HttpMessageConverter是这样转换数据的

原创 tanθ 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=Mzg3NjIxMjA1Ng==&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

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

```
boolean canRead(Class<?> clazz, MediaType mediaType);
boolean canWrite(Class<?> clazz, MediaType mediaType);
List<MediaType> getSupportedMediaTypes();
T read(Class<? extends T> clazz, HttpInputMessage inputMessage) throws IOException, HttpMessageNotReadableException;
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：

```
protected final void addDefaultHttpMessageConverters(List<HttpMessageConverter<?>> messageConverters) {
        StringHttpMessageConverter stringConverter = new StringHttpMessageConverter();
        stringConverter.setWriteAcceptCharset(false);

        messageConverters.add(new ByteArrayHttpMessageConverter());
        messageConverters.add(stringConverter);
        messageConverters.add(new ResourceHttpMessageConverter());
        messageConverters.add(new SourceHttpMessageConverter<Source>());
        messageConverters.add(new AllEncompassingFormHttpMessageConverter());

        if (romePresent) {
            messageConverters.add(new AtomFeedHttpMessageConverter());
            messageConverters.add(new RssChannelHttpMessageConverter());
        }

        if (jackson2XmlPresent) {
            ObjectMapper objectMapper = Jackson2ObjectMapperBuilder.xml().applicationContext(this.applicationContext).build();
            messageConverters.add(new MappingJackson2XmlHttpMessageConverter(objectMapper));
        }
        else if (jaxb2Present) {
            messageConverters.add(new Jaxb2RootElementHttpMessageConverter());
        }

        if (jackson2Present) {
            ObjectMapper objectMapper = Jackson2ObjectMapperBuilder.json().applicationContext(this.applicationContext).build();
            messageConverters.add(new MappingJackson2HttpMessageConverter(objectMapper));
        }
        else if (gsonPresent) {
            messageConverters.add(new GsonHttpMessageConverter());
```

```
        }
    }
```

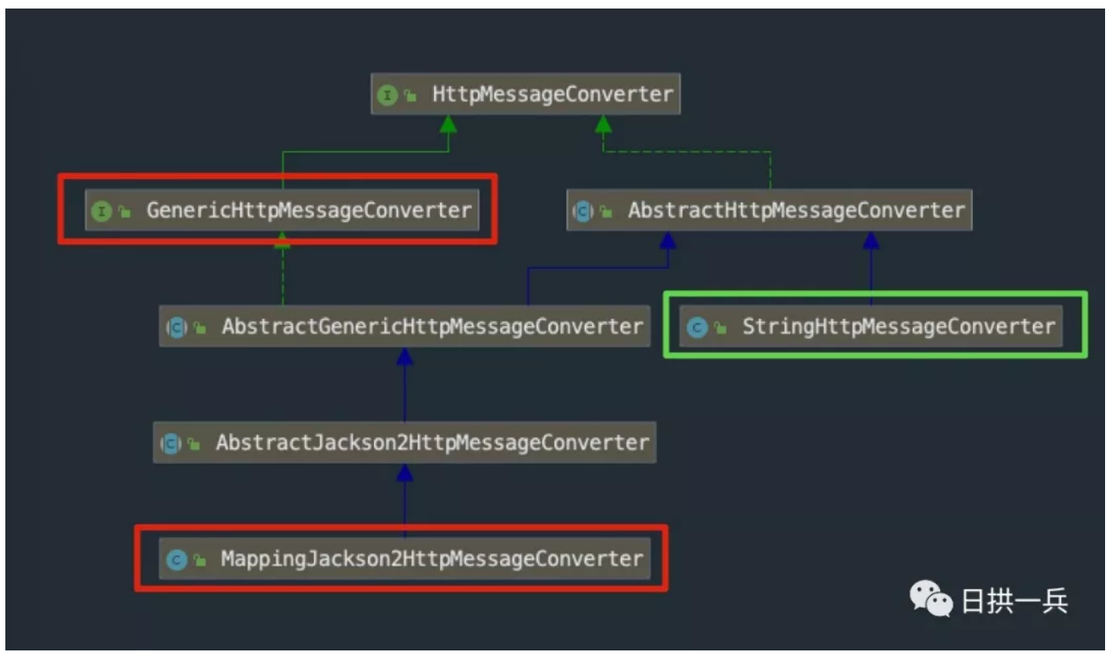
我们看到很熟悉的 `MappingJackson2HttpMessageConverter`，如果我们引入 jackson 相关包，Spring 就会为我们添加该 MessageConverter，但是我们通常在搭建框架的时候还是会手动添加配置 `MappingJackson2HttpMessageConverter`，为什么?

> 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

```
protected final List<HttpMessageConverter<?>> getMessageConverters() {
    if (this.messageConverters == null) {
        this.messageConverters = new ArrayList<HttpMessageConverter<?>>();
        configureMessageConverters(this.messageConverters);
        if (this.messageConverters.isEmpty()) {
            addDefaultHttpMessageConverters(this.messageConverters);
        }
        extendMessageConverters(this.messageConverters);
    }
    return this.messageConverters;
}
```

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：

```
// 这里的 Adapter 实际上是 RequestMappingHandlerAdapter
HandlerAdapter ha = this.getHandlerAdapter(mappedHandler.getHandler());
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
    return;
}
// 实际处理的handler
mv = ha.handle(processedRequest, response, mappedHandler.getHandler());            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：

```
readWithMessageConverters:192, AbstractMessageConverterMethodArgumentResolver (org.springframework.web.servlet.mvc.method.annotation)
readWithMessageConverters:150, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)
resolveArgument:128, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)
resolveArgument:121, HandlerMethodArgumentResolverComposite (org.springframework.web.method.support)
getMethodArgumentValues:158, InvocableHandlerMethod (org.springframework.web.method.support)
invokeForRequest:128, InvocableHandlerMethod (org.springframework.web.method.support)
 // 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
invokeAndHandle:97, ServletInvocableHandlerMethod (org.springframework.web.servlet.mvc.method.annotation)
invokeHandlerMethod:849, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation)
handleInternal:760, RequestMappingHandlerAdapter (org.springframework.web.servlet.mvc.method.annotation)
```

```
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：

```
// 遍历 messageConverters
for (HttpMessageConverter<?> converter : this.messageConverters) {
    Class<HttpMessageConverter<?>> converterType = (Class<HttpMessageConverter<?>>) converter.getClass();
    // 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
    if (converter instanceof GenericHttpMessageConverter) {
        GenericHttpMessageConverter<?> genericConverter = (GenericHttpMessageConverter<?>) converter;
        if (genericConverter.canRead(targetType, contextClass, contentType)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Read [" + targetType + "] as \"" + contentType + "\" with [" + converter + "]");
            }
            if (inputMessage.getBody() != null) {
                inputMessage = getAdvice().beforeBodyRead(inputMessage, parameter, targetType, converterType);
                body = genericConverter.read(targetType, contextClass, inputMessage);
                body = getAdvice().afterBodyRead(body, inputMessage, parameter, targetType, converterType);
            }
            else {
                body = getAdvice().handleEmptyBody(null, inputMessage, parameter, targetType, converterType);
            }
            break;
        }
    }
    else if (targetClass != null) {
        if (converter.canRead(targetClass, contentType)) {
            if (logger.isDebugEnabled()) {
                logger.debug("Read [" + targetType + "] as \"" + contentType + "\" with [" + converter + "]");
            }
            if (inputMessage.getBody() != null) {
                inputMessage = getAdvice().beforeBodyRead(inputMessage, parameter, targetType, converterType);
                body = ((HttpMessageConverter<T>) converter).read(targetClass, inputMessage);
                body = getAdvice().afterBodyRead(body, inputMessage, parameter, targetType, converterType);
            }
            else {
                body = getAdvice().handleEmptyBody(null, inputMessage, parameter, targetType, converterType);
            }
            break;
        }
    }
}
```

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：

```
protected Object _readMapAndClose(JsonParser p0, JavaType valueType) throws IOException{
    try (JsonParser p = p0) {
        Object result;
        JsonToken t = _initForReading(p);
        if (t == JsonToken.VALUE_NULL) {
            // Ask JsonDeserializer what 'null value' to use:
            DeserializationContext ctxt = createDeserializationContext(p,
                    getDeserializationConfig());
            result = _findRootDeserializer(ctxt, valueType).getNullValue(ctxt);
        } else if (t == JsonToken.END_ARRAY || t == JsonToken.END_OBJECT) {
            result = null;
        } else {
            DeserializationConfig cfg = getDeserializationConfig();
            DeserializationContext ctxt = createDeserializationContext(p, cfg);
            JsonDeserializer<Object> deser = _findRootDeserializer(ctxt, valueType);
            if (cfg.useRootWrapping()) {
```

```
                result = _unwrapAndDeserialize(p, ctxt, cfg, valueType, deser);
            } else {
                result = deser.deserialize(p, ctxt);
            }
            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：

```
writeWithMessageConverters:224, AbstractMessageConverterMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)
handleReturnValue:174, RequestResponseBodyMethodProcessor (org.springframework.web.servlet.mvc.method.annotation)
handleReturnValue:81, HandlerMethodReturnValueHandlerComposite (org.springframework.web.method.support)
// 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：

```
for (HttpMessageConverter<?> messageConverter : this.messageConverters) {
    if (messageConverter instanceof GenericHttpMessageConverter) {
        if (((GenericHttpMessageConverter) messageConverter).canWrite(
                declaredType, valueType, selectedMediaType)) {
            outputValue = (T) getAdvice().beforeBodyWrite(outputValue, returnType, selectedMediaType,
                    (Class<? extends HttpMessageConverter<?>>) messageConverter.getClass(),
                    inputMessage, outputMessage);
            if (outputValue != null) {
                addContentDispositionHeader(inputMessage, outputMessage);
                ((GenericHttpMessageConverter) messageConverter).write(
                        outputValue, declaredType, selectedMediaType, outputMessage);
                if (logger.isDebugEnabled()) {
                    logger.debug("Written [" + outputValue + "] as \"" + selectedMediaType +
                            "\" using [" + messageConverter + "]");
                }
            }
            return;
        }
    }
    else if (messageConverter.canWrite(valueType, selectedMediaType)) {
        outputValue = (T) getAdvice().beforeBodyWrite(outputValue, returnType, selectedMediaType,
                (Class<? extends HttpMessageConverter<?>>) messageConverter.getClass(),
                inputMessage, outputMessage);
        if (outputValue != null) {
            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：

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

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

```
@RestControllerAdvice()
public class CommonResultResponseAdvice implements ResponseBodyAdvice<Object> {

    @Override
    public boolean supports(MethodParameter returnType, Class<? extends HttpMessageConverter<?>> converterType) {
        return true;
    }

    @Override
    public Object beforeBodyWrite(Object body, MethodParameter returnType, MediaType selectedContentType,
            Class<? extends HttpMessageConverter<?>> selectedConverterType, ServerHttpRequest request,
            ServerHttpResponse response) {
        if (body instanceof CommonResult) {
            return body;
        }
        return new CommonResult<Object>(body);
    }
}
```

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：

```
@EnableWebMvc
@Configuration
public class MyWebMvcConfig extends WebMvcConfigurerAdapter {
  @Override
  public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(0, new MappingJackson2HttpMessageConverter(){
        @Override
        public ObjectMapper getObjectMapper() {
            super.getObjectMapper().setSerializationInclusion(JsonInclude.Include.NON_NULL);
                    return super.getObjectMapper();
        }
    }
  }
}
```

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：

```
@EnableWebMvc
@Configuration
public class WebConfig extends WebMvcConfigurerAdapter {
  @Override
  public void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(0, new MappingJackson2HttpMessageConverter(){
        @Override
        public ObjectMapper getObjectMapper() {
            super.getObjectMapper().setSerializationInclusion(JsonInclude.Include.NON_NULL);

            // XSS 脚本过滤
            SimpleModule simpleModule = new SimpleModule();
```

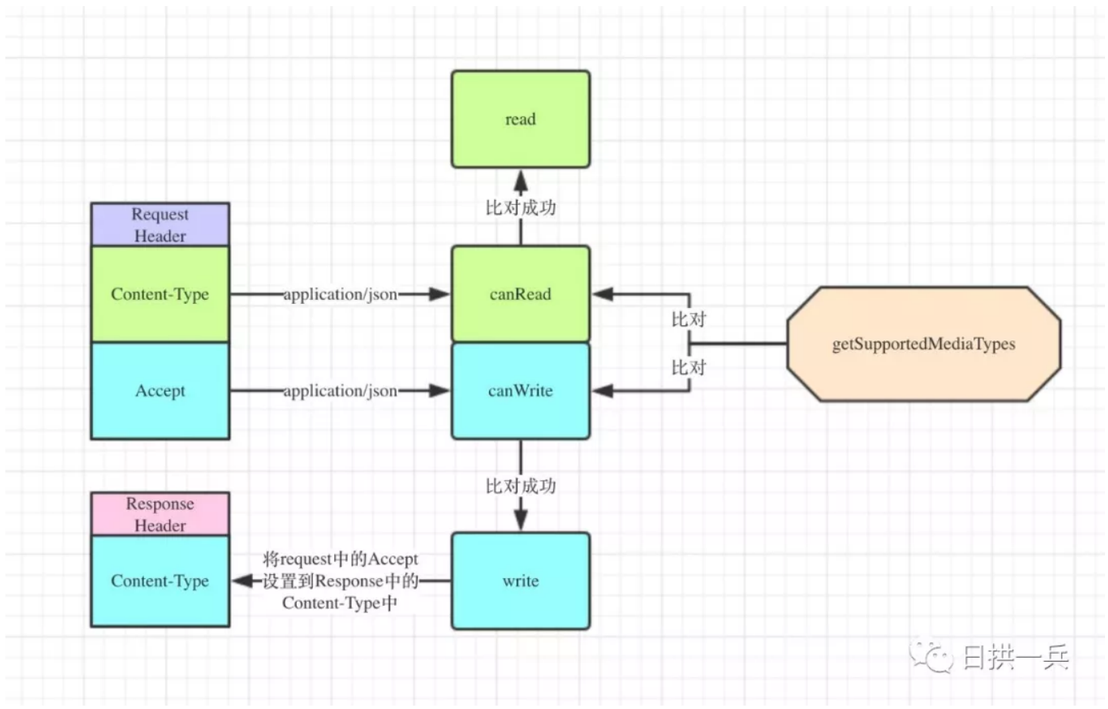
```
            simpleModule.addDeserializer(String.class, new StringXssDeserializer());
            super.getObjectMapper().registerModule(simpleModule);


            return super.getObjectMapper();
        }
    }
  }
}
```

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

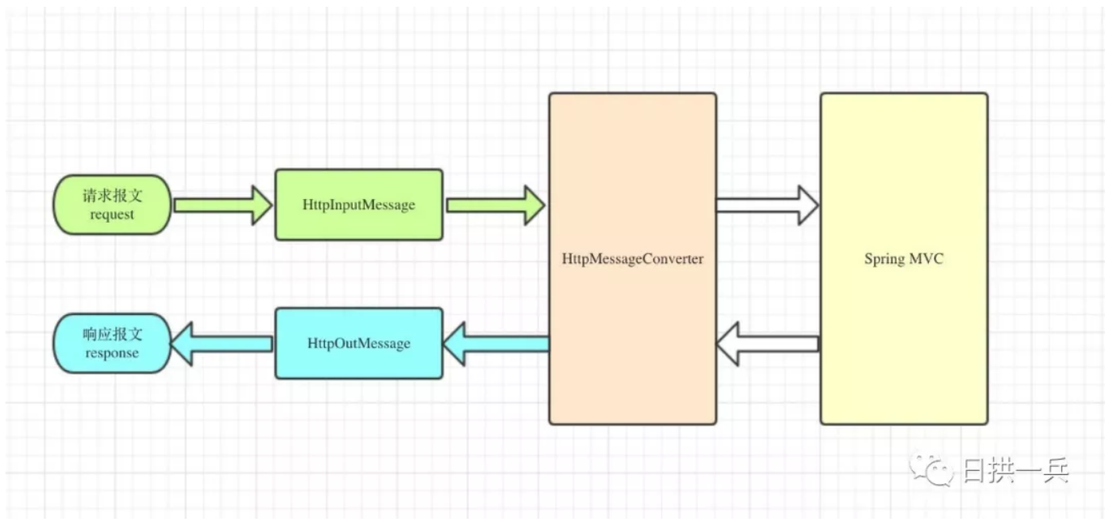
canRead 和 canWrite 的判断逻辑是什么呢？ 请看下图：



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。



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

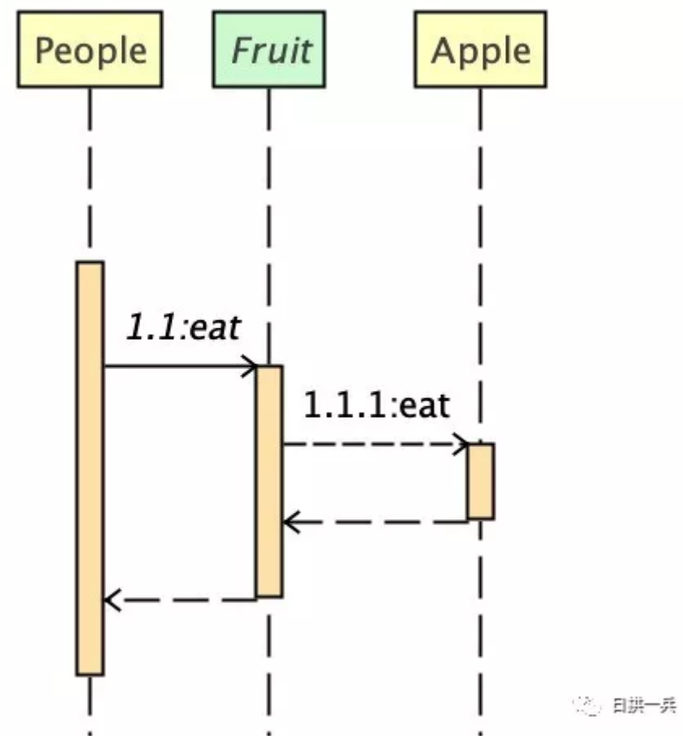
ProcessOn是一个在线作图工具的聚合平台，它可以在线画流程图、思维导图、UI原型图、UML、网络拓扑图、组织结构图等等，您无需担心下载和更新的问题，不管Mac还是Windows，一个浏览器就可以随时随地的发挥创意，规

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