

FROM

```
https://drun1baby.top/2023/12/07/Jackson-
%E5%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E5%BC%88%E4%B8%80%E5%BC%89%E6%BC%8F%E6%B4%9
E%E5%8E%9F%E7%90%86
```

0x01 Jackson 基本使用

Jackson 简介

Jackson 是一个开源的Java序列化和反序列化工具，可以将 Java 对象序列化为 XML 或 JSON 格式的字符串，以及将 XML 或 JSON 格式的字符串反序列化为 Java 对象。

由于其使用简单，速度较快，且不依靠除 JDK 外的其他库，被众多用户所使用。

使用 Jackson 进行序列化与反序列化

使用的 Jackson 包环境为 2.7.9 版本

pom.xml

XML

```
<dependencies>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.7.9</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-core</artifactId>
    <version>2.7.9</version>
  </dependency>
  <dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-annotations</artifactId>
    <version>2.7.9</version>
  </dependency>
</dependencies>
```

自定义 Person 类

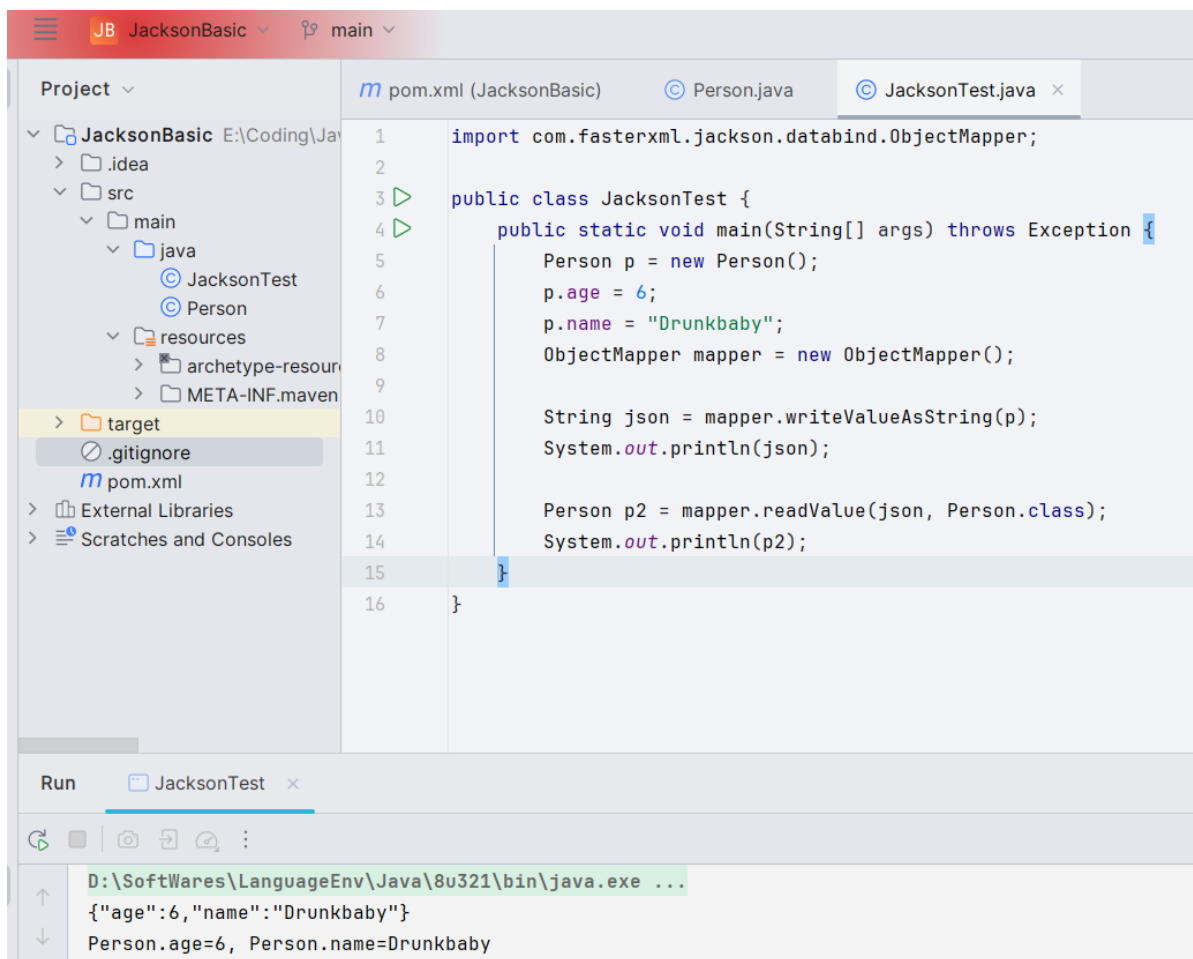
JAVA

```
public class Person {  
    public int age;  
    public String name;  
  
    @Override  
    public String toString() {  
        return String.format("Person.age=%d, Person.name=%s", age, name);  
    }  
}
```

接着编写 Jackson 的序列化与反序列化代码

JAVA

```
import com.fasterxml.jackson.databind.ObjectMapper;  
  
public class JacksonTest {  
    public static void main(String[] args) throws Exception {  
        Person p = new Person();  
        p.age = 6;  
        p.name = "Drunkbaby";  
        ObjectMapper mapper = new ObjectMapper();  
  
        String json = mapper.writeValueAsString(p);  
        System.out.println(json);  
  
        Person p2 = mapper.readValue(json, Person.class);  
        System.out.println(p2);  
    }  
}  
  
// {"age":6,"name":"Drunkbaby"}  
// Person.age=6, Person.name=Drunkbaby
```



0x02 Jackson 对于多态问题的解决 —— JacksonPolymorphicDeserialization

简单地说，Java 多态就是同一个接口使用不同的实例而执行不同的操作。

那么问题来了，如果对多态类的某一个子类实例在序列化后再进行反序列化时，如何能够保证反序列化出来的实例即是我们想要的那个特定子类的实例而非多态类的其他子类实例呢？—— Jackson 实现了 JacksonPolymorphicDeserialization 机制来解决这个问题。

JacksonPolymorphicDeserialization 即 Jackson 多态类型的反序列化：在反序列化某个类对象的过程中，如果类的成员变量不是具体类型（non-concrete），比如 Object、接口或抽象类，则可以在 JSON 字符串中指定其具体类型，Jackson 将生成具体类型的实例。

简单地说，就是将具体的子类信息绑定在序列化的内容中以便于后续反序列化的时候直接得到目标子类对象，其实现有两种，即 `DefaultTyping` 和 `@JsonTypeInfo` 注解。这里和前面学过的 fastjson 是很相似的。

下面具体介绍一下。

DefaultTyping

Jackson 提供一个 `enableDefaultTyping` 设置，其包含 4 个值，查看 `jackson-databind-2.7.9.jar!com/fasterxml/jackson/databind/ObjectMapper.java` 可看到相关介绍信息：

JAVA

```

public enum DefaultTyping {
    /**
     * This value means that only properties that have

```

```

    * {@link java.lang.Object} as declared type (including
    * generic types without explicit type) will use default
    * typing.
    */
    JAVA_LANG_OBJECT,

    /**
     * value that means that default typing will be used for
     * properties with declared type of {@link java.lang.Object}
     * or an abstract type (abstract class or interface).
     * Note that this does not include array types.
     * <p>
     * Since 2.4, this does NOT apply to {@link TreeNode} and its subtypes.
     */
    OBJECT_AND_NON_CONCRETE,

    /**
     * value that means that default typing will be used for
     * all types covered by {@link #OBJECT_AND_NON_CONCRETE}
     * plus all array types for them.
     * <p>
     * Since 2.4, this does NOT apply to {@link TreeNode} and its subtypes.
     */
    NON_CONCRETE_AND_ARRAYS,

    /**
     * value that means that default typing will be used for
     * all non-final types, with exception of small number of
     * "natural" types (String, Boolean, Integer, Double), which
     * can be correctly inferred from JSON; as well as for
     * all arrays of non-final types.
     * <p>
     * Since 2.4, this does NOT apply to {@link TreeNode} and its subtypes.
     */
    NON_FINAL
}

```

默认情况下，即无参数的 `enableDefaultTyping` 是第二个设置，`OBJECT_AND_NON_CONCRETE`。

下面分别对这几个选项进行说明。

JAVA_LANG_OBJECT

JAVA_LANG_OBJECT: 当被序列化或反序列化的类里的属性被声明为一个 Object 类型时，会对该 Object 类型的属性进行序列化和反序列化，并且明确规定类名。（当然，这个 Object 本身也得是一个可被序列化的类）

添加一个 Hacker 类

JAVA

```

package com.drunkbaby.defaultTyping;

public class Hacker {
    public String skill = "hiphop";
}

```

修改 Person 类，添加 Object 类型属性：

JAVA

```
package com.drunkbaby;

public class Person {
    public int age;
    public String name;
    public Object object;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s", age, name,
object == null ? "null" : object);
    }
}
```

新建 `JAVA_LANG_OBJECTTest.java`，添加 `enableDefaultTyping()` 并设置为

`JAVA_LANG_OBJECT`：

JAVA

```
package com.drunkbaby.defaultTyping;

import com.drunkbaby.Person;
import com.fasterxml.jackson.databind.ObjectMapper;

public class JAVA_LANG_OBJECTTest {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        p.age = 6;
        p.name = "drunkbaby";
        p.object = new Hacker();
        ObjectMapper mapper = new ObjectMapper();
        // 设置JAVA_LANG_OBJECT
        mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.JAVA_LANG_OBJECT);

        String json = mapper.writeValueAsString(p);
        System.out.println(json);

        Person p2 = mapper.readValue(json, Person.class);
        System.out.println(p2);
    }
}
```

这里我们同样写一个类，是没有添加 `enableDefaultTyping()` 的，来对比一下

JAVA

```
package com.drunkbaby.defaultTyping;

import com.drunkbaby.Person;
import com.fasterxml.jackson.databind.ObjectMapper;
```

```

public class NoJava_LANG_OBJECT {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        p.age = 6;
        p.name = "drunkbaby";
        p.object = new Hacker();
        ObjectMapper mapper = new ObjectMapper();

        String json = mapper.writeValueAsString(p);
        System.out.println(json);

        Person p2 = mapper.readValue(json, Person.class);
        System.out.println(p2);
    }
}

```



输出对比看到，通过 `enableDefaultTyping()` 设置设置 `JAVA_LANG_OBJECT` 后，会多输出 `Hacker` 类名，且在输出的 `Object` 属性时直接输出的是 `Hacker` 类对象，也就是说同时对 `Object` 属性对象进行了序列化和反序列化操作：

JSON

```

// 设置JAVA_LANG_OBJECT
{"age":6,"name":"milk7ea","object":["com.milk7ea.Hacker",{"skill":"Jackson"}]}
Person.age=6, Person.name=milk7ea, com.milk7ea.Hacker@7f9a81e8

// 未设置JAVA_LANG_OBJECT
{"age":6,"name":"milk7ea","object":{"skill":"Jackson"}}
Person.age=6, Person.name=milk7ea, {skill=Jackson}

```

OBJECT_AND_NON_CONCRETE

`OBJECT_AND_NON_CONCRETE`：除了前面提到的特征，当类里有 `Interface`、`AbstractClass` 类时，对其进行序列化和反序列化（当然这些类本身需要时合法的、可被序列化的对象）。

- 此外，`enableDefaultTyping()` 默认无参数的设置就是此选项。

添加一个 `Sex` 接口

JAVA

```

package com.drunkbaby.defaultTyping.object_and_non_concrete;

public interface Sex {
    public void setSex(int sex);
    public int getSex();
}

```

添加 `MySex` 类实现 `Sex` 接口类：

JAVA

```
package com.drunkbaby.defaultTyping.object_and_non_concrete;

public class MySex implements Sex {
    int sex;

    @Override
    public int getSex() {
        return sex;
    }

    @Override
    public void setSex(int sex) {
        this.sex = sex;
    }
}
```

修改 Person 类:

JAVA

```
public class Person {
    public int age;
    public String name;
    public Object object;
    public Sex sex;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s, %s", age, name,
            object == null ? "null" : object, sex == null ? "null" : sex);
    }
}
```

接着编写序列化与反序列化的代码

JAVA

```
package com.drunkbaby.defaultTyping.object_and_non_concrete;

import com.drunkbaby.Person;
import com.drunkbaby.defaultTyping.java_lang_object.Hacker;
import com.fasterxml.jackson.databind.ObjectMapper;

public class OBJECT_AND_NON_CONCRETE_Test {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        p.age = 6;
        p.name = "drunkbaby";
        p.object = new Hacker();
        p.sex = new MySex();
        ObjectMapper mapper = new ObjectMapper();
        // 设置OBJECT_AND_NON_CONCRETE
    }
}
```

```

mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.OBJECT_AND_NON_CONCRETE);

// 或直接无参调用，输出一样
//mapper.enableDefaultTyping();

String json = mapper.writeValueAsString(p);
System.out.println(json);

Person p2 = mapper.readValue(json, Person.class);
System.out.println(p2);
}
}

```

输出，可以看到该Interface类属性被成功序列化和反序列化：

JSON

```

{"age":6,"name":"drunkbaby","object":
["com.drunkbaby.defaultTyping.java_lang_object.Hacker",{"skill":"hiphop"}],"sex":
["com.drunkbaby.defaultTyping.object_and_non_concrete.MySex",{"sex":0}]}
Person.age=6, Person.name=drunkbaby,
com.drunkbaby.defaultTyping.java_lang_object.Hacker@6d00a15d,
com.drunkbaby.defaultTyping.object_and_non_concrete.MySex@51efea79

```

NON_CONCRETE_AND_ARRAYS

NON_CONCRETE_AND_ARRAYS：除了前面提到的特征外，还支持 Array 类型。

编写序列化与反序列化的代码，在 Object 属性中存在的是数组：

JAVA

```

package com.drunkbaby.defaultTyping.non_concrete_and_arrays;

import com.drunkbaby.Person;
import com.drunkbaby.defaultTyping.java_lang_object.Hacker;
import com.drunkbaby.defaultTyping.object_and_non_concrete.MySex;
import com.fasterxml.jackson.databind.ObjectMapper;

public class NON_CONCRETE_AND_ARRAYS_Test {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        p.age = 6;
        p.name = "drunkbaby";
        Hacker[] hackers = new Hacker[2];
        hackers[0] = new Hacker();
        hackers[1] = new Hacker();
        p.object = hackers;
        p.sex = new MySex();
        ObjectMapper mapper = new ObjectMapper();
        // 设置NON_CONCRETE_AND_ARRAYS

        mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_CONCRETE_AND_ARRAYS);
    }
}

```



```

        String json = mapper.writeValueAsString(p);
        System.out.println(json);

        Person p2 = mapper.readValue(json, Person.class);
        System.out.println(p2);
    }
}

```

输出看到，类名变成了 "[L"+类名+";"，序列化 Object 之后为数组形式，反序列化之后得到 [Lcom.milk7ea.Hacker; 类对象，说明对 Array 类型成功进行了序列化和反序列化：

NON_FINAL

NON_FINAL：除了前面的所有特征外，包含即将被序列化的类里的全部、非 final 的属性，也就是相当于整个类、除 final 外的属性信息都需要被序列化和反序列化。

修改 Person 类，添加 Hacker 属性：

JAVA

```

public class Person {
    public int age;
    public String name;
    public Object object;
    public Sex sex;
    public Hacker hacker;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s, %s, %s", age,
            name, object == null ? "null" : object, sex == null ? "null" : sex, hacker ==
            null ? "null" : hacker);
    }
}

```

编写序列化与反序列化类

JAVA

```

package com.drunkbaby.defaultTyping.non_final;

import com.drunkbaby.Person;
import com.drunkbaby.defaultTyping.java_lang_object.Hacker;
import com.drunkbaby.defaultTyping.object_and_non_concrete.MySex;
import com.fasterxml.jackson.databind.ObjectMapper;

public class NON_FINAL_Test {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        p.age = 6;
        p.name = "drunkbaby";
        p.object = new Hacker();
        p.sex = new MySex();
        p.hacker = new Hacker();
    }
}

```

```

    ObjectMapper mapper = new ObjectMapper();
    // 设置NON_FINAL
    mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);

    String json = mapper.writeValueAsString(p);
    System.out.println(json);

    Person p2 = mapper.readValue(json, Person.class);
    System.out.println(p2);
}
}

```

输出看到，成功对非 final 的 hacker 属性进行序列化和反序列化：

JSON

```

["com.drunkbaby.Person",{"age":6,"name":"drunkbaby","object":
["com.drunkbaby.defaultTyping.java_lang_object.Hacker",{"skill":"hiphop"}],"sex":
["com.drunkbaby.defaultTyping.object_and_non_concrete.MySex",{"sex":0}],"hacker":
["com.drunkbaby.defaultTyping.java_lang_object.Hacker",{"skill":"hiphop"}]]
Person.age=6, Person.name=drunkbaby,
com.drunkbaby.defaultTyping.java_lang_object.Hacker@6d00a15d,
com.drunkbaby.defaultTyping.object_and_non_concrete.MySex@51efea79,
com.drunkbaby.defaultTyping.java_lang_object.Hacker@5034c75a

```

小结

从前面的分析知道，DefaultTyping 的几个设置选项是逐渐扩大适用范围的，如下表：

DefaultTyping类型	描述说明
JAVA_LANG_OBJECT	属性的类型为Object
OBJECT_AND_NON_CONCRETE	属性的类型为Object、Interface、AbstractClass
NON_CONCRETE_AND_ARRAYS	属性的类型为Object、Interface、AbstractClass、Array
NON_FINAL	所有除了声明为final之外的属性

@JsonTypeInfo 注解

@JsonTypeInfo 注解是 Jackson 多态类型绑定的一种方式，支持下面5种类型的取值：

JAVA

```

@JsonTypeInfo(use = JsonTypeInfo.Id.NONE)
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS)
@JsonTypeInfo(use = JsonTypeInfo.Id.MINIMAL_CLASS)
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME)
@JsonTypeInfo(use = JsonTypeInfo.Id.CUSTOM)

```

下面我们逐个看下。

JsonTypeInfo.Id.NONE

JsonTypeInfo_Id_NONE_Test.java

JAVA

```
public class JsonTypeInfo_Id_NONE_Test {
    public static void main(String[] args) throws Exception {
        Person2 p = new Person2();
        p.age = 6;
        p.name = "drunkbaby";
        p.object = new Hacker();
        ObjectMapper mapper = new ObjectMapper();

        String json = mapper.writeValueAsString(p);
        System.out.println(json);

        Person2 p2 = mapper.readValue(json, Person2.class);
        System.out.println(p2);
    }
}
```

Person2 类, 给 object 属性添加 @JsonTypeInfo 注解, 指定为 JsonTypeInfo.Id.NONE:

JAVA

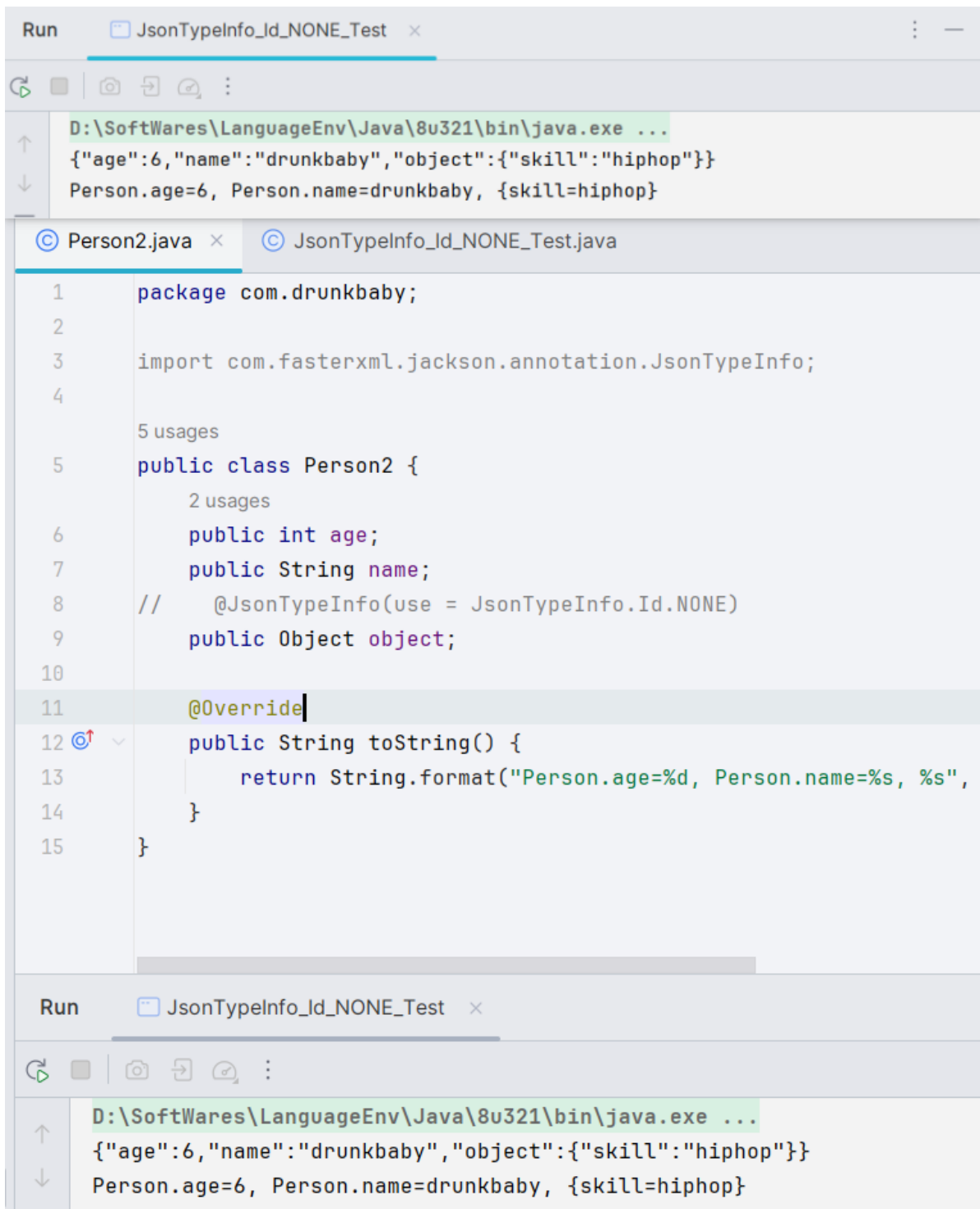
```
public class Person2 {
    public int age;
    public String name;
    @JsonTypeInfo(use = JsonTypeInfo.Id.NONE)
    public Object object;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s", age, name,
            object == null ? "null" : object);
    }
}
```

输出看到, 和没有设置值为 JsonTypeInfo.Id.NONE 的 @JsonTypeInfo 注解是一样的:

JSON

```
{"age":6,"name":"drunkbaby","object":{"skill":"hiphop"}}
Person.age=6, Person.name=drunkbaby, {skill=hiphop}
```



JsonTypeInfo.Id.CLASS

修改 Person2 类中的 object 属性 `@JsonTypeInfo` 注解值为 `JsonTypeInfo.Id.CLASS`

输出看到，object属性中多了 `"@class":"com.drunkbaby.Hacker"`，即含有具体的类的信息，同时反序列化出来的object属性Hacker类对象，即能够成功对指定类型进行序列化和反序列化：

JSON

```
{ "age":6, "name":"drunkbaby", "object":  
  { "@class":"com.drunkbaby.Hacker", "skill":"hiphop" } }  
Person.age=6, Person.name=drunkbaby, com.drunkbaby.Hacker@55f3ddb1
```

也就是说，在Jackson反序列化的时候如果使用了 `JsonTypeInfo.Id.CLASS` 修饰的话，可以通过@class的方式指定相关类，并进行相关调用。

JsonTypeInfo.Id.MINIMAL_CLASS

修改 Person2 类中的object属性 @JsonTypeInfo 注解值为 JsonTypeInfo.Id.MINIMAL_CLASS

输出看到, object属性中多了 "@c":"com.drunkbaby.Hacker", 即使用 @c 替代了 @class, 官方描述中的意思是缩短了相关类名, 实际效果和 JsonTypeInfo.Id.CLASS 类似, 能够成功对指定类型进行序列化和反序列化, 都可以用于指定相关类并进行相关的调用:

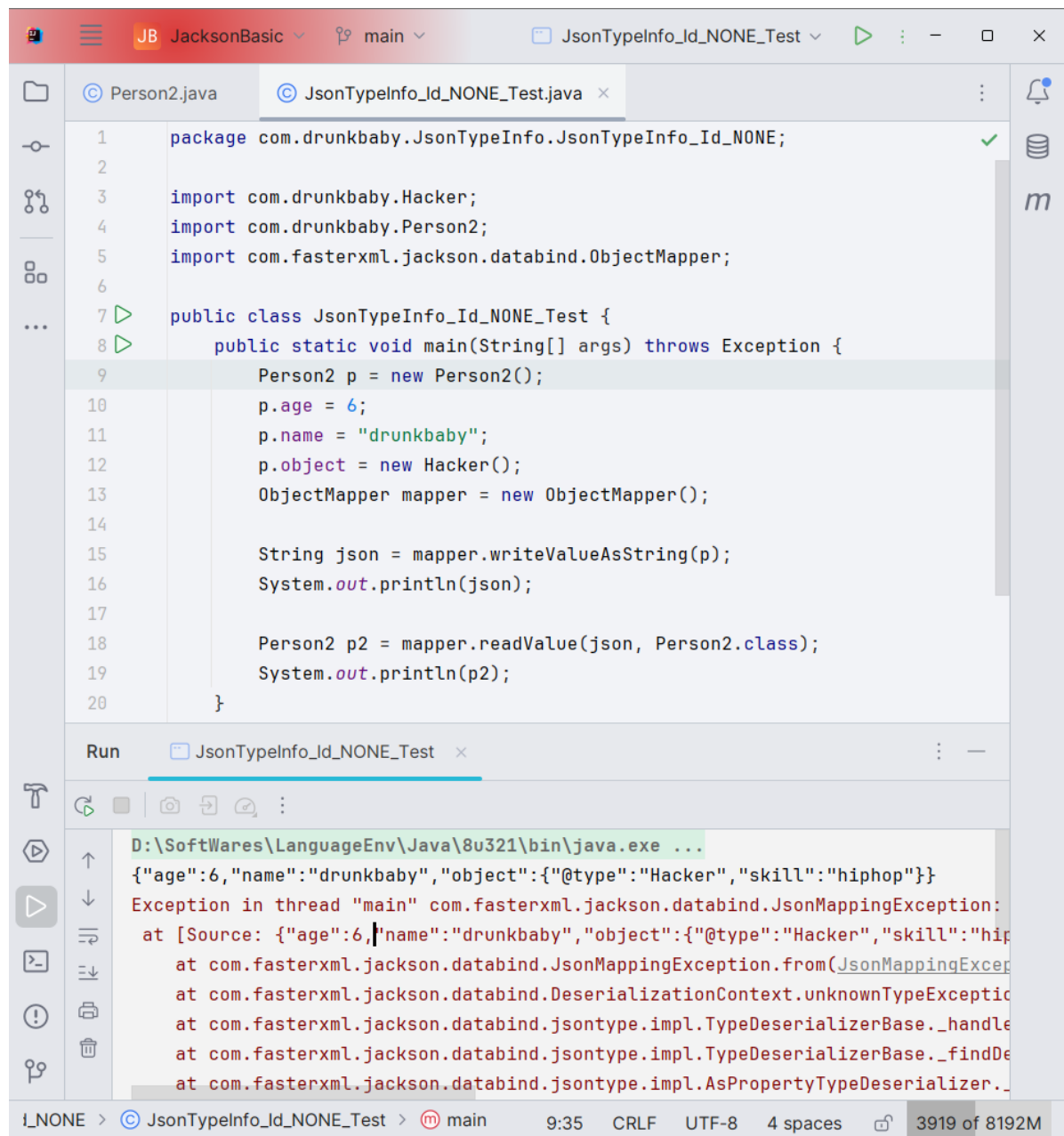
JSON

```
{"age":6,"name":"drunkbaby","object":  
{"@c":"com.drunkbaby.Hacker","skill":"hiphop"}}  
Person.age=6, Person.name=drunkbaby, com.drunkbaby.Hacker@18be83e4
```

JsonTypeInfo.Id.NAME

修改 Person2 类中的object属性 @JsonTypeInfo 注解值为 JsonTypeInfo.Id.NAME

输出看到, object 属性中多了 "@type":"Hacker", 但没有具体的包名在内的类名, 因此在后面的反序列化的时候会报错, 也就是说这个设置值是不能被反序列化利用的:



```
package com.drunkbaby.JsonTypeInfo.JsonTypeInfo_Id_NONE;  
  
import com.drunkbaby.Hacker;  
import com.drunkbaby.Person2;  
import com.fasterxml.jackson.databind.ObjectMapper;  
  
public class JsonTypeInfo_Id_NONE_Test {  
    public static void main(String[] args) throws Exception {  
        Person2 p = new Person2();  
        p.age = 6;  
        p.name = "drunkbaby";  
        p.object = new Hacker();  
        ObjectMapper mapper = new ObjectMapper();  
  
        String json = mapper.writeValueAsString(p);  
        System.out.println(json);  
  
        Person2 p2 = mapper.readValue(json, Person2.class);  
        System.out.println(p2);  
    }  
}
```

Run JsonTypeInfo_Id_NONE_Test

```
D:\SoftWares\LanguageEnv\Java\8u321\bin\java.exe ...  
{ "age":6,"name":"drunkbaby","object":{"@type":"Hacker","skill":"hiphop"}}  
Exception in thread "main" com.fasterxml.jackson.databind.JsonMappingException:  
  at [Source: {"age":6,"name":"drunkbaby","object":{"@type":"Hacker","skill":"hiphop"}}; line: 1, column: 1]  
  at com.fasterxml.jackson.databind.JsonMappingException.from(JsonMappingExceptionSource.java:147)  
  at com.fasterxml.jackson.databind.DeserializationContext.unknownTypeException(DeserializationContext.java:1910)  
  at com.fasterxml.jackson.databind.jsontype.impl.TypeDeserializerBase._handleUnsupportedTypeException(TypeDeserializerBase.java:137)  
  at com.fasterxml.jackson.databind.jsontype.impl.TypeDeserializerBase._findDeserializer(TypeDeserializerBase.java:158)  
  at com.fasterxml.jackson.databind.jsontype.impl.AsPropertyTypeDeserializer._deserializeTypedFromId(AsPropertyTypeDeserializer.java:112)  
  at com.fasterxml.jackson.databind.jsontype.impl.AsPropertyTypeDeserializer.deserializeTypedFromId(AsPropertyTypeDeserializer.java:58)  
  at com.fasterxml.jackson.databind.DeserializationContext.handleTypedFromId(DeserializationContext.java:181)  
  at com.fasterxml.jackson.databind.jsontype.impl.AsPropertyTypeDeserializer.deserializeTypedFromId(AsPropertyTypeDeserializer.java:58)  
  at com.fasterxml.jackson.databind.DeserializationContext.handleTypedFromId(DeserializationContext.java:181)  
  at com.fasterxml.jackson.databind.ObjectMapper._readValueAndGenerateMetadata(ObjectMapper.java:4857)  
  at com.fasterxml.jackson.databind.ObjectMapper.readValue(ObjectMapper.java:1443)  
  at com.drunkbaby.JsonTypeInfo.JsonTypeInfo_Id_NONE_Test.main(JsonTypeInfo_Id_NONE_Test.java:19)
```

JsonTypeInfo.Id.CUSTOM

其实这个值时提供给用户自定义的意思，我们是没办法直接使用的，需要手动写一个解析器才能配合使用，直接运行会抛出异常：

小结

由前面测试发现，当 `@JsonTypeInfo` 注解设置为如下值之一并且修饰的是 `Object` 类型的属性时，可以利用来触发 Jackson 反序列化漏洞：

- `JsonTypeInfo.Id.CLASS`
- `JsonTypeInfo.Id.MINIMAL_CLASS`

0x03 反序列化中类属性方法的调用

这里只针对 Jackson 反序列化过程中存在的一些方法调用进行分析，并且只针对应用 JacksonPolymorphicDeserialization 机制的场景进行分析。

下面简单看下两个实现方式间是否有区别。

当使用 DefaultTyping 时

新增 Person3 类：

JAVA

```
public class Person3 {
    public int age;
    public String name;
    public Sex sex;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s", age, name, sex
== null ? "null" : sex);
    }
}
```

在 MySex 类中的方法中添加输出：

JAVA

```
public class MySex2 implements Sex {
    int sex;
    public MySex2() {
        System.out.println("MySex构造函数");
    }

    @Override
    public int getSex() {
        System.out.println("MySex.getSex");
        return sex;
    }

    @Override
    public void setSex(int sex) {
```

```

        System.out.println("MySex.setSex");
        this.sex = sex;
    }
}

```

修改反序列化代码，只进行反序列化操作并调用无参数的 `enableDefaultTyping()`：

JAVA

```

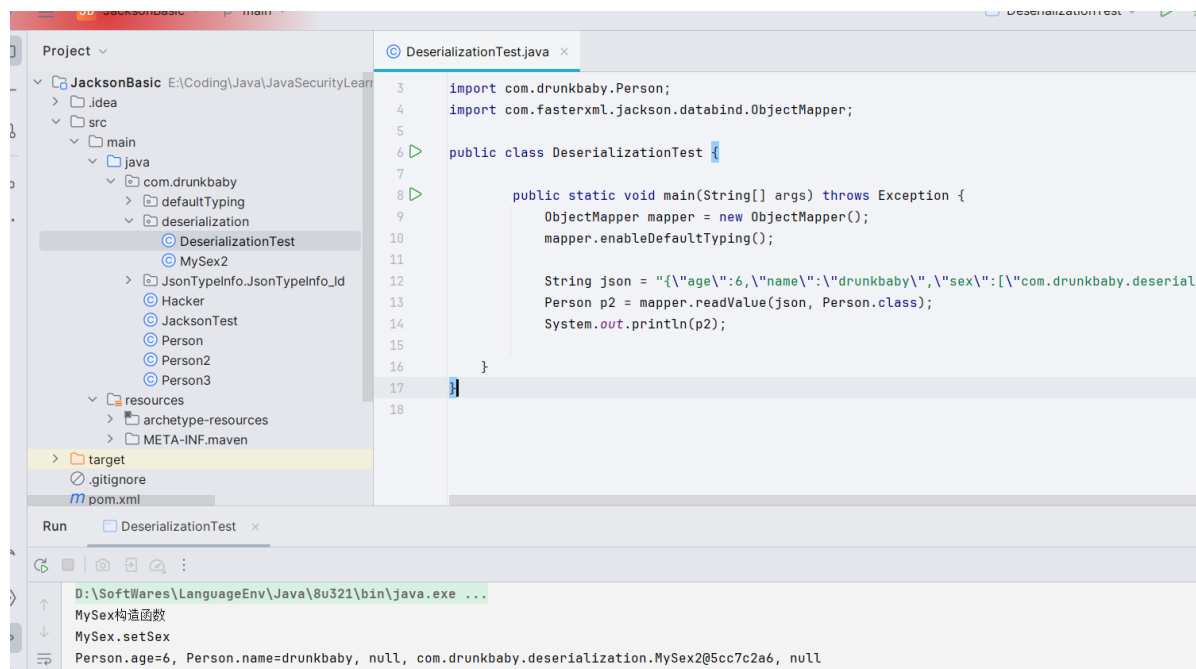
public class DeserializationTest {

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        mapper.enableDefaultTyping();

        String json = "{\"age\":6,\"name\":\"drunkbaby\",\"sex\":[" +
            "\"com.drunkbaby.deserialization.MySex2\",{" +
            "\"sex\":\"1\"}]}";
        Person3 p2 = mapper.readValue(json, Person3.class);
        System.out.println(p2);
    }
}

```

输出，看到调用了目标类的构造函数和 setter 方法：



当使用 @JsonTypeInfo 注解时

修改 Person3 类，在 sex 属性前添加注解：

JAVA

```

public class Person3 {
    public int age;
    public String name;
    @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS)
    // 或 @JsonTypeInfo(use = JsonTypeInfo.Id.MINIMAL_CLASS)
    public Sex sex;

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s", age, name, sex
== null ? "null" : sex);
    }
}

```

修改 DeserializationTest2.java, 注释掉 enableDefaultTyping():

JAVA

```

public class DeserializationTest2 {

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        // mapper.enableDefaultTyping();

        String json = "{\"age\":6,\"name\":\"drunkbaby\",\"sex\":1}";
        Person3 p2 = mapper.readValue(json, Person3.class);
        System.out.println(p2);

    }
}

```

输出看到, 和使用 DefaultTyping 是一样的:

JSON

```

MySex构造函数
MySex.setSex
Person.age=6, Person.name=drunkbaby,
com.drunkbaby.deserialization.MySex2@6a2bcfcb

```

Jackson 反序列化调试分析

Jackson 反序列化的过程其实就分为两步, 第一步是通过构造函数生成实例, 第二部是设置实例的属性值。

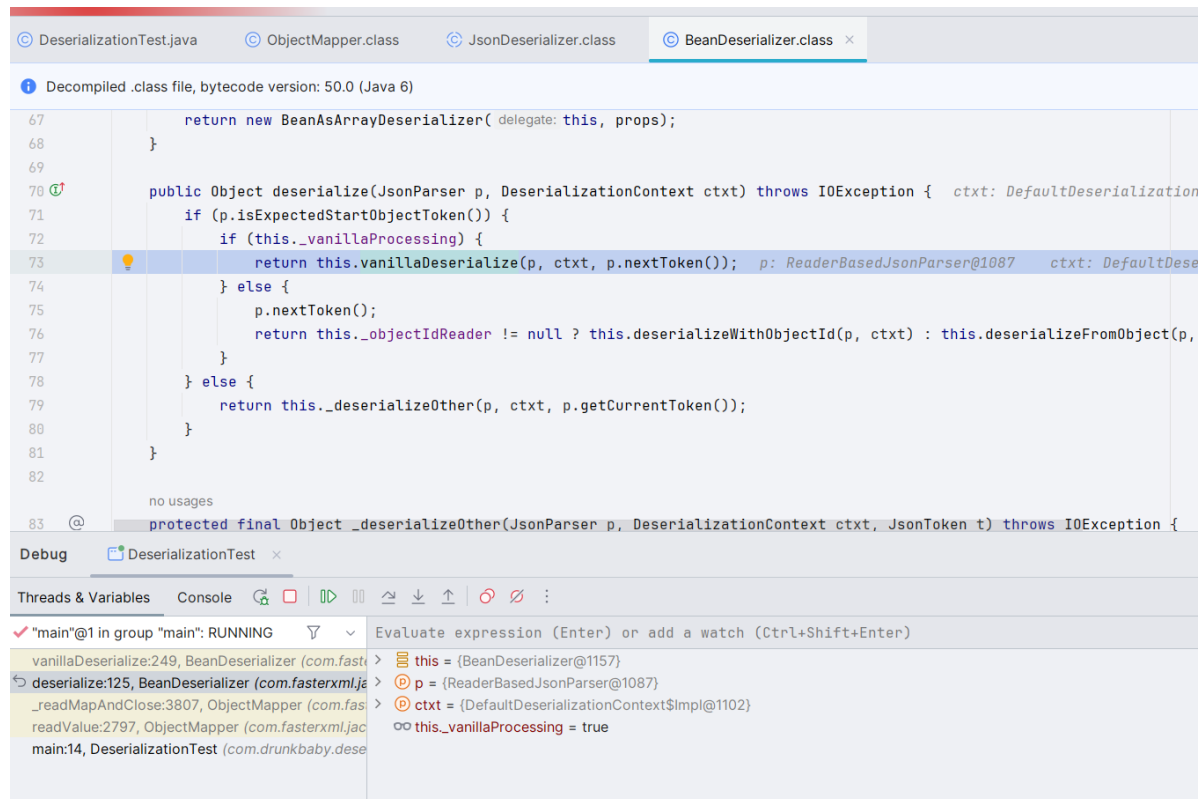
这里以第一个例子来进行 Jackson 反序列化过程的调试分析, 在 `Person p2 = mapper.readValue(json, Person.class);` 处打上断点, 同时在 MySex2 类的构造函数、getter、setter 方法中设置断点, 然后开始调试

另外, 为了方便, 给 Person3 类加上个构造函数, 随后开始调试

首先反序列化跟进来, 在 `com.fasterxml.jackson.databind.ObjectMapper._readMapAndClose()` 方法这里, 先进行 `JsonToken` 的初始化, 随后进行进一步的反序列化操作。

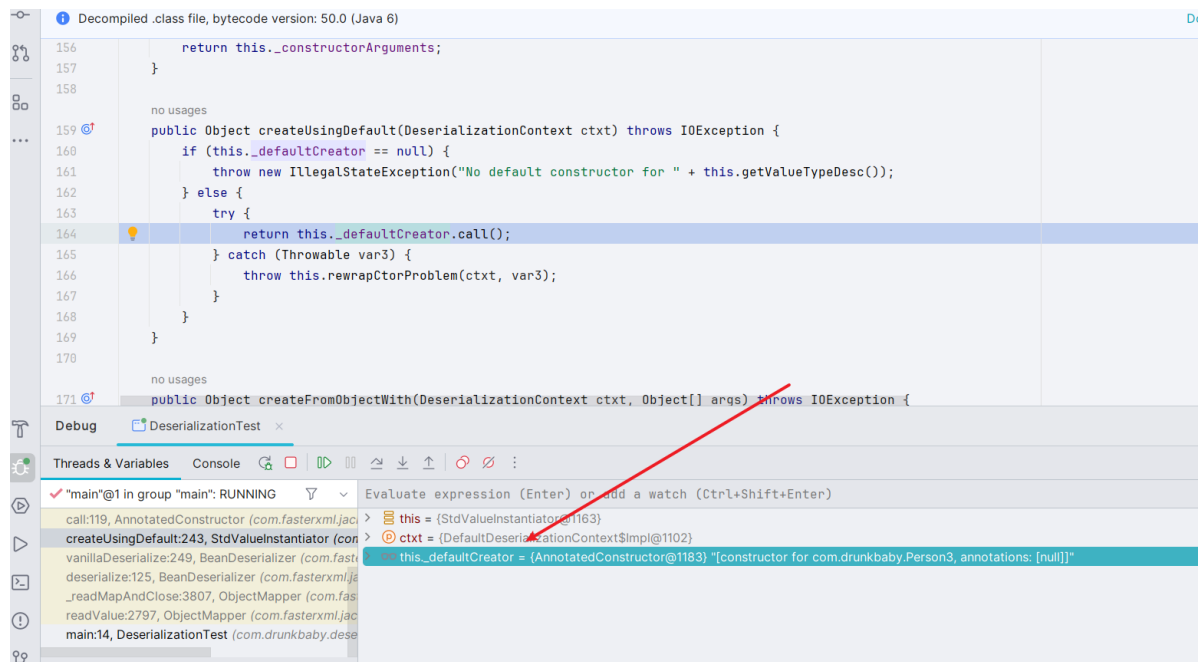
当初始化是第一次的时候，会先调到

`com.fasterxml.jackson.databind.deser.BeanDeserializer#vanillaDeserialize()` 方法

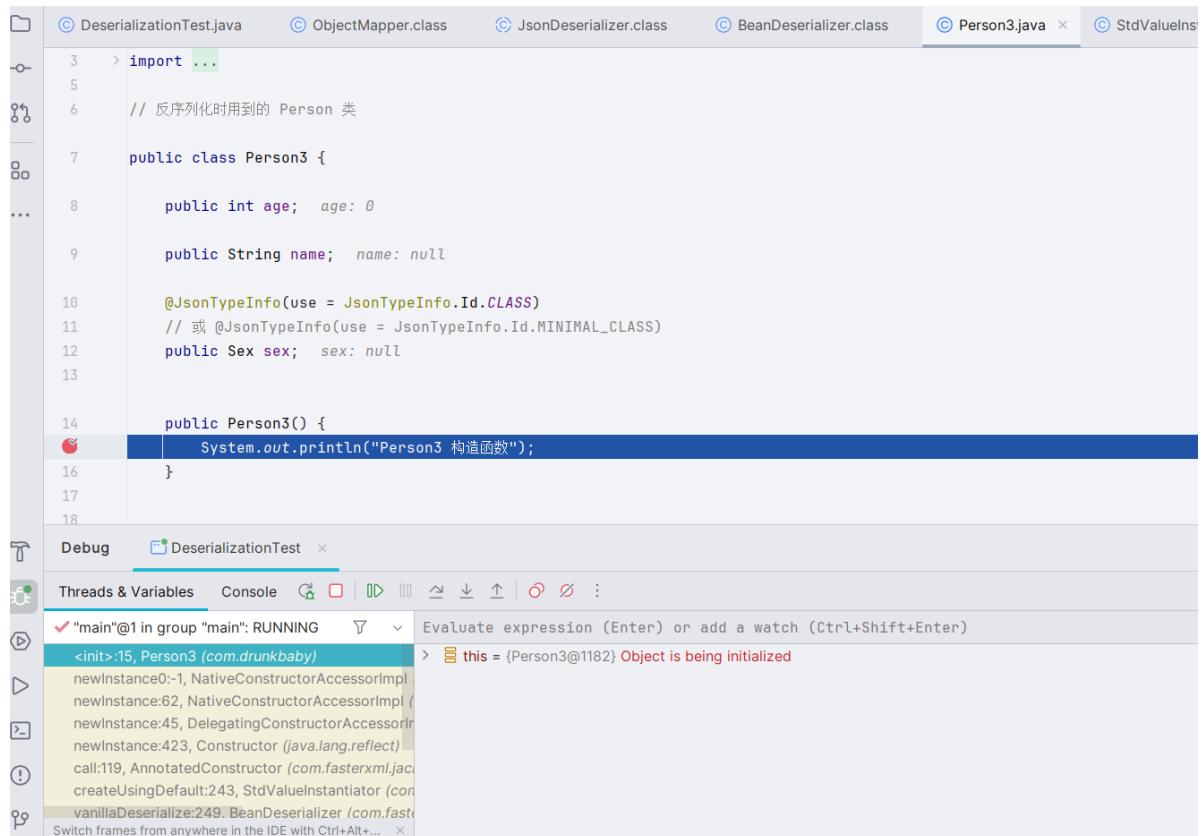


跟进，`vanillaDeserialize()` 方法调用了 `createUsingDefault()` 方法，这个方法的作用是调用指定类的无参构造函数，生成类实例。

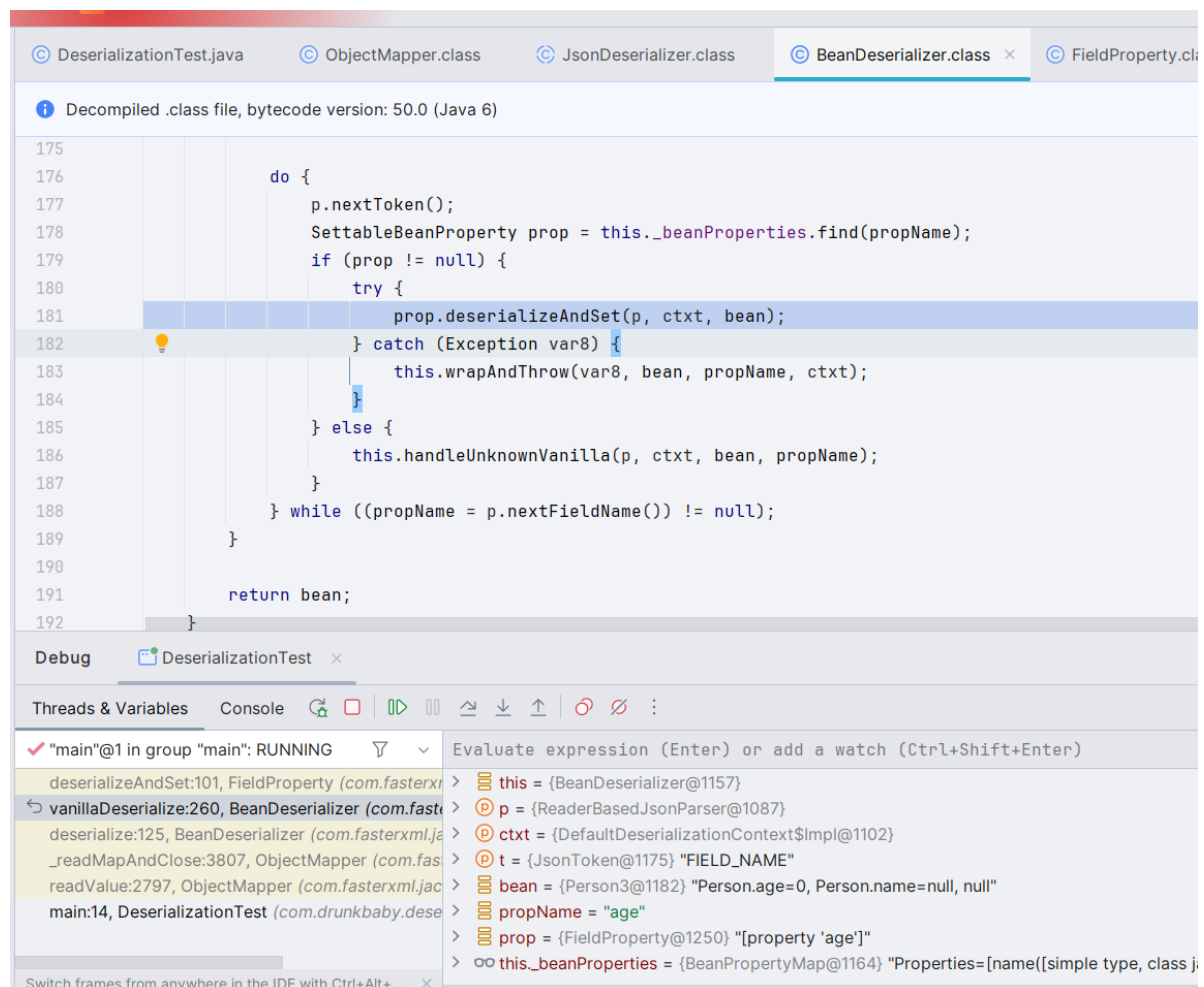
再次跟进就是调用 `call()` 方法了



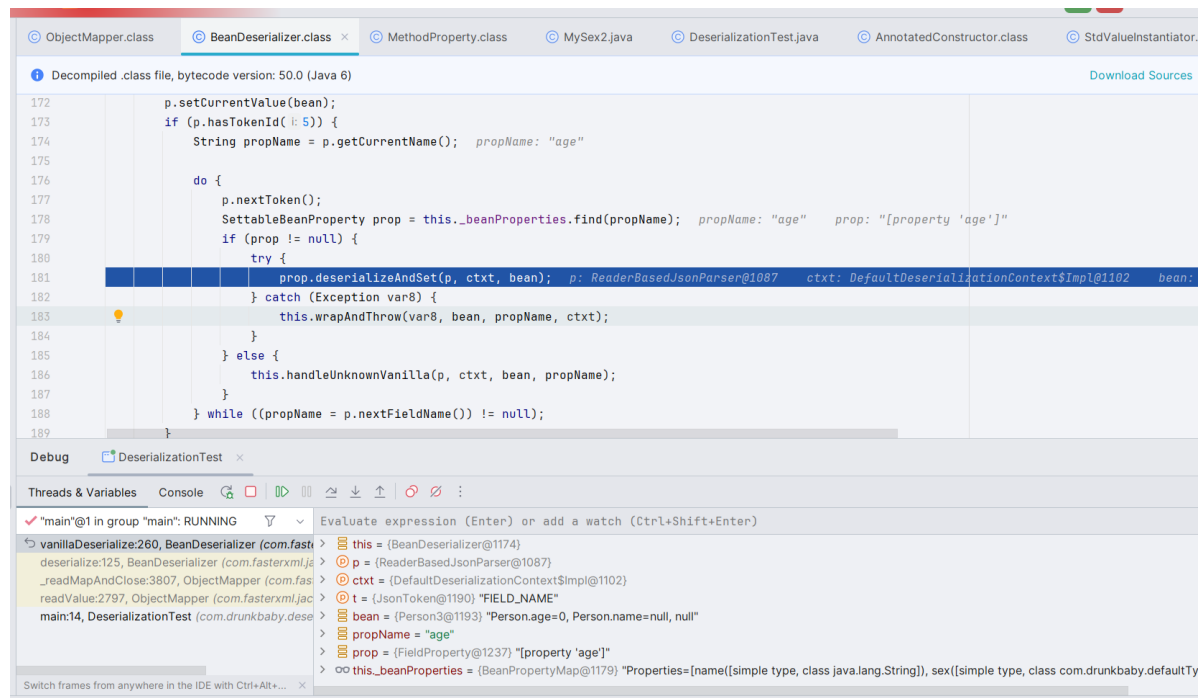
走到了构造函数



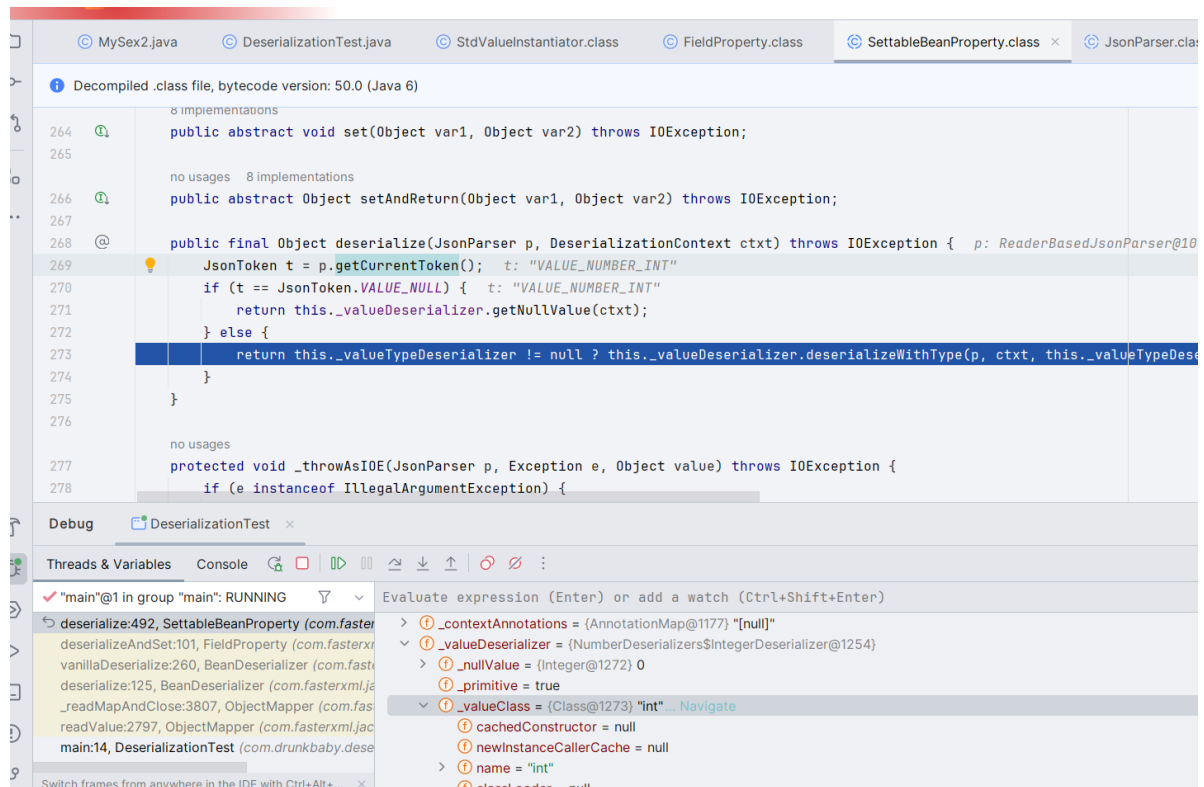
在初始化完毕之后，会调用 `deserializeAndSet()` 方法，完成了一个嵌套的过程，具体的细节后面会讲。



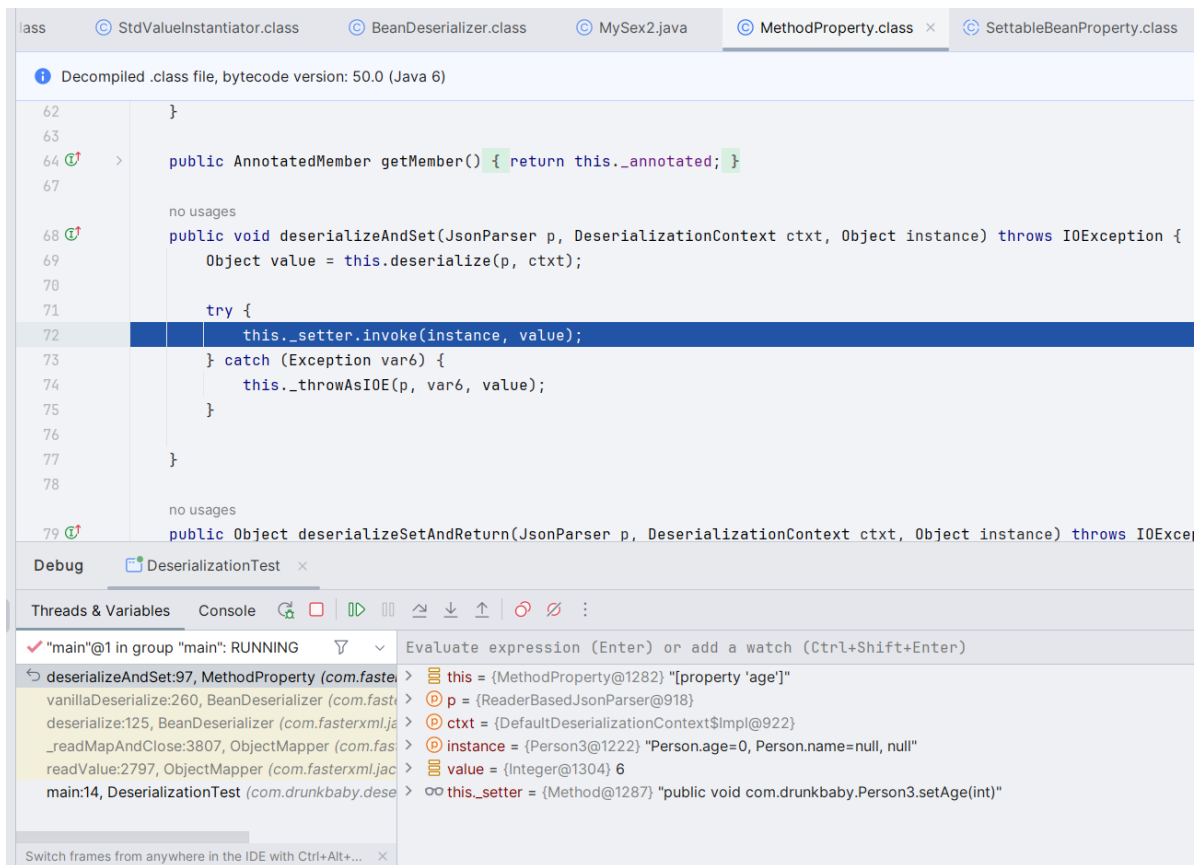
跟进去之后会发现这里调用了 `this.deserialize()` 方法，跟进 `deser.deserialize()`，具体的业务逻辑是在 `com.fasterxml.jackson.databind.deser.BeanDeserializer#deserializeAndSet()` 方法执行的，实际上就是我们前面说的 **Jackson 反序列化的过程其实就分为两步，第一步是通过构造函数生成实例，第二部是设置实例的属性值。** 跟进



继续跟进 `this.deserialize()`，这里先拿了 json 数据的数据类型，接下来判断这个节点的数据类型是否为 null，如果不为 null，再判断 `this._valueTypeDeserializer` 是否为空，如果不为空则继续调用 `this._valueDeserializer.deserialize()` 方法

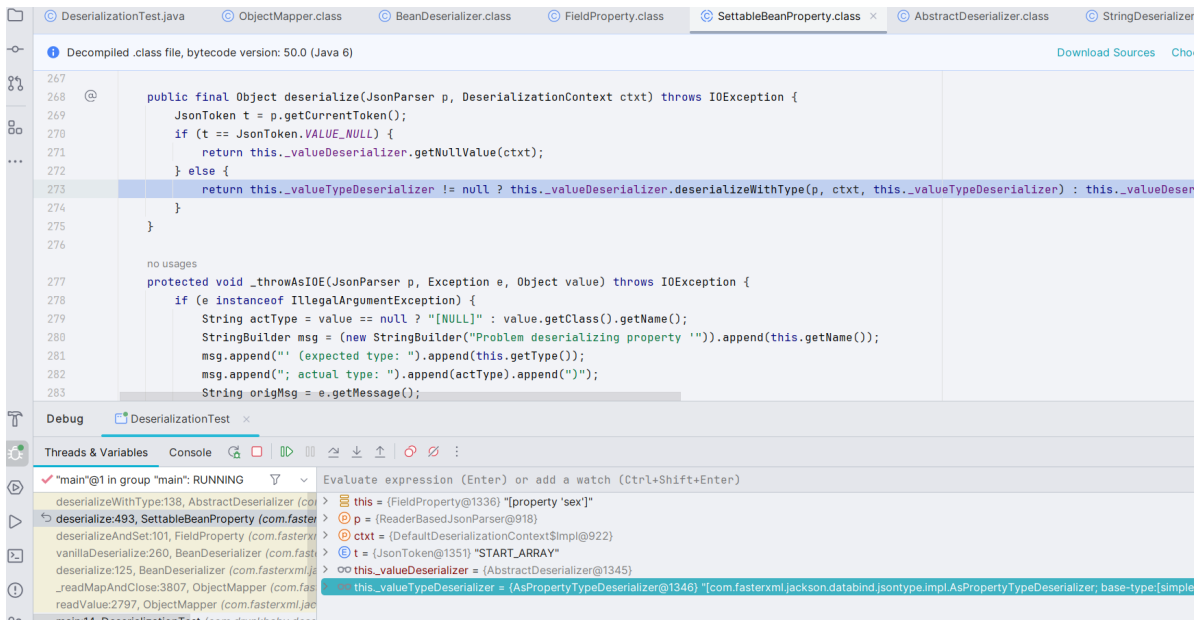


这里 `t` 是存在的，且反序列化程序也是存在的，所以调用 `deserialize()` 方法

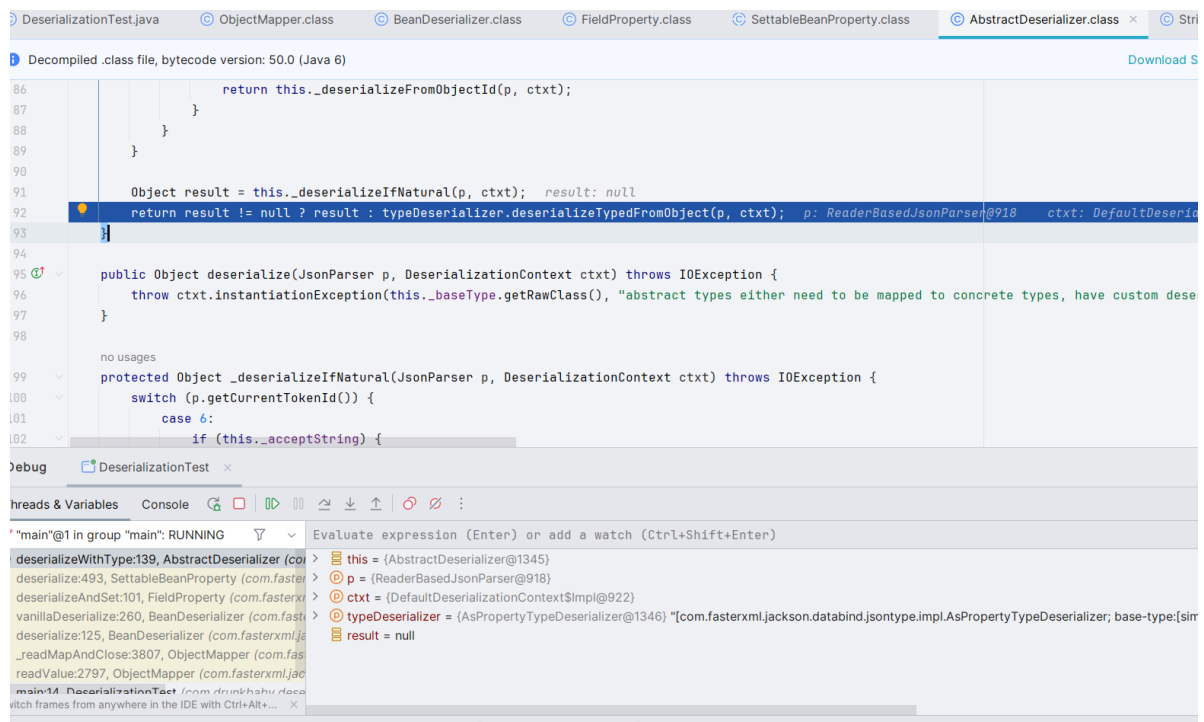


在调用完成之后又会重新回到 `BeanDeserializer.vanillaDeserialize()` 函数中的 do while 循环，继续获取值，继续调用，达到递归的效果。

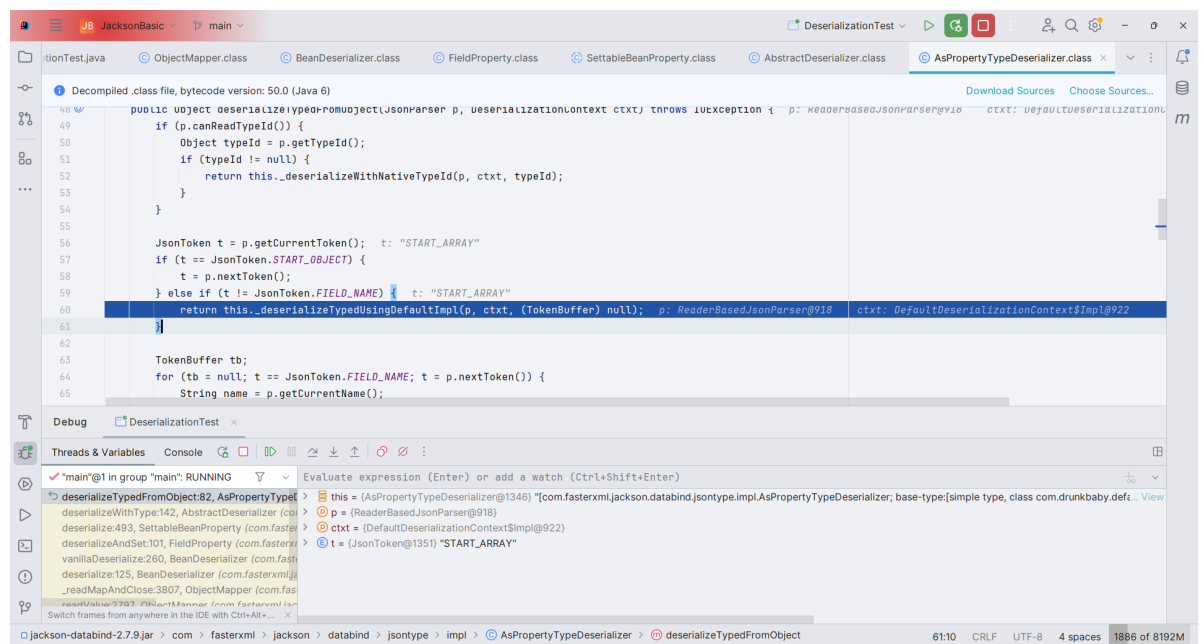
不太一样的是在 `SettableBeanProperty.deserialize()` 函数中进入到了调用 `deserializeWithType()` 函数解析的代码逻辑，因为此时 `_valueTypeDeserializer` 值不为 null：



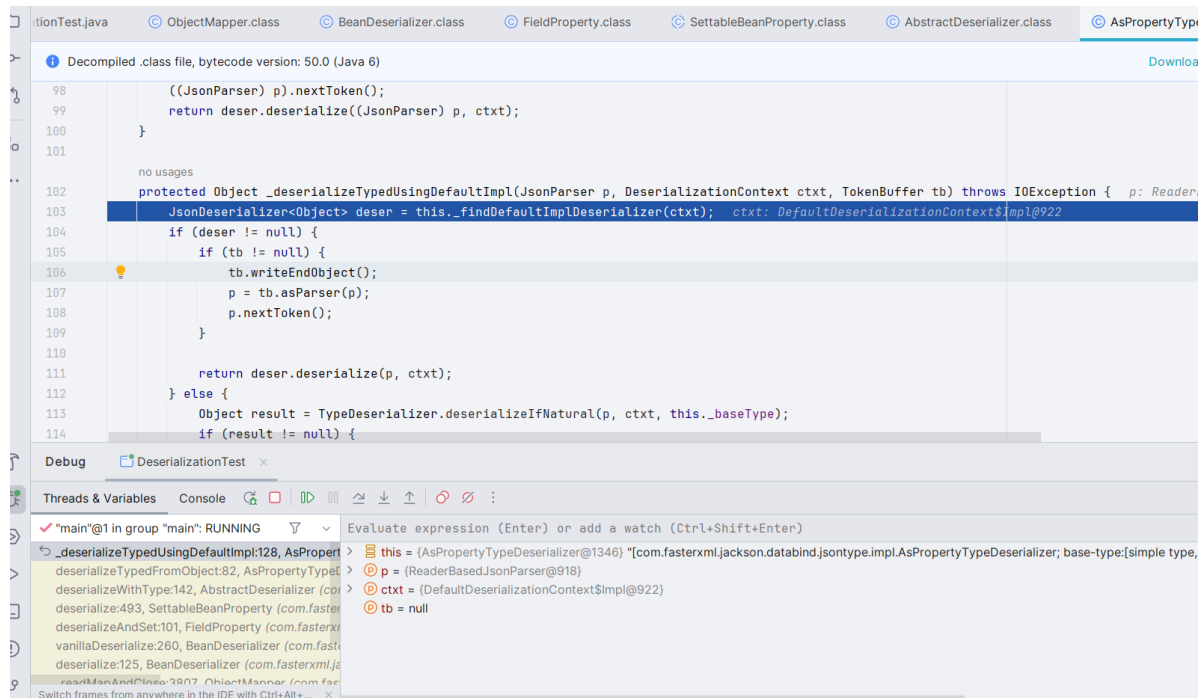
往下，先判断反序列化的类型，因为这里是数组，所以会返回 null，再跟进 `deserializeTypedFromObject()` 方法



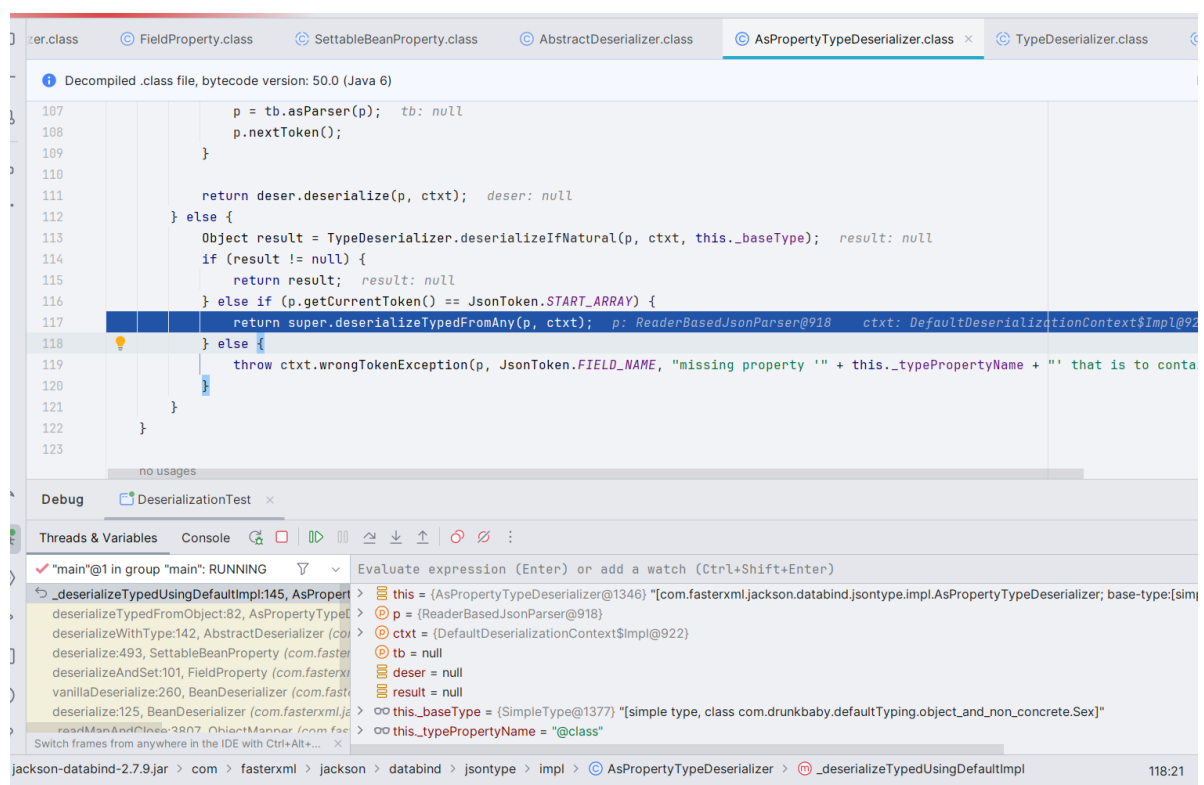
同样因为是数组，跟进 `_deserializeTypedUsingDefaultImpl()` 方法



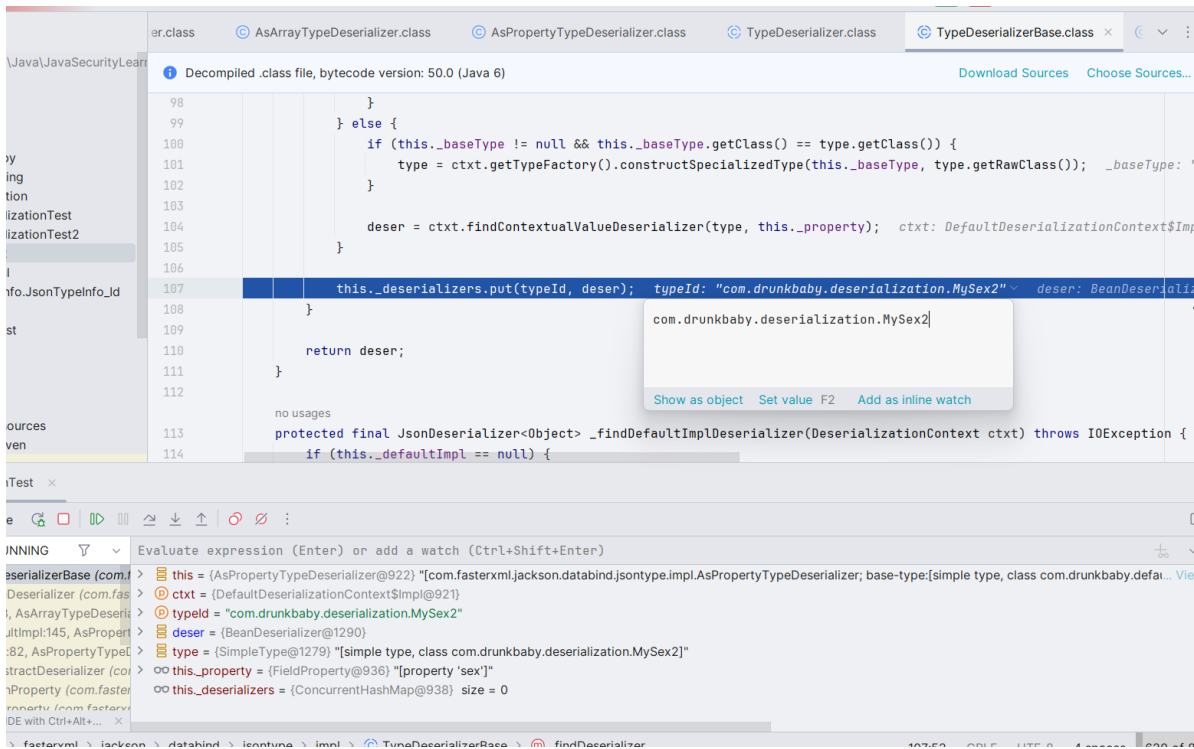
随后寻找反序列化的类



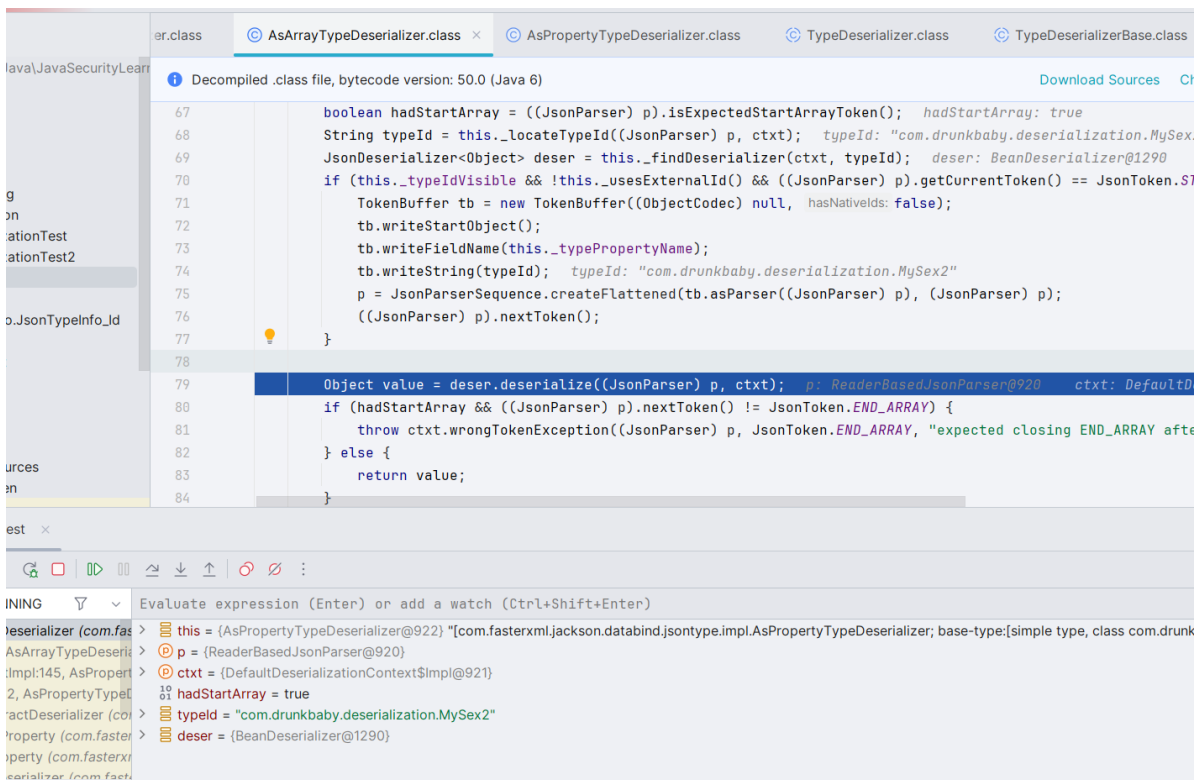
因为这里的数据类型是数组，不匹配任意一项，所以最后调用了 `deserializeTypedFromAny()` 方法，这个方法最终是让程序去已有的类里面找，很明显这里找的是 `com.drunkbaby.serialization.MySex2` 类



跟进去，其中调用 `findContextualValueDeserializer()` 找到 `typed` 类型对应的反序列化器，然后缓存到 `_deserializers` 这个 Map 变量中，然后返回该反序列化器



接着程序回到数组类型解析的 `AsArrayTypeDeserializer._deserialize()` 函数中往下执行，用刚刚获取到的反序列化器来解析 sex 属性中数组内的具体类型实例：



这里的逻辑比较简单，判断目前这个数据类型是否已经有对应的反序列化处理器了，如果没有则最终还是走原来那一套，如果有的话则走 Jackson 自己的规则。

至于后续的逻辑和最开始的反序列化逻辑是很类似的，简单看一下关键部分的调用栈

JAVA


```
<init>: 8, MySex2 (com.drunkbaby.deserialization)
newInstance0:-1, NativeConstructorAccessorImpl (sun.reflect)
newInstance:62, NativeConstructorAccessorImpl (sun.reflect)
newInstance:45, DelegatingConstructorAccessorImpl (sun.reflect)
newInstance:423, Constructor (java.lang.reflect)
call:119, AnnotatedConstructor (com.fasterxml.jackson.databind.introspect)
createUsingDefault:243, StdValueInstantiator
(com.fasterxml.jackson.databind.deser.std)
vanillaDeserialize:249, BeanDeserializer (com.fasterxml.jackson.databind.deser)
deserialize:125, BeanDeserializer (com.fasterxml.jackson.databind.deser)
_deserialize:110, AsArrayTypeDeserializer
(com.fasterxml.jackson.databind.jsontype.impl)
```

后续过程就不再展开了，是相似的调用过程。

至此，整个函数调用过程大致过了一遍。使用@JsonTypeInfo 注解的函数调用过程也是一样的。

- 简单梳理一遍，Jackson 反序列化的过程为，先调用通过无参的构造函数生成目标类实例，接着是根据属性值是否是数组的形式即是否带类名来分别调用不同的函数来设置实例的属性值，其中会调用 Object 类型属性的构造函数和 setter 方法。

结论

在 Jackson 反序列化中，若调用了 `enableDefaultTyping()` 函数或使用 `@JsonTypeInfo` 注解指定反序列化得到的类的属性为 `JsonTypeInfo.Id.CLASS` 或 `JsonTypeInfo.Id.MINIMAL_CLASS`，则会调用该属性的类的构造函数和 setter 方法。

0x04 Jackson 反序列化漏洞

前提条件

满足下面三个条件之一即存在 Jackson 反序列化漏洞：

- 调用了 `ObjectMapper.enableDefaultTyping()` 函数；
- 对要进行反序列化的类的属性使用了值为 `JsonTypeInfo.Id.CLASS` 的 `@JsonTypeInfo` 注解；
- 对要进行反序列化的类的属性使用了值为 `JsonTypeInfo.Id.MINIMAL_CLASS` 的 `@JsonTypeInfo` 注解；

漏洞原理

由之前的结论知道，当使用的 JacksonPolymorphicDeserialization 机制配置有问题时，Jackson 反序列化就会调用属性所属类的构造函数和 setter 方法。

而如果该构造函数或 setter 方法存在危险操作，那么就存在 Jackson 反序列化漏洞。

漏洞场景及 Demo

这里大致以要进行反序列化的类的属性所属的类的类型分为两种：

属性不为Object类时

当要进行反序列化的类的属性所属类的构造函数或 setter 方法本身存在漏洞时，这种场景存在 Jackson 反序列化漏洞。当然这种场景开发几乎不会这么写。

我们看个例子，直接修改 MySex 类的 setSex ()方法，在其中添加命令执行操作（除非程序员自己想留后门、不然不会出现这种写法）：

JAVA

```
public class EvilSex implements Sex {
    int sex;
    public MySex() {
        System.out.println("MySex构造函数");
    }

    @Override
    public int getSex() {
        System.out.println("MySex.getSex");
        return sex;
    }

    @Override
    public void setSex(int sex) {
        System.out.println("MySex.setSex");
        this.sex = sex;
        try {
            Runtime.getRuntime().exec("calc");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Person3 类不变

编写反序列化类，构造 Payload

JAVA

```
public class DeserializationRun {
    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        mapper.enableDefaultTyping();

        String json = "{\"age\":6,\"name\":\"drunkbaby\",\"sex\":1,\"com.drunkbaby.JacksonVul.EvilSex\":{\"sex\":1}}";
        Person3 p2 = mapper.readValue(json, Person3.class);
        System.out.println(p2);
    }
}
```

属性为 Object 类时

当属性类型为 Object 时，因为 Object 类型是任意类型的父类，因此扩大了我们的攻击面，我们只需要寻找出在目标服务端环境中存在的且构造函数或 setter 方法存在漏洞代码的类即可进行攻击利用。

后面出现的 Jackson 反序列化的 CVE 漏洞、黑名单绕过等都是基于这个原理寻找各种符合条件的利用链。

我们编写一个存在漏洞的代码

Evil.java

JAVA

```
public class Evil {
    String cmd;

    public void setCmd(String cmd) {
        this.cmd = cmd;
        try {
            Runtime.getRuntime().exec(this.cmd);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Person4.java

JAVA

```
public class Person4 {
    public int age;
    public String name;
    // @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS)
    // 或 @JsonTypeInfo(use = JsonTypeInfo.Id.MINIMAL_CLASS)    public Object
    object;

    public Person4() {
        System.out.println("Person3 构造函数");
    }

    public void setAge(int age) {
        System.out.println("Person3 setter 函数");
    }

    @Override
    public String toString() {
        return String.format("Person.age=%d, Person.name=%s, %s", age, name,
            object == null ? "null" : object);
    }
}
```

接着编写反序列化代码

JAVA

```

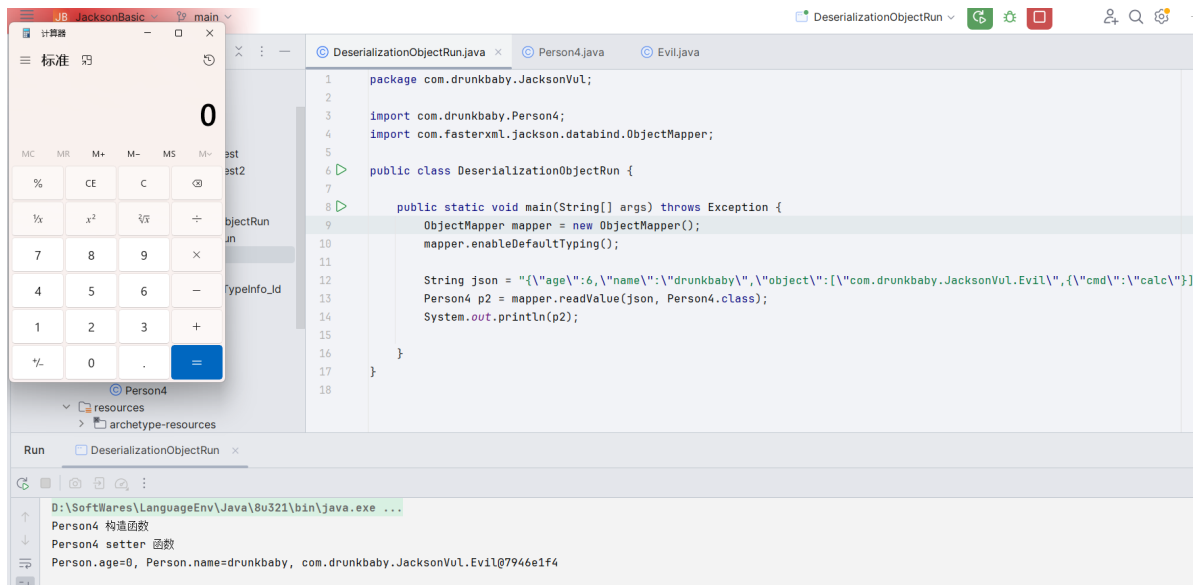
public class DeserializationObjectRun {

    public static void main(String[] args) throws Exception {
        ObjectMapper mapper = new ObjectMapper();
        mapper.enableDefaultTyping();

        String json = "{\"age\":6,\"name\":\"drunkbaby\", \"object\":[" +
        "\"com.drunkbaby.JacksonVul.Evil\", {\"cmd\":\"calc\"}]}";
        Person4 p2 = mapper.readValue(json, Person4.class);
        System.out.println(p2);

    }
}

```



0x05 小结

概念虽然多，但是自己跟一下代码，看起来还是非常快的。总而言之就是开启了特殊的反序列化解析方式时，会调用任意的构造函数与 setter 方法