

15 | 序列化：一来一回你还是原来的你吗？

2020-04-14 朱晔

Java业务开发常见错误100例

[进入课程 >](#)




讲述：王少泽

时长 22:20 大小 20.45M



你好，我是朱晔。今天，我来和你聊聊序列化相关的坑和最佳实践。

序列化是把对象转换为字节流的过程，以方便传输或存储。反序列化，则是反过来把字节流转换为对象的过程。在介绍 [文件 IO](#) 的时候，我提到字符编码是把字符转换为二进制的过程，至于怎么转换需要由字符集制定规则。同样地，对象的序列化和反序列化，也需要由序列化算法制定规则。

关于序列化算法，几年前常用的有 JDK (Java) 序列化、XML 序列化等，但前者不能  言，后者性能较差（时间空间开销大）；现在 RESTful 应用最常用的是 JSON 序列化，追求性能的 RPC 框架（比如 gRPC）使用 protobuf 序列化，这 2 种方法都是跨语言的，而且性能不错，应用广泛。

在架构设计阶段，我们可能会重点关注算法选型，在性能、易用性和跨平台性等中权衡，不过这里的坑比较少。通常情况下，序列化问题常见的坑会集中在业务场景中，比如 Redis、参数和响应序列化反序列化。

今天，我们就一起聊聊开发中序列化常见的一些坑吧。

序列化和反序列化需要确保算法一致

业务代码中涉及序列化时，很重要的一点是要确保序列化和反序列化的算法一致性。有一次我要排查缓存命中率问题，需要运维同学帮忙拉取 Redis 中的 Key，结果他反馈 Redis 中存的都是乱码，怀疑 Redis 被攻击了。其实呢，这个问题就是序列化算法导致的，我们来看下吧。

在这个案例中，开发同学使用 RedisTemplate 来操作 Redis 进行数据缓存。因为相比于 Jedis，使用 Spring 提供的 RedisTemplate 操作 Redis，除了无需考虑连接池、更方便外，还可以与 Spring Cache 等其他组件无缝整合。如果使用 Spring Boot 的话，无需任何配置就可以直接使用。

数据（包含 Key 和 Value）要保存到 Redis，需要经过序列化算法来序列化成字符串。虽然 Redis 支持多种数据结构，比如 Hash，但其每一个 field 的 Value 还是字符串。如果 Value 本身也是字符串的话，能否有便捷的方式来使用 RedisTemplate，而无需考虑序列化呢？

其实是有的，那就是 StringRedisTemplate。

那 StringRedisTemplate 和 RedisTemplate 的区别是什么呢？开头提到的乱码又是怎么回事呢？带着这些问题让我们来研究一下吧。

写一段测试代码，在应用初始化完成后向 Redis 设置两组数据，第一次使用 RedisTemplate 设置 Key 为 redisTemplate、Value 为 User 对象，第二次使用 StringRedisTemplate 设置 Key 为 stringRedisTemplate、Value 为 JSON 序列化后的 User 对象：


```

2 private RedisTemplate redisTemplate;
3 @Autowired
4 private StringRedisTemplate stringRedisTemplate;
5 @Autowired
6 private ObjectMapper objectMapper;
7
8 @PostConstruct
9 public void init() throws JsonProcessingException {
10     redisTemplate.opsForValue().set("redisTemplate", new User("zhuye", 36));
11     stringRedisTemplate.opsForValue().set("stringRedisTemplate", objectMapper.writeValueAsString(new User("zhuye", 36)));
12 }

```

如果你认为，StringRedisTemplate 和 RedisTemplate 的区别，无非是读取的 Value 是 String 和 Object，那就大错特错了，因为使用这两种方式存取的数据完全无法通用。

我们做个小实验，通过 RedisTemplate 读取 Key 为 stringRedisTemplate 的 Value，使用 StringRedisTemplate 读取 Key 为 redisTemplate 的 Value：

 复制代码

```

1 log.info("redisTemplate get {}", redisTemplate.opsForValue().get("stringRedisTemplate"));
2 log.info("stringRedisTemplate get {}", stringRedisTemplate.opsForValue().get("redisTemplate"));

```

结果是，两次都无法读取到 Value：

 复制代码

```

1 [11:49:38.478] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont
2 [11:49:38.481] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont

```


通过 redis-cli 客户端工具连接到 Redis，你会发现根本就没有叫作 redisTemplate 的 Key，所以 StringRedisTemplate 无法查到数据：

```

127.0.0.1:6379> keys *Template
1) "stringRedisTemplate"
2) "\xac\xed\x00\x05t\x00\rredisTemplate"

```


查看 RedisTemplate 的源码发现，默认情况下 RedisTemplate 针对 Key 和 Value 使用了 JDK 序列化：

 复制代码

```
1 public void afterPropertiesSet() {
2     ...
3     if (defaultSerializer == null) {
4         defaultSerializer = new JdkSerializationRedisSerializer(
5             classLoader != null ? classLoader : this.getClass().getClassLoader());
6     }
7     if (enableDefaultSerializer) {
8         if (keySerializer == null) {
9             keySerializer = defaultSerializer;
10            defaultUsed = true;
11        }
12        if (valueSerializer == null) {
13            valueSerializer = defaultSerializer;
14            defaultUsed = true;
15        }
16        if (hashKeySerializer == null) {
17            hashKeySerializer = defaultSerializer;
18            defaultUsed = true;
19        }
20        if (hashValueSerializer == null) {
21            hashValueSerializer = defaultSerializer;
22            defaultUsed = true;
23        }
24    }
25    ...
26 }
```

redis-cli 看到的类似一串乱码的"\xac\xed\x00\x05t\x00\rredisTemplate"字符串，其实就是字符串 redisTemplate 经过 JDK 序列化后的结果。这就回答了之前提到的乱码问题。而 RedisTemplate 尝试读取 Key 为 stringRedisTemplate 数据时，也会对这个字符串进行 JDK 序列化处理，所以同样无法读取到数据。

而 StringRedisTemplate 对于 Key 和 Value，使用的是 String 序列化方式，Key 和 Value 只能是 String：

 复制代码

```
1 public class StringRedisTemplate extends RedisTemplate<String, String> {
2     public StringRedisTemplate() {
3         setKeySerializer(RedisSerializer.string());
```

```

4     setValueSerializer(RedisSerializer.string());
5     setHasKeySerializer(RedisSerializer.string());
6     setHashValueSerializer(RedisSerializer.string());
7 }
8 }
9
10 public class StringRedisSerializer implements RedisSerializer<String> {
11     @Override
12     public String deserialize(@Nullable byte[] bytes) {
13         return (bytes == null ? null : new String(bytes, charset));
14     }
15
16     @Override
17     public byte[] serialize(@Nullable String string) {
18         return (string == null ? null : string.getBytes(charset));
19     }
20 }

```

看到这里，我们应该知道 RedisTemplate 和 StringRedisTemplate 保存的数据无法通用。修复方式就是，让它们读取自己存的数据：

使用 RedisTemplate 读出的数据，由于是 Object 类型的，使用时可以先强制转换为 User 类型；

使用 StringRedisTemplate 读取出的字符串，需要手动将 JSON 反序列化为 User 类型。

 复制代码

```

1 //使用RedisTemplate获取Value，无需反序列化就可以拿到实际对象，虽然方便，但是Redis中保存的
2 User userFromRedisTemplate = (User) redisTemplate.opsForValue().get("redisTemp
3 log.info("redisTemplate get {}", userFromRedisTemplate);
4
5 //使用StringRedisTemplate，虽然Key正常，但是Value存取需要手动序列化成字符串
6 User userFromStringRedisTemplate = objectMapper.readValue(stringRedisTemplate.
7 log.info("stringRedisTemplate get {}", userFromStringRedisTemplate);

```

这样就可以得到正确输出：

 复制代码

```

1 [13:32:09.087] [http-nio-45678-exec-6] [INFO ] [.t.c.s.demo1.RedisTemplateCont
2 [13:32:09.092] [http-nio-45678-exec-6] [INFO ] [.t.c.s.demo1.RedisTemplateCont

```


看到这里你可能会说，使用 RedisTemplate 获取 Value 虽然方便，但是 Key 和 Value 不易读；而使用 StringRedisTemplate 虽然 Key 是普通字符串，但是 Value 存取需要手动序列化成本字符串，有没有两全其美的方式呢？

当然有，自己定义 RedisTemplate 的 Key 和 Value 的序列化方式即可：Key 的序列化使用 RedisSerializer.string()（也就是 StringRedisSerializer 方式）实现字符串序列化，而 Value 的序列化使用 Jackson2JsonRedisSerializer：

 复制代码


```
1 @Bean
2 public <T> RedisTemplate<String, T> redisTemplate(RedisConnectionFactory redisC
3     RedisTemplate<String, T> redisTemplate = new RedisTemplate<>();
4     redisTemplate.setConnectionFactory(redisConnectionFactory);
5     Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2Json
6     redisTemplate.setKeySerializer(RedisSerializer.string());
7     redisTemplate.setValueSerializer(jackson2JsonRedisSerializer);
8     redisTemplate.setHashKeySerializer(RedisSerializer.string());
9     redisTemplate.setHashValueSerializer(jackson2JsonRedisSerializer);
10    redisTemplate.afterPropertiesSet();
11    return redisTemplate;
12 }
```

写代码测试一下存取，直接注入类型为 RedisTemplate<String, User> 的 userRedisTemplate 字段，然后在 right2 方法中，使用注入的 userRedisTemplate 存入一个 User 对象，再分别使用 userRedisTemplate 和 StringRedisTemplate 取出这个对象：

 复制代码

```
1 @Autowired
2 private RedisTemplate<String, User> userRedisTemplate;
3
4 @GetMapping("right2")
5 public void right2() {
6     User user = new User("zhuye", 36);
7     userRedisTemplate.opsForValue().set(user.getName(), user);
8     Object userFromRedis = userRedisTemplate.opsForValue().get(user.getName())
9     log.info("userRedisTemplate get {} {}", userFromRedis, userFromRedis.getClass());
10    log.info("stringRedisTemplate get {} {}", stringRedisTemplate.opsForValue().get
11 }
```


乍一看没啥问题，StringRedisTemplate 成功查出了我们存入的数据：

 复制代码


```
1 [14:07:41.315] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont
2 [14:07:41.318] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont
```

Redis 里也可以查到 Key 是纯字符串，Value 是 JSON 序列化后的 User 对象：

```
127.0.0.1:6379> get zhuye
"{\"name\":\"zhuye\",\"age\":36}"
```


但值得注意的是，这里有一个坑。**第一行的日志输出显示，userRedisTemplate 获取到的 Value，是 LinkedHashMap 类型的**，完全不是泛型的 RedisTemplate 设置的 User 类型。

如果我们把代码里从 Redis 中获取到的 Value 变量类型由 Object 改为 User，编译不会出现问题，但会出现 ClassCastException：

 复制代码


```
1 java.lang.ClassCastException: java.util.LinkedHashMap cannot be cast to org.ge
```

修复方式是，修改自定义 RestTemplate 的代码，把 new 出来的 Jackson2JsonRedisSerializer 设置一个自定义的 ObjectMapper，启用 activateDefaultTyping 方法把类型信息作为属性写入序列化后的数据中（当然了，你也可以调整 JsonTypeInfo.As 枚举以其他形式保存类型信息）：

 复制代码

```
1 ...
2 Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2JsonRedi:
3 ObjectMapper objectMapper = new ObjectMapper();
4 //把类型信息作为属性写入Value
5 objectMapper.activateDefaultTyping(objectMapper.getPolymorphicTypeValidator(),
6 jackson2JsonRedisSerializer.setObjectMapper(objectMapper);
7 ...
```

或者，直接使用 `RedisSerializer.json()` 快捷方法，它内部使用的 `GenericJackson2JsonRedisSerializer` 直接设置了把类型作为属性保存到 Value 中：

 复制代码

```
1 redisTemplate.setKeySerializer(RedisSerializer.string());
2 redisTemplate.setValueSerializer(RedisSerializer.json());
3 redisTemplate.setHashKeySerializer(RedisSerializer.string());
4 redisTemplate.setHashValueSerializer(RedisSerializer.json());
```

重启程序调用 `right2` 方法进行测试，可以看到，从自定义的 `RedisTemplate` 中获取到的 Value 是 `User` 类型的（第一行日志），而且 Redis 中实际保存的 Value 包含了类型完全限定名（第二行日志）：

 复制代码

```
1 [15:10:50.396] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont
2 [15:10:50.399] [http-nio-45678-exec-1] [INFO ] [.t.c.s.demo1.RedisTemplateCont
```

因此，反序列化时可以直接得到 `User` 类型的 Value。

通过对 `RedisTemplate` 组件的分析，可以看到，当数据需要序列化后保存时，读写数据使用一致的序列化算法的必要性，否则就像对牛弹琴。

这里，我再总结下 Spring 提供的 4 种 `RedisSerializer`（Redis 序列化器）：

默认情况下，`RedisTemplate` 使用 `JdkSerializationRedisSerializer`，也就是 JDK 序列化，容易产生 Redis 中保存了乱码的错觉。

通常考虑到易读性，可以设置 Key 的序列化器为 `StringRedisSerializer`。但直接使用 `RedisSerializer.string()`，相当于使用了 UTF_8 编码的 `StringRedisSerializer`，需要注意字符集问题。

如果希望 Value 也是使用 JSON 序列化的话，可以把 Value 序列化器设置为 `Jackson2JsonRedisSerializer`。默认情况下，不会把类型信息保存在 Value 中，即使我们定义 `RedisTemplate` 的 Value 泛型为实际类型，查询出的 Value 也只能是 `LinkedHashMap` 类型。如果希望直接获取真实的数据类型，你可以启用 Jackson

ObjectMapper 的 `activateDefaultTyping` 方法，把类型信息一起序列化保存在 Value 中。


如果希望 Value 以 JSON 保存并带上类型信息，更简单的方式是，直接使用 `RedisSerializer.json()` 快捷方法来获取序列化器。

注意 Jackson JSON 反序列化对额外字段的处理

前面我提到，通过设置 JSON 序列化工具 Jackson 的 `activateDefaultTyping` 方法，可以在序列化数据时写入对象类型。其实，Jackson 还有很多参数可以控制序列化和反序列化，是一个功能强大而完善的序列化工具。因此，很多框架都将 Jackson 作为 JDK 序列化工具，比如 Spring Web。但也正是这个原因，我们使用时要小心各个参数的配置。

比如，在开发 Spring Web 应用程序时，如果自定义了 ObjectMapper，并把它注册成了 Bean，那很可能会导致 Spring Web 使用的 ObjectMapper 也被替换，导致 Bug。

我们来看一个案例。程序一开始是正常的，某一天开发同学希望修改一下 ObjectMapper 的行为，让枚举序列化为索引值而不是字符串值，比如默认情况下序列化一个 Color 枚举中的 `Color.BLUE` 会得到字符串 `BLUE`：

 复制代码

```
1 @Autowired
2 private ObjectMapper objectMapper;
3
4 @GetMapping("test")
5 public void test() throws JsonProcessingException {
6     log.info("color:{}", objectMapper.writeValueAsString(Color.BLUE));
7 }
8
9 enum Color {
10     RED, BLUE
11 }
```


于是，这位同学就重新定义了一个 ObjectMapper Bean，开启了 `WRITE_ENUMS_USING_INDEX` 功能特性：

 复制代码

```
1 @Bean
2 public ObjectMapper objectMapper(){
```


```
3     ObjectMapper objectMapper=new ObjectMapper();
4     objectMapper.configure(SerializationFeature.WRITE_ENUMS_USING_INDEX,true);
5     return objectMapper;
6 }
```

开启这个特性后，Color.BLUE 枚举序列化索引值 1：

 复制代码


```
1 [16:11:37.382] [http-nio-45678-exec-1] [INFO ] [c.s.d.JsonIgnorePropertiesCont
```

修改后处理枚举序列化的逻辑是满足了要求，但线上爆出了大量 400 错误，日志中也出现了很多 UnrecognizedPropertyException：

 复制代码

```
1 JSON parse error: Unrecognized field \"ver\" (class org.geekbang.time.commonmi:
```

从异常信息中可以看到，这是因为反序列化的时候，原始数据多了一个 version 属性。进一步分析发现，我们使用了 UserWrong 类型作为 Web 控制器 wrong 方法的入参，其中只有一个 name 属性：

 复制代码

```
1 @Data
2 public class UserWrong {
3     private String name;
4 }
5
6 @PostMapping("wrong")
7 public UserWrong wrong(@RequestBody UserWrong user) {
8     return user;
9 }
```

而客户端实际传过来的数据多了一个 version 属性。那，为什么之前没这个问题呢？

问题就出在，**自定义 ObjectMapper 启用 WRITE_ENUMS_USING_INDEX 序列化功能特性时，覆盖了 Spring Boot 自动创建的 ObjectMapper**；而这个自动创建的

ObjectMapper 设置过 FAIL_ON_UNKNOWN_PROPERTIES 反序列化特性为 false，以确保出现未知字段时不要抛出异常。源码如下：

 复制代码

```
1 public MappingJackson2HttpMessageConverter() {
2     this(Jackson2ObjectMapperBuilder.json().build());
3 }
4
5
6 public class Jackson2ObjectMapperBuilder {
7
8     ...
9
10    private void customizeDefaultFeatures(ObjectMapper objectMapper) {
11        if (!this.features.containsKey(MapperFeature.DEFAULT_VIEW_INCLUSION)) {
12            configureFeature(objectMapper, MapperFeature.DEFAULT_VIEW_INCLUSION, false);
13        }
14        if (!this.features.containsKey(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES)) {
15            configureFeature(objectMapper, DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
16        }
17    }
18 }
```


要修复这个问题，有三种方式：

第一种，同样禁用自定义的 ObjectMapper 的 FAIL_ON_UNKNOWN_PROPERTIES：

 复制代码

```
1 @Bean
2 public ObjectMapper objectMapper(){
3     ObjectMapper objectMapper=new ObjectMapper();
4     objectMapper.configure(SerializationFeature.WRITE_ENUMS_USING_INDEX,true);
5     objectMapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,false);
6     return objectMapper;
7 }
```

第二种，设置自定义类型，加上 @JsonIgnoreProperties 注解，开启 ignoreUnknown 属性，以实现反序列化时忽略额外的数据：


 复制代码

```
1 @Data
2 @JsonIgnoreProperties(ignoreUnknown = true)
```

```
3 public class UserRight {  
4     private String name;  
5 }
```


第三种，不要自定义 ObjectMapper，而是直接在配置文件设置相关参数，来修改 Spring 默认的 ObjectMapper 的功能。比如，直接在配置文件启用把枚举序列化为索引号：

```
1 spring.jackson.serialization.write_enums_using_index=true
```

 复制代码

或者直接定义 Jackson2ObjectMapperBuilderCustomizer Bean 来启用新特性：

```
1 @Bean  
2 public Jackson2ObjectMapperBuilderCustomizer customizer(){  
3     return builder -> builder.featuresToEnable(SerializationFeature.WRITE_ENUM!  
4 }
```

 复制代码

这个案例告诉我们两点：

Jackson 针对序列化和反序列化有大量的细节功能特性，我们可以参考 Jackson 官方文档来了解这些特性，详见 [SerializationFeature](#)、[DeserializationFeature](#)和 [MapperFeature](#)。

忽略多余字段，是我们写业务代码时最容易遇到的一个配置项。Spring Boot 在自动配置时贴心地做了全局设置。如果需要设置更多的特性，可以直接修改配置文件 `spring.jackson.**` 或设置 Jackson2ObjectMapperBuilderCustomizer 回调接口，来启用更多设置，无需重新定义 ObjectMapper Bean。

反序列化时要小心类的构造方法

使用 Jackson 反序列化时，除了要注意忽略额外字段的问题外，还要小心类的构造方法。我们看一个实际的踩坑案例吧。


有一个 `APIResult` 类包装了 REST 接口的返回体（作为 Web 控制器的出参），其中 `boolean` 类型的 `success` 字段代表是否处理成功、`int` 类型的 `code` 字段代表处理状态码。

开始时，在返回 `APIResult` 的时候每次都根据 `code` 来设置 `success`。如果 `code` 是 2000，那么 `success` 是 `true`，否则是 `false`。后来为了减少重复代码，把这个逻辑放到了 `APIResult` 类的构造方法中处理：

 复制代码

```
1 @Data
2 public class APIResultWrong {
3     private boolean success;
4     private int code;
5
6     public APIResultWrong() {
7     }
8
9     public APIResultWrong(int code) {
10         this.code = code;
11         if (code == 2000) success = true;
12         else success = false;
13     }
14 }
```

经过改动后发现，即使 `code` 为 2000，返回 `APIResult` 的 `success` 也是 `false`。比如，我们反序列化两次 `APIResult`，一次使用 `code==1234`，一次使用 `code==2000`：

 复制代码

```
1 @Autowired
2 ObjectMapper objectMapper;
3
4 @GetMapping("wrong")
5 public void wrong() throws JsonProcessingException {
6     log.info("result :{}", objectMapper.readValue("{\"code\":\"1234\"}", APIResultt
7     log.info("result :{}", objectMapper.readValue("{\"code\":\"2000\"}", APIResultt
8 }
```


日志输出如下：

 复制代码

```
1 [17:36:14.591] [http-nio-45678-exec-1] [INFO ] [DeserializationConstructorCont
2 [17:36:14.591] [http-nio-45678-exec-1] [INFO ] [DeserializationConstructorCont
```


可以看到，两次的 APIResult 的 success 字段都是 false。

出现这个问题的原因是，**默认情况下，在反序列化的时候，Jackson 框架只会调用无参构造方法创建对象**。如果走自定义的构造方法创建对象，需要通过 @JsonCreator 来指定构造方法，并通过 @JsonProperty 设置构造方法中参数对应的 JSON 属性名：

 复制代码

```
1 @Data
2 public class APIResultRight {
3     ...
4
5     @JsonCreator
6     public APIResultRight(@JsonProperty("code") int code) {
7         this.code = code;
8         if (code == 2000) success = true;
9         else success = false;
10    }
11 }
```

重新运行程序，可以得到正确输出：

 复制代码

```
1 [17:41:23.188] [http-nio-45678-exec-1] [INFO ] [DeserializationConstructorCont
2 [17:41:23.188] [http-nio-45678-exec-1] [INFO ] [DeserializationConstructorCont
```

可以看到，这次传入 code==2000 时，success 可以设置为 true。

枚举作为 API 接口参数或返回值的两个大坑

在前面的例子中，我演示了如何把枚举序列化为索引值。但对于枚举，我建议尽量在程序内部使用，而不是作为 API 接口的参数或返回值，原因是枚举涉及序列化和反序列化时会有两个大坑。

第一个坑是，客户端和服务端的枚举定义不一致时，会出异常。比如，客户端版本的枚举定义了 4 个枚举值：

[复制代码](#)

```
1 @Getter
2 enum StatusEnumClient {
3     CREATED(1, "已创建"),
4     PAID(2, "已支付"),
5     DELIVERED(3, "已送到"),
6     FINISHED(4, "已完成");
7
8     private final int status;
9     private final String desc;
10
11     StatusEnumClient(Integer status, String desc) {
12         this.status = status;
13         this.desc = desc;
14     }
15 }
```

服务端定义了 5 个枚举值：

[复制代码](#)

```
1 @Getter
2 enum StatusEnumServer {
3     ...
4     CANCELED(5, "已取消");
5
6     private final int status;
7     private final String desc;
8
9     StatusEnumServer(Integer status, String desc) {
10         this.status = status;
11         this.desc = desc;
12     }
13 }
```


写代码测试一下，使用 RestTemplate 来发起请求，让服务端返回客户端不存在的枚举值：

[复制代码](#)

```
1 @GetMapping("getOrderStatusClient")
2 public void getOrderStatusClient() {
3     StatusEnumClient result = restTemplate.getForObject("http://localhost:4567:
4     log.info("result {}", result);
5 }
6
```


```
7 @GetMapping("getOrderStatus")
8 public StatusEnumServer getOrderStatus() {
9     return StatusEnumServer.CANCELED;
10 }
```

访问接口会出现如下异常信息，提示在枚举 StatusEnumClient 中找不到 CANCELED：

 复制代码

```
1 JSON parse error: Cannot deserialize value of type `org.geekbang.time.commonmi:
```

要解决这个问题，可以开启 Jackson 的 `read_unknown_enum_values_using_default_value` 反序列化特性，也就是在枚举值未知的时候使用默认值：

 复制代码

```
1 spring.jackson.deserialization.read_unknown_enum_values_using_default_value=true
```

并为枚举添加一个默认值，使用 `@JsonEnumDefaultValue` 注解注释：

 复制代码

```
1 @JsonEnumDefaultValue
2 UNKNOWN(-1, "未知");
```

需要注意的是，这个枚举值一定是添加在客户端 StatusEnumClient 中的，因为反序列化使用的是客户端枚举。

这里还有一个小坑是，仅仅这样配置还不能让 RestTemplate 生效这个反序列化特性，还需要配置 RestTemplate，来使用 Spring Boot 的 MappingJackson2HttpMessageConverter 才行：

 复制代码

```
1 @Bean
2 public RestTemplate restTemplate(MappingJackson2HttpMessageConverter mappingJa
3     return new RestTemplateBuilder()
4         .additionalMessageConverters(mappingJackson2HttpMessageConverter)
```

```
5         .build();
6     }
```

现在，请求接口可以返回默认值了：

```
1 [21:49:03.887] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.e.e.EnumUsedInAPIContr
```

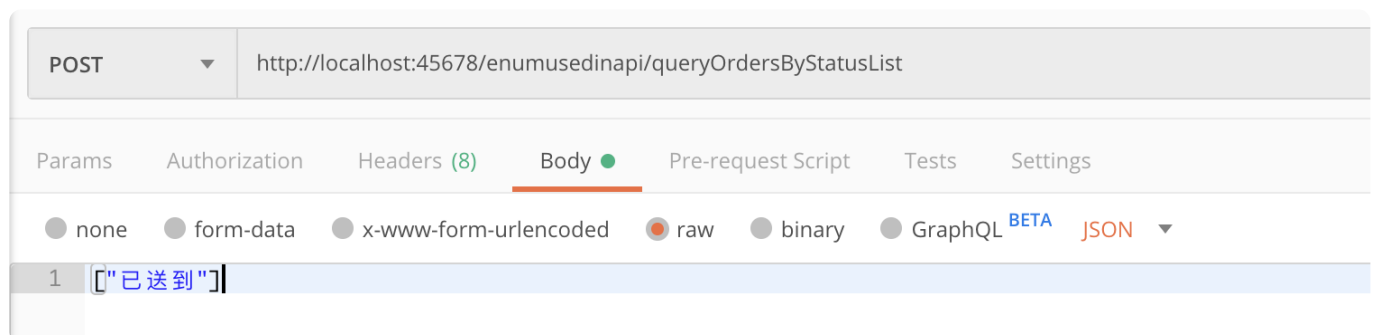
复制代码

第二个坑，也是更大的坑，枚举序列化反序列化实现自定义的字段非常麻烦，会涉及 Jackson 的 Bug。比如，下面这个接口，传入枚举 List，为 List 增加一个 CENCELED 枚举值然后返回：

```
1 @PostMapping("queryOrdersByStatusList")
2 public List<StatusEnumServer> queryOrdersByStatus(@RequestBody List<StatusEnumServer> enumServers) {
3     enumServers.add(StatusEnumServer.CANCELED);
4     return enumServers;
5 }
```

复制代码

如果我们希望根据枚举的 Desc 字段来序列化，传入“已送到”作为入参：

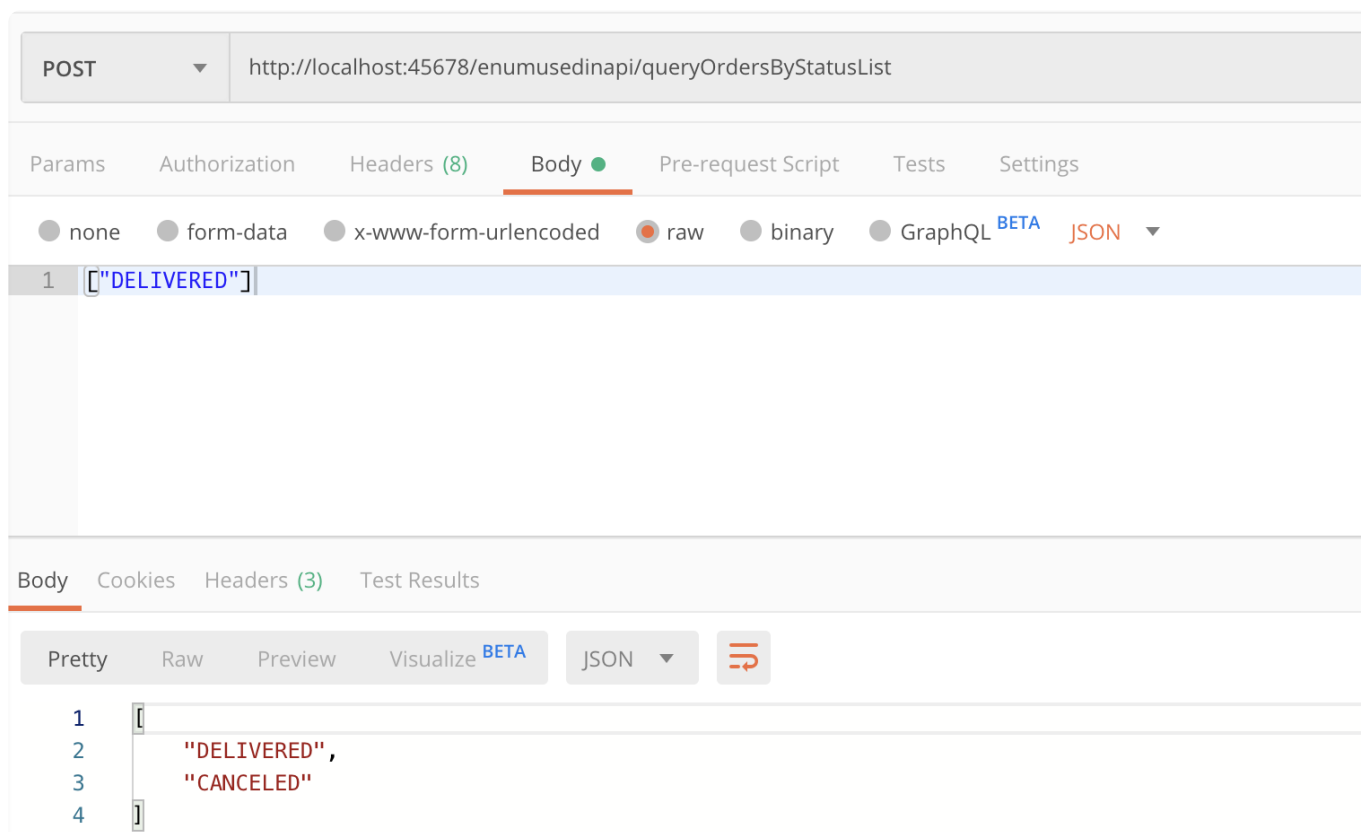


会得到异常，提示“已送到”不是正确的枚举值：

```
1 JSON parse error: Cannot deserialize value of type `org.geekbang.time.commonmi:
```

复制代码

显然，这里反序列化使用的是枚举的 name，序列化也是一样：

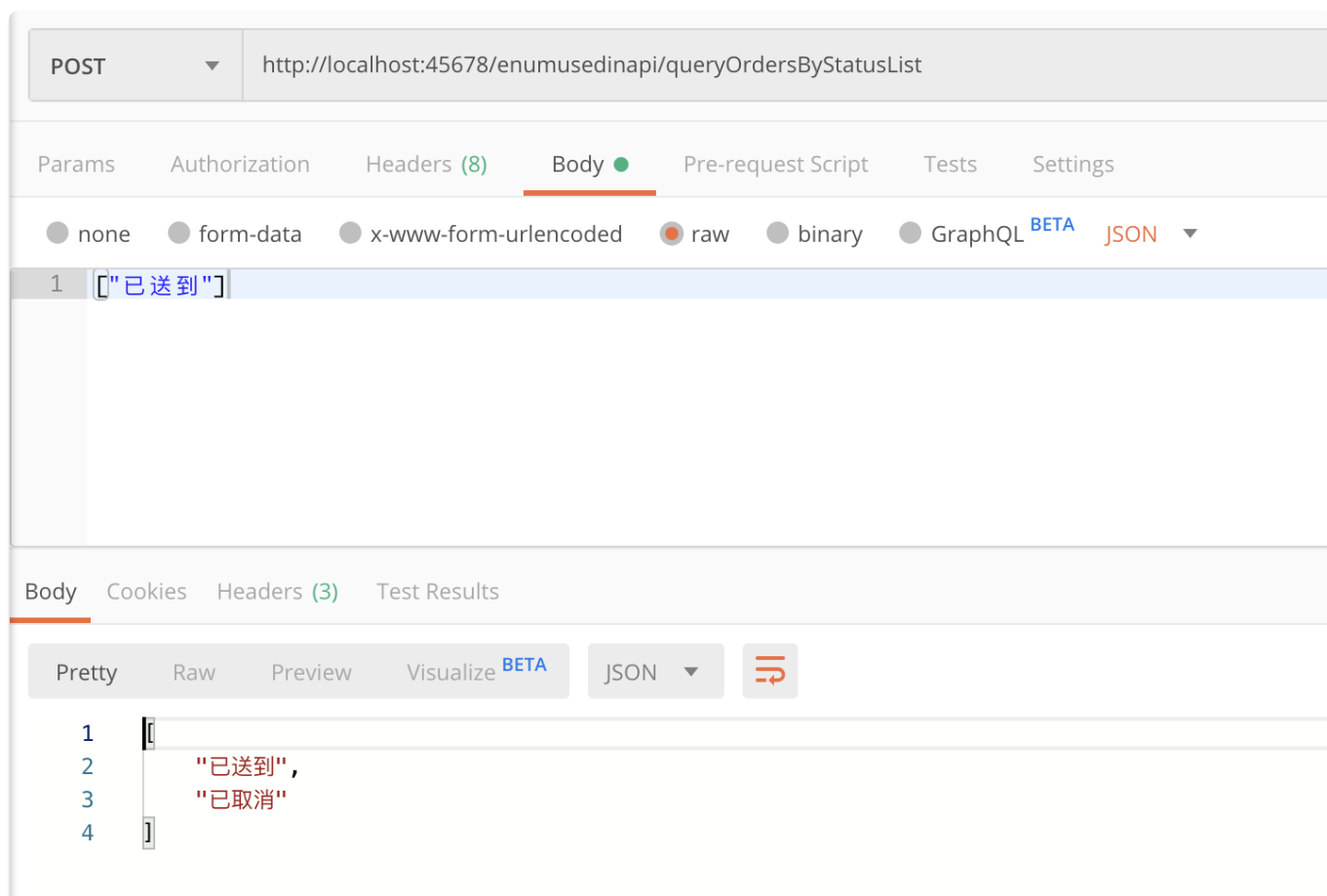


你可能也知道，要让枚举的序列化和反序列化走 desc 字段，可以在字段上加 @JsonValue 注解，修改 StatusEnumServer 和 StatusEnumClient：

```
1 @JsonValue
2 private final String desc;
```

复制代码

然后再尝试下，果然可以用 desc 作为入参了，而且出参也使用了枚举的 desc：



但是，如果你认为这样就完美解决问题了，那就大错特错了。你可以再尝试把 `@JsonValue` 注解加在 `int` 类型的 `status` 字段上，也就是希望序列化反序列化走 `status` 字段：

 复制代码

```
1 @JsonValue
2 private final int status;
```

写一个客户端测试一下，传入 `CREATED` 和 `PAID` 两个枚举值：

 复制代码

```
1 @GetMapping("queryOrdersByStatusListClient")
2 public void queryOrdersByStatusListClient() {
3     List<StatusEnumClient> request = Arrays.asList(StatusEnumClient.CREATED, S
4     HttpEntity<List<StatusEnumClient>> entity = new HttpEntity<>(request, new I
5     List<StatusEnumClient> response = restTemplate.exchange("http://localhost::
6         HttpMethod.POST, entity, new ParameterizedTypeReference<List<Statu:
7     log.info("result {}", response);
8 }
```

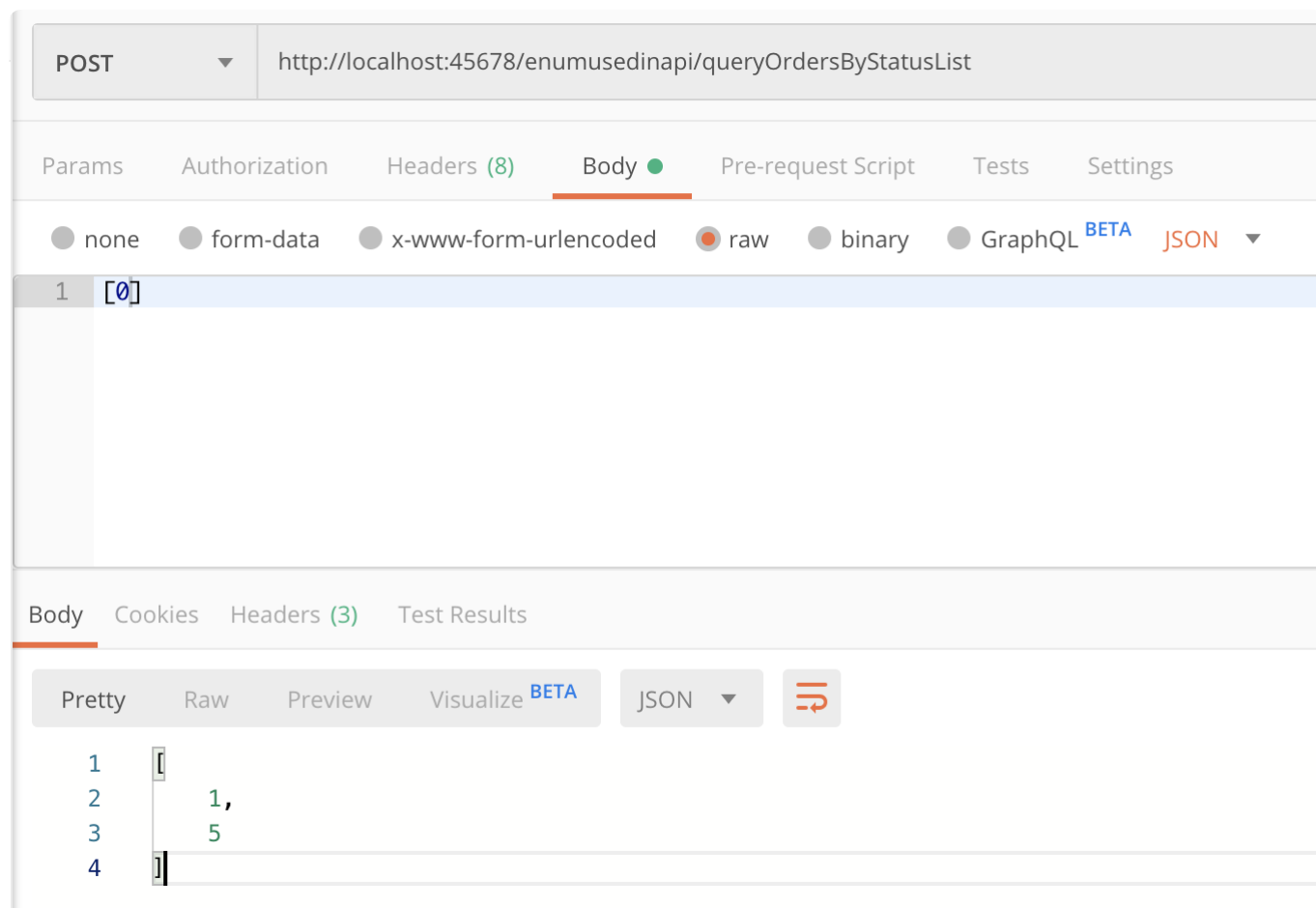
请求接口可以看到，传入的是 CREATED 和 PAID，返回的居然是 DELIVERED 和 FINISHED。果然如标题所说，一来一回你已不是原来的你：

复制代码

```
1 [22:03:03.579] [http-nio-45678-exec-4] [INFO ] [o.g.t.c.e.e.EnumUsedInAPIContr
```

出现这个问题的原因是，**序列化走了 status 的值，而反序列化并没有根据 status 来，还是使用了枚举的 ordinal() 索引值**。这是 Jackson [至今 \(2.10\) 没有解决的 Bug](#)，应该会在 2.11 解决。

如下图所示，我们调用服务端接口，传入一个不存在的 status 值 0，也能反序列化成功，最后服务端的返回是 1：



有一个解决办法是，设置 @JsonCreator 来强制反序列化时使用自定义的工厂方法，可以实现使用枚举的 status 字段来取值。我们把这段代码加在 StatusEnumServer 枚举类中：

[复制代码](#)

```
1 @JsonCreator
2 public static StatusEnumServer parse(Object o) {
3     return Arrays.stream(StatusEnumServer.values()).filter(value->o.equals(value)).findFirst().orElse(null);
4 }
```

要特别注意的是，我们同样要为 StatusEnumClient 也添加相应的方法。因为除了服务端接口接收 StatusEnumServer 参数涉及一次反序列化外，从服务端返回值转换为 List 还会有一次反序列化：

[复制代码](#)

```
1 @JsonCreator
2 public static StatusEnumClient parse(Object o) {
3     return Arrays.stream(StatusEnumClient.values()).filter(value->o.equals(value)).findFirst().orElse(null);
4 }
```

重新调用接口发现，虽然结果正确了，但是服务端不存在的枚举值 CANCELED 被设置为了 null，而不是 @JsonEnumDefaultValue 设置的 UNKNOWN。

这个问题，我们之前已经通过设置 @JsonEnumDefaultValue 注解解决了，但现在又出现了：

[复制代码](#)

```
1 [22:20:13.727] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.e.e.EnumUsedInAPIContr
```

原因也很简单，我们自定义的 parse 方法实现的是找不到枚举值时返回 null。

为彻底解决这个问题，并避免通过 @JsonCreator 在枚举中自定义一个非常复杂的工厂方法，我们可以实现一个自定义的反序列化器。这段代码比较复杂，我特意加了详细的注释：

[复制代码](#)

```
1 class EnumDeserializer extends JsonSerializer<Enum> implements
2     ContextualDeserializer {
3
4     private Class<Enum> targetClass;
5
6     public EnumDeserializer() {
```

```

7     }
8
9     public EnumDeserializer(Class<Enum> targetClass) {
10         this.targetClass = targetClass;
11     }
12
13     @Override
14     public Enum deserialize(JsonParser p, DeserializationContext ctxt) {
15         //找枚举中带有@JsonValue注解的字段，这是我们反序列化的基准字段
16         Optional<Field> valueFieldOpt = Arrays.asList(targetClass.getDeclaredFields())
17             .filter(m -> m.isAnnotationPresent(JsonValue.class))
18             .findFirst();
19
20         if (valueFieldOpt.isPresent()) {
21             Field valueField = valueFieldOpt.get();
22             if (!valueField.isAccessible()) {
23                 valueField.setAccessible(true);
24             }
25             //遍历枚举项，查找字段的值等于反序列化的字符串的那个枚举项
26             return Arrays.stream(targetClass.getEnumConstants()).filter(e -> {
27                 try {
28                     return valueField.get(e).toString().equals(p.getValueAsString());
29                 } catch (Exception ex) {
30                     ex.printStackTrace();
31                 }
32                 return false;
33             }).findFirst().orElseGet(() -> Arrays.stream(targetClass.getEnumConstants())
34                 //如果找不到，就需要寻找默认枚举值来替代，同样遍历所有枚举项，查找@JsonEnum
35                 try {
36                     return targetClass.getField(e.name()).isAnnotationPresent(JsonEnum.class);
37                 } catch (Exception ex) {
38                     ex.printStackTrace();
39                 }
40                 return false;
41             }).findFirst().orElse(null));
42         }
43         return null;
44     }
45
46     @Override
47     public JsonDeserializer<?> createContextual(DeserializationContext ctxt,
48         BeanProperty property) throws IOException {
49         targetClass = (Class<Enum>) ctxt.getContextualType().getRawClass();
50         return new EnumDeserializer(targetClass);
51     }
52 }

```

然后，把这个自定义反序列化器注册到 Jackson 中：

```
1 @Bean
2 public Module enumModule() {
3     SimpleModule module = new SimpleModule();
4     module.addDeserializer(Enum.class, new EnumDeserializer());
5     return module;
6 }
```

第二个大坑终于被完美地解决了：

```
1 [22:32:28.327] [http-nio-45678-exec-1] [INFO ] [o.g.t.c.e.e.EnumUsedInAPIContr
```

这样做，虽然解决了序列化反序列化使用枚举中自定义字段的问题，也解决了找不到枚举值时使用默认值的问题，但解决方案很复杂。因此，我还是建议在 DTO 中直接使用 int 或 String 等简单的数据类型，而不是使用枚举再配合各种复杂的序列化配置，来实现枚举到枚举中字段的映射，会更加清晰明了。

重点回顾

今天，我基于 Redis 和 Web API 的入参和出参两个场景，和你介绍了序列化和反序列化时需要避开的几个坑。

第一，要确保序列化和反序列化算法的一致性。因为，不同序列化算法输出必定不同，要正确处理序列化后的数据就要使用相同的反序列化算法。

第二，Jackson 有大量的序列化和反序列化特性，可以用来微调序列化和反序列化的细节。需要注意的是，如果自定义 ObjectMapper 的 Bean，小心不要和 Spring Boot 自动配置的 Bean 冲突。

第三，在调试序列化反序列化问题时，我们一定要捋清楚三点：是哪个组件在做序列化反序列化、整个过程有几次序列化反序列化，以及目前到底是序列化还是反序列化。

第四，对于反序列化默认情况下，框架调用的是无参构造方法，如果要调用自定义的有参构造方法，那么需要告知框架如何调用。更合理的方式是，对于需要序列化的 POJO 考虑尽量不要自定义构造方法。

第五，枚举不建议定义在 DTO 中跨服务传输，因为会有版本问题，并且涉及序列化反序列化时会很复杂，容易出错。因此，我只建议在程序内部使用枚举。

最后还有一点需要注意，如果需要跨平台使用序列化的数据，那么除了两端使用的算法要一致外，还可能会遇到不同语言对数据类型的兼容问题。这，也是经常踩坑的一个地方。如果你有相关需求，可以多做实验、多测试。

今天用到的代码，我都放在了 GitHub 上，你可以点击 [这个链接](#) 查看。

思考与讨论

1. 在讨论 Redis 序列化方式的时候，我们自定义了 RedisTemplate，让 Key 使用 String 序列化、让 Value 使用 JSON 序列化，从而使 Redis 获得的 Value 可以直接转换为需要的对象类型。那么，使用 RedisTemplate<String, Long> 能否存取 Value 是 Long 的数据呢？这其中有什么坑吗？
2. 你可以看一下 Jackson2ObjectMapperBuilder 类源码的实现（注意 configure 方法），分析一下其除了关闭 FAIL_ON_UNKNOWN_PROPERTIES 外，还做了什么吗？

关于序列化和反序列化，你还遇到过什么坑吗？我是朱晔，欢迎在评论区与我留言分享，也欢迎你把这篇文章分享给你的朋友或同事，一起交流。

点击参与 

进入朱晔老师「读者群」带你
攻克 Java 业务开发常见错误



添加Java班长，报名入群



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

上一篇 14 | 文件IO：实现高效正确的文件读写并非易事

下一篇 16 | 用好Java 8的日期时间类，少踩一些“老三样”的坑

精选留言 (9)

写留言



Darren

2020-04-14

试着回答下今天的问题：

- 1、Long序列化的时候，Redis会认为是int，因此是获取不到的Long数据的，需要处理；
- 2、Jackson2ObjectMapperBuilder的采用了构建者模式创建对象；调用的是build()方法

```
public <T extends ObjectMapper> T build() {  
    ObjectMapper mapper;...
```

展开 ∨

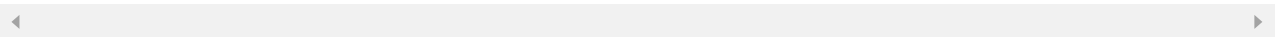
作者回复: 比较坑的是，在Integer区间内返回的是Integer，超过这个区间返回Long

```
@GetMapping("wrong2")  
public void wrong2() {  
    String key = "testCounter";  
    countRedisTemplate.opsForValue().set(key, 1L);  
    log.info("{} {}", countRedisTemplate.opsForValue().get(key), countRedisTemplate.ops  
ForValue().get(key) instanceof Long);  
    Long l1 = getLongFromRedis(key);  
    countRedisTemplate.opsForValue().set(key, Integer.MAX_VALUE + 1L);  
    log.info("{} {}", countRedisTemplate.opsForValue().get(key), countRedisTemplate.ops  
ForValue().get(key) instanceof Long);  
    Long l2 = getLongFromRedis(key);  
    log.info("{} {} {}", l1, l2);  
}  
  
private Long getLongFromRedis(String key) {  
    Object o = countRedisTemplate.opsForValue().get(key);  
    if (o instanceof Integer) {  
        return ((Integer) o).longValue();  
    }  
}
```

```
    if (o instanceof Long) {  
        return (Long) o;  
    }  
    return null;  
}
```

输出：

```
1 false  
2147483648 true  
1 2147483648
```



💬 2

👍 5



左琪

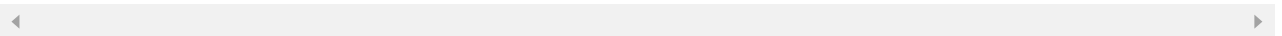
2020-04-16

老师，我之前遇到一个，我用redis存入一个Map<Long,object>，取出时发现却是Map<int,object>,然后响应给前端springmvc就报类型转换异常了，我redis的value也是用的Jackson序列化，自定义了objectmapper，正常对象都能序列化，反序列化，就是Long不行，我想知道该如何修正呢

展开 ∨

作者回复: 这个long和int的问题应该就是我思考题的问题，你可以看看其他网友的回复以及我的回复如何解决。

我不知道你这里的用redis存入Map<Long,object>是不是指key是Long，value是Object，如果是的话，把数字作为Key不是一个好的实践，Redis的Key需要是字符串，并且区分命名空间，比如应用_领域_标识（或是数据库_表_PK），e.g.commonmistakes_redisexample_user123



💬

👍 2



梦倚栏杆

2020-04-14

老师，现在像fastJson, jackson 一般使用序列化和反序列化不都是属性类型兼容就能来回序列化吗？java序列化的时候存储序列化id记录版本号的意义是什么。java序列化一开始存在的意义是什么？为什么要那样处理呢？如果按照现在fastJson 和jackson等的处理方式，toString 不也是一种序列化方式吗？反序列化时按照一种规则解析回去不就行了

展开 ∨

作者回复: 1、有关serialVersionUID的意义:

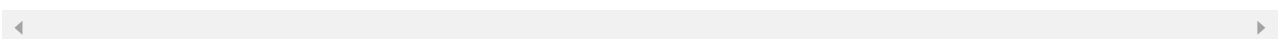
The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization. If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException. A serializable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long:

```
ANY-ACCESS-MODIFIER static final long serialVersionUID = 42L;
```

If a serializable class does not explicitly declare a serialVersionUID, then the serialization runtime will calculate a default serialVersionUID value for that class based on various aspects of the class, as described in the Java(TM) Object Serialization Specification. However, it is strongly recommended that all serializable classes explicitly declare serialVersionUID values, since the default serialVersionUID computation is highly sensitive to class details that may vary depending on compiler implementations, and can thus result in unexpected InvalidClassExceptions during deserialization. Therefore, to guarantee a consistent serialVersionUID value across different java compiler implementations, a serializable class must declare an explicit serialVersionUID value. It is also strongly advised that explicit serialVersionUID declarations use the private modifier where possible, since such declarations apply only to the immediately declaring class serialVersionUID fields are not useful as inherited members.

2、toString也可以认为是一种文本序列化，序列化当然还可以按照自己的方式来做，只要是一致的方式实现对象到字节的转换。

3、java序列化一开始存在的意义是什么？在有xml、json、protobuf等之前，jdk总需要有序列化来实现对象的文件存储、跨服务传输吧，当时确实互联网也没这么发达没考虑到异构体系的交互问题，我们不能以现在的眼光来看当时的技术为什么考虑这么不全面这么鸡肋



梦倚栏杆

2020-04-14

redis 的序列化也遇到过问题，但是当时处理的比较紧急比较模糊，记不清了。

一个类里需要不同结构的redis或者类型不同==>

a> 序列化报错，写不进去。

b> 没报错，阿里云后台查看redis的结构是None,key是乱码，也无法删除。

最终的解决方案是：自己new RedisTemplate, 对于每一种结构对象都设置对应的序列...
展开 ▾



1



小杰

2020-04-14

log.info("longRedisTemplate get {}", (Long)longRedisTemplate.opsForValue().get("longRedisTemplate"));
java.lang.ClassCastException: java.lang.Integer cannot be cast to java.lang.Long
强转异常，也就是说我们获取到这样的值还要自己从Integer转成Long是吗老师？

展开 ▾

作者回复: 是



1



2020-04-14

之前其实一直还是比较喜欢枚举的，一直只是觉得枚举这个是个好的功能，只是我不会用。

现在来看，看来枚举在使用上确实需要谨慎。

个人理解，枚举的本质，其实就是一个Map<Object,Object>，但是扩展性更强一些。...

展开 ▾

作者回复: 内部没关系，也推荐使用枚举，对外是要慎用



1



pedro

2020-04-14

关于枚举，无论是在 dto 还是数据库存储，我们都已经不用 int 来枚举了，而选择语义性的字符串，这在 debug 和维护上十分方便，也有利于迁移，int 枚举太难看了，每次调试，眼睛都花了🤔

作者回复: 😊



1



Michael

2020-04-19

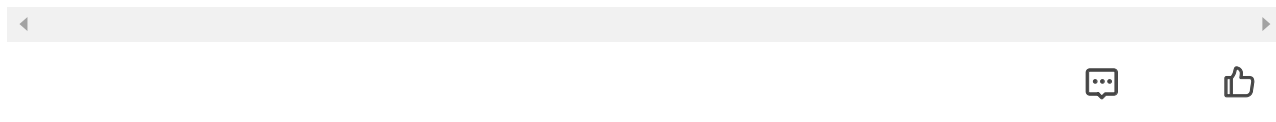
我们项目中遇到的坑是：key是字符串，value是一个自定义对象，我们环境分为inner和prd,inner验证过了才会发生产，但是inner和prd是同一个Redis缓存和DB，value值对应的对象中加了字段，生产和inner同时作用，prd缓存失效了，正好把inner的给存进去了，结果导致生产接口从缓存取数据的时候出现反序列化报错问题，影响了生产。

后面采取方法是在缓存key加上环境前缀来避免这个问题。

展开 ∨

作者回复: 对内环境和生产环境的问题也是比较典型的。对内虽然和生产公用数据库和中间件，但是毕竟还是用于测试验证的。

之前遇到的一个坑是类似的，对内环境和生产公用CDN，对内验证的时候传了一张测试图片上去，然后CDN节点就有了这个文件，到了生产上虽然又传了正式的图片（文件名没有变），但是CDN节点上的测试图片依旧缓存着，部分地区用户还是看到了测试图片。。。



Geek_3b1096

2020-04-18

习惯用枚举表示数据库状态字段
希望老师能指点枚举最佳实践

