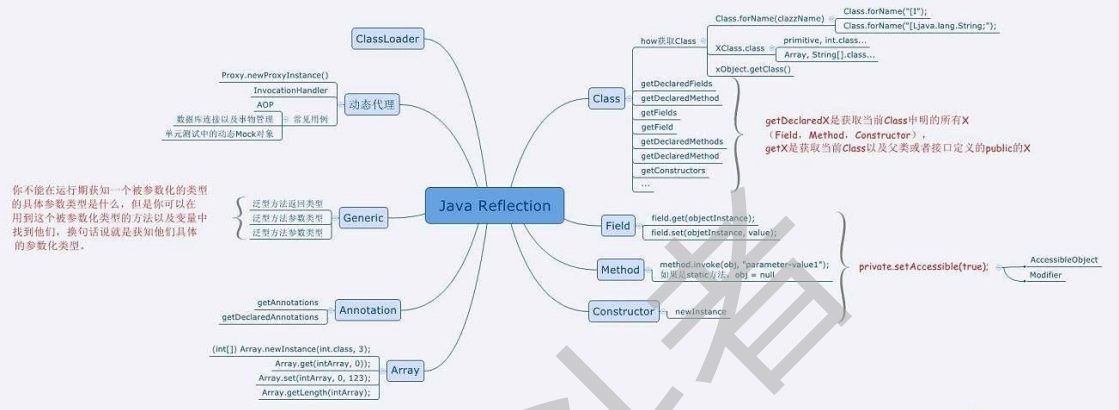
JAVA 反射（非重点）

动态语言

- 动态语言，是指程序在运行时可以改变其结构：新的函数可以引进，已有的函数可以被删除等结构上的变化。比如常见的 JavaScript 就是动态语言，除此之外 Ruby,Python 等也属于动态语言，而 C、C++则不属于动态语言。从反射角度说 JAVA 属于半动态语言。

反射机制概念 （运行状态中知道类所有的属性和方法）

- 

- 在 Java 中的反射机制是指在运行状态中，对于任意一个类都能够知道这个类所有的属性和方法；并且对于任意一个对象，都能够调用它的任意一个方法；这种动态获取信息以及动态调用对象方法的功能成为 Java 语言的反射机制。

反射的应用场合

- 在 Java 程序中许多对象在运行是都会出现两种类型：编译时类型和运行时类型。 编译时的类型由声明对象时实用的类型来决定，运行时的类型由实际赋值给对象的类型决定 。如：

  Person p=new Student();

  其中编译时类型为 Person，运行时类型为 Student。
- 程序在运行时还可能接收到外部传入的对象，该对象的编译时类型为 Object,但是程序有需要调用该对象的运行时类型的方法。为了解决这些问题，程序需要在运行时发现对象和类的真实信息。然而，如果编译时根本无法预知该对象和类属于哪些类，程序只能依靠运行时信息来发现该对象 和类的真实信息，此时就必须使用到反射了。

反射使用步骤（获取 Class 对象、调用对象方法）

- 获取想要操作的类的 Class 对象，他是反射的核心，通过 Class 对象我们可以任意调用类的方法。
- 调用 Class 类中的方法，既就是反射的使用阶段。
- 使用反射 API 来操作这些信息。

获取 Class 对象的 3 种方法

- **调用某个对象的getClass()方法**

  Person p=new Person();

  Class clazz=p.getClass();

- **调用某个类的 class 属性来获取该类对应的 Class 对象**

  Class clazz=Person.class;

- **使用 Class 类中的 forName()静态方法(最安全/性能最好)**

  Class clazz=Class.forName("类的全路径"); (最常用)

反射记录日志实例

使用AOP记录系统接口日志-自定义记录内容**

介绍

在一个微服务的系统中，对外的接口可能分布在不同的服务中，我们需要记录这些接口的日志，可能包括请求的时间、耗时、请求的状态、请求用户、请求参数等；
对于这些需求，可以使用AOP（面向切面编程），来方便的实现。
本篇文章不是侧重于aop的使用，而是针对解决记录接口的请求日志，需要记录请求的类型、请求参数等。对于这些需求，
本文中，基于一个自定义的注解LogAnnotation，来实现对接口的自定义记录方案。而接口方法的具体参数，利用反射来获取参数的具体属性。

日志方案

1.自定义一个日志注解，LogAnnotation，通过该注解，在请求接口方法上，定义需要记录的方法参数的属性和属性的说明
2.在每个需要记录日志的接口方法上，添加LogAnnotation注解
3.使用切面编程，获取每个拥有LogAnnotation注解的方法中的方法参数，结合LogAnnotation注解信息，利用反射，获取方法参数值，拼接日志内容，生成系统日志对象
4.日志处理服务中，将日志入库保存

由于使用了spring cloud微服务，记录日志的方案是：在每个接口服务中，记录日志后，放入响应头，在网关处进行统一的获取处理，放入mq队列，然后由日志服务接收处理，不影响原来请求的响应。这样方式比较方便，不用在每个微服务中进行日志保存等操作，大家可以参考。
下面的代码是核心的生成日志内容的方法。

代码实现

集成AOP

spring boot中添加AOP依赖

```
maven依赖添加如下
<!--引入SpringBoot的Web模块-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!--引入AOP依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

日志类型枚举

通过一个日志类型枚举，来定义不同的日志类型，主要是根据常用的接口请求类型定义的，比如登录、查询、更新等等

```
package com.pu.log;

/**

 * 日志类型枚举
   */
   public enum LogTypeEnum {

   /**

    * 登录
      */
      LOGIN(0, "登录"),

   /**

    * 登出
      */
      LOGOUT(1, "登出"),

   /**

    * 查询
      */
      SELECT(10, "查询"),

   /**

    * 插入，新增
      */
      INSERT(11, "新增"),

   /**

    * 更新
      */
      UPDATE(12, "更新"),

   /**

    * 删除
      */
      DELETE(13, "删除"),

   /**

    * 下载
      */
      DOWLOAD(20, "下载");
private Integer type;
private String message;

LogTypeEnum(Integer type, String message) {
```

```
    this.type = type;
    this.message = message;
}

public Integer getType() {
    return type;
}
public String getMessage() {
    return message;
}
}
```

日志内容记录类

日志基本内容记录类，用来保存日志的类型，日志的内容，请求是否成功，错误原因。
contents属性，即日志内容，会在AOP切面中拼接得到。
该类只是保存了的接口的基本请求信息，一般日志还会加上用户信息等，可以根据自身的项目，进行扩展

```java
package com.pu.log;

import lombok.Data;

import java.util.Date;

/**
 * 日志内容记录
 */
@Data
public class BaseLog {

    /**
     * 日志类型
     */
    private LogTypeEnum logType;

    /**
     * 日志内容
     */
    private String contents;

    /**
     * 时间
     */
     private Date time;

    /**
     * 是否成功 1是 0否
      */
    private Integer success;

    /*
    错误原因
     */
    private String errorReason;

}

```

日志注解

LogAnnotation 注解，用在接口方法上，用来自定义日志的信息，包括：

1.请求的类型（type）
2.接口的方法中需要记录的参数索引（argsIndex）
3.记录方法参数对象里面的哪些属性（field）
4.对应这些属性的前缀说明（prefix）

argsIndex用来记录方法的哪个参数，是需要记录的。所以目前该注解仅支持记录一个参数。
一般在controller接口上，post请求，只会用一个对象来接收request参数；
但是get请求，可能会直接写多个方法参数来接收request参数，所以这种的话，只能记录一个参数，或者将多个参数，写成一个类，用对象来接收即可。

field和prefix，这两个属性，是字符串数组，用来定义，请求参数对象中，需要记录哪些属性和这些属性的说明，
比如 field = [“id”, “name”]，prefix = [“ID”, “名称”]，表示：记录方法参数对象中的，id属性和name属性，分别表示ID和名称。所以field和prefix的数组元素，需要一一对应。

在利用反射进行日志内容拼接时，就是根据field和prefix，来获取属性值，并添加说明后，进行拼接的。

```java
package com.pu.log;

import java.lang.annotation.*;

/**
 * 日志注解
 */
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface LogAnnotation {

    /**
     * 是否需要记录日志，默认需要
     * @return
     */
    boolean need() default true;

    /**
     * 日志类型
     * @return 日志类型
     */
    LogTypeEnum type();

    /**
     * 记录日志时，拼接的前缀(默认后面加":")，当记录多个参数的字段时，前缀一般需要和字段（field）一一对应，
     * 当字段（field）数量大于前缀数量时，默认取最后一个前缀，作为超出的字段的记录前缀。
     */
    String[] prefix() default {};

    /**
     * 日志中记录的方法参数索引，默认记录第0个参数。如果字段（field）为空数组，则记录该参数所有信息。<br>
     * 如果该参数是集合（Collection），则遍历记录每一个元素。
     * @return 方法参数索引
     */
    int argsIndex() default 0;

    /**
     * 方法参数中需要记录的属性字段名，可设置多个需要记录日志的字段
     */
    String[] field() default {};

}
```


AOP切面

定义一个切面，然后使用around(环绕通知)，来获取接口方法的日志内容。
在spliceLogContents()方法中，利用反射，将方法参数的属性提取出来，和前缀拼接成日志内容。
如果LogAnnotation 的field为空，没有定义，则会直接将方法参数toString()后输出。
同时在接口方法执行中捕捉异常，来确定接口是否成功，下面是代码实现：

```
package com.pu.log;

import lombok.extern.slf4j.Slf4j;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
import org.aspectj.lang.reflect.MethodSignature;
import org.springframework.stereotype.Component;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.util.Collection;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;

/**

 * 日志切面
   */
   @Component
   @Aspect
   @Slf4j
   public class SystemLogAop {

   /**

    * 定义切点,控制层所有方法
      */
      @Pointcut("@annotation(com.pu.log.LogAnnotation)")
      public void requestServer() {

   }

   @Around("requestServer()")
   public Object doAround(ProceedingJoinPoint point) throws Throwable {
       // 获取方法
       MethodSignature signature = (MethodSignature) point.getSignature();
       Method method = signature.getMethod();
// 获取类
Class<?> clazz = point.getTarget().getClass();

String methodName = method.getName();
String clazzName = clazz.getSimpleName();

// 看有没有日志注解
LogAnnotation logAnnotation = method.getAnnotation(LogAnnotation.class);
if (logAnnotation == null) {
    return point.proceed();
```

```
}

// 看是不是需要记录日志
if (!logAnnotation.need()) {
    return point.proceed();
}

LogTypeEnum logType = logAnnotation.type();
// 方法参数，需要记录的信息
int argsIndex = logAnnotation.argsIndex();
String[] prefixs = logAnnotation.prefix();
String[] fields = logAnnotation.field();

// 方法参数
Object[] args = point.getArgs();
if (args == null || args.length - 1 < argsIndex) {
    log.error("记录系统日志时，实际的方法参数和LogAnnotation中定义的方法参数索引不一
致，类: {}, 方法: {}",
            clazzName, methodName);
    return point.proceed();
}

// 日志的内容，下面进行拼接
StringBuilder logContents = new StringBuilder();

// 需要记录日志的参数对象，如果参数是个集合，则遍历每一个元素进行记录
Object arg = args[argsIndex];
if (arg instanceof Collection) {
    Collection as = (Collection) arg;
    for (Object a : as) {
        if (logContents.length() > 0) {
            logContents.append(";");
        }
        logContents.append(spliceLogContents(a, fields, prefixs));
    }
} else {
    logContents.append(spliceLogContents(arg, fields, prefixs));
}

// 响应头中存放对象
BaseLog baseLog = new BaseLog();
baseLog.setLogType(logType);
baseLog.setTime(new Date());
baseLog.setContents(logContents.toString());
baseLog.setSuccess(1);

Exception ex = null;
Object proceed = null;
try {
    proceed = point.proceed();
} catch (Exception e) {
    baseLog.setSuccess(0);
    baseLog.setErrorReason(e.getMessage());
    ex = e;
}

log.info("记录日志: {}", baseLog);
// 处理保存日志
```

```
// saveLog(baseLog);

if (ex != null) {
    throw ex;
}

// 继续执行
return proceed;
}
/**
 * 利用反射，从对象中，获取属性字段的值，拼接前缀。
 * @param obj      对象
 * @param fields   字段名称集合
 * @param prefixs 前缀集合
 * @return 拼接内容
 * @throws NoSuchFieldException    找不字段异常
 * @throws IllegalAccessException 字段访问异常
 */
   private String spliceLogContents(Object obj, String[] fields, String[]
prefixs) throws NoSuchFieldException, IllegalAccessException {
   // 如果没有定义属性，则直接将对象toString后记录，如果定义了前缀，则拼接上前缀后记录
   if (fields == null || fields.length == 0) {
       if (prefixs != null && prefixs.length > 0) {
           return prefixs[0] + ":" + obj.toString();
       }
       return obj.toString();
   }

   StringBuilder sb = new StringBuilder();

   boolean hasPre = prefixs.length > 0;
   int prefixMaxIndex = prefixs.length - 1;
   int prefixIndex = 0;

   Class<?> aClass = obj.getClass();

   // 如果该对象中找不到属性，则向上父类查找
   Map<String, Field> fieldMap = new HashMap<>();
   for (; aClass != Object.class; aClass = aClass.getSuperclass()) {
       for (Field f : aClass.getDeclaredFields()) {
           fieldMap.putIfAbsent(f.getName(), f);
       }
   }

   Field field = null;
   Object fieldValue = null;
   for (int i = 0, len = fields.length; i < len; i++) {
       field = fieldMap.get(fields[i]);
       if (field == null) {
           continue;
       }
       field.setAccessible(true);
       fieldValue = field.get(obj);
       if (sb.length() > 0) {
    sb.append(",");
}
if (hasPre) {
    prefixIndex = i < prefixMaxIndex ? i : prefixMaxIndex;
```

```
            sb.append(prefixs[prefixIndex]);
            if (!prefixs[prefixIndex].endsWith(":")) {
                sb.append(":");
            }
        }
        sb.append(fieldValue == null ? "" : fieldValue);
        }
        return sb.toString();
        }
        }
```

使用案例

```java
package com.pu.controller;

import com.pu.log.LogAnnotation;
import com.pu.log.LogTypeEnum;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;

/**
 * @description: 用户接口
 */
@RestController
@RequestMapping("user")
public class UerController {

    /**
     * 该接口在切面中，记录的日志内容BaseLog.contents为：用户ID:XXX
     */
    @LogAnnotation(type = LogTypeEnum.SELECT, prefix = "用户ID")
    @GetMapping
    public Object get(String id) {
        return null;
    }

    /**
     * 该接口在切面中，记录的日志内容BaseLog.contents为:
     * 用户名称:zhangsan,昵称:张三
     */
    @LogAnnotation(type = LogTypeEnum.INSERT, prefix = {"用户名称", "昵称"}, field = {"name", "nickName"})
    @PostMapping
    public Object save(@RequestBody User user) {
        return null;
    }

}
```