

Glory

简介

glory-client

[ClientService](#)

[MockObserver](#)

glory-data

[Audit审计](#)

[动态字段](#)

[统一父类](#)

glory-foundation

[数据安全](#)

[数据加密](#)

[日志脱敏](#)

[Format格式化](#)

简介

Glory代表光辉岁月的意思，旨在总结日常开发中遇到的一些典型问题，提取解决方案，经抽象和完善后形成一系列的功能性框架。Glory着重于具体问题的解决方案，如微服务调用、数据安全、数据处理等领域，技术只是工具，解决问题的思路才是根本。

glory-client

软件开发中，接口调用是很平常的事，特别是微服务，App间调用是非常频繁的，当维护的API接口增多时，如何优雅的调用，及代码维护是需要面对的难题。在微服务中，接口调用常用RestTemplate和FeignClient两种方式，这里提供一种基于RestTemplate的接口调用解决方案。

ClientService

```

1 public abstract class AbstractClientService implements ClientService {
2
3     protected List<HttpRequestEnricher> enrichers; //请求头丰富器, 添加自定义请
    求头
4     @Value("${glory.client.http.log:true}")
5     private boolean logging; //是否记录请求响应报文日志, 默认记录Info日志
6
7     @Override
8     public <T, S> T exchange(HttpRequestWrapper<T, S> wrapper) {
9         Stopwatch watch = new Stopwatch();
10        watch.start();
11        T response = null;
12        enrichHttpRequest(wrapper); //enrich the http request header
13        HttpEntity<S> entity = wrapper.getHttpEntity();
14        RestTemplate restTemplate = getRestTemplate(wrapper); //get effecti
    ve restTemplate, wrapper first.
15        Assert.notNull(restTemplate, "Client restTemplate is null");
16        String url = wrapper.getUrlWrapper().formatUrl(env);
17        logging(true, url, wrapper);
18        ResponseEntity<T> responseEntity = restTemplate
19            .exchange(url, wrapper.getMethod(), entity,
20                ParameterizedTypeReference.forType(wrapper.getResponse
    seClass()),
21                wrapper.getUrlWrapper().getUriVariables());
22        if (responseEntity.getStatusCode() == HttpStatus.OK && responseEnt
    ity.hasBody()) {
23            wrapper.setResult(responseEntity.getBody());
24            response = responseEntity.getBody();
25        }
26        logging(false, url, wrapper);
27        watch.stop();
28        logger.debug("Request[{}] cost time:{}", url, watch.getTotalTimeMi
    llis());
29        return response;
30    }
31    .....
32 }}
33 }

```

抽象类把调用过程中常遇到的问题进行了归纳整理后, 规范RestTemplate调用。HttpRequestWrapper封装了Request常用用到的信息, 包括Host、Uri、Header和RequestBody等内容。

```

1  public class CompensateProcessRemoteImpl extends AbstractClientService
2      implements CompensateProcessRemoteService {
3      @Value("${spark.http.compensate.process:/compensate/item/process}")
4      private String doProcessCompensateUri;//API Uri
5      @Override
6      public CompensateResponse process(CompensateItem item) {
7          UrlWrapper urlWrapper = UrlWrapper.createByAppName(item.getAppName
8              ())
9              .setUri(doProcessCompensateUri)//调用Uri
10             .setHostKey(SPARK_HTTP_HOST);//指定Host,
11             HttpRequestWrapper<CompensateResponse,CompensateItem> wrapper = Ht
12             tpRequestWrapper.
13             create(item,CompensateResponse.class).setUrlWrapper(urlWrapper);
14             return post(wrapper);
15     }
16 }

```

MockObserver

```

1  @Retention(RetentionPolicy.RUNTIME)
2  @Target(ElementType.METHOD)
3  public @interface MockObserver {
4      String condition() default "";//Mock激活条件, 由Env property控制
5      Class<? extends MockCondition> conditionClass()
6          default MockCondition.DefaultCondition.class;//Mock激活条件, MockCo
7          ndition
8      boolean ifMissing() default false;//条件缺失时, 默认不激活
9      String mocker() default "clientMocker";//Mock结果产生类名
10     Class<? extends ClientMocker> mockerClass()
11         default ClientMocker.DefaultClientMocker.class;//Mock结果产生类
12 }
13 public interface MockCondition {//Mock条件判断, 根据方法参数
14     boolean match(Method method, Object[] argus);
15 }
16
17 public interface ClientMocker {//Mock结果产生
18     <T> T mock(Object[] argus, Class<T> responseType);
19 }

```

应用场景：一个完整系统是由非常多的模块构成，某个模块需要暂停使用，或者是系统对外部有依赖，当外部因素不稳定，为保持系统稳定而需要临时关闭对外部依赖。上述场景在软件开发、测试过程经常会遇到，修改代码，不仅会造成代码污染，甚至可能会引起其他bug，MockObserver的引入，最大的优势在于不改变正常代码的前提下，实现对目标方法的替代。

- 激活条件

condition、conditionClass和ifMissing：condition和conditionClass任一即可，conditionClass优先。ifMissing表示在条件缺失时的默认结果，默认不激活

- Mock结果

mock和mockClass：生成Mock结果，任意一个即可，mockClass优先。

注意：Mock只对Spring注入的方法有效，正常方法调用时无效。

Mock示例

Java

```
1 public class DefaultSparkLauncher extends AbstractLauncher{
2     ...
3     @Override
4     @MockObserver(condition = "spark.launcher.disabled")
5     public <E, T> SparkResult<E> transmit(@Nonnull SparkContext<T> context)
6     {
7         ...
8     }
9 }
```

当spark.launcher.disabled=true时，激活Mock

glory-data

DB是一款软件不可缺少的部分，业务数据也需要及时落表。随着业务发展，业务数据模型的复杂度也会随之升高，glory-data模块提供了一种解决复杂数据模型的方案，为统一建模提花了基础。

Audit审计

业务相关的表通常要求具有审计功能，即每条记录都需要知道变更人和时间两个要素。这里提供了一个抽象父类，为每张表默认增加4个审计属性字段，并在保存时，自动更新这几个字段内容。

```
1  @MappedSuperclass
2  public abstract class DomainEntity implements DomainAware, AuditSupport {
3      @Column(name = "create_by", nullable = false)//创建人
4      private long createBy;
5      @Column(name = "modified_by", nullable = false)//最近修改人
6      private long modifiedBy;
7      @Column(name = "create_time", nullable = false)//创建时间
8      @Temporal(TemporalType.TIMESTAMP)
9      private LocalDateTime createTime;
10     @Column(name = "modified_time", nullable = false)//最近修改时间
11     @Temporal(TemporalType.TIMESTAMP)
12     private LocalDateTime modifiedTime;
13     .....
14 }
```

动态字段

业务大变化，模型也常伴随着变化，如何保持数据库稳定也是要面对的问题。

```
1  @MappedSuperclass
2  public abstract class DomainEntityWithDynamicFields extends DomainEntity
3      implements DynamicFieldSupport {
4      @Column(name = "dynamic_fields")
5      @Convert(converter = WithTypeMapConverter.class)
6      private Map<String, Object> _dynamicFields = new HashMap<>();
7
8      @JsonAnyGetter
9      public Map<String, Object> getDynamicFields() {
10         return _dynamicFields;
11     }
12     .....
13     @JsonAnySetter
14     @Override
15     public void setFieldValue(@NonNull String key, Object value) {
16         if (this._dynamicFields.containsKey(key)){
17             this._dynamicFields.put(key, value);
18         }
19     }
20 }
```

数据库表中没有的字段，默认统一存到"dynamic_fields"中，在Json转换中通过@JsonAnyGetter和@JsonAnySetter两个注解对动态字段进行拆箱和封箱。

统一父类

具有了统一父类后，框架在处理模型时，可以进行一些高层次的考虑。

glory-foundation

Foundation是一个功能性包，由众多小模块构成，将会持续维护。

数据安全

技术不断进步，信息安全也越来越重要，特别是敏感信息泄露，会对个人以及社会造成非常大的负面影响，乃至重大损失。系统正式上线运营，安全是不可绕过去的一道墙，数据加密则是信息安全的最后一道防火墙。通常数据泄的途径：API接口报文和数据库。

- API接口

随着网络爬虫泛滥，通过API爬取数据成为常见攻击手段，而防御手段手段之一对接口中敏感数据进行加密，让对方即便拿到数据，但是不能用。

- 数据库

数据泄露的另一个重灾区就是数据库，所以存入DB中的敏感数据也需要加密。

数据加密

- 加密方式

加密方法有很多，常用的几种加密包括：可逆加密AES、DES、RSA和不可逆MD5、Base64等。

▼ Secret定义

Java

```
1  @JacksonAnnotationsInside
2  @Retention(RetentionPolicy.RUNTIME)
3  @Target({ElementType.FIELD,ElementType.PARAMETER,ElementType.METHOD})
4  @JsonSerialize(using = SecretSerializer.class)
5  @JsonDeserialize(using = SecretDeserializer.class)
6  public @interface Secret {
7      String algorithm() default CryptoConstant.CRYPTO_RW_AES;//加密方式，AES、
      DES等
8  }
```

```
1 public class CryptoObject {
2
3     @Secret //默认AES加密
4     private String password;
5     .....
6     @Secret(algorithm = CryptoConstant.CRYPTO_RW_RSA)//RSA加密
7     public String getPhone() {
8         return phone;
9     }
10
11     .....
12 }
13 //对请求参数解密
14 @GetMapping("/para/{text}")
15 public CryptoResult testPara(@Secret @PathVariable("text") String text
16 ){
17     return CryptoResult.create(text);
18 }
19 //数据库属性加密
20 @Convert(converter = SecretSystemStringConverter.class)
21 private String phone;
```

注意，适用加密字段的类型须是字符串类型。

- 扩展加密

若是提供的加密方式不能满足需求，可参照已有的实现方法定制扩展，例如AES：

```
1 public class AesCryptoConfiguration {
2     private final Logger logger = LoggerFactory.getLogger(getClass());
3     private static final String ALGORITHM = "AES";
4     @Value("${glory.crypto.aes.key:12345690ABCDEFKDFJKSDF0DSKFKKSD}")
5     private String secretKey;
6     @Value("${glory.crypto.aes.iv:1234567890ABCDEF}")
7     private String iv;
8
9     @Bean("aesEncryptor")
10    @ConditionalOnMissingBean(name = "aesEncryptor")
11    public Encryptor<String> aesEncryptor(){
12        return str->{
13            if (StringUtils.hasLength(str)){
14                try{
15                    Cipher cipher = getCipher(Cipher.ENCRYPT_MODE);
16                    byte[] bytes = cipher.doFinal(str.getBytes(StandardCharsets.UTF_8));
17                    return Base64.getEncoder().encodeToString(bytes);
18                } catch (Throwable t){
19                    logger.warn(">> AES encrypt[{}] exception:",str,t);
20                    throw new RuntimeException(t);
21                }
22            }
23            return str;
24        };
25    }
26
27    @Bean("aesDecryptor")
28    @ConditionalOnMissingBean(name = "aesDecryptor")
29    public Decryptor aesDecryptor(){
30        return ciphertext -> {
31            if (StringUtils.hasLength(ciphertext)){
32                try{
33                    Cipher cipher = getCipher(Cipher.DECRYPT_MODE);
34                    byte[] bytes = cipher.doFinal(Base64.getDecoder().decode(ciphertext));
35                    return new String(bytes,StandardCharsets.UTF_8);
36                } catch (Throwable t){
37                    logger.warn(">> AES decrypt[{}] exception:",ciphertext,t);
38                    throw new RuntimeException(t);
39                }
40            }
41            return ciphertext;
42        };
43    }
44 }
```

```

43     }
44
45     .....
46 }

```

▼ 加密接口实现与调用

Java

```

1  @Component
2  ▼ public class AesCrypto implements Crypto<String> {
3      @Resource(name = "aesEncryptor")
4      private Encryptor encryptor;
5      @Resource(name = "aesDecryptor")
6      private Decryptor decryptor;
7
8      @Override
9  ▼ public String algorithm() {
10         return CryptoConstant.CRYPTO_RW_AES;
11     }
12
13     @Override
14  ▼ public String decrypt(String ciphertext) {
15         return decryptor.decrypt(ciphertext);
16     }
17
18     @Override
19  ▼ public String encrypt(String str) {
20         return encryptor.encrypt(str);
21     }
22 }
23 //加密
24 CryptoHelper.encrypt(text,algorithm)
25 //解密
26 CryptoHelper.decrypt(ciphertext,algorithm)

```

• 常用配置

glory.secret.algorithm.default:AES //系统默认加密方式

• AES:

glory.crypto.aes.key:12345690ABCDEFKDFJKSDFODSKFKKSD

glory.crypto.aes.iv:1234567890ABCDEF

• DES

glory.crypto.des.key:1234ABCD

glory.crypto.des.iv:ABCD1234

- Hmac-sha256

glory.crypto.hmac-sha256.key:1234ABCD

glory.crypto.hmac-sha256.formatHex:false //密文是否是16进制字符串，默认Base64

- MD5

glory.crypto.md5.formatHex:false//密文是否是16进制字符串，默认Base64

- sha256

glory.crypto.sha256.formatHex:false//密文是否是16进制字符串，默认Base64

- RSA

glory.secret.rsa.private.key //私钥key

glory.secret.rsa.private.file//私钥File，指定文件路径

glory.secret.rsa.public.key//公钥key

glory.secret.rsa.public.file //公钥file，指定文件路径

日志脱敏

系统日志中常包括一些敏感信息，日志泄露也是一个数据安全重灾区，日志中常包含敏感信息。所以数据安全要求日志中的敏感信息需要打掩码，甚至不要记录敏感信息。这里提供两种日志脱敏方式

- 注解

脱敏注解

Java

```
1  @JacksonAnnotationsInside
2  @Target({ElementType.FIELD})
3  @Retention(RetentionPolicy.RUNTIME)
4  @JsonSerialize(using = DesensitizeSerializer.class)
5  public @interface Desensitize {
6      String algorithm() default "PattenMask";//脱敏也是一种加密方式
7      String patten() default ""; //正则表达式
8      String replacement() default ""; //正则匹配后替换字符串
9      int minLength() default 4;//最小长度，字符串大于这个长度，正则脱敏，否则全掩码
    处理
10     boolean withPrefix() default false;//加密时，是否带前缀
11 }
```

```

1 public class CryptoObject {
2
3     @Desensitize
4     private String name;
5 }

```

- 配置脱敏规则

在配置文件中配置脱敏规则，该方案需要在ObjectMapper初始化时，增加一些配置。

```

1 private static void initDesensitize(ObjectMapper mapper){
2     SimpleBeanPropertyFilter filter = new DesensitizePropertyFilter();
3     mapper.setAnnotationIntrospector(new DesensitizeAnnotationIntrospector());
4     Map<Class<?>,Class<?>> mixins = new HashMap<>(8);
5     mixins.put(Object.class,DesensitizeFilterView.class);
6     mapper.setMixIns(mixins);
7     mapper.setFilterProvider(new SimpleFilterProvider()
8         .addFilter("DESENSITIZE_FILTER_NAME",filter));
9 }
10
11 @JsonFilter("DESENSITIZE_FILTER_NAME")
12 interface DesensitizeFilterView{}

```

配置规则方式，对所有使用ObjectMapper的序列化的对象都有效。

- 脱敏规则

glory.desensitize.rules[n].patten //正则表达式

glory.desensitize.rules[n].replacement /替换字符串

glory.desensitize.rules[n].names //属性名，包括Map中的key

glory.desensitize.rules[n].excludes //排除属性名，不序列这些内容

glory.desensitize.rules[n].min-length //掩码时，设定最小长度，小于这个长度直接全掩码

glory.desensitize.rules[n].with-prefix //加密操作时，是否带有前缀，如{AES}

glory.desensitize.rules[n].algorithm //默认是PattenMask，表示掩码，也可指定其他方式

```
1  glory:
2    desensitize:
3      rules[0]:
4        patten: ^(.{2}).*$
5        replacement: $1*****
6        names: test
7        excludes: case
8      rules[1]:
9        patten: ^(.{3}).*$
10       replacement: $1###
11       names: case2
12       min-length: 4
13       with-prefix: true
14     rules[2]:
15       algorithm: AES
16       patten: ^(.{1}).*$
17       replacement: $1*****
18       names: name
```

AppContext.set("DESENSITIZE_ACTIVITY",false);这是一个线程级的动态开关，可通过设置false来关闭日志脱敏功能，默认是开启。略扩散一下思维，通过请求Header来控制当前请求是否日志脱敏。

logback文件配置

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration>
3    <conversionRule conversionWord="msg"
4      converterClass="com.glory.foundation.desensitize.logback.DesensitizeConverter" />
5    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
6      <encoder>
7        <pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg{msg}%n</pattern>
8      </encoder>
9    </appender>
10   <root level="info">
11     <appender-ref ref="CONSOLE" />
12   </root>
13 </configuration>
```

需要在xml 中增加conversionRule配置，并且

converterClass="com.glory.foundation.desensitize.logback.DesensitizeConverter"，在打印日志时，会自动对敏感信息进行脱敏。

▼ 示例

XML

```
1 public CryptoResult testLogback(CryptoObject req){
2     logger.info(">> request ={}",test [{}]",req,req.getPassword());
3     //CryptoObject对象会转换成Json字符串打印，并脱敏
4     return CryptoResult.create(req.getPassword());
5 }
```

Format格式化

提供一个可扩展格式化工具类，用来处理日期和Json模式化，具体可参见FormatHelper类及ObjectFormater

和DateFormater两个接口。