java开发避坑基础篇（三）

- 承前
- 重构
    - 何谓重构
    - 重构的作用
    - 谨慎重构
- 代码坏味道
    - 重复代码
    - 过长函数
    - 依恋情节
    - 狎昵关系
    - switch惊悚
    - 发散式变化
    - 过大的类
    - 过长参数列表
    - 散弹式修改
- 启后

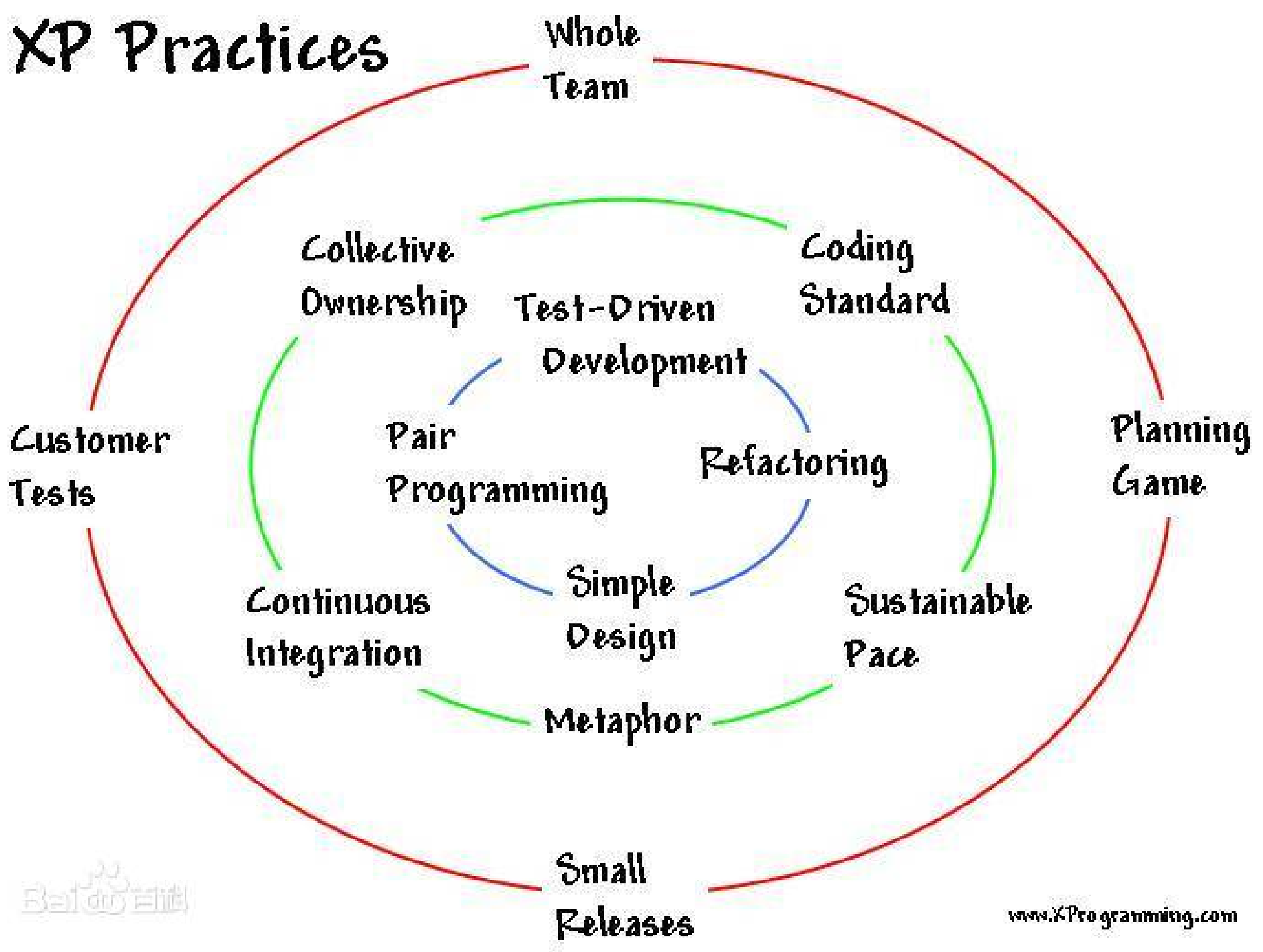
承前

此篇是java开发避坑基础篇中第三篇，也是最后一篇。第一篇康雄同学给大家介绍了有关java语言自身使用上的一些陷阱，第二篇李贺同学就spring事务与大家做了交流沟通。本篇我们就编写代码时常见的坏味道与大家一起交流学习。

重构

何谓重构

提起重构那就不得不提一个我们每个开发人员耳熟能详的名词，那就是敏捷。大家对敏捷应该都不陌生了，至少对这个名词是很不陌生的。从大的方面敏捷分为管理实践与工程实践。管理实践现在最流行的要属SCRM，而工程实践最流行的就是极限编程（XP）。而重构就是作为极限编程中的一个重要环节出现的。下图就是著名的极限编程洋葱环，重构位于13个过程域最最核心的位置。所以说好的代码都是重构出来的。



那什么是重构？下面是重构的两种定义（来自Martin Fowler重构一书）

1、对软件内部结构的一种调整，目的是在不改变软件可观察行为的前提下，提高其可理解性，降低其修改成本；

2、使用一系列重构手法，在不改变软件可观察行为的前提下调整期结构。

上述两种定义有一个共性，那就是不改变软件的可观察行为。那何为可观察行为？这个是相对用户而言的，就是用户在重构前后使用接口的方式是不能变的且接口的功能也是不能变的。这里的接口是系统对外提供的接口也可以是模块之间交互的接口（这里其实牵扯到另外一个问题，设计上的依赖倒置）。总之一句话重构改变的是软件或模块的内部，而不是重新设计，更不是重写。

重构的作用

1、重构改进软件的设计

2、重构使软件更容易被理解

3、重构帮助找到代码中的bug

4、重构提高编程速度

总之重构可以提升代码质量

谨慎重构

重构可以是对软件内部的微调，也可以对软件进行大刀阔斧的修改。那么我们在重构的时候是不是就肆无忌惮了？答案肯定是否定的。我们怎么确保我们的每次重构不会引入新的问题呢？那就是单元测试，单元测试是能够游刃有余的放心大胆的重构的唯一保障，而不是你对系统足够的了解，更不是作为资深程序员的经验。**所以没有单元测试的重构就是在玩火。**

代码坏味道

重复代码

危害

很多书中将代码的重复称作万恶之源。重复代码带来的危害就是对于修改是分散的，在软件变化修改时很容易出现修改遗漏。

案例一：一个类中不同方法之间重复

重构前

```
/**
     *
     */
    @Transactional(rollbackFor = Exception.class)
    public void update(Project project) {
        //
        transactionRegistrar.registryAfterCommitSync(() -> {
            BeanPool.getBean(ProjectManager.class).evictProject(project.getProjectId());
            projectUpdateProducer.produce(project.getProjectId());
        });

    }

    /**
     *
     * @param project
     */
    @Transactional(rollbackFor = Exception.class)
    public void setting(Project project) {
        //
        transactionRegistrar.registryAfterCommitSync(() -> {
            BeanPool.getBean(ProjectManager.class).evictProject(project.getProjectId());
            projectUpdateProducer.produce(project.getProjectId());
        });
    }

    /**
     *
     * @param projectId
     */
    @Transactional(rollbackFor = Exception.class)
    public void delete(Integer projectId) {
        //
        transactionRegistrar.registryAfterCommitSync(() -> {
            BeanPool.getBean(ProjectManager.class).evictProject(projectId);
            projectUpdateProducer.produce(projectId);
        });
    }

    /**
     *
     * @param projectId
     * @param status
     */
    @Transactional(rollbackFor = Exception.class)
    public void audit(Integer projectId, Integer status) {
        //
        transactionRegistrar.registryAfterCommitSync(() -> {
            BeanPool.getBean(ProjectManager.class).evictProject(projectId);
            projectUpdateProducer.produce(projectId);
        });
    }
```

一个类中的重复代码，我们消除的手段一般是提取公共方法。

重构后

```
/**
    *
    */
   @Transactional(rollbackFor = Exception.class)
   public void update(Project project) {
       //
       doAfterTransactionalCommitOfProjectModify(project.getProjectId());
   }

   /**
    *
    * @param project
    */
   @Transactional(rollbackFor = Exception.class)
   public void setting(Project project) {
       //
       doAfterTransactionalCommitOfProjectModify(project.getProjectId());
   }

   /**
    *
    * @param projectId
    * @param status
    */
   @Transactional(rollbackFor = Exception.class)
   public void audit(Integer projectId, Integer status) {
       //
       doAfterTransactionalCommitOfProjectModify(projectId);
   }

   /**
    *
    * @param projectId
    */
   @Transactional(rollbackFor = Exception.class)
   public void delete(Integer projectId) {
       //
       doAfterTransactionalCommitOfProjectModify(projectId);
   }

   private void doAfterTransactionalCommitOfProjectModify(Integer projectId){
       transactionRegistrar.registryAfterCommitSync(() -> {
           BeanPool.getBean(ProjectManager.class).evictProject(projectId);
           projectUpdateProducer.produce(projectId);
       });
   }
```

案例二：多个子类之间的代码重复

子类一

```
@Controller
public class SmsProcessController extends AbstractController {

    protected static final Logger LOGGER = LoggerFactory.getLogger(SmsProcessController.class);

    @VivoLogNote(operation = VivoLogNote.SMS_REGISTER, title = "-", interfaceName = "/account/center/smsLogin
/p1")
    @RequestMapping(value = "/usrprd/account/center/smsLogin/p1", method = RequestMethod.POST, produces =
"application/json")
    @ResponseBody
    public ResponseVo smsLoginP1(CommonLoginReq loginReq) throws Exception {
        if (!this.preprocess(loginReq)) {
            LOGGER.error("smsLoginP1 request parameter:{}", JSON.toJSONString(loginReq));
            return new ResponseVo(ReturnStatEnum.INVALID_PARAMETERS, resource.get());
        }
        if (!loginReq.checkRequestParameter()) {
            LOGGER.error("smsLoginP2 request parameter:{}", loginReq.toString());
            return new ResponseVo(ReturnStatEnum.USER_UPDATE_PHONTE_FORMAT_INVALD, resource.get());
        }
        loginReq.setAccount(loginReq.getPhoneNum());
        loginReq.ip(ip.get()).request(request.get());
        AppRquestEnum registerReq = AppRquestEnum.valueOf(loginReq.getBizzChannel(), loginReq.getVerCode(),
                RequestInterface.SMS_REG_LOGIN_P1);
        ResponseVo responseVo = null ;
        try {
            responseVo = UserProdContext.getInstance(registerReq).handle(loginReq, resource.get());
        } catch (Exception e) {
            LOGGER.error("handle request error. apprequest:{},input:{}", registerReq.toString(), loginReq.
toString(), e);
            return new ResponseVo(ReturnStatEnum.ERROR, resource.get());
        } finally {
            HandlerResultContext.clearHandlerResult();
            removedLocal();
        }
        LOGGER.info("smsLoginP1 response:{}", JSON.toJSONString(responseVo));
        return responseVo;
    }

}
```

多个子类中具有重复代码，我们一般采取的手段是提供公共方法到父类，在子类中调用父类的方法。

子类二

```
@Slf4j
@Controller
public class RegisterController extends AbstractController {

    protected static final Logger LOGGER = LoggerFactory.getLogger(RegisterController.class);

    /**
     * 4.0//
     */
    @VivoLogNote(operation = VivoLogNote.SMS_REGISTER, title = "", interfaceName = "/account/center/register
/p1")
    @RequestMapping(value = "/usrprd/account/center/register/p1", method = RequestMethod.POST, produces =
"application/json")
    @ResponseBody
    public ResponseVo getRegCodeByPhoneOrEmail(PhoneEmailRegisterReq phoneEmailRegisterReq) throws Exception {
        LOGGER.info("getRegCode request:{}", JSON.toJSONString(phoneEmailRegisterReq));
        if (!phoneEmailRegisterReq.checkRequestParameter() || !this.preprocess(phoneEmailRegisterReq)) {
            log.error("get regcode by phone or email request parameter:{}", phoneEmailRegisterReq.toString());
            return new ResponseVo(ReturnStatEnum.INVALID_PARAMETERS, resource.get());
        }
        phoneEmailRegisterReq.ip(ip.get()).request(request.get());
        AppRquestEnum registerReq = AppRquestEnum.valueOf(phoneEmailRegisterReq.getBizzChannel(),
                phoneEmailRegisterReq.getVerCode(), RequestInterface.REGISTER_P1);
        ResponseVo responseVo = null ;
        try {
            responseVo = UserProdContext.getInstance(registerReq).handle(phoneEmailRegisterReq, resource.get());
        } catch (Exception e) {
            log.error("handle request error. apprequest:{},input:{}", registerReq.toString(),
phoneEmailRegisterReq.toString(), e);
            return new ResponseVo(ReturnStatEnum.ERROR, resource.get());
        } finally {
            HandlerResultContext.clearHandlerResult();
            removedLocal();
        }
        LOGGER.info("getRegCode response:{}", JSON.toJSONString(responseVo));
        return responseVo;
    }

}
```

在平时开发中对于代码中一大片的代码重复我们都有动力和意识去消除它，反而对于这种只有一两行的重复代码，我们消除它的动力和意识都比较弱。

过长函数

危害

1、过长的函数导致程序难以被理解

2、过长的函数导致方法不再同一个抽象层级上（只做一件事）

3、代码中出现大量的无用注释

案例一：方法提炼

重构前

```
@Transactional(rollbackFor = Exception.class)
   public void update(Permission permission) {

       validateNameDuplication(permission);

       PermissionDO update = new PermissionDO();
       update.setName(permission.getName());
       update.setGroupId(permission.getGroupId());
       update.setDescription(permission.getDescription());
       update.setType(permission.getType());
       update.setAutoAllocation(permission.getAutoAllocation());

       permissionMapper.updateByExampleSelective(
               update,
               Example.builder(PermissionDO.class)
                       .andWhere(
                               Sqls.custom().andEqualTo("permissionId", permission.getPermissionId())
                       )
                       .build()
       );
       List<Integer> defaultProjectIds = listProjectWhichHasDefaultPermission(permission.getPermissionId());

       deletePermissionFromProject(permission.getPermissionId());

       allocationPermissionToProject(permission);

       resetDefaultPermission(defaultProjectIds, permission.getPermissionId());

       BeanPool.getBean(PermissionManager.class).pipelinedEvictProjectPermission(permission.getProjectIds());
   }
```

绝大部分的场景，处理过大的函数都可以通过函数拆分来解决。拆分函数的原则：每当感觉需要以注释来说明点什么的时候，我们就把需要说明的东西写进一个独立的函数中，并以其用途（而非实现手法）命名。

重构后

```
 @Transactional(rollbackFor = Exception.class)
    public void update(Permission permission) {

        validateNameDuplication(permission);

        updatePermissionElement(permission);

        List<Integer> defaultProjectIds = listProjectWhichHasDefaultPermission(permission.getPermissionId());

        deletePermissionFromProject(permission.getPermissionId());

        allocationPermissionToProject(permission);

        resetDefaultPermission(defaultProjectIds, permission.getPermissionId());

        BeanPool.getBean(PermissionManager.class).pipelinedEvictProjectPermission(permission.getProjectIds());
    }

    private void updatePermissionElement(Permission permission){
        PermissionDO update = new PermissionDO();
        update.setName(permission.getName());
        update.setGroupId(permission.getGroupId());
        update.setDescription(permission.getDescription());
        update.setType(permission.getType());
        update.setAutoAllocation(permission.getAutoAllocation());

        permissionMapper.updateByExampleSelective(
                update,
                Example.builder(PermissionDO.class)
                        .andWhere(
                                Sqls.custom().andEqualTo("permissionId", permission.getPermissionId())
                        )
                        .build()
        );
    }
```

案例二：函数对象

如果我们方法内部有大量的临时变量（要想访问到临时变量，需要程序在临时变量申明的方法内部，最终导致方法过长），它们会对我们方法的提炼产生一定的阻碍。如果我们尝试使用提炼方法，最终就会把临时变量传递给新的方法。导致可读性几乎没有任何的提升。

重构前

```
@Override
    public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) {
        MethodRestConfig config = findMethodRestConfig(method);
        if (config == null) {
            throw new UnsupportedOperationException("can not find rest config of " + method.getDeclaringClass().
getSimpleName() + "." + method.getName());
        }
        HttpResolver httpResolver = config.isAsync() ? asyncHttpResolver : syncHttpResolver;

        RestConfig.RestConfigBuilder builder = httpResolver.custom();

        Map<String, String> paths = new HashMap<>();
        Map<String, String> headers = new HashMap<>();
        Map<String, Object> reals = new HashMap<>();
        Map<String, Object> urlParameters = new HashMap<>();
        Parameter[] parameters = method.getParameters();

        label:
        for (int index = 0; index < parameters.length; index++) {

            if (config.getFutureCallbackIndex() != null && config.getFutureCallbackIndex() == index) {
                builder.futureCallback((FutureCallback) args[index]);
                continue;
            }
            Parameter parameter = parameters[index];
            Annotation[] annotations = parameter.getAnnotations();
            for (Annotation annotation : annotations) {
                if (annotation instanceof Path) {
                    if (args[index] != null) {
                        Path path = (Path) annotation;
                        paths.put(StringUtils.hasText(path.value()) ? path.value() : config.getParameterName
(index), String.valueOf(args[index]));
                    }
                    continue label;
                }
                if (annotation instanceof Header) {
                    if (args[index] != null) {
                        Header header = (Header) annotation;

                        Object headerArgument = args[index];

                        if (headerArgument instanceof Map) {
                            ((Map) headerArgument).forEach((key, value) -> headers.put(String.valueOf(key),
String.valueOf(value)));
                        } else {
                            headers.put(StringUtils.hasText(header.value()) ? header.value() : config.
getParameterName(index), String.valueOf(args[index]));
                        }

                    }
                    continue label;
                }
                if (annotation instanceof Pair) {
                    Pair pair = (Pair) annotation;
                    urlParameters.put(StringUtils.hasText(pair.value()) ? pair.value() : config.getParameterName
(index), args[index]);
                    continue label;
                }
            }
            reals.put(config.getParameterName(index), args[index]);
        }

        processUrlArgument(urlParameters, builder);

        ContentType contentType = method.getDeclaredAnnotation(ContentType.class);
        if (contentType != null && StringUtils.hasText(contentType.value())) {
            builder.contentType(contentType.value());
        }

        builder.socketTimeout(config.getSocketTimeout());
        builder.connectionTimeout(config.getConnectionTimeout());
        builder.returnType(getTypeReference(config, method));
        builder.unpack(config.isUnpack());
        builder.url(pathConfigUrlAnalyzer.analyze(config.getUrlTemplate(), paths, environment));
        builder.header(headers);
        builder.logger(LoggerPool.get(config.getLoggerName()));

        Object result;

        switch (config.getHttpMethod()) {
            case GET:
                processUrlArgument(reals, builder);
                builder.emit0();
                result = httpResolver.get();
                break;
            case DELETE:
                processUrlArgument(reals, builder);
                builder.emit0();
                result = httpResolver.delete();
                break;
            case POST:
                builder.payload(processBodyArgument(reals));
                builder.emit0();
                result = httpResolver.post();
                break;
            case PUT:
                builder.payload(processBodyArgument(reals));
                builder.emit0();
                result = httpResolver.put();
                break;
            default:
                throw new UnsupportedOperationException("http method only support get|post|delete|put");
        }
        return config.hasReturnType() ? result : null;
    }
```

此时我们可以使用以函数对象取代函数。这样我们的临时变量可作为新类的属性出现。

重构后-方法对象

```
public class RestfulExecutor {

    private HttpResolver httpResolver;

    private MethodRestConfig config;

    private Method method;

    private Object[] args;

    private Map<String, String> paths = new HashMap<>();
    private Map<String, String> headers = new HashMap<>();
    private Map<String, Object> reals = new HashMap<>();
    private Map<String, Object> urlParameters = new HashMap<>();
    private Parameter[] parameters;
    private PathConfigUrlAnalyzer pathConfigUrlAnalyzer = new PathConfigUrlAnalyzer();

    public RestfulExecutor(HttpResolver httpResolver,MethodRestConfig config,Method method,Object[] args){
        this.httpResolver = httpResolver;
        this.config = config;
        this.method = method;
        this.args = args;
        this.parameters = method.getParameters();
    }

    public Object execute() {
        RestConfig.RestConfigBuilder builder = httpResolver.custom();
        label:
        for (int index = 0; index < parameters.length; index++) {

            if (config.getFutureCallbackIndex() != null && config.getFutureCallbackIndex() == index) {
                builder.futureCallback((FutureCallback) args[index]);
                continue;
            }


            Parameter parameter = parameters[index];
            Annotation[] annotations = parameter.getAnnotations();
            for (Annotation annotation : annotations) {
                if (annotation instanceof Path) {
                    if (args[index] != null) {
                        Path path = (Path) annotation;
                        paths.put(StringUtils.hasText(path.value()) ? path.value() : config.getParameterName
(index), String.valueOf(args[index]));
                    }
                    continue label;
                }
                if (annotation instanceof Header) {
                    if (args[index] != null) {
                        Header header = (Header) annotation;

                        Object headerArgument = args[index];
                        if (headerArgument instanceof Map) {
                            ((Map) headerArgument).forEach((key, value) -> headers.put(String.valueOf(key),
String.valueOf(value)));
                        } else {
                            headers.put(StringUtils.hasText(header.value()) ? header.value() : config.
getParameterName(index), String.valueOf(args[index]));
                        }

                    }
                    continue label;
                }
                if (annotation instanceof Pair) {
                    Pair pair = (Pair) annotation;
                    urlParameters.put(StringUtils.hasText(pair.value()) ? pair.value() : config.getParameterName
(index), args[index]);
                    continue label;
                }
            }
            reals.put(config.getParameterName(index), args[index]);
        }

        processUrlArgument(urlParameters, builder);

        ContentType contentType = method.getDeclaredAnnotation(ContentType.class);
        if (contentType != null && StringUtils.hasText(contentType.value())) {
            builder.contentType(contentType.value());
        }

        builder.socketTimeout(config.getSocketTimeout());
        builder.connectionTimeout(config.getConnectionTimeout());
        builder.returnType(getTypeReference(config, method));
        builder.unpack(config.isUnpack());
        builder.url(pathConfigUrlAnalyzer.analyze(config.getUrlTemplate(), paths));
        builder.header(headers);
        builder.logger(LoggerPool.get(config.getLoggerName()));

        Object result;

        switch (config.getHttpMethod()) {
            case GET:
                processUrlArgument(reals, builder);
                builder.emit0();
                result = httpResolver.get();
                break;
            case DELETE:
                processUrlArgument(reals, builder);
                builder.emit0();
                result = httpResolver.delete();
                break;
            case POST:
                builder.payload(processBodyArgument(reals));
                builder.emit0();
                result = httpResolver.post();
                break;
            case PUT:
                builder.payload(processBodyArgument(reals));
                builder.emit0();
                result = httpResolver.put();
                break;
            default:
                throw new UnsupportedOperationException("http method only support get|post|delete|put");
        }
        return config.hasReturnType() ? result : null;
    }
}
```

重构后-原方法

```
@Override
    public Object intercept(Object o, Method method, Object[] args, MethodProxy methodProxy) {
        MethodRestConfig config = findMethodRestConfig(method);
        if (config == null) {
            throw new UnsupportedOperationException("can not find rest config of " + method.getDeclaringClass().
getSimpleName() + "." + method.getName());
        }
        HttpResolver httpResolver = config.isAsync() ? asyncHttpResolver : syncHttpResolver;

        RestfulExecutor executor = new RestfulExecutor(httpResolver,config,method,args);
        return executor.execute();

    }
```

思考：我们能不能将这些临时变量提取到当前类中？

经过这一次的重构。原来的过长方法是得到了解决，但是新增的RestfulExecutor的execute方法依旧还是是过长方法（临时变量问题已经不存在了）。现在导致过长方法的原因是for循环。我们将for循环拆开进行进一步的重构。

重构后

```
public class RestfulExecutor {

    private HttpResolver httpResolver;

    private MethodRestConfig config;

    private Method method;

    private Object[] args;

    private Map<String, String> paths = new HashMap<>();
    private Map<String, String> headers = new HashMap<>();
    private Map<String, Object> realArguments = new HashMap<>();
    private Map<String, Object> urlParameters = new HashMap<>();
    private Parameter[] parameters;

    private PathConfigUrlAnalyzer pathConfigUrlAnalyzer = new PathConfigUrlAnalyzer();

    private RestConfig.RestConfigBuilder builder;

    public RestfulExecutor(HttpResolver httpResolver,MethodRestConfig config,Method method,Object[] args){
        this.httpResolver = httpResolver;
        this.config = config;
        this.method = method;
        this.args = args;
        this.parameters = method.getParameters();
        this.builder = httpResolver.custom();
    }

        public Object execute(){
        resolveFutureCallback();
        resolvePath();
        resolveHeader();
        resolvePair();
        resolveArgument();

        resolveContextType();

        richBuilder();

        processUrlArgument(urlParameters);

        return doExecute();
    }
    private void resolveFutureCallback(){
        for (int index = 0; index < parameters.length; index++) {
            if (config.getFutureCallbackIndex() != null && config.getFutureCallbackIndex() == index) {
                builder.futureCallback((FutureCallback) args[index]);
                break;
            }
        }
    }

    private void resolvePath(){
        label:
        for (int index = 0; index < parameters.length; index++){
            for (Annotation annotation:parameters[index].getAnnotations()){
                if (annotation instanceof Path){
                    if (args[index] != null) {
                        Path path = (Path) annotation;
                        paths.put(StringUtils.hasText(path.value()) ? path.value() : config.getParameterName
(index), String.valueOf(args[index]));
                    }
                    continue label;
                }
            }
        }
    }

    private void resolveHeader(){
        label:
        for (int index = 0; index < parameters.length; index++){
            for (Annotation annotation:parameters[index].getAnnotations()){
                if (annotation instanceof Header){
                    if (args[index] != null) {
                        Header header = (Header) annotation;
                        Object headerArgument = args[index];
                        if (headerArgument instanceof Map) {
                            ((Map) headerArgument).forEach((key, value) -> headers.put(String.valueOf(key),
String.valueOf(value)));
                        } else {
                            headers.put(StringUtils.hasText(header.value()) ? header.value() : config.
getParameterName(index), String.valueOf(args[index]));
                        }

                    }
                    continue label;
                }
            }
        }
    }

    private void resolvePair(){
        label:
        for (int index = 0; index < parameters.length; index++){
            for (Annotation annotation:parameters[index].getAnnotations()){
                if (annotation instanceof Pair){
                    if (args[index]!=null){
                        Pair pair = (Pair) annotation;
                        urlParameters.put(StringUtils.hasText(pair.value()) ? pair.value() : config.
getParameterName(index), args[index]);
                    }
                    continue label;
                }
            }
        }
    }

    private boolean hasNoTargetAnnotation(Parameter parameter,List<Class<? extends Annotation>> classes){
        for (Class<? extends Annotation> clazz:classes){
            if (Stream.of(parameter.getAnnotations()).noneMatch(annotation -> annotation.annotationType().equals
(clazz))){
                return true;
            }
        }
        return false;
    }

    private void resolveArgument(){
        for (int index = 0;index<parameters.length;index++){
            if (hasNoTargetAnnotation(parameters[index], Arrays.asList(Path.class,Header.class,Pair.class))){
                realArguments.put(config.getParameterName(index), args[index]);
            }
        }
    }

    private void resolveContextType(){
        ContentType contentType = method.getDeclaredAnnotation(ContentType.class);
        if (contentType != null && StringUtils.hasText(contentType.value())) {
            builder.contentType(contentType.value());
        }
    }

    private void richBuilder(){
        this.builder = httpResolver.custom();
        builder.socketTimeout(config.getSocketTimeout());
        builder.connectionTimeout(config.getConnectionTimeout());
        builder.returnType(getTypeReference(config, method));
        builder.unpack(config.isUnpack());
        builder.url(pathConfigUrlAnalyzer.analyze(config.getUrlTemplate(), paths));
        builder.header(headers);
        builder.logger(LoggerPool.get(config.getLoggerName()));
    }

    private Object doExecute(){
        Object result;

        switch (config.getHttpMethod()) {
            case GET:
                processUrlArgument(realArguments);
                builder.emit0();
                result = httpResolver.get();
                break;
            case DELETE:
                processUrlArgument(realArguments);
                builder.emit0();
                result = httpResolver.delete();
                break;
            case POST:
                builder.payload(processBodyArgument(realArguments));
                builder.emit0();
                result = httpResolver.post();
                break;
            case PUT:
                builder.payload(processBodyArgument(realArguments));
                builder.emit0();
                result = httpResolver.put();
                break;
            default:
                throw new UnsupportedOperationException("http method only support get|post|delete|put");
        }
        return config.hasReturnType() ? result : null;
    }
}
```

我们将循环拆开会不会导致性能问题？代码的可读和性能就像鱼与熊掌二者不能兼得。我对性能的态度是，当性能问题真的发生了，成为了我们代码的主要矛盾时我们在去考虑他。在上述例子我们是循环方法参数，方法的参数不会多到影响性能。所以我们优先考虑代码的可读性。

依恋情节

对象技术的重点在于：这是一种"将数据和对数据的操作行为包装在一块"的技术。讲的简单点就是数据决定对象的行为，行为反过来作用于数据。有一种经典的坏味道是：函数对某个类的兴趣高于对自己所处类的兴趣。这种孺慕之情最通常的焦点便是数据。无数次经验里，我们看到某个类为了计算某个值，从另外一个对象那儿调用了几乎半打的取值函数。

依赖情节是对高内聚的破坏，如果某一个类拥有做一件事情的全部知识，那么我们就让这个类做这件事情。所以消除依恋情节，我们需要将方法提取到它应该在的地方。

案例一：Polaris计算字段是否必填

RequireResolve

```
  /**
     *
     *
     * @param field parameter
     */
    public void resolvedParameterField(Parameter field) {
        Boolean jsrRequired = resolveParameterFieldWithJsrAnnotation(field);
        if (jsrRequired != null) {
            field.setRequired(jsrRequired);
        } else {
            field.setRequired(resolveParameterFieldWithLabel(field));
        }
    }

    /**
     * JSR
     *
     * @param parameter
     * @return true: false null
     */
    private Boolean resolveParameterFieldWithJsrAnnotation(Parameter parameter) {
        JavaAnnotation jsrAnnotation = parameter.annotations()
                .stream()
                .filter(javaAnnotation -> jsrFieldAnnotations.contains(javaAnnotation.getType().
getFullyQualifiedName()))
                .findFirst()
                .orElse(null);
        if (jsrAnnotation == null) {
            return null;
        }
        JavaAnnotation validated = parameter.parent().findAnnotation(VALIDATED);
        if (validated == null) {
            return true;
        }
        List<String> validatedGroups = extractGroups(validated, "value");
        List<String> jsrGroups = extractGroups(jsrAnnotation, "groups");

        if (validatedGroups.isEmpty() && jsrGroups.isEmpty()) {
            return true;
        }

        validatedGroups.retainAll(jsrGroups);
        return !validatedGroups.isEmpty();
    }

    /**
     * required
     *
     * @param parameter
     * @return
     */
    private boolean resolveParameterFieldWithLabel(Parameter parameter) {
        List<Label> requiredLabels = parameter.labels(REQUIRED);

        Label emptyLabel = requiredLabels.stream().filter(Label::empty).findFirst().orElse(null);
        for (Label label : requiredLabels) {
            if (RuntimePolaris.methodFullName().equals(StringUtils.trim(label.getValue()))) {
                return true;
            }
        }
        return emptyLabel != null;
    }
```

上述代码逻辑是根据类属性上的注解和自定义标签计算该属性是否必填。该类频繁的从Parameter对象中获取数据进行计算。这就是依恋情节。消除这种坏味道我们可以将计算必填的逻辑迁移到Parameter对象中。

狎昵关系

有时你会看到两个类关系过于亲密，我们需要花费太多的时间去探究彼此之间的private成分。说的简单点就是相互依赖。修改这种情况我们可以将两个类依赖的部分提取到一个公共的类中。如果我们新增的类在概念上没有一个好的归属的话会导致随机抽象。这种抽象很弱。消除狎昵关系的另一种方法是隐藏委托，但是这在我们的系统中很难应用。我们在无法直接的消除狎昵关系时我们应该遵循依赖倒置原则，将这种对实现的依赖反转为对接口的依赖。

案例一：建站站点和表单模块之间的相互依赖

重构前-站点模块

```
public class SiteManager{
        @Resource
    private FormConfigManager formConfigManager;

        private void assembleSaveSitePlugin(Site site, String userId) {
        sitePluginRelaMapper.delete(SitePluginRelaDO.builder().siteId(site.getSiteId()).siteOnline(0).build());
        if (CollectionUtils.isNotEmpty(site.getPlugins())) {
            List<SitePluginRelaDO> siteInfos = Lists.newArrayListWithCapacity(site.getPlugins().size());
            site.getPlugins().forEach(relaReq -> {
                //
                if (relaReq.getPluginType() == SiteConstants.PluginRelaType.FROM && !formConfigManager.
formMatchUser(userId, relaReq.getPluginId())) {
                    throw new BaseException(PlatformErrorDefinition.COMMON_ERROR.getCode(), "");
                }

                if (relaReq.getPluginType() == SiteConstants.PluginRelaType.ADS_FORM_1 || relaReq.
getPluginType() == SiteConstants.PluginRelaType.ADS_FORM_100) {
                    User user = userManager.findOne(userId);
                    if (!adsFormClient.adsUserHasTheForm(user.getBsAccount(), relaReq.getPluginId())) {
                        throw new BaseException(PlatformErrorDefinition.COMMON_ERROR.getCode(), "");
                    }
                }
            });
            sitePluginRelaMapper.batchInsert(siteInfos);
        }
    }
}
```

重构前-表单模块

```
public class FormConfigManager{
        @Resource
    private SiteManager siteManager;

         public void deleteFormConfig(String userId, String formId) {
        findFormConfigWithMatch(userId, formId);
        if (siteManager.isOnlineSiteRelaForm(formId)) {
            throw new BaseException(FORM_CAN_NOT_DELETE);
        }
        formConfigManager.delete(userId, formId);
    }
}
```

重构后-OnlineSiteJudge

```
package com.vivo.internet.wukongplatform.service.form;

public interface BindingOnlineSiteJudge{

    /**
     *
     * @param formId id
     */
    boolean judge(String formId);
}
```

重构后-FormUserMatcher

```
package com.vivo.internet.wukongplatform.service.form;

public interface FormUserMatcher {

    /**
     *
     * @param userId id
     * @param formId id
     */
    boolean match(String formId,String userId);
}
```

重构后-站点模块

```
public class SiteManager implements BindingOnlineSiteJudge{
        @Resource
    private FormUserMatcher formUserMatcher;

        private void assembleSaveSitePlugin(Site site, String userId) {
        sitePluginRelaMapper.delete(SitePluginRelaDO.builder().siteId(site.getSiteId()).siteOnline(0).build());
        if (CollectionUtils.isNotEmpty(site.getPlugins())) {
            List<SitePluginRelaDO> siteInfos = Lists.newArrayListWithCapacity(site.getPlugins().size());
            site.getPlugins().forEach(relaReq -> {
                //
                if (relaReq.getPluginType() == SiteConstants.PluginRelaType.FROM && !formUserMatcher.match
(userId, relaReq.getPluginId())) {
                    throw new BaseException(PlatformErrorDefinition.COMMON_ERROR.getCode(), "");
                }

                if (relaReq.getPluginType() == SiteConstants.PluginRelaType.ADS_FORM_1 || relaReq.
getPluginType() == SiteConstants.PluginRelaType.ADS_FORM_100) {
                    User user = userManager.findOne(userId);
                    if (!adsFormClient.adsUserHasTheForm(user.getBsAccount(), relaReq.getPluginId())) {
                        throw new BaseException(PlatformErrorDefinition.COMMON_ERROR.getCode(), "");
                    }
                }
            });
            sitePluginRelaMapper.batchInsert(siteInfos);
        }
    }

        public boolean judge(String formId){
        }
}
```

重构后-表单模块

```
public class FormConfigManager implements FormUserMatcher {

        @Resource
    private BindingOnlineSiteJudge bindingOnlineSiteJudge;

         public void deleteFormConfig(String userId, String formId) {
        findFormConfigWithMatch(userId, formId);
        if (bindingOnlineSiteJudge.judge(formId)) {
            throw new BaseException(FORM_CAN_NOT_DELETE);
        }
        formConfigManager.delete(userId, formId);
    }

        public boolean match(String formId,String userId){

        }
}
```

switch惊悚

面向对象程序的一个最明显的特征就是少英switch语句。从本质上来说switch语句导致的问题在于重复。你经常会发现同样的switch语句散步在不同的地点。如果要为它添加一个新的case字句就必须找到所有的switch语句并修改他们。所有的switch都可以使用面向对象多态进行消除。

案例一：由switch语句导致的代码重复

重构前-SyncHttpResolver

```
package com.vivo.doraemon.rest.client;

public class SyncHttpResolver implements HttpResolver {

    private Object request(HttpMethod httpMethod) {
        try {
            RestConfig restConfig = this.threadLocal.get();
            restConfig.validate(httpMethod);
            String response;
            switch (httpMethod) {
                case GET:
                    response = syncHttpFacade.requestGet(restConfig);
                    break;
                case DELETE:
                    response = syncHttpFacade.requestDelete(restConfig);
                    break;
                case POST:
                    response = syncHttpFacade.requestPost(restConfig);
                    break;
                case PUT:
                    response = syncHttpFacade.requestPut(restConfig);
                    break;
                default:
                    throw new UnsupportedOperationException("http method only support get & post & delete &
put");
            }
            return restConfig.isUnpack() ? WrapHelper.unpack(response, restConfig.getTypeReference())
                    : WrapHelper.directWrap(response, restConfig.getTypeReference());
        } finally {
            threadLocal.remove();
        }
    }
}
```

重构前-AsyncHttpRsolver

```
package com.vivo.doraemon.rest.client;

public class AsyncHttpResolver implements HttpResolver {

    private AsyncFuture<Object> request(HttpMethod httpMethod) {
        try {
            RestConfig restConfig = this.threadLocal.get();
            restConfig.validate(httpMethod);
            AsyncFuture<Object> asyncFuture;

            switch (httpMethod) {
                case GET:
                    asyncFuture = asyncHttpFacade.requestGet(restConfig, getHttpResponseDecoder(restConfig), 
restConfig.getFutureCallback());
                    break;
                case DELETE:
                    asyncFuture = asyncHttpFacade.requestDelete(restConfig, getHttpResponseDecoder(restConfig), 
restConfig.getFutureCallback());
                    break;
                case POST:
                    asyncFuture = asyncHttpFacade.requestPost(restConfig, getHttpResponseDecoder(restConfig), 
restConfig.getFutureCallback());
                    break;
                case PUT:
                    asyncFuture = asyncHttpFacade.requestPut(restConfig, getHttpResponseDecoder(restConfig), 
restConfig.getFutureCallback());
                    break;
                default:
                    throw new UnsupportedOperationException("http method only support get and post");
            }
            return asyncFuture;
        } finally {
            threadLocal.remove();
        }
    }
}
```

重构后类较多，这里就不一一贴出来，只展示部分代码

重构后-AbstractHttpResolver

```java
package com.vivo.doraemon.rest.client;

import com.vivo.doraemon.rest.client.executor.HttpMethodExecutor;

public abstract class AbstractHttpResolver implements HttpResolver {

    private final ThreadLocal<RestConfig> threadLocal = new ThreadLocal<>();

    public RestConfig.RestConfigBuilder custom() {
        return RestConfig.custom(this);
    }

    public Object post() {
        return request(HttpMethod.POST);
    }

    public Object put() {
        return request(HttpMethod.PUT);
    }

    public Object get() {
        return request(HttpMethod.GET);
    }

    public Object delete() {
        return request(HttpMethod.DELETE);
    }

    private Object request(HttpMethod httpMethod) {
        try {
            threadLocal.get().validate(httpMethod);
            return getHttpMethodExecutor(httpMethod).execute(threadLocal.get());
        } finally {
            threadLocal.remove();
        }
    }

    protected abstract HttpMethodExecutor getHttpMethodExecutor(HttpMethod httpMethod);

    @Override
    public void emit(RestConfig restConfig) {
        this.threadLocal.set(restConfig);
    }

}
```

重构后-AsyncHttpResolver

```
package com.vivo.doraemon.rest.client;

import com.vivo.doraemon.rest.client.executor.DeleteAsyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.GetAsyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.HttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.PostAsyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.PutAsyncHttpMethodExecutor;

import java.util.HashMap;
import java.util.Map;

public class AsyncHttpResolver extends AbstractHttpResolver {

    private final Map<HttpMethod, HttpMethodExecutor> map = new HashMap<>();

    public AsyncHttpResolver(AsyncHttpFacade asyncHttpFacade) {
        map.put(HttpMethod.GET,new GetAsyncHttpMethodExecutor(asyncHttpFacade));
        map.put(HttpMethod.POST,new PostAsyncHttpMethodExecutor(asyncHttpFacade));
        map.put(HttpMethod.PUT,new PutAsyncHttpMethodExecutor(asyncHttpFacade));
        map.put(HttpMethod.DELETE,new DeleteAsyncHttpMethodExecutor(asyncHttpFacade));
    }

    @Override
    protected HttpMethodExecutor getHttpMethodExecutor(HttpMethod httpMethod) {
        HttpMethodExecutor httpMethodExecutor = map.get(httpMethod);
        if (httpMethodExecutor == null){
            throw new UnsupportedOperationException("async http method only support get|post|put|delete");
        }
        return httpMethodExecutor;
    }
}
```

重构后-SyncHttpResolver

```
package com.vivo.doraemon.rest.client;

import com.vivo.doraemon.rest.client.executor.DeleteSyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.GetSyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.HttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.PostSyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.executor.PutSyncHttpMethodExecutor;

import java.util.HashMap;
import java.util.Map;

public class SyncHttpResolver extends AbstractHttpResolver {

    private final Map<HttpMethod, HttpMethodExecutor> map = new HashMap<>();

    public SyncHttpResolver(SyncHttpFacade syncHttpFacade) {
        map.put(HttpMethod.GET,new GetSyncHttpMethodExecutor(syncHttpFacade));
        map.put(HttpMethod.POST,new PostSyncHttpMethodExecutor(syncHttpFacade));
        map.put(HttpMethod.PUT,new PutSyncHttpMethodExecutor(syncHttpFacade));
        map.put(HttpMethod.DELETE,new DeleteSyncHttpMethodExecutor(syncHttpFacade));
    }

    @Override
    protected HttpMethodExecutor getHttpMethodExecutor(HttpMethod httpMethod) {
        HttpMethodExecutor httpMethodExecutor = map.get(httpMethod);
        if (httpMethodExecutor == null){
            throw new UnsupportedOperationException("http method only support get|post|put|delete");
        }
        return httpMethodExecutor;
    }
}
```

重构后-HttpMethodExecutor

```
package com.vivo.doraemon.rest.client.executor;

import com.vivo.doraemon.rest.client.RestConfig;

public interface HttpMethodExecutor {

    /**
     *
     * @param restConfig
     */
    Object execute(RestConfig restConfig);


}
```

重构后-AsyncHttpMethodExecutor

```
package com.vivo.doraemon.rest.client;

import com.vivo.doraemon.exception.BaseException;
import com.vivo.doraemon.exception.Exceptions;
import com.vivo.doraemon.rest.client.executor.HttpMethodExecutor;
import com.vivo.doraemon.rest.log.RestLog;
import org.apache.http.Consts;
import org.apache.http.util.EntityUtils;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;

public abstract class AsyncHttpMethodExecutor implements HttpMethodExecutor {

    private Logger log = LoggerFactory.getLogger(AbstractHttpResolver.class);

    @Override
    public Object execute(RestConfig restConfig) {
        return doExecute(restConfig, (httpResponse, trace) -> {
            try {
                int statusCode = httpResponse.getStatusLine().getStatusCode();

                String body = EntityUtils.toString(httpResponse.getEntity(), Consts.UTF_8);

                if (restConfig.getLogger().isDebugEnabled()) {
                    restConfig.getLogger().debug(new RestLog(trace, restConfig.getUrl(), statusCode, body).
toString());
                }

                if (statusCode != HttpStatus.OK.value()) {
                    String statusText = httpResponse.getStatusLine().getReasonPhrase();
                    throw new NotOkStatusException(statusCode, statusText);
                }
                return restConfig.isUnpack() ? WrapHelper.unpack(body, restConfig.getTypeReference()) :
WrapHelper.directWrap(body, restConfig.getTypeReference());

            } catch (Exception e) {
                if (e instanceof BaseException && restConfig.getFutureCallback() != null) {
                    try {
                        restConfig.getFutureCallback().businessFailed(restConfig.getUrl(), (BaseException) e);
                    } catch (Exception ex) {
                        throw Exceptions.sneakyThrow(ex);
                    }
                }
                log.error("{}", restConfig.getUrl(), e);
                throw Exceptions.sneakyThrow(e);
            }
        });
    }

    protected abstract AsyncFuture<Object> doExecute(RestConfig restConfig, HttpResponseDecoder decoder);
}
```

重构后-GetAsyncHttpMethodExecutor

```
package com.vivo.doraemon.rest.client.executor;

import com.vivo.doraemon.rest.client.AsyncFuture;
import com.vivo.doraemon.rest.client.AsyncHttpFacade;
import com.vivo.doraemon.rest.client.AsyncHttpMethodExecutor;
import com.vivo.doraemon.rest.client.HttpResponseDecoder;
import com.vivo.doraemon.rest.client.RestConfig;

public class GetAsyncHttpMethodExecutor extends AsyncHttpMethodExecutor {

    private AsyncHttpFacade asyncHttpFacade;

    public GetAsyncHttpMethodExecutor(AsyncHttpFacade asyncHttpFacade) {
        this.asyncHttpFacade = asyncHttpFacade;
    }

    @Override
    protected AsyncFuture<Object> doExecute(RestConfig restConfig, HttpResponseDecoder decoder) {
        return asyncHttpFacade.requestGet(restConfig, decoder);
    }
}
```

发散式变化

对于软件的修改，我们希望能够跳到系统的某一点，只在该处做修改。如果某个类经常因为不同的方向上发生变化，那么就出现发散式修改。

案例一：悟空建站页面检测，根据宙斯提供的元数据并结合页面的插件数给页面计算一个分值并给出整改建议

重构前

```
public class SiteScoreManager {

    /**
     *
     *
     * @param overview
     * @param resources
     */
    public void calcScoreAndDeductionDetails(SiteDetectionDetail detail, Overview overview, List<Resource>
resources) {
        //10
        double totalPoints = 10.0;

        List<DeductionDetail> deductionDetails = new ArrayList<>();

        //1.3s1
        if (null != overview.getWst() && DetectionParseUtils.milliToSecondScale2(overview.getWst()) > 1.3) {
            totalPoints -= 1;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.WHITE_SCREEN_TIME.getName())
                    .deductionValue(1D)
                    .type(DeductionTypeEnum.PAGE.getName())
                    .suggestion("" + DetectionParseUtils.milliToSecondScale2(overview.getWst())
                            + "1.3")
                    .build());
        }

        ///1.8s0.10.11
        if (null != overview.getFpt() && DetectionParseUtils.milliToSecondScale2(overview.getFpt()) > 1.8) {
            double deducted = deductNum(
                    MathUtils.div(MathUtils.reduce(DetectionParseUtils.milliToSecondScale2(overview.getFpt()),
1.8), 0.1),
                    0.1, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.FIRST_RENDERING_TIME.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.PAGE.getName())
                    .suggestion("" + DetectionParseUtils.milliToSecondScale2(overview.getFpt())
                            + "1.8")
                    .build());
        }

        //1.3X1.3s0.4
        List<Resource> overLoadedResources = getOverLoadedResources(resources);
        if (overLoadedResources.size() > 0) {
            double deducted = deductNum(overLoadedResources.size(), 0.4, false);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.RESOURCE_LOAD_TIME.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.RESOURCE.getName())
                    .suggestion("1.3" + overLoadedResources.size() + "")
                    .resources(parseResources(overLoadedResources))
                    .build());
        }

        //X250.0751
        List<Resource> imagesResources = getTypeSpecificResources(resources, ResourceTypeEnum.IMG.getName());
        if (imagesResources.size() > 25) {
            double deducted = deductNum(imagesResources.size() - 25, 0.075, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.IMAGE_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.IMAGE.getName())
                    .suggestion("" + imagesResources.size() + "25")
                    .build());
        }

        //230KBX0.21
        List<Resource> overSizedImages = getOverSizedImages(imagesResources);
        if (overSizedImages.size() > 0) {
            double deducted = deductNum(overSizedImages.size(), 0.2, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.IMAGE_OVERSIZE_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.IMAGE.getName())
                    .suggestion("230KB" + overSizedImages.size() + "")
                    .resources(parseResources(overSizedImages))
                    .build());
        }

        //AjaxX50.11
        List<Resource> ajaxResources = getTypeSpecificResources(resources, ResourceTypeEnum.AJAX.getName());
        if (ajaxResources.size() > 5) {
            double deducted = deductNum(ajaxResources.size() - 5, 0.1, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.AJAX_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.AJAX.getName())
                    .suggestion("Ajax" + ajaxResources.size() + "5")
                    .build());
        }

        //X0.31
        if (null != overview.getRedirectCount() && overview.getRedirectCount() > 0) {
            double deducted = deductNum(overview.getRedirectCount(), 0.3, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.REDIRECT_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.PAGE.getName())
                    .suggestion("" + overview.getRedirectCount() + "")
                    .build());
        }

        //X80.11
        List<Resource> jsResources = getTypeSpecificResources(resources, ResourceTypeEnum.JS.getName());
        if (jsResources.size() > 8) {
            double deducted = deductNum(jsResources.size() - 8, 0.1, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.SCRIPT_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.SCRIPT.getName())
                    .suggestion("" + jsResources.size() + "8")
                    .build());
        }

        //X80.11
        if (null != detail.getPluginCount() && detail.getPluginCount() > 8) {
            double deducted = deductNum(detail.getPluginCount() - 8, 0.1, true);
            totalPoints -= deducted;

            deductionDetails.add(DeductionDetail.builder()
                    .deductionItem(DeductionItemEnum.PLUGIN_COUNT.getName())
                    .deductionValue(deducted)
                    .type(DeductionTypeEnum.PAGE.getName())
                    .suggestion("" + detail.getPluginCount() + "8")
                    .build());
        }
        //0
        detail.setScore(Math.max(MathUtils.round(totalPoints, 1), 0));
        detail.setDeductionDetails(deductionDetails);
    }
}
```

上述代码从直观商第一感觉是过长方法，但是我们无法使用过长方法的消除手段去重构它。深入分析，上述方法的坏味道应该是关注点过多

上面这段代码至少有四个原因会引起其修改：

1、增加一种新的扣分规则

2、根据插件的扣分规则发生变化

3、根据概览的扣分规则发生变化

4、根据资源的扣分规则发生变化

重构代码较多，这里只展示原先方法

重构后

```
public void calcScoreAndDeductionDetails(String url,SiteDetectionDetail detail, Overview overview,
List<Resource> resources) {

        List<LinkScoreComputer> computers = new ArrayList<>();
        computers.add(new ZeusOverviewLinkScoreComputer(overview.getWst(),overview.getFpt(),overview.
getRedirectCount()));
        computers.add(new ZeusResourcesLinkScoreComputer(resources));
        computers.add(new PluginLinkScoreComputer(siteManager.revertSiteId(url),pluginUsageManager));

        List<ComputeResult> computeResults = computers
                .stream()
                .flatMap(computer -> computer.compute().stream())
                .collect(Collectors.toList());

        double deduct = computeResults
                .stream()
                .mapToDouble(ComputeResult::getScore)
                .sum();

        List<DeductionDetail> deductionDetails = computeResults
                .stream()
                .map(computeResult ->
                    DeductionDetail
                            .builder()
                            .deductionItem(computeResult.getItem().getName())
                            .type(computeResult.getType().getName())
                            .deductionValue(computeResult.getScore())
                            .suggestion(computeResult.getSuggestion())
                            .build()
                )
                .collect(Collectors.toList());

        detail.setScore(Math.max(MathUtils.round(deduct, 1), 0));
        detail.setDeductionDetails(deductionDetails);
    }
```

我们平时做接口设计时，受到spring框架的影响太大。我们经常会把需要的参数全部放到参数列表中。有的参数可能只会在某一个实现中用到。这样的接口设计是及其脆弱的，要么参数列表过长，那么掉进状态的深渊中，很容易崩溃。我们应该关注接口的行为，而不是数据，除非这个数据是描述接口行为必不可少的。

过大的类

危害

过大的类最直接的危害就是这个类很难别阅读和理解。甚至导致不遵循单一职责原则。

以悟空建站SiteManager为例，该类有1000多行代码，无论该类是否遵循单一职责原则，我们都需要重构他。由于该模块是建站最核心的模块。我们这里不给出具体的重构方法。

过长参数列表

危害

1、过长的参数列表直接导致过长方法

2、过长的参数列表导致方法不可测，或者方法难以被测试

散弹式修改

散弹式修改类似发散式变化，但是两者是正好相反的。发散式变化是一个类受到多种变化的影响，是对单一职责的破快。而散弹式修改是指一种变化引发了多个类的修改。这是一种低内聚的表现。

启后

本篇主要介绍了重构的一些基本概念和常见的代码坏味道及消除手段，作为重构的第一篇。后续可就重构做更加深入的交流与学习