URL)
      .request().post(Entity.entity(tokenBody,
MediaType.APPLICATION_JSON));
  TokenBody responseBody =
      response.readEntity(TokenBody.class);
  return responseBody.getAccessToken();
}
```

2.4.2 `使用 WebClient 改造`

* [docs.spring.io/spring-fram...](http://cxyroad.com/"https://docs.spring.io/spring-framework/reference/web/webflux.html")



```
@Configuration
public class HttpInterfaceConfig {

  @Autowired
  GsonEncoder encoder;

  @Autowired
  GsonDecoder decoder;

  @SneakyThrows
  @Bean
  WebClient webClient() {
    // 设置SSL
    SslContext sslContext = SslContextBuilder.forClient()
        .trustManager(InsecureTrustManagerFactory.INSTANCE).build();

    HttpClient httpClient = HttpClient.create()
        // To configure a connection timeout
        .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, 30000)
        // To configure a read or write timeout
        .doOnConnected(conn -> conn
            .addHandlerLast(new ReadTimeoutHandler(30))
```

```
            .addHandlerLast(new WriteTimeoutHandler(30)))
        // 设置ssl
        .secure(ssl -> {
          ssl.sslContext(sslContext);
        });

    ClientHttpConnector httpConnector = new
ReactorClientHttpConnector(
        httpClient);

    return WebClient.builder()
        .clientConnector(httpConnector)
        // 使用gson实现解编码
        .codecs(clientCodecConfigurer -> {
          clientCodecConfigurer.customCodecs().register(encoder);
          clientCodecConfigurer.customCodecs().register(decoder);
        })
        .build();
  }
 }

```


```
@Autowired
WebClient webClient;

private String getAccessToken() {
  Mono<ResponseEntity<TokenBody>> responseMono = webClient.post()
      .uri(URL)
      .contentType(MediaType.APPLICATION_JSON)
      .bodyValue(tokenBody)
      .retrieve()
      .toEntity(TokenBody.class);

  TokenBody responseBody = responseMono.block().getBody();
  return responseBody.getAccessToken();

```

2.5. Spring @Async 替换自定义注解@Async

* 原来的框架并不支持异常注解在方法上面的操作, 也没有引入特别重的框架，而是自定义一个异步注解，使用线程池实现异步的功能，但是所有的异步处理都在一个线程池上面，后面采用Spring自带的异步注解/配置对应的线程池，主要在性能和监控上面有所改进。

2.5.1 自定义异步注解和实现方法

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Async {
}
```

> 异步注解拦截器 逻辑实现(线程池 submit)

```
import com.google.common.util.concurrent.ListeningExecutorService;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;


public class AsyncInterceptor implements MethodInterceptor {

  private final ListeningExecutorService executorService;

  public AsyncInterceptor(ListeningExecutorService executorService) {
    this.executorService = executorService;
  }

  @Override
  public Object invoke(MethodInvocation methodInvocation) {
    executorService.submit(() -> {
      try {
        methodInvocation.proceed();
      } catch (Throwable t) {
        logError(LOG, t);
      }
```

```
    });
    return null;
  }
}
```

> 绑定到Guice中

```
public class RpcBaseModule extends AbstractModule {
RetryInterceptor retryInterceptor = new RetryInterceptor();
bindInterceptor(Matchers.any(), Matchers.annotatedWith(Retry.class),
    retryInterceptor);
```

2.5.2 ThreadPoolConfig 线程池配置 和使用方法

```
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

import com.google.common.util.concurrent.ThreadFactoryBuilder;
import lombok.Setter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableAsync;

@Setter
@EnableAsync
@Configuration
public class ThreadPoolConfig {

  /**
   * 线程池中的核心线程数量,默认为1.
   */
  private int corePoolSize = 6;
  /**
   * 线程池中的最大线程数量.
   */
```

```
  private int maxPoolSize = 12;
  /**
   * 线程池中允许线程的空闲时间,默认为 60s.
   */
  private int keepAliveTime = ((int) TimeUnit.SECONDS.toSeconds(60));
  /**
   * 线程池中的队列最大数量.
   */
  private int queueCapacity = 1024;

  /**
   * 线程的名称前缀.
   */
  private static final String THREAD_PREFIX = "business-thread-call-
runner-%d";

  private static final String THREAD_PREFIX_V2 =
      "business-v2-thread-call-runner-%d";

  private static final String ASYNC_THREAD_PREFIX =
      "business-async-thread-call-runner-%d";

  private static final String THREAD_PREFIX_FUTURE =
      "chatGpt-future-thread-call-runner-%d";

  @Bean("chatGptThreadPool")
  public ThreadPoolExecutor threadPoolExecutor() {
    ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()
        .setNameFormat(THREAD_PREFIX).build();
    return new ThreadPoolExecutor(
        corePoolSize, maxPoolSize,
        keepAliveTime, TimeUnit.SECONDS,
        new LinkedBlockingDeque<>(queueCapacity),
        namedThreadFactory,
        new ThreadPoolExecutor.CallerRunsPolicy());
  }
}
```

```
@Autowired
@Qualifier("chatGptThreadPool")
ThreadPoolExecutor chatGptThreadPool;
```

2.6. Spring @Retryable 替换自定义注解@Retry

* 原来的重试方式也是使用自定义注解，异常重试直接进行次数重试操作,虽然能够满足业务需求,但是对比与Spring的@Retryable注解提供了更多的功能特性，比如设置最大重试次数/重试等待策略(重试等待时间/倍率)/抛出指定异常才会重试等等。

2.6.1 自定义注解@Retry

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Retry {
  int value();
}
```

> 重试注解拦截器 逻辑实现 其实就是catch异常再重试

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class RetryInterceptor implements MethodInterceptor {

  @Override
  public Object invoke(MethodInvocation methodInvocation) throws
Throwable {
    int retryTimes = methodInvocation.getMethod()
        .getAnnotation(Retry.class).value();
    return invoke(methodInvocation, retryTimes);
  }

  private Object invoke(MethodInvocation methodInvocation, int
```

```
retryTimes)
      throws Throwable {
    try {
      return methodInvocation.proceed();
    } catch (Throwable e) {
      if (--retryTimes > 0) {
        return invoke(methodInvocation, retryTimes);
      }
      throw e;
    }
  }
}
```

> 注册到Guice中

```
public class RpcBaseModule extends AbstractModule {
    AsyncInterceptor asyncInterceptor = new
AsyncInterceptor(executorService);
    bindInterceptor(Matchers.any(), Matchers.annotatedWith(Async.class),
        asyncInterceptor);
```

2.6.2 `Spring @Retryable`

```
<!-- retry -->
<dependency>
    <groupId>org.springframework.retry</groupId>
    <artifactId>spring-retry</artifactId>
    <version>2.0.0</version>
</dependency>
```

> 启动类上加上开启重试注解

```
@EnableRetry
```

```
@Retryable(backoff = @Backoff(delay = 2000, multiplier = 1.5))
```

2.7. RedisClient 替换 JedisClient

* Redis使用Spring扩展Redis的starter组件, 第一个是`RedisTemplate` 提供了很多数据结构API操作入口，比如字符串String、哈希hash、列表list、排序集合zset等，以及对事务Transactional、管道pipline和lua脚本。第二个的话是注入和依赖管理对于Spring项目更加丝滑。至于底层Lettuce的实现原理不是特别了解(非阻塞IO/响应式/线程安全/故障恢复？有时间学习一下),jedis太原生了, 不是Jedis不行了，而是RedisTemplate更有性价比。

2.7.1 JedisClient 初始化 废弃

```
import com.google.inject.AbstractModule;
import com.google.inject.Module;
import com.google.inject.Singleton;
import com.google.inject.matcher.Matchers;
import com.google.inject.name.Names;

public class RpcBaseModule extends AbstractModule {
    bind(JedisClient.class).in(Singleton.class);
```

```
import com.google.inject.Inject;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import redis.clients.jedis.BitOP;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.ScanParams;
```

```
import redis.clients.jedis.ScanResult;
import redis.clients.jedis.params.SetParams;

public class JedisClient {

  private static final Log LOG = LogFactory.getLog(JedisClient.class);

  private JedisPool jedisPool;

  private static final String LOCK_SUCCESS = "OK";

  private static final Long RELEASE_SUCCESS = 1L;

  @Inject
  JedisClient() {
    initPool();
  }

  private void initPool() {
    try {
      JedisPoolConfig jedisPoolConfig = new JedisPoolConfig();
      jedisPoolConfig.setMaxTotal(200);
      jedisPoolConfig.setMaxIdle(20);
      jedisPoolConfig.setMinIdle(5);
      jedisPoolConfig.setTestOnBorrow(true); //取出时检查连接有效性
      jedisPoolConfig.setTestWhileIdle(true); //空闲时检查连接有效性
      jedisPoolConfig.setMaxWaitMillis(30000); //最大等待时间 ms
      String jedisHost = ConfigUtil.getProperty(
          Constants.JEDIS_CONNECTION_HOST);
      int jedisPort = ConfigUtil.getIntProperty(
          Constants.JEDIS_CONNECTION_PORT);
      jedisPool = new JedisPool(jedisPoolConfig, jedisHost, jedisPort);
      LOG.info("===jedis pool初始化成功===");
    } catch (Exception e) {
      LOG.error("===jedis pool初始化失败===");
      if (jedisPool != null) {
        jedisPool.close();
      }
    }
  }

  public Jedis getJedis() {
    if (jedisPool == null) {
      synchronized (JedisClient.class) {
        if (jedisPool == null) {
          initPool();
        }
      }
```

```
    }
    return jedisPool.getResource();
  }
  ..........
}

```

2.7.2 RedisClient 改造

> 增加Redis Client，完成各常用数据结构的基本操作方法

* [redis.io/docs/latest...](http://cxyroad.com/”https://redis.io/docs/latest/develop/connect/clients/”)

```
<!-- redis -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
<!-- redis连接池 -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-pool2</artifactId>
</dependency>

```

```
# redis配置
data:
  redis:
    host: ${jedis.connection.host:localhost}
    port: ${jedis.connection.port:6379}
    password:
    database: 0
    timeout: 5000ms
    lettuce:
      pool:
        # 连接池中的最大空闲连接
        max-idle: 16
```

```
        # 连接池最大连接数(使用负值表示没有限制)
        max-active: 32
        # 连接池中的最小空闲连接
        min-idle: 8
        # 连接池最大阻塞等待时间(使用负值表示没有限制)
        max-wait: 6000ms
        # 每100s运行一次空闲连接回收器（独立线程）
        time-between-eviction-runs: 100000ms
      # 关闭超时时间, 默认100ms
      shutdown-timeout: 100ms

```

```
import com.fasterxml.jackson.annotation.JsonAutoDetect;
import com.fasterxml.jackson.annotation.JsonTypeInfo;
import com.fasterxml.jackson.annotation.PropertyAccessor;
import com.fasterxml.jackson.databind.ObjectMapper;
import
com.fasterxml.jackson.databind.jsontype.impl.LaissezFaireSubTypeValida
tor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import
org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import
org.springframework.data.redis.serializer.GenericToStringSerializer;
import
org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;
import org.springframework.data.redis.serializer.StringRedisSerializer;


public class BaseRedisTemplateConfig {

  @Autowired
  RedisTemplate redisTemplate;

  @Bean(”redisTemplate”)
  public RedisTemplate<String, Object> redisTemplate(
      RedisConnectionFactory factory) {
    RedisTemplate<String, Object> redisTemplate = new
RedisTemplate<>();
    redisTemplate.setConnectionFactory(factory);
    Jackson2JsonRedisSerializer jackson2JsonRedisSerializer =
        new Jackson2JsonRedisSerializer(Object.class);
    ObjectMapper objectMapper = new ObjectMapper();
```

```
    objectMapper.setVisibility(PropertyAccessor.ALL,
        JsonAutoDetect.Visibility.ANY);
objectMapper.activateDefaultTyping(LaissezFaireSubTypeValidator.instance,
        ObjectMapper.DefaultTyping.NON_FINAL,
JsonTypeInfo.As.PROPERTY);
    jackson2JsonRedisSerializer.setObjectMapper(objectMapper);
    StringRedisSerializer stringRedisSerializer = new
StringRedisSerializer();
    redisTemplate.setKeySerializer(stringRedisSerializer);
    // redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);
    GenericToStringSerializer genericToStringSerializer =
        new GenericToStringSerializer(Object.class);
    redisTemplate.setValueSerializer(genericToStringSerializer);
    redisTemplate.setHashKeySerializer(stringRedisSerializer);
    redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);
    redisTemplate.setDefaultSerializer(stringRedisSerializer);
    redisTemplate.afterPropertiesSet();
    return redisTemplate;
  }

  @Bean
  public RedisClient getRedisClient() {
    return new RedisClient(redisTemplate);
  }

}
```

```
import org.springframework.data.redis.core.RedisTemplate;

public class RedisClient {

  private RedisTemplate redisTemplate;

  public RedisClient(RedisTemplate redisTemplate) {
    this.redisTemplate = redisTemplate;
  }

  /**
   * 给一个指定的 key 值附加过期时间.
   *
   * @param key
   * @param time
   * @return
```

```
     */
    public boolean setExpire(String key, long time) {
        return redisTemplate.expire(key, time, TimeUnit.SECONDS);
    }
    ..........

}

```

2.8. Redission 替换jedis实现的setnx 分布式锁

* 对于Redis分布式锁我印象深刻的还是Redlock作者antirez和一个分布式专家**Martin**两个人的争论事件（神仙打架 我们看戏）。
* 直接使用Redission提供的分布式锁机制，我们这边redis是单机版配置没有采用集群,我记得Redission操作都是原子性的(基于lua脚本),锁自动续期操作(看门狗机制)

2.8.1 jedis实现的setnx 分布式锁 废弃

```
/**
 * 尝试获取分布式锁.
 *
 * @param lockKey       锁的key
 * @param value         值
 * @param milliseconds 毫秒过期时间
 * @return 是否获取成功
 */
public Boolean tryGetDistributedLock(String lockKey, String value,
    int milliseconds) throws Exception {
    Jedis jedis = getJedis();
    if (jedis == null) {
        throw new Exception("get jedis resource failed.");
    }

    boolean returnResult = Boolean.FALSE;
    try {
        SetParams setParams = new SetParams();
        setParams.nx();
        setParams.px(milliseconds);
        String result = jedis.set(lockKey, value, setParams);
        if (LOCK_SUCCESS.equals(result)) {
            returnResult = Boolean.TRUE;
```

```
    }
  } catch (Exception e) {
    LOG.error("jedis tryGetDistributedLock lockKey " + lockKey +
        " error: " + e.getMessage());
  } finally {
    returnJedisResource(jedis);
  }
  return returnResult;
}

/**
 * 释放分布式锁.
 *
 * @param lockKey 锁的key.
 * @param value   值
 * @return 是否释放成功
 */
public Boolean releaseDistributedLock(String lockKey, String value)
    throws Exception {
  Jedis jedis = getJedis();
  if (jedis == null) {
    throw new Exception("get jedis resource failed.");
  }

  boolean returnResult = Boolean.FALSE;
  try {
    String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return " +
        "redis.call('del', KEYS[1]) else return 0 end";
    Object result = jedis.eval(script, Collections.singletonList(lockKey),
        Collections.singletonList(value));
    if (RELEASE_SUCCESS.equals(result)) {
      returnResult = Boolean.TRUE;
    }
  } catch (Exception e) {
    LOG.error("jedis releaseDistributedLock lockKey " + lockKey +
        " error: " + e.getMessage());
  } finally {
    returnJedisResource(jedis);
  }
  return returnResult;
}

```

2.8.2 Redission 实现分布式锁 改造

* [github.com/redisson/re...](http://cxyroad.com/

”https://github.com/redisson/redisson”)
* [redisson.pro/](http://cxyroad.com/ ”https://redisson.pro/”)

```
<!-- redisson -->
<dependency>
    <groupId>org.redisson</groupId>
    <artifactId>redisson</artifactId>
    <version>3.21.3</version>
</dependency>
```

```
import org.redisson.Redisson;
import org.redisson.api.RedissonClient;
import org.redisson.config.Config;
import org.redisson.config.SingleServerConfig;
import org.redisson.config.TransportMode;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;


@Configuration
public class RedissonConfig {

  @Value(”${redisson.address}”)
  private String redissonAddress;

  @Value(”${redisson.password}”)
  private String redissonPassword;

  @Value(”${redisson.database}”)
  private Integer redissonDatabase;

  @Bean
  public RedissonClient redissonClient() {
    Config config = new Config();
    config.setTransportMode(TransportMode.NIO);
    SingleServerConfig singleServerConfig = config.useSingleServer();
    // 可以用”rediss://”来启用SSL连接
    singleServerConfig.setAddress(redissonAddress);
    // 密码没设置会报错
    // singleServerConfig.setPassword(redissonPassword);
```

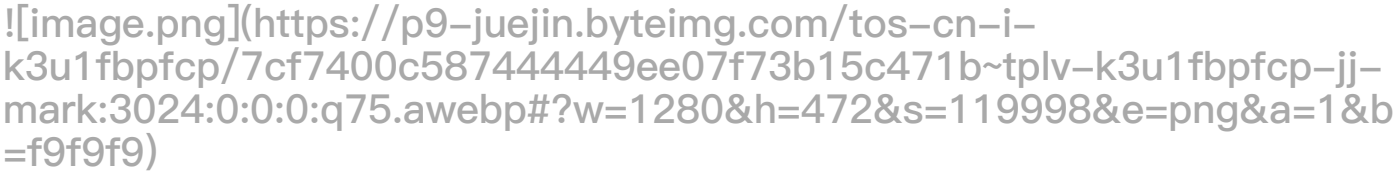
```
    singleServerConfig.setDatabase(redissonDatabase);
    return Redisson.create(config);
  }
}

```

```
String key =”lock”;
RLock rLock = redissonClient.getLock(key);
Boolean resultLock = rLock.tryLock();
if (resultLock) {
  try {
    // 逻辑处理

  } finally {
    rLock.unlock();
  }

```

2.9. Spring 内置@Scheduled 替换Quartz定时任务 （后面有okr指标 又改成xxl-job了）

> 废弃的Quartz定时任务 我就不讲了 没意思
> Spring 内置@Scheduled

* 定时任务表达式问题，老框架Quartz和新框架Spring 星期的表达不一致 一个从1开始/一个从0开始-0630




```
import java.util.UUID;

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.apache.logging.log4j.ThreadContext;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnExpressi
on;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;


@Slf4j
@Component
@ConditionalOnExpression("${scheduling.enabled} and ${hello.switch}")
@RequiredArgsConstructor(onConstructor = @__(@Autowired))
public class HelloTaskJob {

  private final XXXXXClientV2 xxxxxClient;

  @Scheduled(cron = "${hello.time}")
  public void adhdHelloTask() {
    try {
      String requestId = UUID.randomUUID().toString().substring(0, 8);
      ThreadContext.put("requestId", requestId);
      log.info("HelloTaskJob requestId {}", requestId);
      //逻辑处理.....
    } catch (Exception e) {
      log.error("send HelloTaskJob fail ", e);
    } finally {
      log.info("clear requestId HelloTaskJob");
      ThreadContext.remove("requestId");
    }
  }

}

```

2.11·0. lombok 替换 原生get/set/toString()

* 原来的get/set/toString 方法都是依赖于IDEA自动生成的, 简化代码操作和开发工作量，考虑使用lombok全面替换。


```
<!-- lombok -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
```

```
    <version>1.18.24</version>
</dependency>

```

```
import java.io.Serializable;

import com.baomidou.mybatisplus.annotation.IdType;
import com.baomidou.mybatisplus.annotation.TableId;
import com.baomidou.mybatisplus.annotation.TableName;
import lombok.Data;


@Data
@TableName("analysis_text")
public class xxxxTextModel implements Serializable {

  private static final long serialVersionUID = 1L;

  /**
   * id.
   */
  @TableId(value = "id", type = IdType.AUTO)
  private int id;

}

```

2.12. 过滤器`org.springframework.web.filter`替换`javax.ws.rs.container.ContainerRequestFilter`

* 都使用Spring生态了, 对于过滤器的操作也需要重构。

2.12.1 过滤器`javax.ws.rs.container.ContainerRequestFilter` 废弃

```
import javax.annotation.Priority;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;

@PreMatching
```

```
@Priority(Priorities.AUTHENTICATION)
public class AuthenticationFilter implements ContainerRequestFilter {
}
```

2.12.2 过滤器`org.springframework.web.filter` 改造

* 自定义过滤器, Spring 提供了一个方便的基类 `OncePerRequestFilter`，确保过滤器只执行一次：

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.filter.OncePerRequestFilter;

@Slf4j
@Component
@Order(Priorities.AUTHENTICATION)
public class AuthFilter extends OncePerRequestFilter {

}
```

2.13. GlobalExceptionHandler 全局异常日志处理

* 主要是拦截异常, 进行统一处理, 比如在接口层面参数异常/参数类型不匹配/body参数为空等等.

```
import org.springframework.validation.BindException;
import
org.springframework.web.HttpRequestMethodNotSupportedException;
import
org.springframework.web.bind.MethodArgumentNotValidException;
import
org.springframework.web.bind.MissingServletRequestParameterExceptio
n;
```

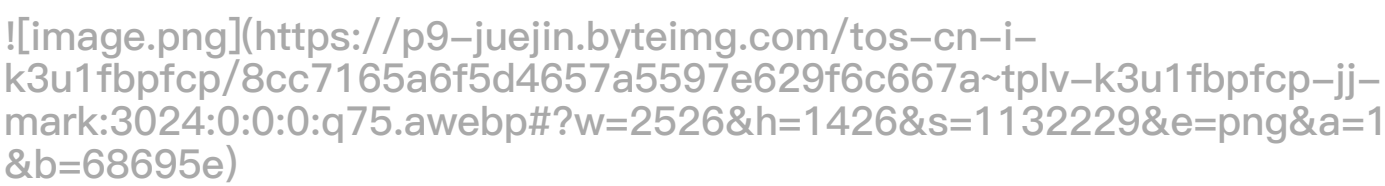
```
import org.springframework.web.bind.annotation.RestControllerAdvice;
import
org.springframework.web.method.annotation.MethodArgumentTypeMism
atchException;


@RestControllerAdvice
@Slf4j
public class GlobalExceptionHandler {

    /**
     * 处理参数不能null异常(该异常类继承的是RuntimeException).
     *
     * @param request   请求
     * @param exception 异常
     * @return
     */
    @ExceptionHandler(value = ParamIsNullException.class)
    public ResponseEntity<LollypopError> paramIsNullExceptionHandler(
        HttpServletRequest request, ParamIsNullException exception) {
      log.error("paramIsNullExceptionHandler requestUrl:{},
errorMessage:{}",
          request.getRequestURI(), exception.getMessage());
      return
ResponseEntity.status(exception.getErrorCode().getStatusCode())
          .body(new LollypopError(exception.getErrorCode().getErrorCode(),
              exception.getMessage()));
    }
    ..............
}

```


2.14. 还有一些我记得技术点的改进（不详细介绍了）

* Nacos(`spring-cloud-starter-alibaba-nacos-discovery`) 替换 MySQL存储服务器节点信息和健康状态
* SpringBoot整合GRPC 配置 (`grpc-server-spring-boot-starter` `grpc-client-spring-boot-starter`)
(grpc原来就有的)
* xxxl-job (`xxl-job-core`) 替换Spring内置@Scheduled (今年Q1季度换的同事写的)
* 302重定向等状态码 替换
* resource下资源读取的方式 替换
* 每个服务器JVM 堆大小参数调整
* sktwalking 链路追踪接入 (用的很少)

* Controller层/webclient层编解码 Gson序列化策略 GsonHttpMessageConverter (同事写的)
* GRPC context上下文传递requestId 拦截器
* .....

3. `下班娱乐局`




4. 参考
---

* [spring.io/projects/sp...](http://cxyroad.com/"https://spring.io/projects/spring-boot#overview")
* [reflectoring.io/java-releas...](http://cxyroad.com/"https://reflectoring.io/java-release-notes/")
* [# 从 Java8 到 Java11 再到 Java17 的新特性(1)](http://cxyroad.com/"https://zguishen.com/posts/36db0091.html")
* [@Component , @Repository , @ Controller , @Service , @Configration注解的作用和联系](http://cxyroad.com/"https://blog.csdn.net/Li2992973708/article/details/133586341")
* [# Java 17 新特性概述](http://cxyroad.com/"https://pdai.tech/md/java/java8up/java17.html")
* [【Java】Java 17 新特性概览](http://cxyroad.com/"https://blog.csdn.net/weixin_42201180/article/details/133957082")
* [# Spring Boot 3.0正式发布及新特性解读](http://cxyroad.com/"https://blog.csdn.net/k316378085/article/details/134307395")
* [# 网络请求客户端WebClient的使用](http://cxyroad.com/"https://blog.csdn.net/qq_74748121/article/details/139305215")
* [# 官方推荐的 WebClient](http://cxyroad.com/"https://zhuanlan.zhihu.com/p/659885945")
* [# Spring定时任务@Scheduled的使用](http://cxyroad.com/"https://blog.csdn.net/W1324926684/article/details/130983571")

原文链接: https://juejin.cn/post/7374683456728760374