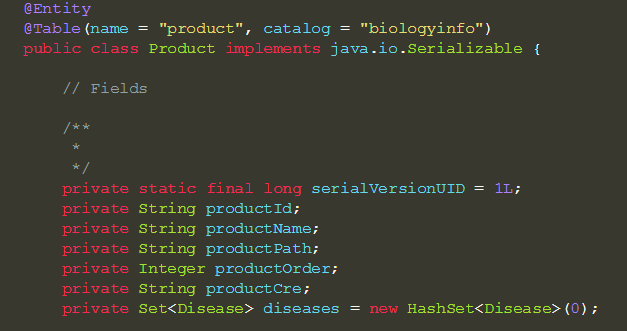
## 项目中出现的问题

### Json循环引用

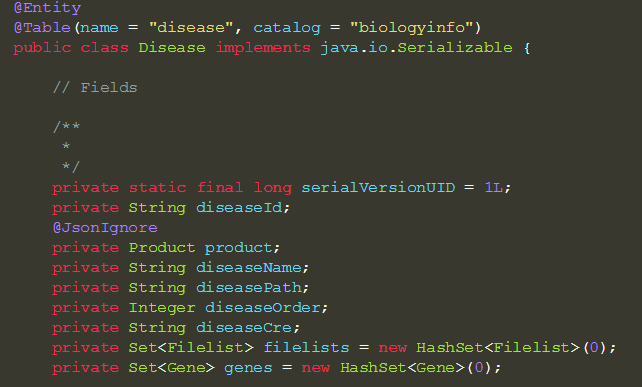
视频服务平台：

两个实体类，Video和Category，都持有对方的引用，在进行json数据转换时，会出现栈溢出异常。

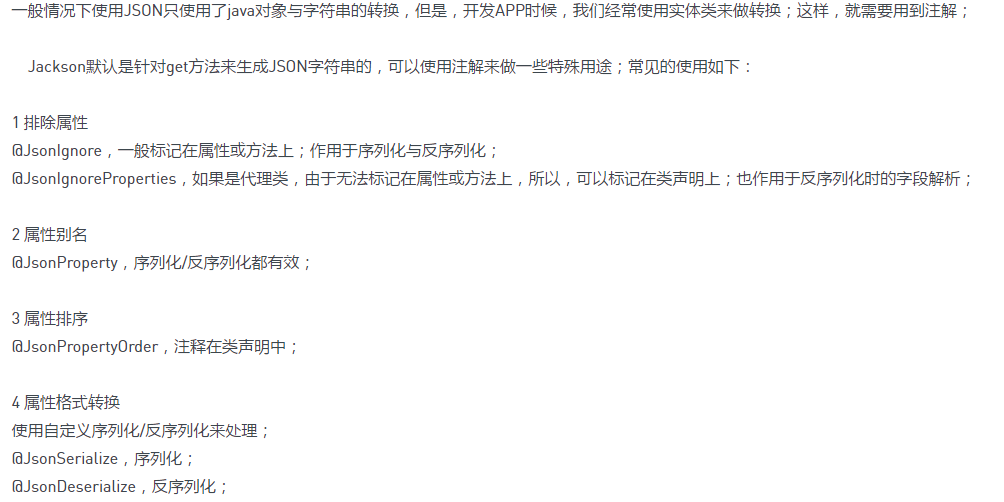
原因：**循环引用：实体A与实体B有关系，A中有B作为字段，B中有A作为一个字段。查询A对象后，将A对象转化为JSON格式数据时，会因为序列化过程中导致A中有B字段，B字段中又有A，这样就引起了循环引用的问题。**



下面是正确的实体类，添加了@JsonIgnore注解



**这样仅仅解决了可以查询Disease实体类的问题，如果要将实体类转化为JSON格式的数据传递给前台，就会报错！！！！**

### Servlet过滤器中使用Spring IOC功能

在过滤器中想使用spring ioc的功能，但是由于加载顺序的问题，是先加载filter再加载spring。所以需要配置监听器listener，让wen容器启动时就会去加载配置文件。

## Jackson详解

<http://blog.csdn.net/sdyy321/article/details/40298081>

**jackson的自动检测机制**

jackson允许使用任意的构造方法或工厂方法来构造实例

**使用@JsonAutoDetect（作用在类上）来开启/禁止自动检测**

fieldVisibility:字段的可见级别

ANY:任何级别的字段都可以自动识别

NONE:所有字段都不可以自动识别

NON\_PRIVATE:非private修饰的字段可以自动识别

PROTECTED\_AND\_PUBLIC:被protected和public修饰的字段可以被自动识别

PUBLIC\_ONLY:只有被public修饰的字段才可以被自动识别

DEFAULT:同PUBLIC\_ONLY

jackson默认的字段属性发现规则如下：

所有被public修饰的字段->所有被public修饰的getter->所有被public修饰的setter

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. **public** **static** **class** TestPOJO{
2. TestPOJO(){}
4. TestPOJO(String name){
5. **this**.name = name;
6. }
7. **private** String name;
9. @Override
10. **public** String toString() {
11. **return** "TestPOJO{" +
12. "name='" + name + '\'' +
13. '}';
14. }
15. }

这个类我们只有一个private的name属性，并且没有提供对应的get,set方法，如果按照默认的属性发现规则我们将无法序列化和反序列化name字段(如果没有get,set方法，只有被public修饰的属性才会被发现)，你可以通过修改@JsonAutoDetect的fieldVisibility来调整自动发现级别，为了使name被自动发现，我们需要将级别调整为ANY

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonAutoDetect(fieldVisibility = JsonAutoDetect.Visibility.ANY)

同理，除了fieldVisibility可以设置外，还可以设置getterVisibility、setterVisibility、isGetterVisibility、creatorVisibility级别，不再多讲

除了上面的方式，你还可以有一些其他方式可以配置methods,fields和creators(构造器和静态方法)的自动检测，例如：

你可以配置MapperFeature来启动/禁止一些特别类型(getters,setters,fields,creators)的自动检测

比如下面的MapperFeature配置：

SORT\_PROPERTIES\_ALPHABETICALLY：按字母顺序排序属性

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. ObjectMapper objectMapper = **new** ObjectMapper();
2. objectMapper.configure(MapperFeature.SORT\_PROPERTIES\_ALPHABETICALLY,**true**);

**配置SerializationFeature**

一些我们比较常用的SerializationFeature配置：

**SerializationFeature.WRAP\_ROOT\_VALUE：是否环绕根元素，默认false，如果为true，则默认以类名作为根元素，你也可以通过@JsonRootName来自定义根元素名称**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRAP\_ROOT\_VALUE,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonRootName("myPojo")
2. **public** **static** **class** TestPOJO{
3. **private** String name;
5. **public** String getName() {
6. **return** name;
7. }
9. **public** **void** setName(String name) {
10. **this**.name = name;
11. }
12. }

该类在序列化成json后类似如下:{"myPojo":{"name":"aaaa"}}

**SerializationFeature.INDENT\_OUTPUT：是否缩放排列输出，默认false，有些场合为了便于排版阅读则需要对输出做缩放排列**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.INDENT\_OUTPUT,**true**);

举例：

如果一个类中有a、b、c、d四个可检测到的属性，那么序列化后的json输出类似下面：

{  
  "a" : "aaa",  
  "b" : "bbb",  
  "c" : "ccc",  
  "d" : "ddd"  
}

**SerializationFeature.WRITE\_DATES\_AS\_TIMESTAMPS：序列化日期时以timestamps输出，默认true**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_DATES\_AS\_TIMESTAMPS,**true**);

比如如果一个类中有private Date date;这种日期属性，序列化后为：{"date" : 1413800730456}，若不为true，则为{"date" : "2014-10-20T10:26:06.604+0000"}

**SerializationFeature.WRITE\_ENUMS\_USING\_TO\_STRING：序列化枚举是以toString()来输出，默认false，即默认以name()来输出**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_ENUMS\_USING\_TO\_STRING,**true**);

**SerializationFeature.WRITE\_ENUMS\_USING\_INDEX：序列化枚举是以ordinal()来输出，默认false**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_ENUMS\_USING\_INDEX,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** enumTest() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. testPOJO.setMyEnum(TestEnum.ENUM01);
6. ObjectMapper objectMapper = **new** ObjectMapper();
7. objectMapper.configure(SerializationFeature.WRITE\_ENUMS\_USING\_TO\_STRING,**false**);
8. objectMapper.configure(SerializationFeature.WRITE\_ENUMS\_USING\_INDEX,**false**);
9. String jsonStr1 = objectMapper.writeValueAsString(testPOJO);
10. Assert.assertEquals("{\"myEnum\":\"ENUM01\",\"name\":\"myName\"}",jsonStr1);
12. ObjectMapper objectMapper2 = **new** ObjectMapper();
13. objectMapper2.configure(SerializationFeature.WRITE\_ENUMS\_USING\_TO\_STRING,**true**);
14. String jsonStr2 = objectMapper2.writeValueAsString(testPOJO);
15. Assert.assertEquals("{\"myEnum\":\"enum\_01\",\"name\":\"myName\"}",jsonStr2);
17. ObjectMapper objectMapper3 = **new** ObjectMapper();
18. objectMapper3.configure(SerializationFeature.WRITE\_ENUMS\_USING\_INDEX,**true**);
19. String jsonStr3 = objectMapper3.writeValueAsString(testPOJO);
20. Assert.assertEquals("{\"myEnum\":0,\"name\":\"myName\"}",jsonStr3);
21. }
22. **public** **static** **class** TestPOJO{
23. TestPOJO(){}
24. **private** TestEnum myEnum;
25. **private** String name;
27. //getters、setters省略
28. }
30. **public** **static** **enum** TestEnum{
31. ENUM01("enum\_01"),ENUM02("enum\_01"),ENUM03("enum\_01");
33. **private** String title;
35. TestEnum(String title) {
36. **this**.title = title;
37. }
39. @Override
40. **public** String toString() {
41. **return** title;
42. }
43. }

**SerializationFeature.WRITE\_SINGLE\_ELEM\_ARRAYS\_UNWRAPPED：序列化单元素数组时不以数组来输出，默认false**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_ENUMS\_USING\_TO\_STRING,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** singleElemArraysUnwrap() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. List<Integer> counts = **new** ArrayList<>();
6. counts.add(1);
7. testPOJO.setCounts(counts);
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. objectMapper.configure(SerializationFeature.WRITE\_SINGLE\_ELEM\_ARRAYS\_UNWRAPPED,**false**);
10. String jsonStr1 = objectMapper.writeValueAsString(testPOJO);
11. Assert.assertEquals("{\"name\":\"myName\",\"counts\":[1]}",jsonStr1);
13. ObjectMapper objectMapper2 = **new** ObjectMapper();
14. objectMapper2.configure(SerializationFeature.WRITE\_SINGLE\_ELEM\_ARRAYS\_UNWRAPPED,**true**);
15. String jsonStr2 = objectMapper2.writeValueAsString(testPOJO);
16. Assert.assertEquals("{\"name\":\"myName\",\"counts\":1}",jsonStr2);
17. }
19. **public** **static** **class** TestPOJO{
20. **private** String name;
21. **private** List<Integer> counts;
23. //getters、setters省略
24. }

**SerializationFeature.ORDER\_MAP\_ENTRIES\_BY\_KEYS：序列化Map时对key进行排序操作，默认false**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.ORDER\_MAP\_ENTRIES\_BY\_KEYS,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** orderMapBykey() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. Map<String,Integer> counts = **new** HashMap<>();
6. counts.put("a",1);
7. counts.put("d",4);
8. counts.put("c",3);
9. counts.put("b",2);
10. counts.put("e",5);
11. testPOJO.setCounts(counts);
12. ObjectMapper objectMapper = **new** ObjectMapper();
13. objectMapper.configure(SerializationFeature.ORDER\_MAP\_ENTRIES\_BY\_KEYS,**false**);
14. String jsonStr1 = objectMapper.writeValueAsString(testPOJO);
15. Assert.assertEquals("{\"name\":\"myName\",\"counts\":{\"d\":4,\"e\":5,\"b\":2,\"c\":3,\"a\":1}}",jsonStr1);
17. ObjectMapper objectMapper2 = **new** ObjectMapper();
18. objectMapper2.configure(SerializationFeature.ORDER\_MAP\_ENTRIES\_BY\_KEYS,**true**);
19. String jsonStr2 = objectMapper2.writeValueAsString(testPOJO);
20. Assert.assertEquals("{\"name\":\"myName\",\"counts\":{\"a\":1,\"b\":2,\"c\":3,\"d\":4,\"e\":5}}",jsonStr2);
21. }
23. **public** **static** **class** TestPOJO{
24. **private** String name;
25. **private** Map<String,Integer> counts;
27. //getters、setters省略
28. }

**SerializationFeature.WRITE\_CHAR\_ARRAYS\_AS\_JSON\_ARRAYS：序列化char[]时以json数组输出，默认false**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_CHAR\_ARRAYS\_AS\_JSON\_ARRAYS,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** charArraysAsJsonArrays() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. **char**[] counts = **new** **char**[]{'a','b','c','d'};
6. testPOJO.setCounts(counts);
7. ObjectMapper objectMapper = **new** ObjectMapper();
8. objectMapper.configure(SerializationFeature.WRITE\_CHAR\_ARRAYS\_AS\_JSON\_ARRAYS,**false**);
9. String jsonStr1 = objectMapper.writeValueAsString(testPOJO);
10. Assert.assertEquals("{\"name\":\"myName\",\"counts\":\"abcd\"}",jsonStr1);
12. ObjectMapper objectMapper2 = **new** ObjectMapper();
13. objectMapper2.configure(SerializationFeature.WRITE\_CHAR\_ARRAYS\_AS\_JSON\_ARRAYS,**true**);
14. String jsonStr2 = objectMapper2.writeValueAsString(testPOJO);
15. Assert.assertEquals("{\"name\":\"myName\",\"counts\":[\"a\",\"b\",\"c\",\"d\"]}",jsonStr2);
16. }
18. **public** **static** **class** TestPOJO{
19. **private** String name;
20. **private** **char**[] counts;
22. //getters、setters省略
23. }

**SerializationFeature.WRITE\_BIGDECIMAL\_AS\_PLAIN：序列化BigDecimal时之间输出原始数字还是科学计数，默认false，即是否以toPlainString()科学计数方式来输出**

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(SerializationFeature.WRITE\_CHAR\_ARRAYS\_AS\_JSON\_ARRAYS,**true**);

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** bigDecimalAsPlain() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. testPOJO.setCount(**new** BigDecimal("1e20"));
7. ObjectMapper objectMapper = **new** ObjectMapper();
8. objectMapper.configure(SerializationFeature.WRITE\_BIGDECIMAL\_AS\_PLAIN,**false**);
9. String jsonStr1 = objectMapper.writeValueAsString(testPOJO);
10. Assert.assertEquals("{\"name\":\"myName\",\"count\":1E+20}",jsonStr1);
12. ObjectMapper objectMapper2 = **new** ObjectMapper();
13. objectMapper2.configure(SerializationFeature.WRITE\_BIGDECIMAL\_AS\_PLAIN,**true**);
14. String jsonStr2 = objectMapper2.writeValueAsString(testPOJO);
15. Assert.assertEquals("{\"name\":\"myName\",\"count\":100000000000000000000}",jsonStr2);
16. }

配置DeserializationFeature

需要注意的是对于第二种通过配置SerializationConfig和DeserializationConfig方式只能启动/禁止自动检测，无法修改我们所需的可见级别

有时候对每个实例进行可见级别的注解可能会非常麻烦，这时候我们需要配置一个全局的可见级别，通过objectMapper.setVisibilityChecker()来实现，默认的VisibilityChecker实现类为VisibilityChecker.Std，这样可以满足实现复杂场景下的基础配置。

也有一些实用简单的可见级别配置，比如：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. ObjectMapper objectMapper = **new** ObjectMapper();
2. objectMapper.setVisibility(PropertyAccessor.FIELD, JsonAutoDetect.Visibility.ANY) // auto-detect all member fields
3. .setVisibility(PropertyAccessor.GETTER, JsonAutoDetect.Visibility.NONE) // but only public getters
4. .setVisibility(PropertyAccessor.IS\_GETTER, JsonAutoDetect.Visibility.NONE) // and none of "is-setters"
5. ;

你也可以通过下面方式来禁止所有的自动检测功能

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. ObjectMapper objectMapper = **new** ObjectMapper();
2. objectMapper.setVisibilityChecker(objectMapper.getVisibilityChecker().with(JsonAutoDetect.Visibility.NONE));

**jackson的常用注解**

**1、@JsonAutoDetect**

看上面自动检测，不再重复

**2、@JsonIgnore**

作用在字段或方法上，用来完全忽略被注解的字段和方法对应的属性，即便这个字段或方法可以被自动检测到或者还有其他的注解

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonIgnoreTest() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setId(111);
5. testPOJO.setName("myName");
6. testPOJO.setCount(22);
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. String jsonStr = objectMapper.writeValueAsString(testPOJO);
10. Assert.assertEquals("{\"id\":111}",jsonStr);
12. String jsonStr2 = "{\"id\":111,\"name\":\"myName\",\"count\":22}";
13. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2, TestPOJO.**class**);
14. Assert.assertEquals(111,testPOJO2.getId());
15. Assert.assertNull(testPOJO2.getName());
16. Assert.assertEquals(0,testPOJO2.getCount());
17. }
19. **public** **static** **class** TestPOJO{
20. **private** **int** id;
21. @JsonIgnore
22. **private** String name;
23. **private** **int** count;
25. **public** **int** getId() {
26. **return** id;
27. }
29. **public** **void** setId(**int** id) {
30. **this**.id = id;
31. }
33. **public** String getName() {
34. **return** name;
35. }
37. **public** **void** setName(String name) {
38. **this**.name = name;
39. }
41. **public** **int** getCount() {
42. **return** count;
43. }
44. @JsonIgnore
45. **public** **void** setCount(**int** count) {
46. **this**.count = count;
47. }
48. }

当@JsonIgnore不管注解在getters上还是setters上都会忽略对应的属性

**3、@JsonProperty**

作用在字段或方法上，用来对属性的序列化/反序列化，可以用来避免遗漏属性，同时提供对属性名称重命名，比如在很多场景下**Java**对象的属性是按照规范的驼峰书写，但是实际展示的却是类似C-style或C++/Microsolft style

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonPropertyTest() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.wahaha(111);
5. testPOJO.setFirstName("myName");
7. ObjectMapper objectMapper = **new** ObjectMapper();
8. String jsonStr = objectMapper.writeValueAsString(testPOJO);
9. Assert.assertEquals("{\"id\":111,\"first\_name\":\"myName\"}",jsonStr);
11. String jsonStr2 = "{\"id\":111,\"first\_name\":\"myName\"}";
12. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2, TestPOJO.**class**);
13. Assert.assertEquals(111, testPOJO2.wahaha());
14. Assert.assertEquals("myName", testPOJO2.getFirstName());
15. }
17. **public** **static** **class** TestPOJO{
18. @JsonProperty//注意这里必须得有该注解，因为没有提供对应的getId和setId函数，而是其他的getter和setter，防止遗漏该属性
19. **private** **int** id;
20. @JsonProperty("first\_name")
21. **private** String firstName;
23. **public** **int** wahaha() {
24. **return** id;
25. }
27. **public** **void** wahaha(**int** id) {
28. **this**.id = id;
29. }
31. **public** String getFirstName() {
32. **return** firstName;
33. }
35. **public** **void** setFirstName(String firstName) {
36. **this**.firstName = firstName;
37. }
38. }

**4、@JsonIgnoreProperties**

作用在类上，用来说明有些属性在序列化/反序列化时需要忽略掉，可以将它看做是@JsonIgnore的批量操作，但它的功能比@JsonIgnore要强，比如一个类是代理类，我们无法将将@JsonIgnore标记在属性或方法上，此时便可用@JsonIgnoreProperties标注在类声明上，它还有一个重要的功能是作用在反序列化时解析字段时过滤一些未知的属性，否则通常情况下解析到我们定义的类不认识的属性便会抛出异常。

可以注明是想要忽略的属性列表如@JsonIgnoreProperties({"name","age","title"})，

也可以注明过滤掉未知的属性如@JsonIgnoreProperties(ignoreUnknown=true)

举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test(expected = UnrecognizedPropertyException.**class**)
2. **public** **void** JsonIgnoreProperties() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setId(111);
5. testPOJO.setName("myName");
6. testPOJO.setAge(22);
7. ObjectMapper objectMapper = **new** ObjectMapper();
8. String jsonStr = objectMapper.writeValueAsString(testPOJO);
9. Assert.assertEquals("{\"id\":111}",jsonStr);//name和age被忽略掉了
11. String jsonStr2 = "{\"id\":111,\"name\":\"myName\",\"age\":22,\"title\":\"myTitle\"}";
12. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2, TestPOJO.**class**);
13. Assert.assertEquals(111, testPOJO2.getId());
14. Assert.assertNull(testPOJO2.getName());
15. Assert.assertEquals(0,testPOJO2.getAge());
16. String jsonStr3 = "{\"id\":111,\"name\":\"myName\",\"count\":33}";//这里有个未知的count属性，反序列化会报错
17. objectMapper.readValue(jsonStr3, TestPOJO.**class**);
18. }
20. @JsonIgnoreProperties({"name","age","title"})
21. **public** **static** **class** TestPOJO{
22. **private** **int** id;
23. **private** String name;
24. **private** **int** age;
26. //getters、setters省略
27. }

如果将上面的

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonIgnoreProperties({"name","age","title"})

更换为

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonIgnoreProperties(ignoreUnknown=**true**)

那么测试用例中在反序列化未知的count属性时便不会抛出异常了

**5、@JsonUnwrapped**

作用在属性字段或方法上，用来将子JSON对象的属性添加到封闭的JSON对象，说起来比较难懂，看个例子就很清楚了，不多解释

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonUnwrapped() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setId(111);
5. TestName testName = **new** TestName();
6. testName.setFirstName("张");
7. testName.setSecondName("三");
8. testPOJO.setName(testName);
9. ObjectMapper objectMapper = **new** ObjectMapper();
10. String jsonStr = objectMapper.writeValueAsString(testPOJO);
11. //如果没有@JsonUnwrapped，序列化后将为{"id":111,"name":{"firstName":"张","secondName":"三"}}
12. //因为在name属性上加了@JsonUnwrapped，所以name的子属性firstName和secondName将不会包含在name中。
13. Assert.assertEquals("{\"id\":111,\"firstName\":\"张\",\"secondName\":\"三\"}",jsonStr);
14. String jsonStr2 = "{\"id\":111,\"firstName\":\"张\",\"secondName\":\"三\"}";
15. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2,TestPOJO.**class**);
16. Assert.assertEquals(111,testPOJO2.getId());
17. Assert.assertEquals("张",testPOJO2.getName().getFirstName());
18. Assert.assertEquals("三",testPOJO2.getName().getSecondName());
19. }
21. **public** **static** **class** TestPOJO{
22. **private** **int** id;
23. @JsonUnwrapped
24. **private** TestName name;
26. //getters、setters省略
27. }                                                                                                                             **public** **static** **class** TestName{
28. **private** String firstName;
29. **private** String secondName;
31. //getters、setters省略
32. }

在2.0+版本中@JsonUnwrapped添加了prefix和suffix属性，用来对字段添加前后缀，这在有关属性分组上比较有用，在上面的测试用例中，如果我们将TestPOJO的name属性上的@JsonUnwrapped添加前后缀配置，即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonUnwrapped(prefix = "name\_",suffix = "\_test")

那么TestPOJO序列化后将为{"id":111,"name\_firstName\_test":"张","name\_secondName\_test":"三"}，反序列化时也要加上前后缀才会被解析为POJO

**6、@JsonIdentityInfo**

2.0+版本新注解，作用于类或属性上，被用来在序列化/反序列化时为该对象或字段添加一个对象识别码，通常是用来解决循环嵌套的问题，比如**数据库**中的多对多关系，通过配置属性generator来确定识别码生成的方式，有简单的，配置属性property来确定识别码的名称，识别码名称没有限制。  
对象识别码可以是虚拟的，即存在在JSON中，但不是POJO的一部分，这种情况下我们可以如此使用注解

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.**class**,property = "@id")

对象识别码也可以是真实存在的，即以对象的属性为识别码，通常这种情况下我们一般以id属性为识别码，可以这么使用注解

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.**class**,property = "id")

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonIdentityInfo() **throws** Exception {
3. Parent parent = **new** Parent();
4. parent.setName("jack");
5. Child child = **new** Child();
6. child.setName("mike");
7. Child[] children = **new** Child[]{child};
8. parent.setChildren(children);
9. child.setParent(parent);
10. ObjectMapper objectMapper = **new** ObjectMapper();
11. String jsonStr = objectMapper.writeValueAsString(parent);
12. Assert.assertEquals("{\"@id\":1,\"name\":\"jack\",\"children\":[{\"name\":\"mike\",\"parent\":1}]}",jsonStr);
13. }
15. @JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.**class**,property = "@id")
16. **public** **static** **class** Parent{
17. **private** String name;
18. **private** Child[] children;
20. //getters、setters省略
21. }
23. **public** **static** **class** Child{
24. **private** String name;
25. **private** Parent parent;
27. //getters、setters省略
28. }

这里需要提醒一下的是在1.6版本中提供了@JsonManagedReference和@JsonBackReference来解决循环嵌套问题，因为属于过时注解这里就不解释了，有兴趣的可以自己看

**7、@JsonNaming**

jackson 2.1+版本的注解，作用于类或方法，注意这个注解是在jackson-databind包中而不是在jackson-annotations包里，它可以让你定制属性命名策略，作用和前面提到的@JsonProperty的重命名属性名称相同。比如  
你有一个JSON串{"in\_reply\_to\_user\_id":"abc123"}，需要反序列化为POJO，POJO一般情况下则需要如此写

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. **public** **static** **class** TestPOJO{
3. **private** String in\_reply\_to\_user\_id;
5. **public** String getIn\_reply\_to\_user\_id() {
6. **return** in\_reply\_to\_user\_id;
7. }
9. **public** **void** setIn\_reply\_to\_user\_id(String in\_reply\_to\_user\_id) {
10. **this**.in\_reply\_to\_user\_id = in\_reply\_to\_user\_id;
11. }
12. }

但这显然不符合JAVA的编码规范，你可以用@JsonProperty，比如：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. **public** **static** **class** TestPOJO{
3. @JsonProperty("in\_reply\_to\_user\_id")
4. **private** String inReplyToUserId;
6. **public** String getInReplyToUserId() {
7. **return** inReplyToUserId;
8. }
10. **public** **void** setInReplyToUserId(String inReplyToUserId) {
11. **this**.inReplyToUserId = inReplyToUserId;
12. }
13. }

这样就符合规范了，可是如果POJO里有很多属性，给每个属性都要加上@JsonProperty是多么繁重的工作，这里就需要用到@JsonNaming了，它不仅能制定统一的命名规则，还能任意按自己想要的方式定制

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonNaming() **throws** Exception{
3. String jsonStr = "{\"in\_reply\_to\_user\_id\":\"abc123\"}";
4. ObjectMapper objectMapper = **new** ObjectMapper();
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("abc123",testPOJO.getInReplyToUserId());
8. TestPOJO testPOJO2 = **new** TestPOJO();
9. testPOJO2.setInReplyToUserId("abc123");
10. String jsonStr2 = objectMapper.writeValueAsString(testPOJO2);
11. Assert.assertEquals(jsonStr,jsonStr2);
12. }
14. @JsonNaming(PropertyNamingStrategy.LowerCaseWithUnderscoresStrategy.**class**)
15. **public** **static** **class** TestPOJO{
17. **private** String inReplyToUserId;
19. **public** String getInReplyToUserId() {
20. **return** inReplyToUserId;
21. }
23. **public** **void** setInReplyToUserId(String inReplyToUserId) {
24. **this**.inReplyToUserId = inReplyToUserId;
25. }
26. }

@JsonNaming使用了jackson已经实现的PropertyNamingStrategy.LowerCaseWithUnderscoresStrategy，它可以将大写转换为小写并添加下划线。你可以自定义，必须继承类PropertyNamingStrategy，建议继承PropertyNamingStrategyBase，我们自己实现一个类似LowerCaseWithUnderscoresStrategy的策略，只是将下划线改为破折号

举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonNaming() **throws** Exception{
3. String jsonStr = "{\"in-reply-to-user-id\":\"abc123\"}";
4. ObjectMapper objectMapper = **new** ObjectMapper();
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("abc123", testPOJO.getInReplyToUserId());
8. TestPOJO testPOJO2 = **new** TestPOJO();
9. testPOJO2.setInReplyToUserId("abc123");
10. String jsonStr2 = objectMapper.writeValueAsString(testPOJO2);
11. Assert.assertEquals(jsonStr, jsonStr2);
12. }
14. @JsonNaming(MyPropertyNamingStrategy.**class**)
15. **public** **static** **class** TestPOJO{
17. **private** String inReplyToUserId;
19. **public** String getInReplyToUserId() {
20. **return** inReplyToUserId;
21. }
23. **public** **void** setInReplyToUserId(String inReplyToUserId) {
24. **this**.inReplyToUserId = inReplyToUserId;
25. }
26. }
28. **public** **static** **class** MyPropertyNamingStrategy **extends** PropertyNamingStrategy.PropertyNamingStrategyBase {
29. @Override
30. **public** String translate(String input) {
31. **if** (input == **null**) **return** input; // garbage in, garbage out
32. **int** length = input.length();
33. StringBuilder result = **new** StringBuilder(length \* 2);
34. **int** resultLength = 0;
35. **boolean** wasPrevTranslated = **false**;
36. **for** (**int** i = 0; i < length; i++)
37. {
38. **char** c = input.charAt(i);
39. **if** (i > 0 || c != '-') // skip first starting underscore
40. {
41. **if** (Character.isUpperCase(c))
42. {
43. **if** (!wasPrevTranslated && resultLength > 0 && result.charAt(resultLength - 1) != '-')
44. {
45. result.append('-');
46. resultLength++;
47. }
48. c = Character.toLowerCase(c);
49. wasPrevTranslated = **true**;
50. }
51. **else**
52. {
53. wasPrevTranslated = **false**;
54. }
55. result.append(c);
56. resultLength++;
57. }
58. }
59. **return** resultLength > 0 ? result.toString() : input;
60. }
61. }

如果你想让自己定制的策略对所有解析都实现，除了对每个具体的实体类对应的位置加上@JsonNaming外你还可以如下做全局配置

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. ObjectMapper objectMapper = **new** ObjectMapper();
2. objectMapper.setPropertyNamingStrategy(**new** MyPropertyNamingStrategy());

**多态类型处理**

jackson允许配置多态类型处理，当进行反序列话时，JSON数据匹配的对象可能有多个子类型，为了正确的读取对象的类型，我们需要添加一些类型信息。可以通过下面几个注解来实现：

**@JsonTypeInfo**

作用于类/接口，被用来开启多态类型处理，对基类/接口和子类/实现类都有效

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.NAME,include = JsonTypeInfo.As.PROPERTY,property = "name")

这个注解有一些属性，

use:定义使用哪一种类型识别码，它有下面几个可选值：

1、JsonTypeInfo.Id.CLASS：使用完全限定类名做识别

2、JsonTypeInfo.Id.MINIMAL\_CLASS：若基类和子类在同一包类，使用类名(忽略包名)作为识别码

3、JsonTypeInfo.Id.NAME：一个合乎逻辑的指定名称

4、JsonTypeInfo.Id.CUSTOM：自定义识别码，由@JsonTypeIdResolver对应，稍后解释

5、JsonTypeInfo.Id.NONE：不使用识别码

include(可选):指定识别码是如何被包含进去的，它有下面几个可选值：

1、JsonTypeInfo.As.PROPERTY：作为数据的兄弟属性

2、JsonTypeInfo.As.EXISTING\_PROPERTY：作为POJO中已经存在的属性

3、JsonTypeInfo.As.EXTERNAL\_PROPERTY：作为扩展属性

4、JsonTypeInfo.As.WRAPPER\_OBJECT：作为一个包装的对象

5、JsonTypeInfo.As.WRAPPER\_ARRAY：作为一个包装的数组

property(可选):制定识别码的属性名称

此属性只有当use为JsonTypeInfo.Id.CLASS（若不指定property则默认为@class）、JsonTypeInfo.Id.MINIMAL\_CLASS(若不指定property则默认为@c)、JsonTypeInfo.Id.NAME(若不指定property默认为@type)，include为JsonTypeInfo.As.PROPERTY、JsonTypeInfo.As.EXISTING\_PROPERTY、JsonTypeInfo.As.EXTERNAL\_PROPERTY时才有效

defaultImpl(可选)：如果类型识别码不存在或者无效，可以使用该属性来制定反序列化时使用的默认类型

visible(可选，默认为false)：是否可见

属性定义了类型标识符的值是否会通过JSON流成为反序列化器的一部分，默认为fale,也就是说,jackson会从JSON内容中处理和删除类型标识符再传递给JsonDeserializer。

**@JsonSubTypes**

作用于类/接口，用来列出给定类的子类，只有当子类类型无法被检测到时才会使用它

一般是配合@JsonTypeInfo在基类上使用，比如：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.NAME,include = JsonTypeInfo.As.PROPERTY,property = "typeName")
2. @JsonSubTypes({@JsonSubTypes.Type(value=Sub1.**class**,name = "sub1"),@JsonSubTypes.Type(value=Sub2.**class**,name = "sub2")})

@JsonSubTypes的值是一个@JsonSubTypes.Type[]数组，里面枚举了多态类型(value对应类)和类型的标识符值(name对应@JsonTypeInfo中的property标识名称的值，此为可选值，若不制定需由@JsonTypeName在子类上制定)

**@JsonTypeName**

作用于子类，用来为多态子类指定类型标识符的值

比如：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeName(value = "sub1")

value属性作用同上面@JsonSubTypes里的name作用

**@JsonTypeResolver和@JsonTypeIdResoler**

作用于类，可以自定义多态的类型标识符，这个平时很少用到，主要是现有的一般就已经满足绝大多数的需求了，如果你需要比较特别的类型标识符，建议使用这2个注解，自己定制基于TypeResolverBuilder和TypeIdResolver的类即可

我们看几个jackson处理多态的例子

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonTypeInfo() **throws** Exception{
3. Sub1 sub1 = **new** Sub1();
4. sub1.setId(1);
5. sub1.setName("sub1Name");
6. Sub2 sub2 = **new** Sub2();
7. sub2.setId(2);
8. sub2.setAge(33);
9. ObjectMapper objectMapper = **new** ObjectMapper();
10. TestPOJO testPOJO = **new** TestPOJO();
11. testPOJO.setMyIns(**new** MyIn[]{sub1, sub2});
12. String jsonStr = objectMapper.writeValueAsString(testPOJO);
13. Assert.assertEquals("{\"myIns\":[{\"id\":1,\"name\":\"sub1Name\"},{\"id\":2,\"age\":33}]}", jsonStr);
14. System.out.println(jsonStr);
15. }
17. **public** **static** **abstract** **class** MyIn{
18. **private** **int** id;
20. //getters、setters省略
21. }
23. **public** **static** **class** Sub1 **extends** MyIn{
24. **private** String name;
26. //getters、setters省略
27. }
29. **public** **static** **class** Sub2 **extends** MyIn{
30. **private** **int** age;
32. //getters、setters省略
33. }

这是序列化时最简单的一种多态处理方式，因为没有使用任何多态处理注解，即默认使用的识别码类型为JsonTypeInfo.Id.NONE，而jackson没有自动搜索功能，所以只能序列化而不能反序列化，上面序列化测试的结果为{"myIns":[{"id":1,"name":"sub1Name"},{"id":2,"age":33}]}，我们可以看到JSON串中是没有对应的多态类型识别码的。

下面我们在基类MyIn上加上多态处理相关注解，首先我们在基类MyIn上添加@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS)即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS)
2. **public** **static** **abstract** **class** MyIn{
3. **private** **int** id;
5. //getters、setters省略
6. }

执行上面的序列化测试代码结果将会是

{"myIns":[{"@class":"cn.yangyong.fodder.util.JacksonUtilsTest$Sub1","id":1,"name":"sub1Name"},{"@class":"cn.yangyong.fodder.util.JacksonUtilsTest$Sub2","id":2,"age":33}]}

我们可以看到多了相应的多态类型识别码，识别码名称为默认的@class（因为没有指定名称），识别码的值为JsonTypeInfo.Id.CLASS即子类完全限定名

我们再添加上property属性@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS,property = "typeName")即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS,property = "typeName")
2. **public** **static** **abstract** **class** MyIn{
3. **private** **int** id;
5. //getters、setters省略
6. }

再次执行上面的序列化测试代码结果将会是

{"myIns":[{"typeName":"cn.yangyong.fodder.util.JacksonUtilsTest$Sub1","id":1,"name":"sub1Name"},{"typeName":"cn.yangyong.fodder.util.JacksonUtilsTest$Sub2","id":2,"age":33}]}

这次多态类型识别码的名称已经变成了我们指定的typeName而不是默认的@class了

上面的例子都是默认选择的include为JsonTypeInfo.As.PROPERTY，下面我们更改include方式，看看有什么变化，将include设置为JsonTypeInfo.As.WRAPPER\_OBJECT即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS,include = JsonTypeInfo.As.WRAPPER\_OBJECT,property = "typeName")
2. **public** **static** **abstract** **class** MyIn{
3. **private** **int** id;
5. //getters、setters省略
6. }

再次执行序列化测试，结果为

{"myIns":[{"cn.yangyong.fodder.util.JacksonUtilsTest$Sub1":{"id":1,"name":"sub1Name"}},{"cn.yangyong.fodder.util.JacksonUtilsTest$Sub2":{"id":2,"age":33}}]}

我们看到类型识别码不再成为兄弟属性包含进去了而是为父属性将其他属性包含进去，此时我们指定的property=“typeName”已经无用了

再次修改use属性指定为JsonTypeInfo.Id.MINIMAL\_CLASS，即@JsonTypeInfo(use = JsonTypeInfo.Id.MINIMAL\_CLASS,include = JsonTypeInfo.As.PROPERTY,property = "typeName")

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.MINIMAL\_CLASS,include = JsonTypeInfo.As.PROPERTY,property = "typeName")
2. **public** **static** **abstract** **class** MyIn{
3. **private** **int** id;
5. //getters、setters省略
6. }

测试序列化结果为

{"myIns":[{"typeName":".JacksonUtilsTest$Sub1","id":1,"name":"sub1Name"},{"typeName":".JacksonUtilsTest$Sub2","id":2,"age":33}]}

发现已经没有同包的package名称，识别码的值更加简短了

**测试**反序列化

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonTypeInfo() **throws** Exception{
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr2 = "{\"myIns\":[{\"typeName\":\".JacksonUtilsTest$Sub1\",\"id\":1,\"name\":\"sub1Name\"},{\"typeName\":\".JacksonUtilsTest$Sub2\",\"id\":2,\"age\":33}]}";
5. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2,TestPOJO.**class**);
6. MyIn[] myIns = testPOJO2.getMyIns();
7. **for** (MyIn myIn : myIns) {
8. System.out.println(myIn.getClass().getSimpleName());
9. }
10. }

结果将会显示为Sub1和Sub2说明是可以实现多态的反序列化的

可能我们在反序列化时觉得如此传递识别码很不友好，最好可以自定义识别码的值，可以选择use = JsonTypeInfo.Id.NAME和@JsonSubTypes配合即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @JsonTypeInfo(use = JsonTypeInfo.Id.NAME,include = JsonTypeInfo.As.PROPERTY,property = "typeName")
2. @JsonSubTypes({@JsonSubTypes.Type(value=Sub1.**class**,name="sub1"),@JsonSubTypes.Type(value=Sub2.**class**,name="sub2")})
3. **public** **static** **abstract** **class** MyIn{
4. **private** **int** id;
5. //getters、setters省略
6. }

执行序列化结果为

{"myIns":[{"typeName":"sub1","id":1,"name":"sub1Name"},{"typeName":"sub2","id":2,"age":33}]}

使用这个结果反序列化也可以得到我们想要的结果，或者在子类上添加@JsonTypeName(value = "sub1")和@JsonTypeName(value = "sub2")以便取代@JsonSubTypes里的name

如果想不使用@JsonSubTypes来实现反序列化，我们可以在ObjectMapper上注册子类实现，即

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. ObjectMapper objectMapper = **new** ObjectMapper();
2. objectMapper.registerSubtypes(**new** NamedType(Sub1.**class**,"sub1"));
3. objectMapper.registerSubtypes(**new** NamedType(Sub2.**class**,"sub2"));

更多多态处理的例子还请大家自己研究

**用于序列化和反序列化的注解类**

**1、@JsonSerialize和@JsonDeserialize**

作用于方法和字段上，通过 using(JsonSerializer)和using(JsonDeserializer)来指定序列化和反序列化的实现，通常我们在需要自定义序列化和反序列化时会用到，比如下面的例子中的日期转换

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonSerializeAndDeSerialize() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. testPOJO.setBirthday(**new** Date());
6. ObjectMapper objectMapper = **new** ObjectMapper();
7. String jsonStr = objectMapper.writeValueAsString(testPOJO);
8. System.out.println(jsonStr);
10. String jsonStr2 = "{\"name\":\"myName\",\"birthday\":\"2014-11-11 19:01:58\"}";
11. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2,TestPOJO.**class**);
12. System.out.println(testPOJO2.toString());
13. }
15. **public** **static** **class** TestPOJO{
16. **private** String name;
17. @JsonSerialize(using = MyDateSerializer.**class**)
18. @JsonDeserialize(using = MyDateDeserializer.**class**)
19. **private** Date birthday;
21. //getters、setters省略
23. @Override
24. **public** String toString() {
25. **return** "TestPOJO{" +
26. "name='" + name + '\'' +
27. ", birthday=" + birthday +
28. '}';
29. }
30. }
32. **private** **static** **class** MyDateSerializer **extends** JsonSerializer<Date>{
33. @Override
34. **public** **void** serialize(Date value, JsonGenerator jgen, SerializerProvider provider) **throws** IOException, JsonProcessingException {
35. DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
36. String dateStr = dateFormat.format(value);
37. jgen.writeString(dateStr);
38. }
39. }
41. **private** **static** **class** MyDateDeserializer **extends** JsonDeserializer<Date>{
42. @Override
43. **public** Date deserialize(JsonParser jp, DeserializationContext ctxt) **throws** IOException, JsonProcessingException {
44. String value = jp.getValueAsString();
45. DateFormat dateFormat = **new** SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
46. **try** {
47. **return** dateFormat.parse(value);
48. } **catch** (ParseException e) {
49. e.printStackTrace();
50. }
51. **return** **null**;
52. }
53. }

上面的例子中自定义了日期的序列化和反序列化方式，可以将Date和指定日期格式字符串之间相互转换。  
  
也可以通过使用as(JsonSerializer)和as(JsonDeserializer)来实现多态类型转换，上面我们有提到多态类型处理时可以使用@JsonTypeInfo实现，还有一种比较简便的方式就是使用@JsonSerialize和@JsonDeserialize指定as的子类类型，注意这里必须指定为子类类型才可以实现替换运行时的类型

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonSerializeAndDeSerialize() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. Sub1 sub1 = **new** Sub1();
6. sub1.setId(1);
7. sub1.setName("sub1Name");
8. Sub2 sub2 = **new** Sub2();
9. sub2.setId(2);
10. sub2.setAge(22);
11. testPOJO.setSub1(sub1);
12. testPOJO.setSub2(sub2);
13. ObjectMapper objectMapper = **new** ObjectMapper();
14. String jsonStr = objectMapper.writeValueAsString(testPOJO);
15. System.out.println(jsonStr);
17. String jsonStr2 = "{\"name\":\"myName\",\"sub1\":{\"id\":1,\"name\":\"sub1Name\"},\"sub2\":{\"id\":2,\"age\":22}}";
18. TestPOJO testPOJO2 = objectMapper.readValue(jsonStr2,TestPOJO.**class**);
19. System.out.println(testPOJO2.toString());
20. }
22. **public** **static** **class** TestPOJO{
23. **private** String name;
24. @JsonSerialize(as = Sub1.**class**)
25. @JsonDeserialize(as = Sub1.**class**)
26. **private** MyIn sub1;
27. @JsonSerialize(as = Sub2.**class**)
28. @JsonDeserialize(as = Sub2.**class**)
29. **private** MyIn sub2;
31. //getters、setters省略
33. @Override
34. **public** String toString() {
35. **return** "TestPOJO{" +
36. "name='" + name + '\'' +
37. ", sub1=" + sub1 +
38. ", sub2=" + sub2 +
39. '}';
40. }
41. }
43. **public** **static** **class** MyIn{
44. **private** **int** id;
46. //getters、setters省略
47. }
49. **public** **static** **class** Sub1 **extends** MyIn{
50. **private** String name;
52. //getters、setters省略
54. @Override
55. **public** String toString() {
56. **return** "Sub1{" +
57. "id=" + getId()  +
58. "name='" + name + '\'' +
59. '}';
60. }
61. }
62. **public** **static** **class** Sub2 **extends** MyIn{
63. **private** **int** age;
64. //getters、setters省略
66. @Override
67. **public** String toString() {
68. **return** "Sub1{" +
69. "id=" + getId() +
70. "age='" + age +
71. '}';
72. }
73. }

上面例子中通过as来指定了需要替换实际运行时类型的子类，实际上上面例子中序列化时是可以不使用@JsonSerialize(as = Sub1.class)的，因为jackson可以自动将POJO转换为对应的JSON，而反序列化时由于无法自动检索匹配类型必须要指定@JsonDeserialize(as = Sub1.class)方可实现

最后@JsonSerialize可以配置include属性来指定序列化时被注解的属性被包含的方式，默认总是被包含进来，但是可以过滤掉空的属性或有默认值的属性，举个简单的过滤空属性的例子如下

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonSerializeAndDeSerialize() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("");
5. ObjectMapper objectMapper = **new** ObjectMapper();
6. String jsonStr = objectMapper.writeValueAsString(testPOJO);
7. Assert.assertEquals("{}",jsonStr);
8. }
10. **public** **static** **class** TestPOJO{
11. @JsonSerialize(include = JsonSerialize.Inclusion.NON\_EMPTY)
12. **private** String name;
14. //getters、setters省略
15. }

**2、@JsonPropertyOrder**

作用在类上，被用来指明当序列化时需要对属性做排序，它有2个属性

一个是alphabetic：布尔类型，表示是否采用字母拼音顺序排序，默认是为false，即不排序

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonPropertyOrder() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setA("1");
5. testPOJO.setB("2");
6. testPOJO.setC("3");
7. testPOJO.setD("4");
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. String jsonStr = objectMapper.writeValueAsString(testPOJO);
10. Assert.assertEquals("{\"a\":\"1\",\"c\":\"3\",\"d\":\"4\",\"b\":\"2\"}",jsonStr);
11. }
13. **public** **static** **class** TestPOJO{
14. **private** String a;
15. **private** String c;
16. **private** String d;
17. **private** String b;
19. //getters、setters省略
20. }

我们先看一个默认的排序方式，序列化单元测试结果依次为{"a":"1","c":"3","d":"4","b":"2"}，即是没有经过排序操作的，在TestPOJO上加上@jsonPropertyOrder(alphabetic = true)再执行测试结果将会为{"a":"1","b":"2","c":"3","d":"4"}  
还有一个属性是value：数组类型，表示将优先其他属性排序的属性名称

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonPropertyOrder() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setA("1");
5. testPOJO.setB("2");
6. testPOJO.setC("3");
7. testPOJO.setD("4");
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. String jsonStr = objectMapper.writeValueAsString(testPOJO);
10. System.out.println(jsonStr);
11. Assert.assertEquals("{\"c\":\"3\",\"b\":\"2\",\"a\":\"1\",\"d\":\"4\"}",jsonStr);
12. }
14. @JsonPropertyOrder(alphabetic = **true**,value = {"c","b"})
15. **public** **static** **class** TestPOJO{
16. **private** String a;
17. **private** String c;
18. **private** String d;
19. **private** String b;
21. //getters、setters省略
22. }

上面例子可以看到value指定了c和b属性优先排序，所以序列化后为{"c":"3","b":"2","a":"1","d":"4"}

还记得本文上面最开始配置MapperFeature时也有属性排序么，对，就是

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. objectMapper.configure(MapperFeature.SORT\_PROPERTIES\_ALPHABETICALLY,**true**);

只不过@JsonPropertyOrder颗粒度要更细一点，可以决定哪些属性优先排序

**3、@JsonView**

视图模板，作用于方法和属性上，用来指定哪些属性可以被包含在JSON视图中，在前面我们知道已经有@JsonIgnore和@JsonIgnoreProperties可以排除过滤掉不需要序列化的属性，可是如果一个POJO中有上百个属性，比如订单类、商品详情类这种属性超多，而我们可能只需要概要简单信息即序列化时只想输出其中几个或10几个属性，此时使用@JsonIgnore和@JsonIgnoreProperties就显得非常繁琐，而使用@JsonView便会非常方便，只许在你想要输出的属性(或对应的getter)上添加@JsonView即可，举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonView() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setA("1");
5. testPOJO.setB("2");
6. testPOJO.setC("3");
7. testPOJO.setD("4");
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. objectMapper.configure(MapperFeature.DEFAULT\_VIEW\_INCLUSION, **false**);
10. String jsonStr = objectMapper.writerWithView(FilterView.OutputA.**class**).writeValueAsString(testPOJO);
11. Assert.assertEquals("{\"a\":\"1\",\"c\":\"3\"}",jsonStr);
12. String jsonStr2 = objectMapper.writerWithView(FilterView.OutputB.**class**).writeValueAsString(testPOJO);
13. Assert.assertEquals("{\"d\":\"4\",\"b\":\"2\"}",jsonStr2);
14. }
16. **public** **static** **class** TestPOJO{
17. @JsonView(FilterView.OutputA.**class**)
18. **private** String a;
19. @JsonView(FilterView.OutputA.**class**)
20. **private** String c;
21. @JsonView(FilterView.OutputB.**class**)
22. **private** String d;
23. @JsonView(FilterView.OutputB.**class**)
24. **private** String b;
25. //getters、setters忽略
26. }

29. **private** **static** **class** FilterView {
30. **static** **class** OutputA {}
31. **static** **class** OutputB {}
32. }

上面的测试用例中，我们在序列化之前先设置了objectMapper.configure(MapperFeature.DEFAULT\_VIEW\_INCLUSION, false)，看javadoc说这是一个双向开关，开启将输出没有JsonView注解的属性，false关闭将输出有JsonView注解的属性，可惜我在测试中开启开关后有JsonView注解的属性任然输出了，大家可以研究下。序列化时使用了objectMapper.writerWithView(FilterView.OutputA.class).writeValueAsString(testPOJO)，即使用哪个视图来输出。在上面的例子中又2种视图，我们在序列化的时候可以选择想要的视图来输出，这在一些地方比较好用，比如安卓、苹果、桌面等不同的客户端可能会输出不同的属性。在1.6版本中这个@JsonView注解同时也会强制性自动发现，也就是说不管属性的可见性以及是否设置了自动发现这些属性都将会自动被发现，在上例中TestPOJO中的getters、setters可以不需要也能输出我们想要的结果。

**4、@JsonFilter**

Json属性过滤器，作用于类，作用同上面的@JsonView，都是过滤掉不想要的属性，输出自己想要的属性。和@FilterView不同的是@JsonFilter可以动态的过滤属性，比如我不想输出以system开头的所有属性等待，应该说@JsonFilter更高级一点，举个简单的例子

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonFilter() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setA("1");
5. testPOJO.setB("2");
6. testPOJO.setC("3");
7. testPOJO.setD("4");
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. FilterProvider filters = **new** SimpleFilterProvider().addFilter("myFilter",SimpleBeanPropertyFilter.filterOutAllExcept("a"));
10. objectMapper.setFilters(filters);
11. String jsonStr = objectMapper.writeValueAsString(testPOJO);
12. Assert.assertEquals("{\"a\":\"1\"}",jsonStr);
13. }
15. @JsonFilter("myFilter")
16. **public** **static** **class** TestPOJO{
17. **private** String a;
18. **private** String c;
19. **private** String d;
20. **private** String b;
22. //getters、setters省略
23. }

上面例子中在我们想要序列化的POJO上加上了@JsonFilter，表示该类将使用名为myFilter的过滤器。在测试中定义了一个名为myFilter的SimpleFilterProvider，这个过滤器将会过滤掉所有除a属性以外的属性。这只是最简单的输出指定元素的例子，你可以自己实现FilterProvider来满足你的过滤需求。

有时候我们可能需要根据现有的POJO来过滤属性，而这种情况下通常不会让你修改已有的代码在POJO上加注解，这种情况下我们就可以结合@JsonFilter和MixInAnnotations来实现过滤属性，如下例所示，不再多做解释

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonFilter() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setA("1");
5. testPOJO.setB("2");
6. testPOJO.setC("3");
7. testPOJO.setD("4");
8. ObjectMapper objectMapper = **new** ObjectMapper();
9. FilterProvider filters = **new** SimpleFilterProvider().addFilter("myFilter",SimpleBeanPropertyFilter.filterOutAllExcept("a"));
10. objectMapper.setFilters(filters);
11. objectMapper.addMixInAnnotations(TestPOJO.**class**,MyFilterMixIn.**class**);
12. String jsonStr = objectMapper.writeValueAsString(testPOJO);
13. Assert.assertEquals("{\"a\":\"1\"}",jsonStr);
14. }
16. **public** **static** **class** TestPOJO{
17. **private** String a;
18. **private** String c;
19. **private** String d;
20. **private** String b;
21. //getters、setters省略
22. }
24. @JsonFilter("myFilter")
25. **private** **static** **interface** MyFilterMixIn{
26. }

**5、@JsonIgnoreType**

作用于类，表示被注解该类型的属性将不会被序列化和反序列化，也跟上面几个一样属于过滤属性功能的注解，举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonFilter() **throws** Exception {
3. TestPOJO testPOJO = **new** TestPOJO();
4. testPOJO.setName("myName");
5. Sub1 sub1 = **new** Sub1();
6. sub1.setId(1);
7. sub1.setName("sub1");
8. Sub2 sub2 = **new** Sub2();
9. sub2.setId(2);
10. sub2.setAge(22);
11. testPOJO.setMyIn(sub1);
12. testPOJO.setSub1(sub1);
13. testPOJO.setSub2(sub2);
14. ObjectMapper objectMapper = **new** ObjectMapper();
15. String jsonStr = objectMapper.writeValueAsString(testPOJO);
16. System.out.println(jsonStr);
17. }
19. **public** **static** **class** TestPOJO{
20. **private** Sub1 sub1;
21. **private** Sub2 sub2;
22. **private** MyIn myIn;
23. **private** String name;
24. //getters、setters省略
25. }
27. **public** **static** **class** MyIn{
28. **private** **int** id;
29. //getters、setters省略
30. }
32. @JsonIgnoreType
33. **public** **static** **class** Sub1 **extends** MyIn{
34. **private** String name;
35. //getters、setters省略
36. }
38. @JsonIgnoreType
39. **public** **static** **class** Sub2 **extends** MyIn{
40. **private** **int** age;
41. //getters、setters省略
42. }

上面例子中我们在类Sub1和Sub2上都加上了@JsonIgnoreType，那么需要序列化和反序列时POJO中所有Sub1和Sub2类型的属性都将会被忽略，上面测试结果为{"myIn":{"id":1,"name":"sub1"},"name":"myName"}，只输出了name和myIn属性。需要注意的是@JsonIgnoreType是可以继承的，即如果在基类上添加了该注解，那么子类也相当于加了该注解。在上例中，如果只在基类MyIn上添加@JsonIgnoreType那么序列化TestPOJO时将会过滤掉MyIn、Sub1、Sub2。输出结果为{"name":"myName"}

**6、@JsonAnySetter**

作用于方法，在反序列化时用来处理遇到未知的属性的时候调用，在本文前面我们知道可以通过注解@JsonIgnoreProperties(ignoreUnknown=true)来过滤未知的属性，但是如果需要这些未知的属性该如何是好?那么@JsonAnySetter就可以派上用场了，它通常会和map属性配合使用用来保存未知的属性，举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonAnySetter() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"name\":\"myName\",\"code\":\"12345\",\"age\":12}";
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("myName",testPOJO.getName());
7. Assert.assertEquals("12345",testPOJO.getOther().get("code"));
8. Assert.assertEquals(12,testPOJO.getOther().get("age"));
9. }
11. **public** **static** **class** TestPOJO{
12. **private** String name;
14. **private** Map other = **new** HashMap();
16. @JsonAnySetter
17. **public** **void** set(String name,Object value) {
18. other.put(name,value);
19. }
21. //getters、setters省略
22. }

测试用例中我们在set方法上标注了@JsonAnySetter，每当遇到未知的属性时都会调用该方法

**7、@JsonCreator**

作用于方法，通常用来标注构造方法或静态工厂方法上，使用该方法来构建实例，默认的是使用无参的构造方法，通常是和@JsonProperty或@JacksonInject配合使用，举例

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonCreator() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"full\_name\":\"myName\",\"age\":12}";
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("myName",testPOJO.getName());
7. Assert.assertEquals(12, testPOJO.getAge());
8. }
10. **public** **static** **class** TestPOJO{
11. **private** String name;
12. **private** **int** age;
14. @JsonCreator
15. **public** TestPOJO(@JsonProperty("full\_name") String name,@JsonProperty("age") **int** age){
16. **this**.name = name;
17. **this**.age = age;
18. }
19. **public** String getName() {
20. **return** name;
21. }
22. **public** **int** getAge() {
23. **return** age;
24. }
25. }

上面示例中是在构造方法上标注了@JsonCreator，同样你也可以标注在静态工厂方法上，比如：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonCreator() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"name\":\"myName\",\"birthday\":1416299461556}";
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("myName",testPOJO.getName());
7. System.out.println(testPOJO.getBirthday());
8. }
10. **public** **static** **class** TestPOJO{
11. **private** String name;
12. **private** Date birthday;
14. **private** TestPOJO(String name,Date birthday){
15. **this**.name = name;
16. **this**.birthday = birthday;
17. }
19. @JsonCreator
20. **public** **static** TestPOJO getInstance(@JsonProperty("name") String name,@JsonProperty("birthday") **long** timestamp){
21. Date date = **new** Date(timestamp);
22. **return** **new** TestPOJO(name,date);
23. }
25. **public** String getName() {
26. **return** name;
27. }
29. **public** Date getBirthday() {
30. **return** birthday;
31. }
32. }

这个实例中，TestPOJO的构造方法是私有的，外面无法new出来该对象，只能通过工厂方法getInstance来构造实例，此时@JsonCreator就标注在工厂方法上。

除了这2种方式外，还有一种构造方式成为授权式构造器，也是我们平常比较常用到的，这个构造器只有一个参数，且不能使用@JsonProperty。举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jsonCreator() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"full\_name\":\"myName\",\"age\":12}";
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("myName",testPOJO.getName());
7. Assert.assertEquals(12,testPOJO.getAge());
8. }
10. **public** **static** **class** TestPOJO{
11. **private** String name;
12. **private** **int** age;
13. @JsonCreator
14. **public** TestPOJO(Map map){
15. **this**.name = (String)map.get("full\_name");
16. **this**.age = (Integer)map.get("age");
17. }
19. **public** String getName() {
20. **return** name;
21. }
23. **public** **int** getAge() {
24. **return** age;
25. }
26. }

**8、@JacksonInject**

作用于属性、方法、构造参数上，被用来反序列化时标记已经被注入的属性，举例：

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jacksonInject() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"age\":12}";
5. InjectableValues inject = **new** InjectableValues.Std().addValue("name","myName");
6. TestPOJO testPOJO = objectMapper.reader(TestPOJO.**class**).with(inject).readValue(jsonStr);
7. Assert.assertEquals("myName", testPOJO.getName());
8. Assert.assertEquals(12,testPOJO.getAge());
9. }
11. **public** **static** **class** TestPOJO{
12. @JacksonInject("name")
13. **private** String name;
14. **private** **int** age;
16. //getters、setters省略
17. }

上面例子中我们在反序列化前通过InjectableValues来进行注入我们想要的属性

**9、@JsonPOJOBuilder**

作用于类，用来标注如何定制构建对象，使用的是builder模式来构建，比如Value v = new ValueBuilder().withX(3).withY(4).build();这种就是builder模式来构建对象，通常会喝@JsonDeserialize.builder来配合使用，我们举个例子：

**[java]** view plain copy

[在CODE上查看代码片](https://code.csdn.net/snippets/524404)

1. @Test
2. **public** **void** jacksonInject() **throws** Exception {
3. ObjectMapper objectMapper = **new** ObjectMapper();
4. String jsonStr = "{\"name\":\"myName\",\"age\":12}";
5. TestPOJO testPOJO = objectMapper.readValue(jsonStr,TestPOJO.**class**);
6. Assert.assertEquals("myName", testPOJO.getName());
7. Assert.assertEquals(12,testPOJO.getAge());
8. }
10. @JsonDeserialize(builder=TestPOJOBuilder.**class**)
11. **public** **static** **class** TestPOJO{
12. **private** String name;
13. **private** **int** age;
15. **public** TestPOJO(String name, **int** age) {
16. **this**.name = name;
17. **this**.age = age;
18. }
20. **public** String getName() {
21. **return** name;
22. }
24. **public** **int** getAge() {
25. **return** age;
26. }
27. }
29. @JsonPOJOBuilder(buildMethodName = "create",withPrefix = "with")
30. **public** **static** **class** TestPOJOBuilder{
31. **private** String name;
32. **private** **int** age;
34. **public** TestPOJOBuilder withName(String name) {
35. **this**.name = name;
36. **return** **this**;
37. }
39. **public** TestPOJOBuilder withAge(**int** age) {
40. **this**.age = age;
41. **return** **this**;
42. }
44. **public** TestPOJO create() {
45. **return** **new** TestPOJO(name,age);
46. }
47. }

在TestPOJOBuilder上有@JsonPOJOBuilder注解，表示所有的参数传递方法都是以with开头，最终构建好的对象是通过create方法来获得，而在TestPOJO上使用了@JsonDeserializer，告诉我们在反序列化的时候我们使用的是TestPOJOBuilder来构建此对象的

还有一些过期不推荐使用的注解，我们一笔带过，主要知道他们是跟哪些其他注解功能一样即可

@JsonGetter

作用于方法，1.0版本开始的注解，已经过期，不推荐使用，改用@JsonProperty  
@JsonUseSerializer

作用于类和方法，1.5版本开始被移除了，改用@JsonSerialize

@JsonSetter  
作用于方法，1.0版本开始的注解，已过期，不推荐使用，改用@JsonProperty

@JsonClass

作用于方法和类，1.9版本开始被移除了，改为@JsonDeserialize.as

@JsonContentClass

作用于方法，1.9版本开始被移除了，改为@JsonDeserialize.contentAs

@JsonKeyClass

作用于方法和类，1.9版本开始被移除了，改为@JsonDeserialize.keyAs

@JsonUseDeserializer

作用于方法和类，1.5版本开始被移除了，改为@JsonDeserialize

## Redis与数据库处理方式

我采用的是方式一。Nginx和tomcat安装在windows上，三个tomcat服务，修改端口号加以识别。Redis安装在linux上。

修改tomcat端口号，需要修改三个地方：

Server port="18005"

<Connector connectionTimeout="20000" port="18080" protocol="HTTP/1.1" redirectPort="8443"/>

<Connector port="18009" protocol="AJP/1.3" redirectPort="8443"/>

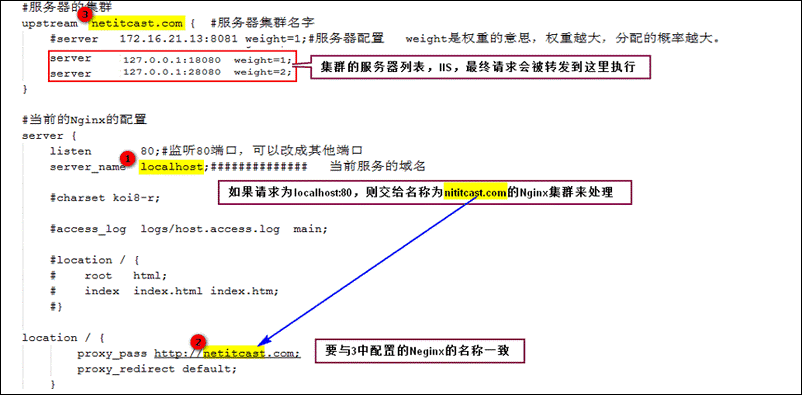
同理第二个tomcat端口修改为28080,28005,28009

然后启动两个tomcat，localhost:18080 和localhost:28080

为了加以区分，修改两个默认的页面。Webapps/root/index.jsp

配置nginx实现负载均衡

核心配置如下



需要配置proxy\_pass,upstream服务器集群。

浏览器端发送请求，nginx监听8010端口，收到请求后，会将该请求交给配置的服务器集群进行处理，服务器集群会根据配置的权重进行分发请求。

启动nginx服务。

Windows中在命令行中进入nginx目录下：

启动 start nginx

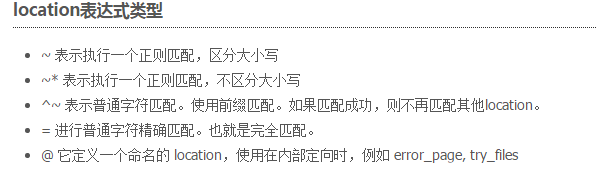
停止nginx.exe –s stop或者nginx –s quit stop是快速停止，可能不会保存相关信息。Quit是有序的停止nginx，并保存相关信息

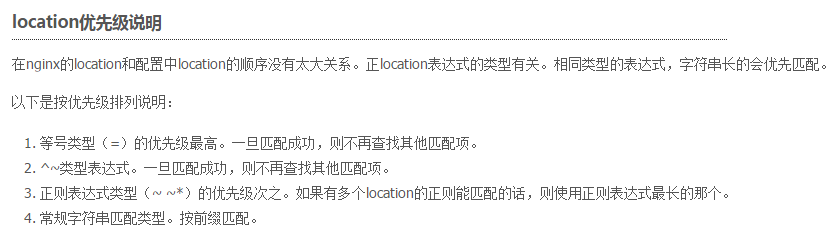
重新载入:nginx –s reload 当配置信息修改后，需要重新载入

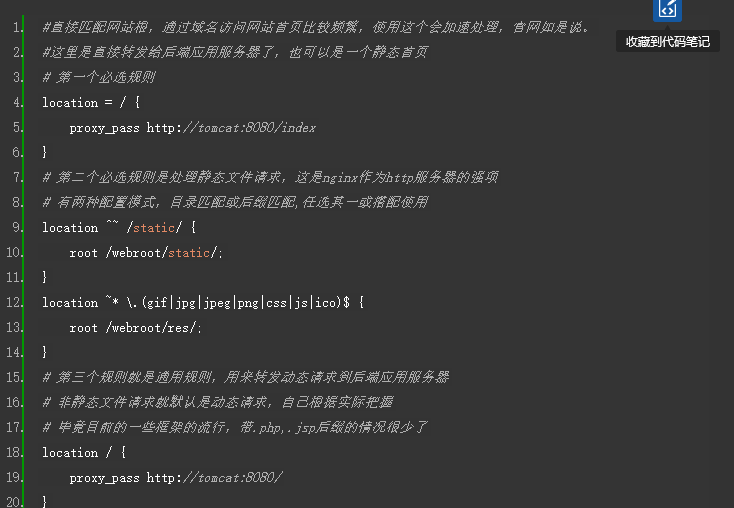
查看nginx版本:nginx -v

访问<http://localhost:8010/>后，发现页面跳转到tomcat，不停的刷新，发现服务会在tomcat1和tomcat2之间跳转。

这里可以通过location配置让nginx处理静态资源htmp .mpg css js .png等，让tomcat处理jsp动态资源。







匹配的顺序：先匹配=，再匹配^~，其次是按文件中顺序的正则匹配，最后交给/匹配。当有匹配成功后，按当前匹配规则处理。

如果此时tomcat1 down了，则下面的服务由tomcat2提供。

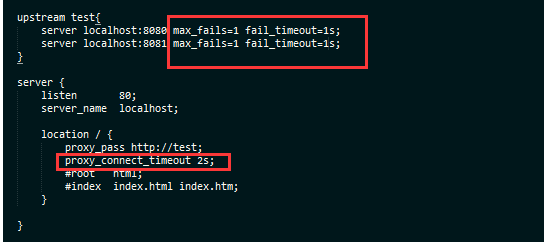
但是会发现，有时响应的时间会比较长，因为nginx将请求转发到tomcat1时，由于已经挂了，就会出现长达一分钟的等待响应过程。因此需要修改配置文件：

* proxy\_connect\_timeout：与服务器连接的超时时间，默认60s
* fail\_timeout：当该时间内服务器没响应，则认为服务器失效，默认10s
* max\_fails：允许连接失败次数，默认为1

这里我们所需等待时间 = proxy\_connect\_timeout + fail\_timeout\*max\_fails，所以我如下配置只需等待3秒，nginx便会将请求转给tomcat2,还是在能接受范围内的。

**项目中只是使用了redis缓存的功能，因此把redis的持久化功能关闭了。**

**添加缓存的原则：不能影响原有的正常的业务逻辑执行。也就是说需要在缓存逻辑的代码上加try catch语句。如果出现异常，则直接捕获，程序继续向下执行。**



### 方式一

Tomcat2

Redis(缓存)

Mysql

Tomcat3

Tomcat1

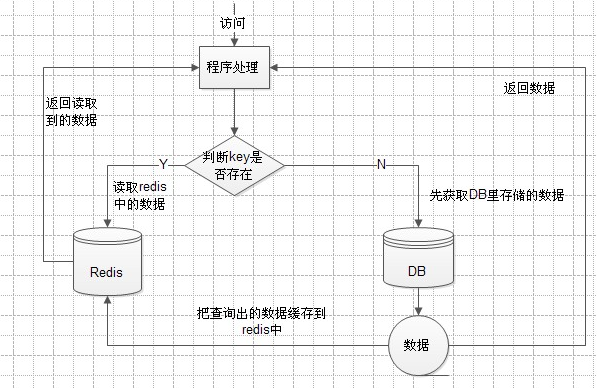
Nginx（负载均衡）

客户端

Nginx作为反向代理，将来自客户端的请求按照权重随机分配给三台tomcat服务器，redis作为缓存使用，存储了数据库中的热点数据，并且作为session共享的解决方案。

流程：客户端发送请求，nginx接收到请求之后，会根据权重将请求随机分配给某台tomcat服务器，tomcat服务器在取数据时，首先会去redis中查看是否存在该数据，如果存在则直接返回缓存好的数据，如果不存在，则再去数据库中取数据，并把数据存放在redis中，最近再将数据返回给nginx服务器，nginx会把响应送回客户端。

在增加 删除或者修改数据时，首先会去查看redis中有没有这个数据，如果存在，则直接删除，如果不存在，则在数据库中进行添加 删除或者修改数据，同时对应的在redis中做相应的操作，最后进行返回。



应用场景：数据量比较大，但是不经常更新的情况

### 方式二

Tomcat2

Redis(缓存)

Mysql

Tomcat3

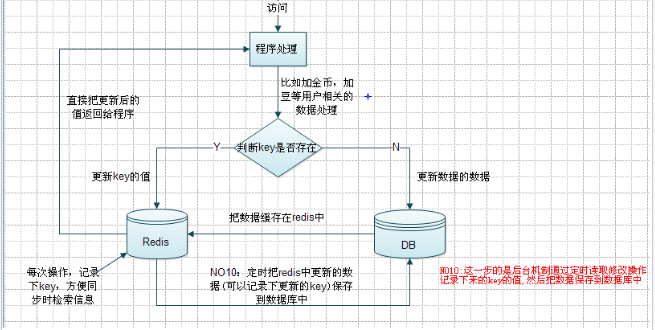
Tomcat1

Nginx（负载均衡）

客户端

把数据缓存在redis中

定时把redis中更新的数据保存在数据库中



这里我们会先去redis中判断数据是否存在，如果存在，则直接更新对应的数据(这一步会把对应更新过的key记录下来，比如也保存到redis中比如：key为：save\_update\_keys【用lpush列表记录】)，并把更新后的数据返回给页面。而如果不存在的话，就会去先更新数据库中内容，然后把数据保存一份到Redis中。后面的工作：后台会有相关机制把Redis中的save\_update\_keys存储的key，分别读取出来，找到对应的数据，更新到DB中。

　　优点：这个流程的主要目的是把Redis当作数据库使用，更新获取数据比DB快。非常适合大数据量的频繁变动(比如微博)。

　　缺点：对Redis的依赖很大，要做好宕机时的数据保存。(不过可以使用redis的快照AOF，快速恢复的话，应该不会有多大影响，因为就算Redis不工作了，也不会影响后续数据的处理。)

　　难点：在前期规划key的格式，存储类型很重要，因为这会影响能否把数据同步到DB。

## 同一个tomcat不同项目的session共享

解决思路： 1 将session存储在公共的地方 2 在不同的数据源之间做数据同步

解决方案：

### 修改tomcat的server.xml文件

每一个web应用程序都有唯一一个ServletContext实例对象，被该web应用下面的每一个servlet共享。通过修改tomcat的server.xml，使不同web应用的ServletContext可以互相访问。然后，用ServletContext的setAttribute()方法把session存入ServletContext中，在另一个web程序就可以用getAttribute()方法取得传递过来的session。

server.xml文件修改如下：

<Host name="localhost" appBase="webapps"unpackWARs="true" autoDeploy="true">

<Context path="/WebappA" debug="9" reloadable="true" crossContext="true"/> //WebappA为项目名，crossContext="true"是关键

<Context path="/WebappB" debug="9" reloadable="true" crossContext="true"/>

</Host>

crossContext属性的意思是：如果设置为true，你可以通过ServletContext.getContext() 调用另外一个WEB应用程序，获得ServletContext 然后再调用其getAttribute() 得到你要的对象。



### 数据库存储session

将session数据存入数据库，不同web程序读取数据库数据即可。这样做的好处是实现简单，缺点是每次请求都要读写一次数据库，加大了系统开销。

### Cookie代替session

cookie是存放在本地硬盘上的文件，适合用于存放小量数据，且可以通过cookie.setPath(“/”)方法设置cookie为同一tomcat下的所有webapp共享。cookie.setPath(“/”)方法的参数”/”表示tomcat的webapps文件夹下的所有应用共享该cookie。



## Redis缓存主页

之前使用了redis对数据库的读写做了缓存，这里实现了对主页的缓存。

### 思路

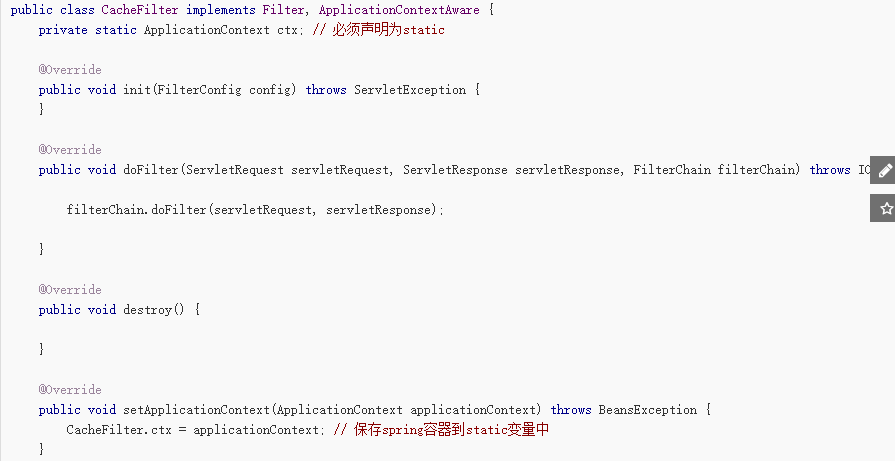
自定义一个过滤器，实现Filter和ApplicationContextAware接口（可以使用ApplicationContext）。

在过滤器中拦截对主页的访问请求。当访问主页时，首先向redis查询主页html的缓存，如果有，则直接返回给客户端。如果没有，则在过滤器中获取截获jsp页面的渲染效果（通过HttpServletResponseWrapper），然后存放进redis中，以供下次使用。设定缓存过期时间10分钟。

### 实现

#### 1 在servlet的过滤器中使用spring容器提供的IOC功能

实现了ApplicationContextAware接口，定义一个全局的ApplicationContext引用，重写setApplicationContext方法。



然后在spring配置文件中配置该filter，使之成为容器管理的bean

<bean id=*"cacheFilter"* class=*"com.wy.filter.CacheFilter"*/>

这样就可以在doFilter方法中直接使用ApplicationContext了。

StringRedisTemplate redis = (StringRedisTemplate)ctx.getBean("redisTemplate");

（注意：哎spring中对StringRedisTemplate进行管理配置）

**第二种：**

**http://www.codesenior.com/en/tutorial/Spring-ContextLoaderListener-And-DispatcherServlet-Concepts**

web.xml中的加载顺序为:listener>>filter>>servlet>>spring。filter的执行顺序是filter-mapping在web.xml中配置的先后顺序。

加载的顺序会影响spring bean的调用，比如filter需要用到bean,但是问题是先加载filter再加载spring的，则filter中初始化操作中的bean为null。所以，如果过滤器中要使用bean，可以将spring的加载改成listener模式。

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>classpath:applicationContext.xml</param-value>

</context-param>

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>

</listener>

ContextLoaderListener作用是在启动web容器时，自动装配ApplicationContext配置信息。

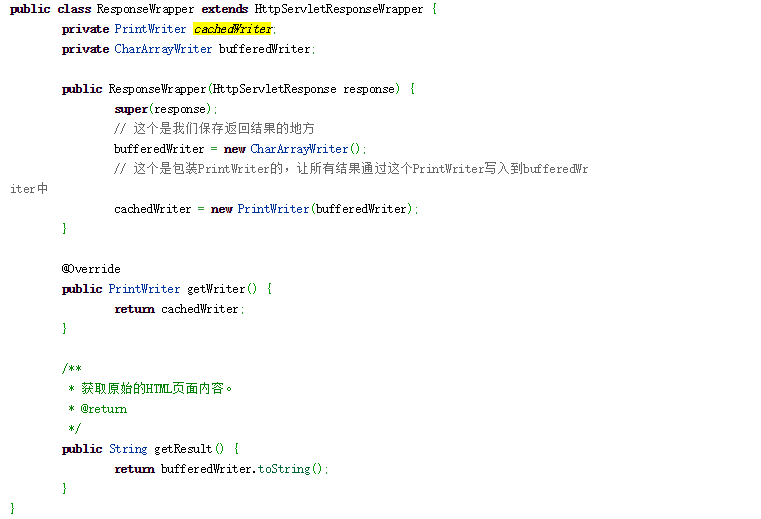


#### 2 如果截获JSP渲染的结果

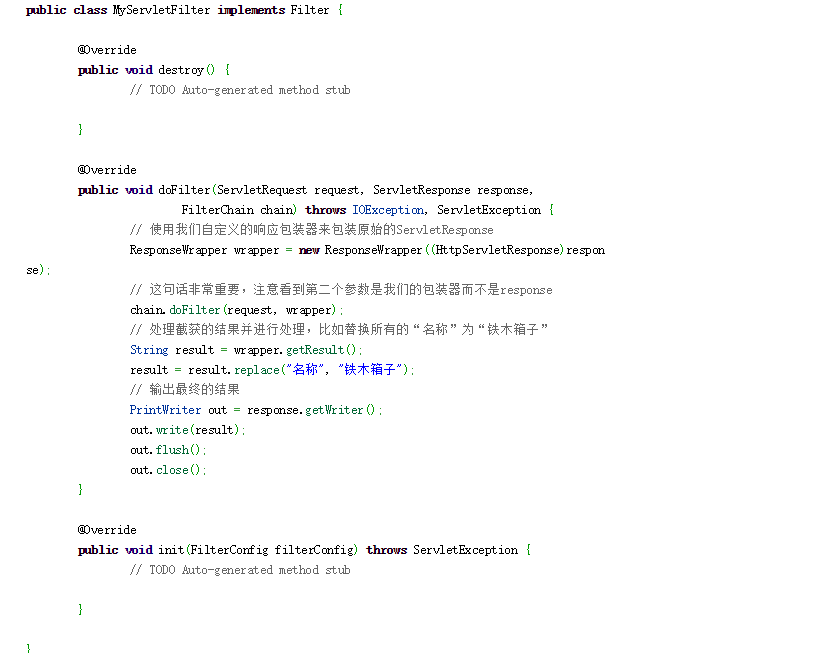
通过继承HttpServletResponseWrapper类实现。

这是一个包装器类，可以在请求输出之前对response对象进行一些额外的操作。比如从response读取相应的数据，或者修改数据，最后再传给客户端。

截获返回给客户端的页面，然后分析html代码，最终装饰页面效果后返回给客户端。



然后在过滤器中使用该类。

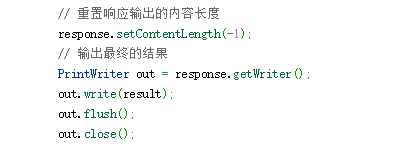


在web.xml中配置过滤器。

这样发现返回的页面，之前的“名称”都被替换为“铁木箱子”了。

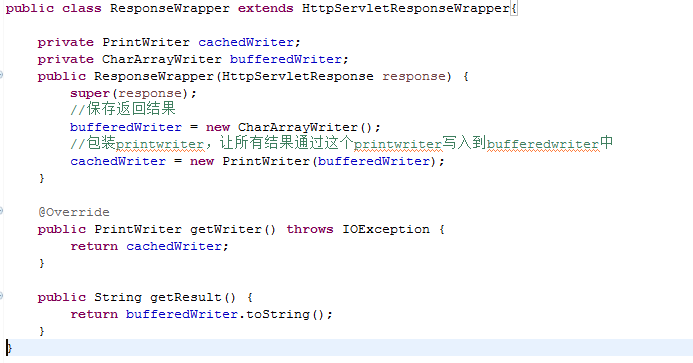
可能遇到的问题：页面只输出了部分内容。。。。。。

在经过多个过滤器或者框架处理后，很有可能在其他框架中设置了输出内容的长度，导致浏览器只根据得到的长度来显示部分内容。因此需要在结果输出前充值一个ContentLength。

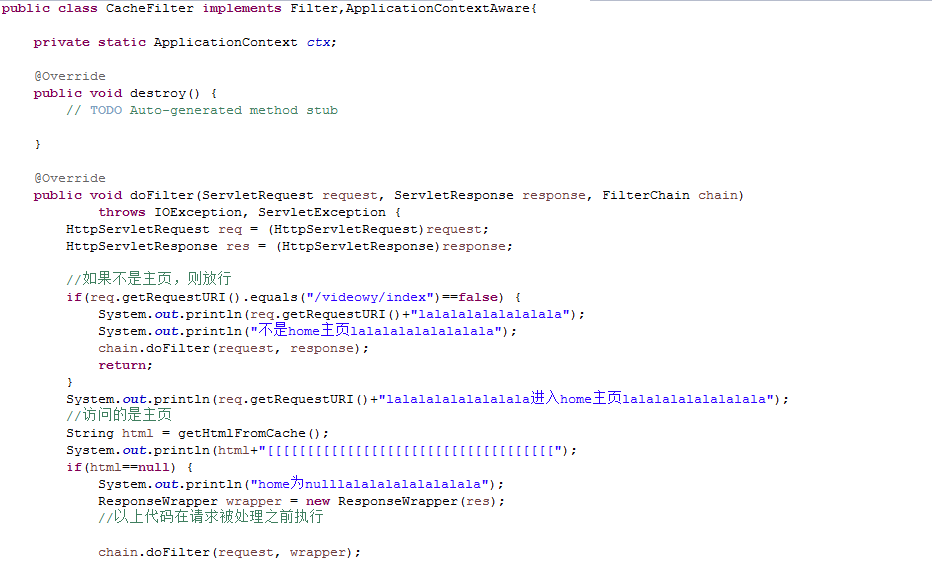


### 代码实现

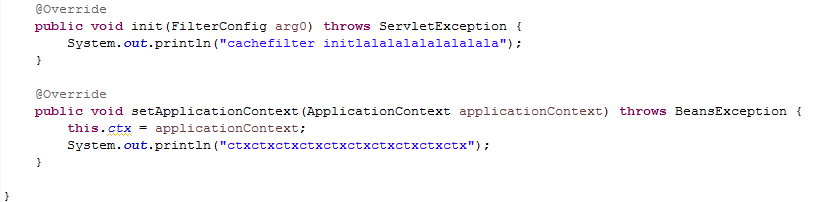
#### ResponseWrapper类



#### CacheFilter类

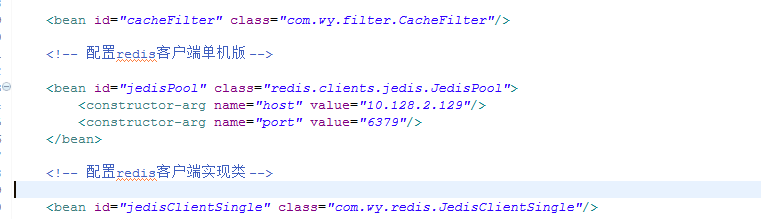


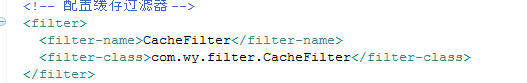


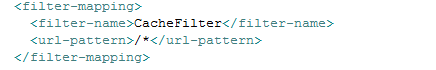


#### 配置

需要让spring管理CacheFilter类，这样才会去加载这个类。

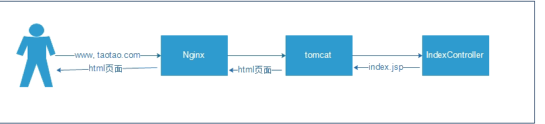






## 静态化

静态是指html页面。



正常的流程如上，如果实现静态化，则请求到达nginx后就会返回响应，不会经过tomcat了。

问题：

1 静态页面如何生成？

1. 请求tomcat
2. 在nginx中配置，如果静态化文件不存在就访问tomcat，如果存在就返回
3. 后台任务，访问tomcat生成静态页面

2 后台数据和静态页面的同步？

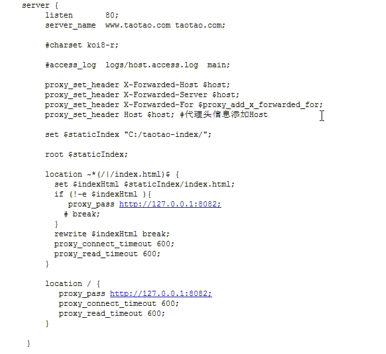
1. 后台数据变更，发送消息，重新生成静态页面（删除这个静态页面）
2. 从业务角度考虑，能否定时生成静态页面

1 使用Quartz定时任务+httpclient实现

2 请求地址变了

3 使用linux中wget命令下载页面保存到某地，然后nginx直接取出这个静态页面，使用crontab实现定时任务

### 静态化配置



## Tomcat优化

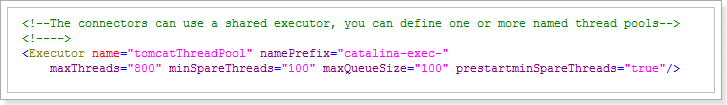
运行模式：使用的Tomcat7，默认是BIO方式，这里开启了NIO（新IO）方式。

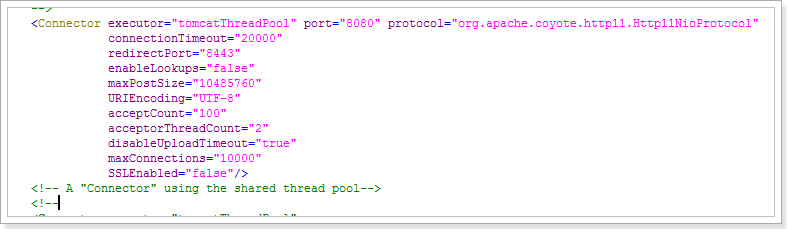
默认没有采用线程池，这里开启了线程池

<Executor name="tomcatThreadPool" namePrefix="catalina-exec-"

maxThreads="500" minSpareThreads="4" prestartminSpareThreads="true" maxQueueSize="50"/>

关闭enablelookups，即禁用了dns查询。





Apache和tomcat进行通讯是用过ajp协议，但是项目中使用的是nginx和tomcat，因此禁用了ajp协议。(相当于禁用了ajp服务)

对jvm优化：

因为tomcat运行在jvm之上，调整了jvm参数，可以提升tomcat运行性能。

将初始堆内存和最大堆内存大小设置相等，这样避免了在后面扩容过程中所带来的性能开销。-Xms –Xmx 1024m

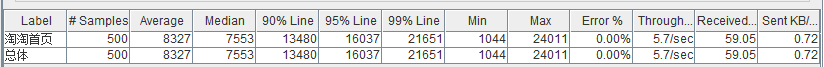
修改/bin/catalina.bat (windows)

set JAVA\_OPTS=-Dfile.encoding=UTF-8 -server -Xms1024m -Xmx2048m -XX:NewSize=512m -XX:MaxNewSize=1024m -XX:PermSize=256m -XX:MaxPermSize=256m

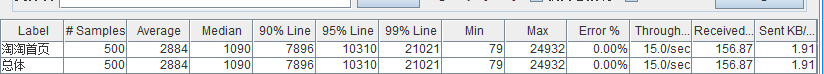
-XX:MaxTenuringThreshold=10 -XX:NewRatio=2 -XX:+DisableExplicitGC

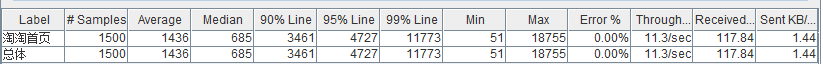
## 压力测试

1个tomcat（windows下）



1个tomcat（windows下） 1个redis作为缓存使用（未缓存首页）





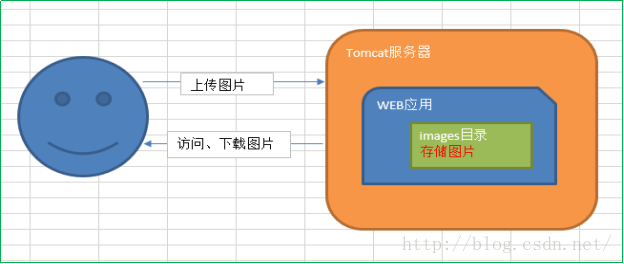
1个tomcat（windows下） 1个redis作为缓存使用（缓存首页）



静态化

## FasfDFS

传统的图片展示



存在的问题：

1 大并发量访问图片时，需要对web应用做负载均衡，**但是会存在图片共享问题**

2 web应用服务器的存储空间有限，它是web应用服务器，而不是存储服务器

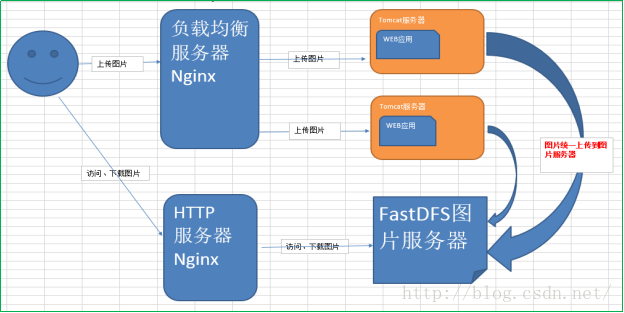
3 web服务器本身的io读写性能不高，图片上传和下载时，速度较慢

4 web应用服务器访问图片时，由于图片内容较大，并发量大的时候，会占用web服务器的带宽，这样该web应用服务器的其他功能就会受到较大影响

解决方案：

1 将图片集中存储在IO读写性能高的图片服务器中

2 下载访问图片时，使用http服务器直接读取图片服务器中的图片



### 简介

原始的是，把文件上传到文件服务器(FTP)中，也就是本地文件夹中，然后配置nginx来访问文件服务器即可。

但是如果本地存储不够，要使用分布式文件存储。

FastDFS是用c语言编写的一款开源的分布式文件系统。FastDFS为互联网量身定制，充4 分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，使用FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。



客户端Tracker server进行文件上传，下载，通过Tracker server调度最终由Storage server完成文件上传和下载。

Tracker server作用是负载均衡和调度，通过Tracker server在文件上传时，可以根据一些策略找到storage server提供文件上传服务。可以将tracker称为追踪服务器或者调度服务器。

Storage server作用是文件存储，客户端上传的文件最终存储在storage server服务器上，storage server没有实现自己的文件系统，而是利用操作系统的文件系统来管理文件。

Tracker server可以有多台，它们之间是平等关系，同时对外提供服务，不存在单点故障，客户端请求tracker server采用轮询方式，如果请求的tracker 无法提供服务，则自动换另一个tracker.

Storage集群采用分组存储方式，storage集群由一个或者多个组构成，集群存储的总容量为集群中所有组的存储容量之和。一个组由一台或多台存储服务器组成，组内的storage server是平等关系，不同组的storage server之间不会相互通信。同组内的storage server之间相互连接并进行文件同步，从而保证同组内每个storage上的文件完全一致。一个组的存储容量为该组内存储服务器容量最小的那个。

采用分组存储的好处是灵活 可控性较强。比如上传文件时，可以由客户端直接指定上传到的组也可以由tracker进行调度选择。一个分组的存储服务器访问压力较大时，可以在该组中增加存储服务器来扩充服务能力（纵向扩容）。当系统容量不足时，可以增加组来扩充存储容量（横向扩容）。

文件上传流程：



文件下载流程：

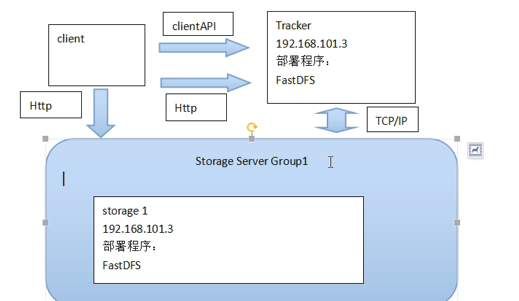
客户端上传文件后存储服务器将文件ID返回给客户端，此文件ID用于以后访问该文件的索引信息。文件索引信息包括：组名，虚拟磁盘路径，数据两级目录，文件名。



* 组名：文件上传后所在的storage组名称，在文件上传成功后有storage服务器返回，需要客户端自行保存。
* 虚拟磁盘路径：storage配置的虚拟路径，与磁盘选项store\_path\*对应。如果配置了store\_path0则是M00，如果配置了store\_path1则是M01，以此类推。
* 数据两级目录：storage服务器在每个虚拟磁盘路径下创建的两级目录，用于存储数据文件。

文件名：与文件上传时不同。是由存储服务器根据特定信息生成，文件名包含：源存储服务器IP地址、文件创建时间戳、文件大小、随机数和文件拓展名等信息。

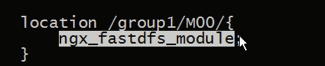
### 项目中采用的fastdfs架构



使用一台服务器搭建。只有一个tracker和一个storage服务。访问图片时，是配置nginx访问图片。（需要对nginx重新编译，有个fastdfs-nginx-module插件，进入nginx目录,

./configure --prefix=/usr/local/nginx --add-module=/usr/local/fastdfs-nginx-module-master/src

然后再编译安装即可 make make install）



如果请求为/group1/M00，则请求交给这个ngx\_fastdfs\_module插件进行处理。这个插件读取配置文件，就可以找到图片。

### 安装

<http://www.linuxidc.com/Linux/2016-09/135537.htm>

第一步：把fastDFS都上传到linux系统。

第二步：安装FastDFS之前，先安装libevent工具包。

yum -y install libevent

第三步：安装libfastcommonV1.0.7工具包。

1. 解压缩
2. ./make.sh
3. ./make.sh install
4. 把/usr/lib64/libfastcommon.so文件向/usr/lib/下复制一份

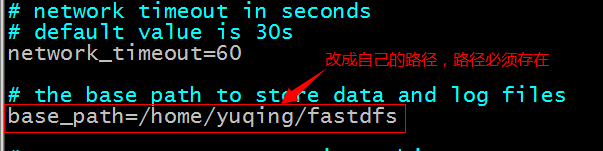
第四步：安装Tracker服务。

1. 解压缩
2. ./make.sh
3. ./make.sh install

安装后在/usr/bin/目录下有以fdfs开头的文件都是编译出来的。

配置文件都放到/etc/fdfs文件夹

1. 把/root/FastDFS/conf目录下的所有的配置文件都复制到/etc/fdfs下。
2. 配置tracker服务。修改/root/FastDFS/conf/tracker.conf文件。

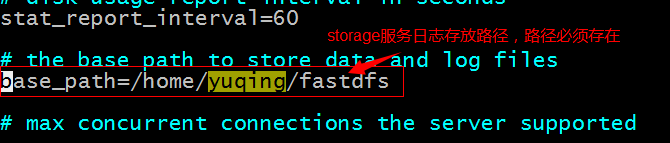


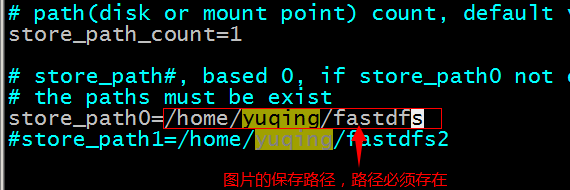
1. 启动tracker。/usr/bin/fdfs\_trackerd /etc/fdfs/tracker.conf

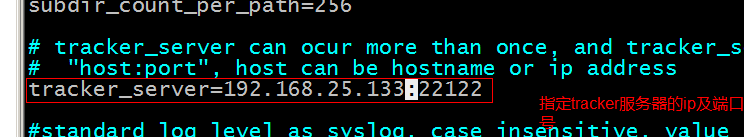
重启使用命令：/usr/bin/fdfs\_trackerd /etc/fdfs/tracker.conf restart

第五步：安装storage服务。

1. 如果是在不同的服务器安装，第四步的1~4需要重新执行。
2. 配置storage服务。修改/root/FastDFS/conf/storage.conf文件





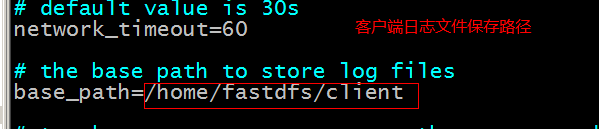


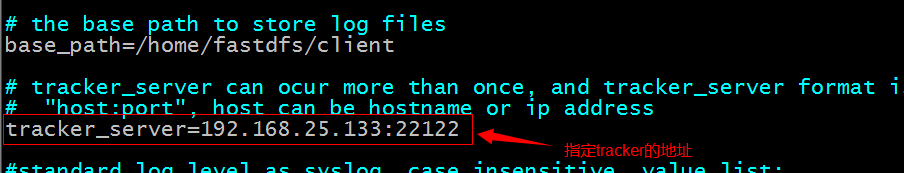
1. 启动storage服务。

/usr/bin/fdfs\_storaged /etc/fdfs/storage.conf **restart**

第六步：测试服务。

1. 修改配置文件/etc/fdfs/client.conf





1. 测试

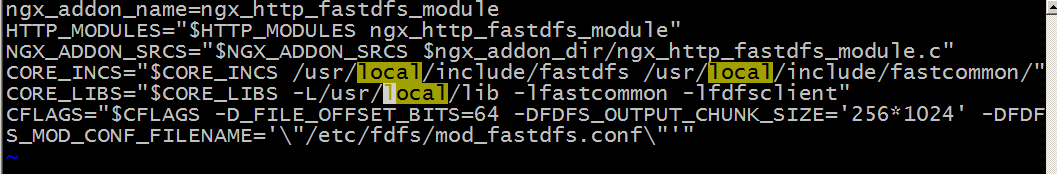
/usr/bin/fdfs\_test /etc/fdfs/client.conf upload anti-steal.jpg

第七步：搭建nginx提供http服务。

可以使用官方提供的nginx插件。要使用nginx插件需要重新编译。

fastdfs-nginx-module\_v1.16.tar.gz

1. 解压插件压缩包
2. 修改/root/fastdfs-nginx-module/src/config文件，把其中的local去掉。



1. 对nginx重新config

./configure \

--prefix=/usr/local/nginx \

--pid-path=/var/run/nginx/nginx.pid \

--lock-path=/var/lock/nginx.lock \

--error-log-path=/var/log/nginx/error.log \

--http-log-path=/var/log/nginx/access.log \

--with-http\_gzip\_static\_module \

--http-client-body-temp-path=/var/temp/nginx/client \

--http-proxy-temp-path=/var/temp/nginx/proxy \

--http-fastcgi-temp-path=/var/temp/nginx/fastcgi \

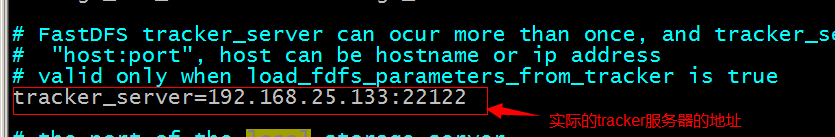
--http-uwsgi-temp-path=/var/temp/nginx/uwsgi \

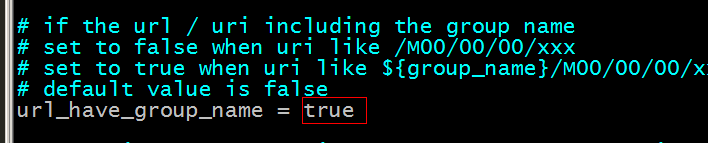
--http-scgi-temp-path=/var/temp/nginx/scgi \

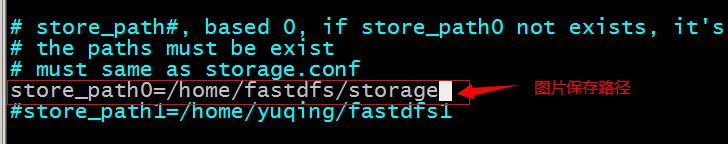
--add-module=/root/fastdfs-nginx-module/src

1. make
2. make install
3. 把/root/fastdfs-nginx-module/src/mod\_fastdfs.conf文件复制到/etc/fdfs目录下。编辑：









1. nginx的配置

在nginx的配置文件中添加一个Server：

server {

listen 80;

server\_name 192.168.101.3;

location /group1/M00/{

#root /home/FastDFS/fdfs\_storage/data;

ngx\_fastdfs\_module;

}

}

1. 将libfdfsclient.so拷贝至/usr/lib下

cp /usr/lib64/libfdfsclient.so /usr/lib/

1. 启动nginx

### 使用Demo

官方提供一个jar包

使用方法：

1. 把FastDFS提供的jar包添加到工程中
2. 初始化全局配置。加载一个配置文件。
3. 创建一个TrackerClient对象。
4. 创建一个TrackerServer对象。
5. 声明一个StorageServer对象，null。
6. 获得StorageClient对象。

直接调用StorageClient对象方法上传文件即可

|  |
| --- |
| @Test  **public** **void** testUpload() **throws** Exception {  // 1、把FastDFS提供的jar包添加到工程中  // 2、初始化全局配置。加载一个配置文件。  ClientGlobal.*init*("D:\\workspaces-itcast\\JaveEE18\\taotao-manager\\taotao-manager-web\\src\\main\\resources\\properties\\client.conf");  // 3、创建一个TrackerClient对象。  TrackerClient trackerClient = **new** TrackerClient();  // 4、创建一个TrackerServer对象。  TrackerServer trackerServer = trackerClient.getConnection();  // 5、声明一个StorageServer对象，null。  StorageServer storageServer = **null**;  // 6、获得StorageClient对象。  StorageClient storageClient = **new** StorageClient(trackerServer, storageServer);  // 7、直接调用StorageClient对象方法上传文件即可。  String[] strings = storageClient.upload\_file("D:\\Documents\\Pictures\\images\\2f2eb938943d.jpg", "jpg", **null**);  **for** (String string : strings) {  System.***out***.println(string);  }  } |

Client.conf

|  |
| --- |
| tracker\_server=192.168.25.133:22122 |

使用工具类上传

@Test

**public** **void** testFastDfsClient() **throws** Exception {

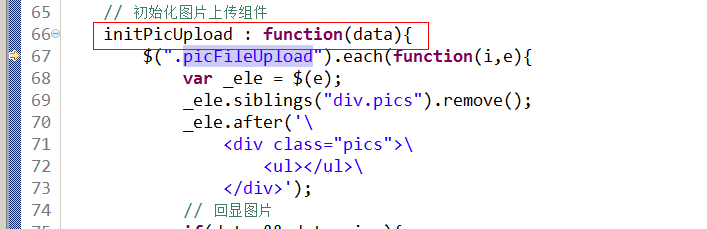
FastDFSClient client = **new** FastDFSClient("D:\\workspaces-itcast\\JaveEE18\\taotao-manager\\taotao-manager-web\\src\\main\\resources\\properties\\client.conf");

String uploadFile = client.uploadFile("D:\\Documents\\Pictures\\images\\200811281555127886.jpg", "jpg");

System.***out***.println(uploadFile);

}

### 项目中使用fastdfs







添加jar包：

Commons-io、fileupload，两个jar包。

#### 在springmvc.xml中配置多媒体解析器

<bean id=*"multipartResolver"*

class=*"org.springframework.web.multipart.commons.CommonsMultipartResolver"*>

<!-- 设定默认编码 -->

<property name=*"defaultEncoding"* value=*"UTF-8"*></property>

<!-- 设定文件上传的最大值5MB，5\*1024\*1024 -->

<property name=*"maxUploadSize"* value=*"5242880"*></property>

</bean>

#### 响应的内容

返回格式(JSON)

|  |
| --- |
| //成功时  {  "error" : 0,  "url" : "http://www.example.com/path/to/file.ext"  }  //失败时  {  "error" : 1,  "message" : "错误信息"  } |

需要创建一个pojo描述返回值。

Pojo中有三个属性：error、url、message，可以放到taotao-common工程中。

|  |
| --- |
| **public** **class** PictureResult {  **private** **int** error;  **private** String url;  **private** String message;  **public** **int** getError() {  **return** error;  }  **public** **void** setError(**int** error) {  **this**.error = error;  }  **public** String getUrl() {  **return** url;  }  **public** **void** setUrl(String url) {  **this**.url = url;  }  **public** String getMessage() {  **return** message;  }  **public** **void** setMessage(String message) {  **this**.message = message;  }    } |

使用responsebody会自动将pojo转换为json

#### Service层

接收图片数据，将图片上传到图片服务器，然后返回结果。

参数：MultiPartFile pictureFile

返回值：PictureResult

@Service

**public** **class** PictureServiceImpl **implements** PictureService {

@Override

**public** PictureResult uploadPic(MultipartFile picFile) {

PictureResult result = **new** PictureResult();

//判断图片是否为空

**if** (picFile.isEmpty()) {

result.setError(1);

result.setMessage("图片为空");

**return** result;

}

//上传到图片服务器

**try** {

//取图片扩展名

String originalFilename = picFile.getOriginalFilename();

//取扩展名不要“.”

String extName = originalFilename.substring(originalFilename.lastIndexOf(".") + 1);

FastDFSClient client = **new** FastDFSClient("classpath:properties/client.conf");

String url = client.uploadFile(picFile.getBytes(), extName);

//把url响应给客户端

result.setError(0);

result.setUrl(url);

} **catch** (Exception e) {

e.printStackTrace();

result.setError(1);

result.setMessage("图片上传失败");

}

**return** result;

}

}

#### controller层

接收上传的图片信息，调用Service把图片上传到图片服务器。返回json数据。需要使用@ResponseBody

@Controller

**public** **class** PictureController {

@Autowired

**private** PictureService pictureService;

@RequestMapping("/pic/upload")

@ResponseBody

**public** PictureResult uploadFile(MultipartFile uploadFile) {

PictureResult result = pictureService.uploadPic(uploadFile);

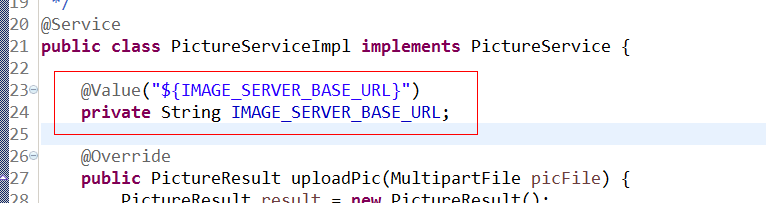
**return** result;

}

}

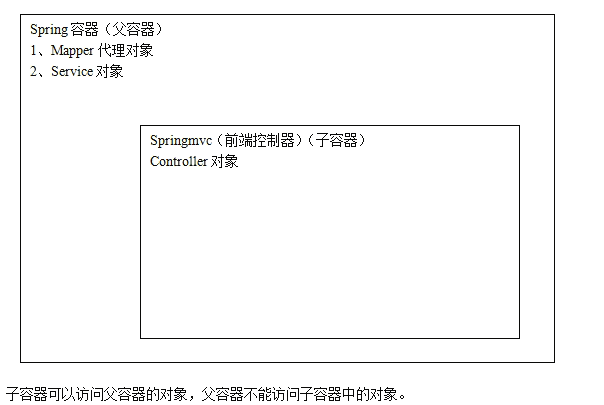
#### 加载属性文件

1. 创建一个属性文件
2. 使用spring容器扫描属性文件。
3. @Value注解取属性的值。



#### 项目中父子容器问题

在web.xml中通过要配置spring容器，也需要配置springmvc容器。当然如果只配置springmvc(前端控制器），也是可以正常使用的。只不过在配置spring容器后，可以和其他框架一起使用了。



在springmvc.xml(子容器）中配置扫描controller包注解，在applicationContext.xml（父容器）中配置扫描service包注解。如果在子容器中也配置了扫描service包的注解，则会导致事务不生效。

### 视频 图片添加的实现

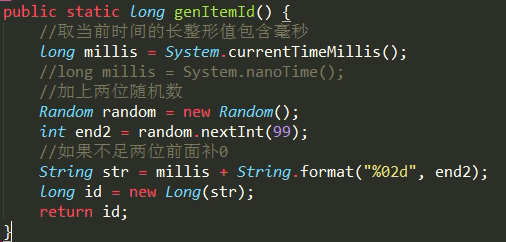
结合上面使用的fastdfs，实现了将图片上传到文件服务器中。下面需要将数据写到数据库中。

写到user和video表中。

执行insert操作，可以使用逆向工程生成的代码。

商品id应该是一串数字。（我项目中的id是主键 自动增长的，这里是非自动增长）

可以使用毫秒+两位随机数生成一个id。（如果使用redis，可以使用incr，由于是单线程，所以不会有并发问题，也就不会有相同的id）

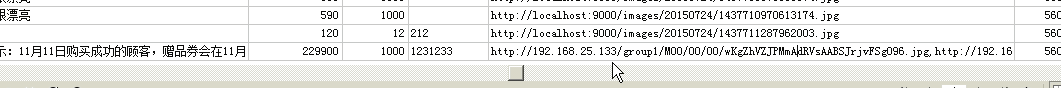


自增id的问题：如果把数据导出，那么这个数据没有了，当再次导入到表中时，id变了，那么如果之前的id有外键关系，那么这种关系就不存在了。另外，在大数据时，将表水平拆分，将表中的数据拆分到多个数据库中存储，那么如果使用自增长，那么每个数据库都生成自己的id，可能就会发生重复。理论上互不干扰，但是实际上还是一个表，id重复。

#### service层

表单提交数据，实际上传递过来的是一个pojo对象。然后调用dao层方法插入数据。

注意：这里的插入和更新操作，都开启了事务



可以看到，这里将上传到文件服务器中的url写进了数据库中。