TD-BASE 1.1 使用文档

- 版本: 1.1-beta
- 最后修订时间: 2017-11-28
- Gitlab地址: http://git.tuandai888.com/architecture/td-base

1. 项目介绍

1.1. Maven模块划分

1. apiutils
   主要包含异常类, 异常对象, 以及一些常量.
   业务项目的API模块会依赖于这个模块.
2. utils
   主要包含一些工具类.
   业务项目的CORE模块可能会依赖于这个模块, 或者直接依赖于下面的COMMON模块.
3. common
   包含Spring Cloud大部分组件的依赖.
   包含Spring Boot与Spring Cloud默认的配置类以及工具类.
   业务项目的CORE模块可能会依赖于这个模块.

1.2. 配置类

1. BaseConfiguration
   基础配置类, 基本所有项目都会引用.
2. ServiceClientConfiguration
   服务调用客户端配置类, 如果需要调用其他服务, 需要引入该类.
3. WebApplication
   Spring MVC配置类, 如果提供服务调用, 需要引入该类.
4. MultipleDataSourceConfiguration
   多数据源配置类, 如果项目中有多个数据源, 需要引入该类,
   并且配置MultipleDataSource Bean, 具体步骤请查看2.4的多数据源使用.

2. 基本用法

2.1. DTO的定义

1. DTO所有需要转为JSON的属性都需要定义setter/getter方法
2. DTO如果定义了构造方法, 需要保证有一个空构造方法

2.2. 常用工具类

1. JsonUtils
   JSON工具类.
2. ApplicationContextHolder
   可以静态获取Spring Context和Spring Bean. 用法:
   `ApplicationContextHolder.context.getBean(...)`
3. ApplicationConstant
   可以获取一些项目参数, 例如当前服务名, 服务编号, 当前运行环境(dev/test/prev/prod)等. 使用的时候直接注入即可,
   例如: `@Autowired ApplicationConstant applicationConstant;`

2.3. 异常处理

2.3.1. 异常状态码定义与使用

参照CommonErrorCode, 实现ErrorCode接口, 例如

```
public enum CgErrorCode implements ErrorCode {

    ERR_2(500, "已处理过"),
    ERR_3(503, "网络异常"),
    ERR_4(400, "验签失败"),
    ERR_5(500, "处理异常");
}
```

service里抛出AppBusinessException: `throw new AppBusinessException(CgErrorCode.ERR_4);`

服务器将会返回:

```
{
code:"ERR_4"
message:"验签失败"
requestUri:""
}
```

2.3.2. 服务端抛出异常

在业务代码中可以手动抛出AppBusinessException, 该异常会直接抛到Controller的异常处理器中, 由异常处理器返回统一的异常JSON对象到客户端.

2.4. 多数据源的使用

2.4.1. 数据源定义

2.4.1.1 创建一个常量类, 定义自己系统需要使用到的数据源的简称和配置前缀, 例如:

```
public interface DataSourceNames {

    /**
     * 消息写(MySQL)
     */
    String WDS_MESSAGE = "db.ds.write.message";

    /**
     * 通用读(sql server)
     */
    String RDS_COMMON = "db.ds.read.common";

}
```

2.4.1.2 在配置文件中加入数据源相关配置(略)

2.4.1.3 在SpringBoot启动类引入`MultipleDataSourceConfiguration`, 例如:

```
@Import({BaseConfiguration.class, ServiceClientConfiguration.class, WebApplicat
ion.class,
        ConfigTestConfiguration.class, MultipleDataSourceConfiguration.class})
public class ConfigTestApplication {

    public static void main(String[] args) {
        SpringApplication.run(ConfigTestApplication.class, args);
    }

}
```

2.4.1.4 创建`MultipleDataSource`这个Bean, 传入定义的数据源, 注意传入的第一个数据源为默认数据源, 例如:

```
    @Bean
    public MultipleDataSource multipleDataSource() {
        return new MultipleDataSource(Lists.newArrayList(RDS_COMMON,
WDS_MESSAGE));
    }
```

2.4.2. 数据源使用

2.4.2.1 使用`@DataSource`注解在方法级别切换数据源(推荐)

用法:

```
@DataSource(DataSourceNames.WDS_INFO)
```

```
public int getCountUnReadBycid(MessageInfoDTO messageInfoDTO){
    return   messageDAO.getCountByUnread(messageInfoDTO);
}
```

1. `@DataSource`配置的数据源有效范围在本方法内, 就是本方法内无论调用多少次DAO查询都会使用该数据源.
2. `@DataSource`注解只能加在Service方法上, 加在DAO上是无效的.
3. 在Service方法内调用本实例的其他方法, 将导致其他方法的`@DataSource`配置失效. 例如:

```
@DataSource(DataSourceNames.WDS_INFO)
public void method1(MessageInfoDTO messageInfoDTO){
    //此处使用数据源WDS_INFO
    messageDAO.getCountByUnread(messageInfoDTO);
    //调用method2不会使用数据源RDS_COMMON
    method2();
}

@DataSource(DataSourceNames.RDS_COMMON)
public void method2(){
    readDAO.readSomething();
}
```

因为在method1方法内部调用method2, 导致AOP失效, `@DataSource`注解不会被读取, 所以method2执行的时候依然使用WDS_INFO数据源.
这种情况需要用到下面的手动切换数据源.

2.4.2.2. 手动切换数据源(不推荐)

手动切换数据源需要考虑数据源的作用范围以及回收, 如果使用不慎很容易出现BUG, 所以推荐使用DataSource注解的方式切换数据源

用法:

```
try {
    DataSourceHolder.putDataSource(DataSourceNames.RDS_COMMON);
    //do your work
} finally {
    DataSourceHolder.removeDataSource(DataSourceNames.RDS_COMMON);
}
```

1. 调用`DataSourceHolder.putDataSource`切换到你想设置的数据源
2. 此时可以发起DAO调用, 无论调用多少次, 都会使用你手动设置的数据源.

3. 如果发起对其他Service的调用, 而且Service方法上有`@DataSource`注解, 则在Service方法范围内会使用该注解配置的数据源, 调用完成会退回成你手动设置的数据源.
4. 手动设置数据源的话, 大部分情况下需要在finally块中删除你设置的数据源, 以免对后面代码的数据源产生影响.

2.4.2.3. 多数据源对事务的影响

1. 单数据源单事务
   `@DataSource`注解可以和`@Transactional`注解配合使用.
2. 多数据源多事务
   如果Service方法A已经开启了事务, 需要在A方法中调用其他Service类的B方法, 而且B方法使用的是不同的数据源. 需要配置B方法的事务传播特性为`REQUIRES_NEW`(开启新事务).

```
@Service
class ClassA {

    @Autowired
    ClassB b;

    @Transactional
    @DataSource(DataSourceNames.D1)
    void A() {
        // some work.
        //invoke B method
        b.B();
    }
}

@Service
class ClassB {

    @Transactional(propagation = Propagation.REQUIRES_NEW)
    @DataSource(DataSourceNames.D2)
    void B() {
        // some work.
    }
}
```

上面代码, A方法在执行的时候会使用D1数据源, A方法在调用B方法的时候, A方法的事务会暂时挂起, B方法中会使用D2数据源并且开启一个新的事务, 并且B方法的事务范围是独立的, 即无论B方法提交还是回滚都不会影响A方法.
B方法执行完毕并且B的事务提交后, A方法会继续运行.
如果B方法没有加`propagation = Propagation.REQUIRES_NEW`, 数据源将不会切换, 也不会开启新事务, 需要注意.

2.4.2.4. 多数据源的注意事项

1. 如果加了`@Transactional`注解, 手动设置数据源的操作将失效! 在需要事务的时候, 就不能使用手动设置数据源的方法. 如果需要事务, 而且又是多数据源, 则在实现Service的时候需要将同一个数据源的操作放在一个方法内并且加上`@DataSource`注解和`@Transactional`注解.
2. 现在提到的多数据源多事务是指每个数据源可以有自己的事务, 事务之间是相互独立的. 不涉及到多个数据源的事务一致性(同时成功或失败).
3. 手动设置的数据源记得在finally中进行删除.
4. 如果Service方法没有加`@Transactional`注解, 并且没有被其他开启了事务的Service进行调用, 那么就是没有开启事务. 此时所有SQL操作都会在执行完之后自动提交到数据库(默认开启了AutoCommit), 如果Service的后续操作出现异常, 无法进行回滚. 所以在实现Service方法的时候需要考虑, 到底要不要使用事务?
5. 开启事务的话, 需要尽量减小事务的边界, 因为执行时间很长的事务会影响性能并且导致较多的行锁. 一些查询操作如果不需要事务的话可以放在事务边界之外.

2.4.2.5. 为数据源指定超时时间

通过修改配置文件, 可以为数据源指定超时时间. 例如:
文件: config-data/application.yml:

```
app:
  ds.timeout:
    db.ds.read.common: 30 #通用读数据源设置30s的sql超时时间
```

2.5. RestTemplate与Feign的使用

2.5.1. 调用内部微服务接口

调用内部其他微服务接口, 第一选择使用Feign, 如果请求参数非常复杂Feign满足不了需求, 可以考虑使用RestTemplate.
微服务RestTemplate注入方法:

```
@Autowired
private RestTemplate restTemplate;
```

2.5.2. 调用外部HTTP接口或者内部非微服务的HTTP接口

除了部分遗留代码会使用到HttpClient之外, 新开发的接口都要使用RestTemplate.
非微服务接口RestTemplate注入方法:

```
@Autowired
@Qualifier(Constants.RAW_REST_TEMPLATE)
```

```
3.  private RestTemplate rawRestTemplate;
```

2.5.3. RestTemplate使用示例

```
1.  String userId = "1";
2.  String category = "category1";
3.
4.  //GET查询用户资金
5.  BigDecimal balance = restTemplate.getForObject("http://USER-
    SERVICE/users/{userId}/balance",
6.          BigDecimal.class, userId);
7.
8.  //查询列表数据, url中带有PathVariable和QueryParam, 并且带有特殊字符
9.  URI uri = UriComponentsBuilder
10.         .fromHttpUrl("http://USER-SERVICE/users/{category}")
11.         .queryParam("queryParam1", "value1")
12.         .queryParam("queryParam2", "%value2%")
13.         .buildAndExpand(category).encode().toUri();
14.
15. CgtUserDTO[] cgtUserDTOS = restTemplate.getForObject(uri, CgtUserDTO[].class);
16. List<CgtUserDTO> cgtUserDTOList = Arrays.asList(cgtUserDTOS);
```

2.5.4 RestTemplate异常处理

使用RestTemlate请求其他服务, 如果服务返回正常请求结果(http状态码2xx), 则调用RestTemplate会正常返回. 如果服务返回的http状态码是4xx或者5xx, RestTemplate会抛出下面两个自定义异常之一:

1. **ServiceUnavailableException**: RestTemplate在连接不上服务端的时候会抛出这个异常.
2. **RemoteCallException**: RestTemplate能够请求到服务端, 但是服务端返回的http状态码不是2xx的情况下, 会抛出这个异常. 通过getOriginError()方法能够拿到服务端抛出的具体错误code信息, 如果解析服务端code失败, 则会拿到默认的CommonErrorCode.REQUEST_SERVICE_ERROR.

2.5.5 Feign异常处理

使用Feign请求其他服务, 如果服务返回正常请求结果(http状态码2xx), 则调用Feign会正常返回. 如果服务返回的http状态码是4xx或者5xx, Feign会抛出下面两个自定义异常之一:

1. **HystrixRuntimeException**: Feign在连接不上服务端的时候会抛出这个异常.
2. **RemoteCallException**: 定义同上.

2.5.6 RestTemplate与Feign异常处理示例

客户端在使用RestTemplate或Feign进行服务调用的时候如果不对上面的异常进行try/catch, 则异常会一直抛到Controller层的异常处理类中进行统一处理, 返回统一的JSON格式错误信息或者错误页面.

根据业务需要, 在使用RestTemplate或Feign进行服务调用的时候也可以手动处理异常情况.例如:

RestTemplate:

```
        try {
            UserBasicInfoDTO userBasicInfoDTO = restTemplate.getForObject("http
://USER-SERVICE-CLIENTS/user/{userId}/info",
                    UserBasicInfoDTO.class, "userid");
            //正确返回的业务处理

        } catch (RemoteCallException e) {
            ErrorInfo originError = e.getOriginError();
            //服务端报了业务异常，错误情况处理

        } catch (ServiceUnavailableException e) {
            //连接不上服务端，错误情况处理
        }
```

Feign:

```
        try {
            UserBasicInfoDTO userBasicInfo =
userClient.getUserBasicInfo("userid");
            //正确返回的业务处理

        } catch (RemoteCallException e) {
            ErrorInfo originError = e.getOriginError();
            //服务端报了业务异常，错误情况处理

        } catch (HystrixRuntimeException e) {
            //连接不上服务端，错误情况处理
        }

```

2.5.7 RestTemplate与Feign的请求超时配置

Feign会优先读取Ribbon的超时配置. 所以如果同时配置了okhttp和ribbon的超时时间, Feign会使用ribbon的配置. Feign的调用是在Hystrix里完成的. Hystrix命令执行开启了默认的超时时间(10s), 如果超过这个时间, 会抛出HystrixRuntimeException.

RestTemplate只会读取okhttp的超时配置, 并且默认不会使用Hystrix.
具体如何配置请查看下面的td-base配置项.

2.5.8 RestTemplate与Feign的请求重试机制

客户端使用Feign调用服务端如果出现异常情况, 会在一定规则下自动进行重试.
1. 如果是GET之外的请求(POST/DELET/PUT), 在找不到服务端或者服务端连接超时(ConnectionException或SocketTimeoutException)的情况下, Ribbon会在服务列表中找寻下一台服务

器再发起一次重试请求.
2. 如果是GET请求, 除了上面的情况, 在服务端返回超时的情况下也会进行一次重试.

因为GET请求在服务端超时的情况下可能会请求多次, 所以保证GET接口的幂等性很重要.

RestTemplate默认不会进行重试.

2.5.9 如何使用Feign上传文件

2.5.9.1. 定义feign接口

```
    @RequestMapping(method = RequestMethod.POST, value = "/file", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    String uploadFile(@RequestPart MultipartFile file);
```

2.5.9.2. 客户端调用

```
    String name = "photo.jpg";
    String contentType = "image/jpg";
    byte[] content = FileUtils.readFile(...)
    MultipartFile file = new MockMultipartFile(name,
                name, contentType, content);
    photoClient.uploadFile(file);
```

2.5.9.3. 服务端接收

```
    @RequestMapping(method = RequestMethod.POST, value = "/file", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    @ResponseBody
    public String uploadFile(@RequestPart(value = "photo.jpg", required = false) MultipartFile photo) {
        //...
    }
```

如果文件名是动态的, 可以用下面方式拿到文件

```
    @RequestMapping(method = RequestMethod.POST, value = "/file", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
    @ResponseBody
    public String uploadFile(AbstractMultipartHttpServletRequest request) {
        Map<String, MultipartFile> fileMap = request.getFileMap();
        //...
    }
```

2.6. 日志记录

2.6.1. 自动记录性能日志

默认情况下, td-base会拦截到spring mvc, feign, restTemplate的请求和响应, 将请求参数, 请求响应http头, 请求URL, 响应时间等信息以统一的格式打印到日志里. 日志示例:

```
PerfLog_1:{"type":"SPRING_RESP","time":525,"uri":"/userinfo","status":500,"header":"{\"Transfer-Encoding\":\"chunked\",\"Cache-Control\":\"no-store\",\"Connection\":\"keep-alive\",\"X-Application-Context\":\"config-test:datasource\\n:0\",\"Content-Language\":\"zh-CN\",\"Date\":\"Tue, 25 Jul 2017 11:36:39 GMT\",\"Content-Type\":\"application/json;charset=UTF-8\"}","error":""}
```

具体日志格式可以查看TD-BASE输出的性能日志格式.

可以修改配置文件来控制打印的内容, 配置项如下:

```
app.performance.log: all/simple/minimum/none
```

例如none就是不打印日志, 具体各参数说明如下:

```
    /**
     * 不打印任何内容
     */
    NONE,

    /**
     * 日志最小化, 不打印请求消息体, 请求参数, 请求响应头
     */
    MINIMUM,

    /**
     * 简化, 消息体, 请求参数, 请求响应头最长只打印1000个字符
     */
    SIMPLE,

    /**
     * 打印所有信息
     */
    ALL,

    /**
     * 没有设置, 如果是prod环境, 按照SIMPLE打印日志, 否则按照ALL打印日志.
     */
    NOTSET;
```

还可以修改配置文件来过滤某些url不打印日志, 例如:

```
app:
  #login和register请求不打印日志
  performance.log.ignore.urls: /login, /register
```

2.7. OkHttp使用

2.7.1 OkHttp介绍

td-base已经集成了OkHttp, 现在Feign和RestTemplate的默认Http类库已经是OkHttp3. 使用OkHttp能够复用TCP连接, 提升TCP连接效率, 减少以往读写超时的出现次数, 并且OkHttp的API更为简洁.

2.7.2 使用td-base提供的OkHttpClient

在BaseConfiguration中已经初始化了一个OkHttpClient的bean, 有需要的时候可以通过注入的方式直接使用.

1. 获取方式1: `@Autowired OkHttpClient okHttpClient`

2. 获取方式2:

```
OkHttpClient okHttpClient = ApplicationContextHolder.context.getBean(OkHttpClient.class)
```

可以在配置文件里设置OkHttpClient的相关参数:

```
app:
  okhttp:
    read.timeout: 10000
    connect.timeout: 10000
    write.timeout: 10000
    #最大空闲连接数
    max.idle: 5
    #空闲连接存活时间
    alive.duration: 300
```

2.7.3 自定义OkHttpClient

某些情况下, 可能需要自定义OkHttpClient, 比如设置特定的超时时间, 添加拦截器等. 此时可以按下面方式创建一个OkHttpClient.Builder.

```
OkHttpClient.Builder builder =
OkHttpUtils.okHttpClientBuilder(applicationConstant);
```

OkHttpUtils里设置了连接参数与日志拦截器, 你可以覆盖连接参数与添加自己的拦截器.

大部分情况下, 不应该直接使用OkHttp, 而应该使用RestTemplate这种抽象层次更高的类库. 只在有必

要的情况才使用OkHttp(例如管理https密钥).

2.8. 如何在部署多实例的时候只在Leader节点执行定时任务

只有不需要管理的重复执行的任务才推荐使用spring定时任务, 其他任务请使用xxl job

2.8.1. 引入SchedulerConfiguration

在启动类的import数组中, 加入SchedulerConfiguration类. 例如:

```
@Import({BaseConfiguration.class, ServiceClientConfiguration.class, WebApplicat
ion.class,
        MultipleDataSourceConfiguration.class, SchedulerConfiguration.class})
```

2.8.2. 定义定时任务Bean

```
public class JobScheduler {

    @Autowired
    private UserService userService;

    //两次调用之间固定间隔3秒
    @Scheduled(fixedRate = 3000L)
    @ZkLeaderSchedule
    public void sendBonusToRegisterUser() {
        userService.sendBonusToRegisterUser();
    }

    //每5秒执行一次
    @Scheduled(cron = "*/5 * * * * ?")
    @ZkLeaderSchedule
    public void checkUserLoginStatus() {
        userService.checkUserLoginStatus();
    }

}
```

注意, 如果需要只在leader节点上执行的定时任务, 方法上要加`@ZkLeaderSchedule`注解.

2.8.3. 在Spring容器创建Bean或者将Bean定义为@Component

2.8.4. 定时任务相关配置

```
app:
  #如果引用了SchedulerConfiguration, zookeeper.address必须要配置
  zookeeper.address: 10.100.11.13:2181,10.100.11.14:2181
  #定时任务执行线程数, 不配默认为10
```

```
    scheduler.thread.count: 10
```

2.8.5 实现方式

为了避免多实例环境下任务的并发执行, 对Spring的任务执行器做了修改. 系统在启动的时候先连接zookeeper进行leader选举, 只有被选举为leader的实例才能执行定时任务. 如果leader挂掉, 其他实例可以被选举为leader来执行任务. 同一时刻只会有一台实例执行定时任务.

leader选举的zk结点为`/tdjava/{serviceName}/schedulers`.

2.9. 使用xxl-job执行定时任务

2.9.1. 引入XxlJobConfiguration

在启动类的import数组中, 加入XxlJobConfiguration类. 例如:

```
@Import({BaseConfiguration.class, ServiceClientConfiguration.class, WebApplicat
ion.class,
        MultipleDataSourceConfiguration.class, XxlJobConfiguration.class})
```

2.9.2. 配置文件加入必须定义的配置

文件位置: `config-data/application-dev.yml`:

```
xxl.job:
  admin:
    addresses: http://10.100.11.195:9107
  executor:
    #appname: appname可以不填写, 默认为服务名-环境, 例如assets-service-prev
    port: 50021
```

addresses不同环境的地址不同:
1. 开发测试环境: http://10.100.11.195:9107
2. 灰度生产环境: http://task.paiconf.com
port端口号可以在Java服务端口分配中自行填写, 然后使用. 注意保证端口号不重复.

关于如何定义其他参数以及创建任务, 请查看任务调度中心接入与开发指南

2.10 定义默认的错误页面(包含页面的项目才需要提供)

如果项目中使用了thymeleaf模版, 请在模版文件夹的目录下提供`401.html, 404.html, 500.html`这三个默认的错误页面, 可以是静态html页面或者thymeleaf模版. 默认的模版文件夹位置

为`src/main/resources/templates`, 例如401.html的默认路径
为`src/main/resources/templates/401.html`. 前端的页面请求过来如果报错, 系统会根据http状态码跳转到对应的页面(除这三个之外的4xx, 5xx状态码统一返回500.html).

2.11 使用TdKeyGenerator类生成全局有序唯一ID(基于snowflake)

使用TdKeyGenerator类生成全局有序唯一ID(基于snowflake)

2.12 使用Sharding-JDBC进行分库分表

TD-BASE结合Sharding-JDBC进行分库分表

3. 其他问题

3.1. 预防SQL注入

Mybatis中尽量使用#{}表达式, 因为#{}表达式会使用JDBC的预编译功能, 生成?占位符来预防SQL注入.
例如在MySQL中, 进行like查询使用#{}表达式 `LIKE CONCAT('%',#{name},'%' )`.
${}表达式属于字符串替换, 很容易引发SQL注入. 使用的时候需要很谨慎, 并且要调用工具类来检测危险字符串.

3.2. JSON对象转换的问题

3.2.1. SpringMVC和RestTemplate不能正确反序列化包含泛型和list的复杂对象(如分页的Page对象)的问题

注意, 如果json对应的数据是包含泛型和list的复杂对象(例如分页的Page对象), 在Spring MVC的Controller方法声明中, 不能直接使用对象,
并且RestTemplate的返回值也不能声明该对象, 不然会出现反序列化失败的问题. 例如下面的代码就不能正常工作:

```
1.  //不能反序列化为newPage
2.  Page<DqUserFund> newPage = restTemplate.postForObject("http://trade/test2", Res
    tTemplateUtils.createJsonEntity(page), Page.class);
3.
4.  //不能正常接收page参数, 服务端会报http status code 为415的错误
5.  @RequestMapping(method = RequestMethod.POST, value = "/test2")
6.  @ResponseBody
7.  String test2(@RequestBody Page<DqUserFund> page) throws IOException {
```

```
}
```

上面的情况, 应该使用字符串类型接收数据, 然后在代码中手动将数据转换成目标类型. 例如:

```
String json = restTemplate.postForObject("http://trade/test2",
RestTemplateUtils.createJsonEntity(page), String.class);
Page<DqUserFund> newPage = JsonUtils.json2GenericObject(json, new TypeReference
<Page<DqUserFund>>() {});

@RequestMapping(method = RequestMethod.POST, value = "/test2")
@ResponseBody
Page<DqUserFund> test2(@RequestBody String json) throws IOException {
    Page<DqUserFund> page = JsonUtils.json2GenericObject(json, new
TypeReference<Page<DqUserFund>>() {});
}
```

使用Feign不会存在RestTemplate这种问题.

3.3. 服务名的版本化与定制化

3.3.1. 服务名版本化

项目上线之后, 可能会出现一个项目多个分支的情况, 并且一个服务可能会演化出不同的版本. 如果不同版本用相同的服务名, 在服务注册中心会出现混乱. 这种情况下服务名可以加上版本号后缀, 比如用户服务(user-service)最开始的第一个版本是1.0, 服务名是user-service. 服务上线之后又继续在开发1.1版本, 并且一段时间内1.0和1.1版本要同时并存(线上有两个版本的user-service), 这种情况下, 用户服务1.1的服务名可以取作user-service-1.1. user-service-1.1和user-service可以用相同的配置仓库, 也可以用不同, 具体看业务需求. 如果要使用相同的仓库, 修改user-service-1.1项目里的bootstrap.xml, 修改`spring.cloud.config.name: user-service`即可.

3.3.2. 开发环境的服务名定制

开发环境中, 如果服务名在服务注册中心已经存在, 而又想让客户端只调用自己服务的话, 可以在服务名后面加上自己名字的后缀来启动服务. 例如修改bootstrap.xml:

```
spring.application.name: user-service-lb
spring.cloud.config.name: user-service
```

客户端在调用的时候指定请求user-service-lb服务, 就只会访问你机器上的服务了.

bootstrap.xml中两个重要的配置:
1. spring.application.name 服务名称, 一般来说是[项目名-版本号]或者在开发环境下[项目名-你的代号]

2. spring.cloud.config.name git仓库名称前缀, 配置中心会读取这个配置来查找配置仓库位置

4. td-base相关配置项

下面配置都可以配置在application.yml中, 列出来的是默认配置. 如果不需要修改默认配置, 则不需要将下面的配置加到application.yml.

4.1. td-base配置项

```
app:
  #是否是web站点(对外提供服务), 默认false.
  #如果设置为true, swagger文档默认不显示. 如果项目有页面, 需要在templates目录下提供404和401的thymeleaf页面
  web.project: false

  # 统一拦截的日志级别
  # NONE 不打印任何内容
  # MINIMUM 日志最小化, 不打印请求消息体, 请求参数, 请求响应头
  # SIMPLE 简化, 消息体, 请求参数, 请求响应头最长只打印1000个字符
  # ALL 打印所有日志
  # NOTSET 非生产环境为ALL, 生产环境为SIMPLE
  performance.log: NOTSET
  #login和register请求不打印日志
  performance.log.ignore.urls: /login, /register


  #okhttp相关配置
  okhttp:
    read.timeout: 10000
    connect.timeout: 10000
    write.timeout: 10000
    #最大空闲连接数
    max.idle: 5
    #空闲连接存活时间
    alive.duration: 300

  #zookeeper与定时任务配置项
  #如果引用了SchedulerConfiguration, zookeeper.address必须要配置
  zookeeper.address: 10.100.11.13:2181,10.100.11.14:2181
  #定时任务执行线程数, 默认为10
  scheduler.thread.count: 10

  #sharding jdbc相关配置
  #是否打印sharding jdbc日志, 默认为true
  sjdbc.show.log: true
  #sharding jdbc执行的线程池数量, 默认为可用CPU核心数
```

```
  sjdbc.executor.size: 8

  #数据源超时配置
  ds.timeout:
    db.ds.read.common: 30 #通用读数据源设置30s的sql超时时间
```

4.2. Hystrix主要配置项

```
hystrix:
  # hystrix执行命令的默认超时时间，超过这个时间feign调用将抛出HystrixRuntimeException
  command.default.execution.isolation.thread.timeoutInMilliseconds: 10000
  # 线程池大小，Feign每个类使用不同的线程池
  threadpool.default.coreSize: 10
```

4.3. Ribbon主要配置项

```
ribbon:
  # 默认的连接超时时间，Feign会优先读取这个配置
  ConnectTimeout: 2000
  # 默认的读取超时时间，  Feign会优先读取这个配置
  ReadTimeout: 5000
```

4.4. xxl job主要配置项

```
xxl.job:
  accessToken: #可以为空
  admin.addresses: #不能为空，xxl job控制台地址
  executor:
    appname: #可以为空，默认为服务名-环境，例如assets-service-prev
    ip: #可以为空
    port: #不能为空，需要在Java服务端口分配文档中填写然后使用
    logpath: #可以为空，默认为/data/logs/td-task/jobhandler/
```