

# HttpMessageConverter是这样转换数据的

原创 tan0 日拱一兵 2019-05-27

Java Web 人员经常要设计 RESTful API ([如何设计好的RESTful API < https://mp.weixin.qq.com/s?\\_\\_biz=Mzg3NjlxMjA1Ng==&mid=2247483661&idx=1&sn=048af6543c7baf6cefa691f80587b4c3&chksm=cf34fb3af843722c839977948df95b881b9bc3a4124b36a091af8435451643470d7db0af51de&token=1506224503&lang=zh\\_CN&scene=21#wechat\\_redirect](https://mp.weixin.qq.com/s?__biz=Mzg3NjlxMjA1Ng==&mid=2247483661&idx=1&sn=048af6543c7baf6cefa691f80587b4c3&chksm=cf34fb3af843722c839977948df95b881b9bc3a4124b36a091af8435451643470d7db0af51de&token=1506224503&lang=zh_CN&scene=21#wechat_redirect))，通过 json 数据进行交互。那么前端传入的 json 数据如何被解析成 Java 对象作为 API入参，API 返回结果又如何将 Java 对象解析成 json 格式数据返回给前端？

其实在整个数据流转过程中， `HttpMessageConverter` 起到了重要作用；本文我们除了关注数据是如何转换的，另外还会关注在转换的过程我们可以加入哪些定制化内容

🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹  
本文因包含源码与调用栈信息，通过浏览器阅读，阅读体验更佳  
🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹🌹

## HttpMessageConverter 介绍

`org.springframework.http.converter.HttpMessageConverter` 是一个策略接口，接口说明如下：

Strategy interface that specifies a converter that can convert from and to HTTP requests and responses.

简单说就是 HTTP request (请求)和response (响应)的转换器

该接口有只有5个方法，就是获取支持的 `MediaType` (`application/json`之类)，接收到请求时判断是否能读 (`canRead`)，能读则读 (`read`)；返回结果时判断是否能写 (`canWrite`)，能写则写 (`write`)。这几个方法先有个印象即可

```
1 boolean canRead(Class<?> clazz, MediaType mediaType);
2 boolean canWrite(Class<?> clazz, MediaType mediaType);
3
4 List<MediaType> getSupportedMediaTypes();
5
6 T read(Class<? extends T> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException;
7
8 void write(T t, MediaType contentType, HttpOutputMessage outputMessage) throws IOException, HttpMessageNotWritableException;
```

### 缺省配置

我们写 Demo 没有配置任何 `MessageConverter`，但是数据前后传递依旧好用，是因为 `SpringMVC` 启动时会自动配置一些`HttpMessageConverter`，在 `WebMvcConfigurationSupport` 类中添加了缺省 `MessageConverter`：

```
1 protected final void addDefaultHttpMessageConverters(List<HttpMessageConverter<?>> messageConverters) {
2     StringHttpMessageConverter stringConverter = new StringHttpMessageConverter();
3     stringConverter.setWriteAcceptCharset(false);
4
5
6     messageConverters.add(new ByteArrayHttpMessageConverter());
7     messageConverters.add(stringConverter);
8     messageConverters.add(new ResourceHttpMessageConverter());
9     messageConverters.add(new SourceHttpMessageConverter<Source>());
10    messageConverters.add(new AllEncompassingFormHttpMessageConverter());
11
12
13
14
15    if (romePresent) {
16        messageConverters.add(new AtomFeedHttpMessageConverter());
17        messageConverters.add(new RssChannelHttpMessageConverter());
18    }
19
20
21
22    if (jackson2XmlPresent) {
23        ObjectMapper objectMapper = Jackson20bjectMapperBuilder.xml().applicationContext(this.applicationContext).build();
24        messageConverters.add(new MappingJackson2XmlHttpMessageConverter(objectMapper));
25    }
26
27    else if (jaxb2Present) {
28        messageConverters.add(new Jaxb2RootElementHttpMessageConverter());
29    }
30
31
32
33
34    if (jackson2Present) {
35        ObjectMapper objectMapper = Jackson20bjectMapperBuilder.json().applicationContext(this.applicationContext).build();
36        messageConverters.add(new MappingJackson2HttpMessageConverter(objectMapper));
37    }
38
39    else if (gsonPresent) {
40        messageConverters.add(new GsonHttpMessageConverter());
41    }
42 }
```

```
    }  
}  
}
```

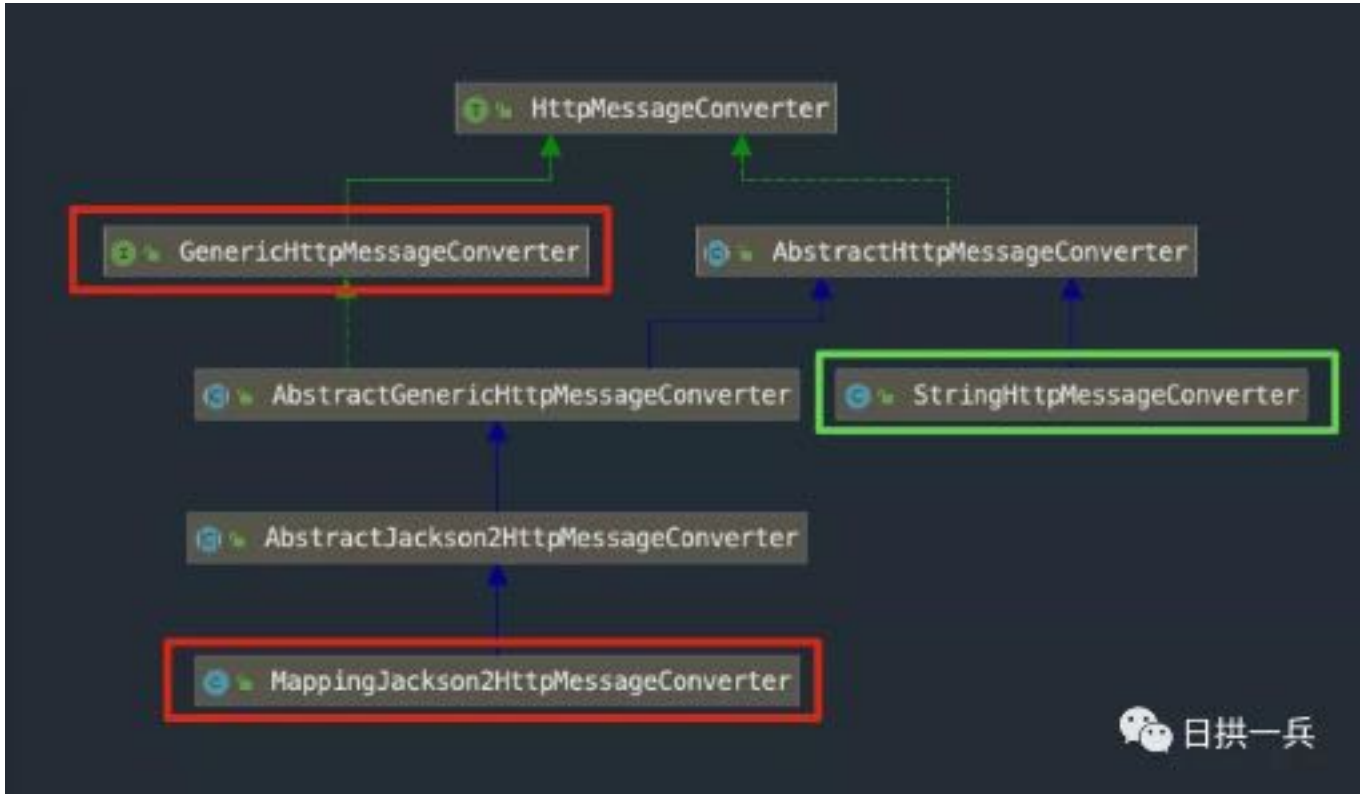
我们看到很熟悉的 MappingJackson2HttpMessageConverter，如果我们引入 jackson 相关包，Spring 就会为我们添加该 MessageConverter，但是我们通常在搭建框架的时候还是会手动添加配置 MappingJackson2HttpMessageConverter，为什么？

因为，当我们配置了自己的 MessageConverter，SpringMVC 启动过程就不会调用 addDefaultHttpMessageConverters 方法，且看下面代码 if 条件，这样做也是为了定制化我们自己的 MessageConverter

```
1  protected final List<HttpMessageConverter<?>> getMessageConverters() {  
2      if (this.messageConverters == null) {  
3  
4          this.messageConverters = new ArrayList<HttpMessageConverter<?>>();  
5  
6          configureMessageConverters(this.messageConverters);  
7  
8          if (this.messageConverters.isEmpty()) {  
9  
10             addDefaultHttpMessageConverters(this.messageConverters);  
11  
12         }  
13  
14         extendMessageConverters(this.messageConverters);  
15  
16     }  
17  
18     return this.messageConverters;  
19 }  
20 }
```

类关系图

在此处仅列出 MappingJackson2HttpMessageConverter 和 StringHttpMessageConverter 两个转换器，我们发现，前者实现了 GenericHttpMessageConverter 接口, 而后者却没有，留有这个关键印象，这是数据流转过程分析的关键逻辑判断



数据流转解析

数据的请求和响应都要经过 DispatcherServlet 类的 doDispatch(HttpServletRequest request, HttpServletResponse response) 方法的处理

请求过程解析

看 doDispatch 方法中的关键代码：

```
1  // 这里的 Adapter 实际上是 RequestMappingHandlerAdapter  
2  HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());  
3  
4  if (!mappedHandler.applyPreHandle(processedRequest, response)) {  
5      return;  
6  }  
7  
8  // 实际处理的handler  
9  
10 mv = ha.handle(processedRequest, response, mappedHandler.getHandler());  
11  
12 mappedHandler.applyPostHandle(processedRequest, response, mv);
```

我将进入 ha.handle 方法后的调用栈粘贴在此处，希望小伙伴可以按照调用栈路线动手跟踪尝试：

```
1  readWithMessageConverters:192, AbstractMessageConverterMethodArgumentResolver (org.springframework.web.servlet.mvc.method.annotation)  
2  readWithMessageConverters:150, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)  
3  resolveArgument:128, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)  
4  resolveArgument:121, HandlerMethodArgumentResolverComposite (org.springframework.web.method.support)  
5  getMethodArgumentValues:158, InvocableHandlerMethod (org.springframework.web.method.support)  
6  invokeForRequest:128, InvocableHandlerMethod (org.springframework.web.method.support)  
7  // 下面的调用栈重点关注，处理请求和返回值的分叉口就在这里  
8  invokeAndHandle:97, ServletInvocableHandlerMethod (org.springframework.web.servlet.mvc.method.annotation)  
9  invokeHandlerMethod:849, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation)  
10  
11 handleInternal:760, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation)
```

2020/1/8	HttpMessageConverter是这样转换数据的
handle:85, AbstractHandlerMethodAdapter (org.springframework.web.servlet.mvc.method)	
doDispatch:967, DispatcherServlet (org.springframework.web.servlet)	

这里重点说明调用栈最顶层 readWithMessageConverters 方法中内容：

```
1 // 遍历 messageConverters
2 for (HttpMessageConverter<?> converter : this.messageConverters) {
3
4     Class<HttpMessageConverter<?>> converterType = (Class<HttpMessageConverter<?>>) converter.getClass();
5
6     // 上文类关系图处要重点记住的地方，主要判断 MappingJackson2HttpMessageConverter 是否是 GenericHttpMessageConverter 类型
7     if (converter instanceof GenericHttpMessageConverter) {
8
9         GenericHttpMessageConverter<?> genericConverter = (GenericHttpMessageConverter<?>) converter;
10
11         if (genericConverter.canRead(targetType, contextClass, contentType)) {
12
13             if (logger.isDebugEnabled()) {
14
15                 logger.debug("Read [" + targetType + "] as \"" + contentType + "\" with [" + converter + "]");
16
17             }
18
19             if (inputMessage.getBody() != null) {
20
21                 inputMessage = getAdvice().beforeBodyRead(inputMessage, parameter, targetType, converterType);
22
23                 body = genericConverter.read(targetType, contextClass, inputMessage);
24
25                 body = getAdvice().afterBodyRead(body, inputMessage, parameter, targetType, converterType);
26
27             }
28
29             else {
30
31                 body = getAdvice().handleEmptyBody(null, inputMessage, parameter, targetType, converterType);
32
33             }
34
35             break;
36
37         }
38
39     }
40
41     else if (targetClass != null) {
42
43         if (converter.canRead(targetClass, contentType)) {
44
45             if (logger.isDebugEnabled()) {
46
47                 logger.debug("Read [" + targetType + "] as \"" + contentType + "\" with [" + converter + "]");
48
49             }
50
51             if (inputMessage.getBody() != null) {
52
53                 inputMessage = getAdvice().beforeBodyRead(inputMessage, parameter, targetType, converterType);
54
55                 body = ((HttpMessageConverter<T>) converter).read(targetClass, inputMessage);
56
57                 body = getAdvice().afterBodyRead(body, inputMessage, parameter, targetType, converterType);
58
59             }
60
61             else {
62
63                 body = getAdvice().handleEmptyBody(null, inputMessage, parameter, targetType, converterType);
64
65             }
66
67             break;
68
69         }
70
71     }
72
73 }
```

然后就判断是否canRead，能读就read，最终走到下面代码处将输入的内容反序列化出来：

```
1 protected Object _readMapAndClose(JsonParser p0, JavaType valueType) throws IOException{
2
3     try (JsonParser p = p0) {
4
5         Object result;
6
7         JsonToken t = _initForReading(p);
8
9         if (t == JsonToken.VALUE_NULL) {
10
11             // Ask JsonDeserializer what 'null value' to use:
12
13             DeserializationContext ctxt = createDeserializationContext(p,
14
15                 getDeserializationConfig());
16
17             result = _findRootDeserializer(ctxt, valueType).getNullValue(ctxt);
18
19         } else if (t == JsonToken.END_ARRAY || t == JsonToken.END_OBJECT) {
20
21             result = null;
22
23         } else {
24
25             DeserializationConfig cfg = getDeserializationConfig();
26
27             DeserializationContext ctxt = createDeserializationContext(p, cfg);
28
29             JsonDeserializer<Object> deser = _findRootDeserializer(ctxt, valueType);
30
31             if (cfg.useRootWrapping()) {
```

2020/1/8

HttpMessageConverter是这样转换数据的

23

result = \_unwrapAndDeserialize(p, ctxt, cfg, valueType, deser);

24

25

} else {

26

result = deser.deserialize(p, ctxt);

27

}

ctxt.checkUnresolvedObjectId();

}

*// Need to consume the token too*

p.clearCurrentToken();

return result;

}

}

到这里从请求中解析参数过程的分析就到此结束了，趁热打铁来看将响应结果返回给前端的过程

### 返回过程解析

在上面调用栈请求和返回结果分叉口处同样处理返回值的内容：

1

writeWithMessageConverters:224, AbstractMessageConverterMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)

2

handleReturnValue:174, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)

3

4

handleReturnValue:81, HandlerMethodReturnValueHandlerComposite (org.springframework.web.method.support)

5

// 分叉口

invokeAndHandle:113, ServletInvocableHandlerMethod (org.springframework.web.servlet.mvc.method.annotation)

重点关注调用栈顶层内容，是不是很熟悉的样子，完全一样的逻辑, 判断是否能写canWrite，能写则write：

1

for (HttpMessageConverter<?> messageConverter : this.messageConverters) {

2

if (messageConverter instanceof GenericHttpMessageConverter) {

3

4

if (((GenericHttpMessageConverter) messageConverter).canWrite(

5

declaredType, valueType, selectedMediaType)) {

6

7

outputValue = (T) getAdvice().beforeBodyWrite(outputValue, returnType, selectedMediaType,

8

(Class<? extends HttpMessageConverter<?>>) messageConverter.getClass(),

9

inputMessage, outputMessage);

10

11

if (outputValue != null) {

12

addContentDispositionHeader(inputMessage, outputMessage);

13

14

((GenericHttpMessageConverter) messageConverter).write(

15

outputValue, declaredType, selectedMediaType, outputMessage);

16

17

if (logger.isDebugEnabled()) {

18

logger.debug("Written [" + outputValue + "] as \"" + selectedMediaType +

19

"\" using [" + messageConverter + "]");

20

21

}

22

}

23

24

return;

25

}

26

}

27

else if (messageConverter.canWrite(valueType, selectedMediaType)) {

28

29

outputValue = (T) getAdvice().beforeBodyWrite(outputValue, returnType, selectedMediaType,

30

(Class<? extends HttpMessageConverter<?>>) messageConverter.getClass(),

31

inputMessage, outputMessage);

32

33

if (outputValue != null) {

34

addContentDispositionHeader(inputMessage, outputMessage);

((HttpMessageConverter) messageConverter).write(outputValue, selectedMediaType, outputMessage);

if (logger.isDebugEnabled()) {

logger.debug("Written [" + outputValue + "] as \"" + selectedMediaType +

"\" using [" + messageConverter + "]");

}

}

}

return;

}

}

上面代码第5行，我们看到有这样代码：

1

outputValue = (T) getAdvice().beforeBodyWrite(outputValue, returnType, selectedMediaType,



```
2         (Class<? extends HttpMessageConverter<?>>) messageConverter.getClass(),
3         inputMessage, outputMessage);
```

其实，我们在设计 RESTful API 接口的时候通常会将返回的数据封装成统一格式，通常我们会实现 `ResponseBodyAdvice` 接口来处理所有 API 的返回值，在真正 `write` 之前将数据进行统一的封装：

```
1  @RestControllerAdvice()
2
3  public class CommonResultResponseAdvice implements ResponseBodyAdvice<Object> {
4
5
6      @Override
7      public boolean supports(MethodParameter returnType, Class<? extends HttpMessageConverter<?>> converterType) {
8
9          return true;
10     }
11
12     @Override
13     public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType selectedContentType,
14                                  Class<? extends HttpMessageConverter<?>> selectedConverterType, ServerHttpRequest request,
15                                  ServerHttpResponse response) {
16
17         if (body instanceof CommonResult) {
18
19             return body;
20         }
21
22         return new CommonResult<Object>(body);
23     }
24 }
```

至此，通过 `HttpMessageConverter` 转换请求和响应数据的流程就是这样，整个实现过程细节还需小伙伴自行追踪发现（一定要亲自尝试），在文章开头我们说过添加自己的 `MessageConverter` 能更好的满足我们的定制化，都有哪些内容可以定制的呢？

## 定制化

### 空值处理

请求和返回的数据有很多空值，这些值有时候并没有实际意义，我们可以过滤掉和不返回，或设置成默认值。比如通过重写 `getObjectMapper` 方法，将返回结果的空值不进行序列化处理：

```
1  @EnableWebMvc
2
3  @Configuration
4  public class MyWebMvcConfig extends WebMvcConfigurerAdapter {
5
6      @Override
7      public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
8
9          converters.add(0, new MappingJackson2HttpMessageConverter(){
10
11              @Override
12              public ObjectMapper getObjectMapper() {
13
14                  super.getObjectMapper().setSerializationInclusion(JsonInclude.Include.NON_NULL);
15
16                  return super.getObjectMapper();
17              }
18          }
19      }
20  }
```

## XSS 脚本攻击

为了确保输入的数据更安全，防止 XSS 脚本攻击，我们可以添加自定义的反序列化器：

```
1  @EnableWebMvc
2
3  @Configuration
4  public class WebConfig extends WebMvcConfigurerAdapter {
5
6      @Override
7      public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
8
9          converters.add(0, new MappingJackson2HttpMessageConverter(){
10
11              @Override
12              public ObjectMapper getObjectMapper() {
13
14                  super.getObjectMapper().setSerializationInclusion(JsonInclude.Include.NON_NULL);
15
16                  // XSS 脚本过滤
17
18                  SimpleModule simpleModule = new SimpleModule();
```

2020/1/8

HttpMessageConverter是这样转换数据的

18

19

20

```
simpleModule.addDeserializer(String.class, new StringXssDeserializer());

super.get.ObjectMapper().registerModule(simpleModule);

return super.get.ObjectMapper();

}

}

}

}
```

这里是数据转换的关键，所有涉及到数据转换需要统一处理的地方，我们都可以考虑如何在此处进行定制化处理。

### 细节分析

canRead 和 canWrite 的判断逻辑是什么呢？ 请看下图：

```
graph TD
    subgraph Request_Header [Request Header]
        CT1[Content-Type]
        Acc1[Accept]
    end
    subgraph Response_Header [Response Header]
        CT2[Content-Type]
    end
    subgraph Converter [HttpMessageConverter]
        canRead
        canWrite
        write
    end
    subgraph MediaTypes [getSupportedMediaTypes]
        getSupportedMediaTypes
    end
    subgraph Actions [Actions]
        read
    end

    CT1 -- "application/json" --> canRead
    Acc1 -- "application/json" --> canWrite
    canRead -- "比对成功" --> read
    canWrite -- "比对成功" --> write
    write -- "将request中的Accept设置到Response中的Content-Type中" --> CT2
    getSupportedMediaTypes -- "比对" --> canRead
    getSupportedMediaTypes -- "比对" --> canWrite
```

客户端 Request Header 中设置好 Content-Type（传入的数据格式）和Accept（接收的数据格式）， 根据配置好的 MessageConverter 来判断是否 canRead 或 canWrite，然后决定 response.body 的 Content-Type 的第一要素是对应的request.headers.Accept 属性的值。如果服务端支持这个 Accept，那么应该按照这个 Accept 来确定返回response.body 对应的格式，同时把 response.headers.Content-Type 设置成自己支持的符合那个 Accept 的 MediaType

### 总结与思考

站在上帝视角看，整个流程可以按照下图进行概括，请求报文先转换成 HttpInputMessage, 然后再通过 HttpMessageConverter 将其转换成 SpringMVC 的 java 对象，反之亦然。

```
graph LR
    Request([请求报文 request]) --> Input[HttpInputMessage]
    Input --> Converter[HttpMessageConverter]
    Converter --> MVC[Spring MVC]
    MVC --> Converter
    Converter --> Output[HttpOutMessage]
    Output --> Response([响应报文 response])
```

将各种常用 HttpMessageConverter 支持的MediaType 和 JavaType 以及对应关系总结在此处：

类名	支持的JavaType	支持的MediaType
ByteArrayHttpMessageConverter	byte[]	application/octet-stream, */*
StringHttpMessageConverter	String	text/plain, */*
MappingJackson2HttpMessageConverter	Object	application/json, application/*+json
AllEncompassingFormHttpMessageConverter FormHttpMessageConverter	Map<K, List<?>>	application/x-www-form-urlencoded, multipart/form-data
SourceHttpMessageConverter	Source	application/xml, text/xml, application/*+xml

🤔 思考

为什么 HttpMessageConverter 在写的逻辑中，先判断 canWrite 后判断是否有统一的 responseBodyAdvice 数据封装呢？ 如果先进行统一的 responseBodyAdvice 数据封装后判断 canWrite 会怎样呢？

### 提高效率工具

依旧介绍写该文章用到的一些好的工具，在后续内容中有好用的工具也会在公众号中推荐

processon

https://mp.weixin.qq.com/s?\_\_biz=Mzg3NjlxMjA1Ng==&mid=2247483668&idx=1&sn=69e83dded8f92539084a095be50440f8&scene=21#wechat\_redirect

6/7

ProcessOn是一个在线作图工具的聚合平台，它可以在线画流程图、思维导图、UI原型图、UML、网络拓扑图、组织结构图等等，您无需担心下载和更新的问题，不管Mac还是Windows，一个浏览器就可以随时随地的发挥创意，规

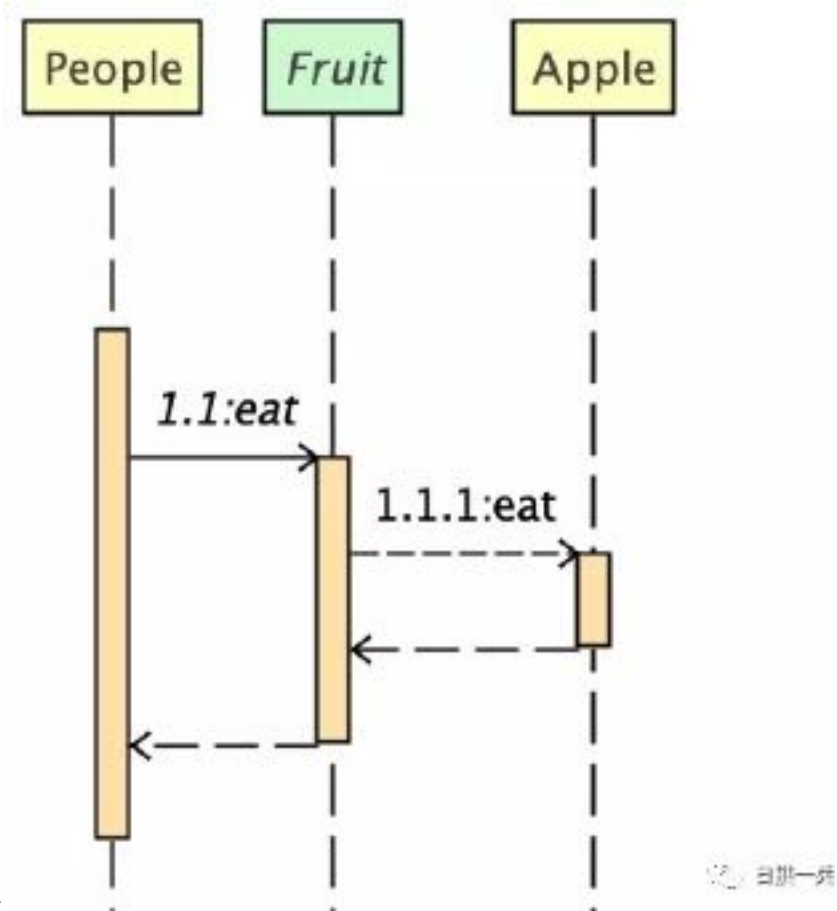


划工作，同时您可以把作品分享给团队成员或好友，无论何时何地大家都可以对作品进行编辑、阅读和评论

SequenceDiagram

SequenceDiagram 是 IntelliJ IDEA 的一个插件，有了这个插件，你可以

- 1. 生成简单序列图。
- 2. 单击图形形状来导航代码。
- 3. 从图中删除类。
- 4. 将图表导出为图像。
- 5. 通过“设置”>“其他设置”>“序列”从图表中排除类



方便快速的定位方法和理解类的调用过程

最后还是希望小伙伴亲自按照调用栈追踪调用过程，另外如果这篇文章对你有帮助，烦请关注公众号，我们一起探讨 Coding 那些趣事