

目录

2021年11月28日 15:05

[基础语法](#)

[面向对象](#)

[面向对象（二）](#)

[面向对象（三）](#)

[异常处理](#)

[多线程](#)

[反射](#)

Javaweb2020

[html初步](#)

[css初步](#)

[JavaScript初步](#)

[jQuery初步](#)

[XML&Tomcat](#)

[servlet技术](#)

[数据库初步](#)

[jdbc初步](#)

[第一个小项目--书城](#)

Javaweb2022

[页面网站](#)

[Thymeleaf初步](#)

[servlet之session](#)

[第一个小项目--水果](#)

[第二个小项目--QQZone](#)

[Vue 和 Axios 初步](#)

电商推荐

[项目体系架构设计](#)

[数据加载](#)

[谷粒学院 -- 微服务架构](#)

[设计思路](#)

[MyBatis初步](#)

[数据库与代码生成器](#)

[swagger测试器](#)

[同一数据格式 Json](#)

[讲师后台搭建](#)

[ES6, Vue和Node](#)

[后台的前端页面](#)

[阿里云oss与easyExcel](#)

[课程管理系统](#)

[阿里云视频](#)

[Javaweb开发步骤（一）](#)

[微服务架构](#)

[NUXT前台](#)

[redis 缓存](#)

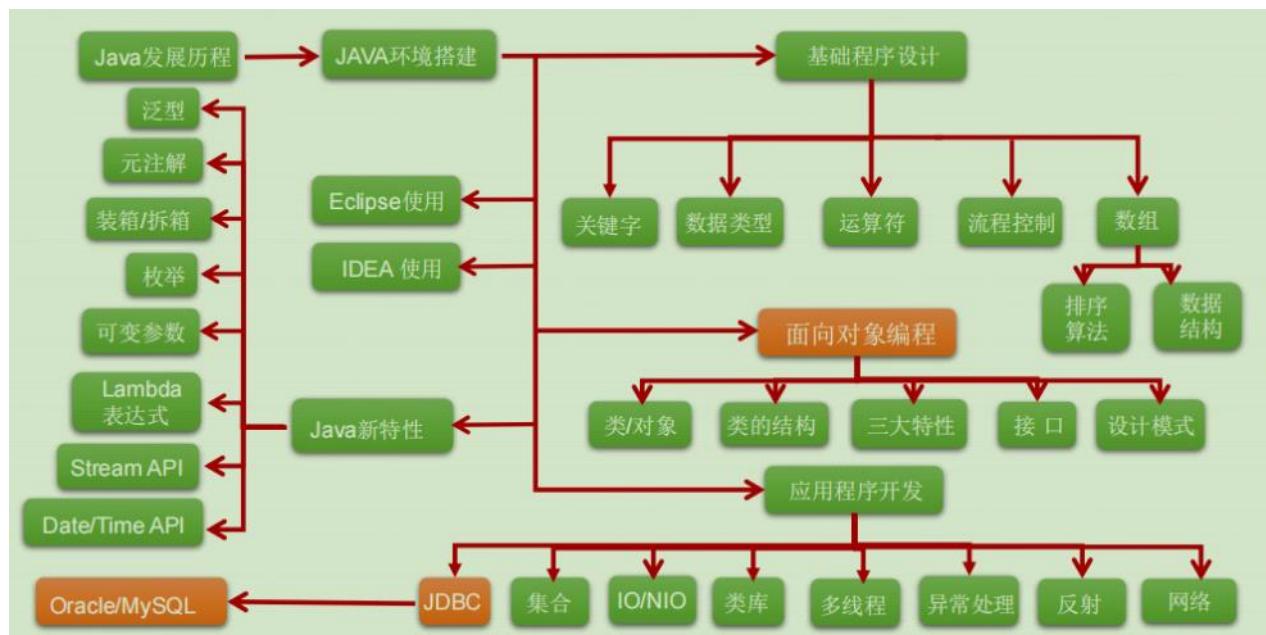
[redis 秒杀技术](#)

[单点登录与JWT](#)

[Jenkins 部署](#)

基础语法

2021年11月28日 15:05



用于定义数据类型的关键字

class	interface	enum	byte	short
int	long	float	double	char
boolean	void			

用于定义流程控制的关键字

if	else	switch	case	default
while	do	for	break	continue
return				

用于定义访问权限修饰符的关键字

private	protected	public		
---------	-----------	--------	--	--

用于定义类，函数，变量修饰符的关键字

abstract	final	static	synchronized	
----------	-------	--------	--------------	--

用于定义类与类之间关系的关键字

extends	implements			
---------	------------	--	--	--

用于定义建立实例及引用实例，判断实例的关键字

new	this	super	instanceof	
-----	------	-------	------------	--

用于异常处理的关键字

try	catch	finally	throw	throws
-----	-------	---------	-------	--------

用于包的关键字

package	import			
---------	--------	--	--	--

其他修饰符关键字

native	strictfp	transient	volatile	assert
--------	----------	-----------	----------	--------

* 用于定义数据类型值的字面值

true	false	null		
------	-------	------	--	--

Java中的名称命名规范

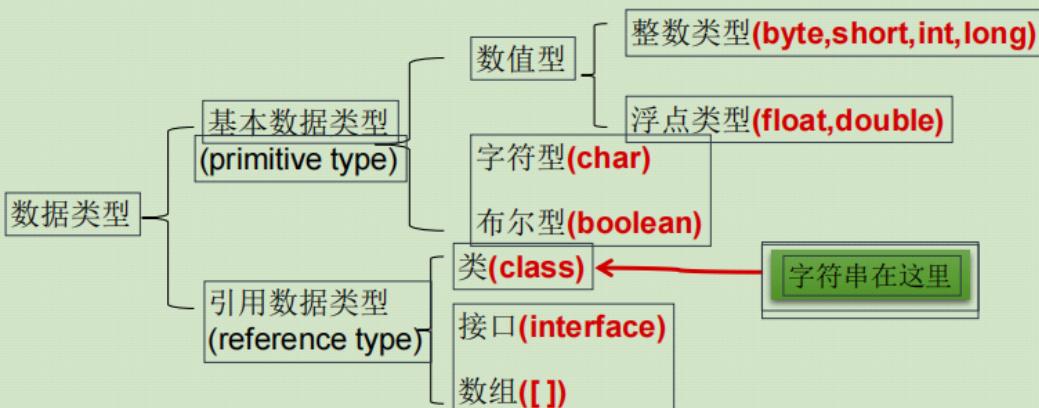
●Java中的名称命名规范：

➤ **包名**：多单词组成时所有字母都小写： xxxyyzzz

➤ **类名、接口名**：多单词组成时，所有单词的首字母大写： XxxYyyZzz

➤ **变量名、方法名**：多单词组成时，第一个单词首字母小写，第二个单词开始每个单词首字母大写： xxxYyyZzz

➤ **常量名**：所有字母都大写。多单词时每个单词用下划线连接： XXX_YYY_ZZZ



整数类型：byte、short、int、long

●Java各整数类型有固定的表数范围和字段长度，不受具体OS的影响，以保证java程序的可移植性。

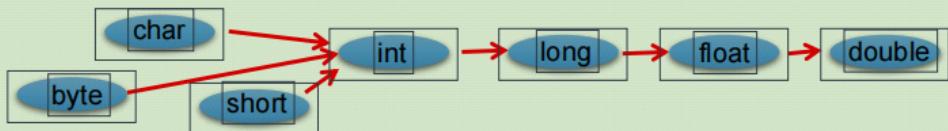
●java的整型常量默认为 int 型，声明long型常量须后加 ‘l’ 或 ‘L’

●java程序中变量通常声明为int型，除非不足以表示较大的数，才使用long

类 型	占用存储空间	表数范围
byte	1字节=8bit位	-128 ~ 127
short	2字节	-2 ¹⁵ ~ 2 ¹⁵ -1
int	4字节	-2 ³¹ ~ 2 ³¹ -1 (约21亿)
long	8字节	-2 ⁶³ ~ 2 ⁶³ -1

基本数据类型转换

●**自动类型转换：**容量小的类型自动转换为容量大的数据类型。数据类型按容量大小排序为：



●有多种类型的数据混合运算时，系统首先自动将所有数据转换成容量最大的那种数据类型，然后再进行计算。

●byte,short,char之间不会相互转换，他们三者在计算时首先转换为int类型。

●boolean类型不能与其它数据类型运算。

●当把任何基本数据类型的值和字符串(String)进行连接运算时(+)，基本数据类型的值将自动转化为字符串(String)类型。

让天下没有难学的技

位运算符

运算符	运算	范例
<<	左移	$3 << 2 = 12 \rightarrow 3*2^2=12$
>>	右移	$3 >> 1 = 1 \rightarrow 3/2=1$
>>>	无符号右移	$3 >>> 1 = 1 \rightarrow 3/2=1$
&	与运算	$6 \& 3 = 2$
	或运算	$6 3 = 7$
^	异或运算	$6 ^ 3 = 5$
~	取反运算	$\sim 6 = -7$

```

3. if(条件表达式1){
    执行代码块1;
}
else if (条件表达式2){
    执行代码块2;
}
.....
else{
    执行代码块n;
}

```

```

int i = 1;
switch (i) {
case 0:
    System.out.println("zero");
    break;
case 1:
    System.out.println("one");
    break;
default:
    System.out.println("default");
    break;
}

```

Java语言中声明数组时不能指定其长度(数组中元素的数), 例如: int a[5]; //非法

●动态初始化：数组声明且为数组元素分配空间与赋值的操作分开进行

```
int[] arr = new int[3];  
arr[0] = 3;  
arr[1] = 9;  
arr[2] = 8;
```

```
String names[];  
names = new String[3];  
names[0] = "钱学森";  
names[1] = "邓稼先";  
names[2] = "袁隆平";
```

●静态初始化：在定义数组的同时就为数组元素分配空间并赋值。

```
int arr[] = new int[]{ 3, 9, 8};  
或  
int[] arr = {3,9,8};
```

```
String names[] = {  
    "李四光", "茅以升", "华罗庚"  
}
```

二维数组 $[\cdot][\cdot]$ ：数组中的数组

格式1（动态初始化）：`int $[\cdot][\cdot]$ arr = new int[3][2];`

定义了名称为`arr`的二维数组

二维数组中有3个一维数组

每一个一维数组中有2个元素

一维数组的名称分别为`arr[0], arr[1], arr[2]`

给第一个一维数组1脚标位赋值为78写法是：~~`arr[0][1] = 78;`~~

格式2（动态初始化）：`int $[\cdot][\cdot]$ arr = new int[3][3];`

二维数组中有3个一维数组。

爱了爱了

每个一维数组都是默认初始化值`null` (注意：区别于格式1)

可以对这个三个一维数组分别进行初始化

`arr[0] = new int[3]; arr[1] = new int[1]; arr[2] = new int[2];`

注：

`int $[\cdot][\cdot]$ arr = new int $[\cdot][3]$;` //非法

格式3（静态初始化）：`int $[\cdot][\cdot]$ arr = new int $[\cdot][\cdot]$ {{3,8,2},{2,7},{9,0,1,6}};`

定义一个名称为`arr`的二维数组，二维数组中有三个一维数组

每一个一维数组中具体元素也都已初始化

第一个一维数组 `arr[0] = {3,8,2};`

第二个一维数组 `arr[1] = {2,7};`

第三个一维数组 `arr[2] = {9,0,1,6};`

第三个一维数组的长度表示方式：`arr[2].length;`

➤ 注意特殊写法情况：`int[] x,y[];` x 是一维数组， y 是二维数组。

➤ Java中多维数组不必都是规则矩阵形式

java.util.Arrays类即为操作数组的工具类，包含了用来操作数组（比如排序和搜索）的各种方法。

1	boolean equals(int[] a,int[] b)	判断两个数组是否相等。
2	String toString(int[] a)	输出数组信息。
3	void fill(int[] a,int val)	将指定值填充到数组之中。
4	void sort(int[] a)	对数组进行排序。
5	int binarySearch(int[] a,int key)	对排序后的数组进行二分法检索指定的值。

array方法

`booleanequals(int[] a, int[] b)`:判断两个数组是否相等。

`StringtoString(int[] a)`:输出数组信息。

`voidfill(int[] a, intval)`:将指定值填充到数组之中。

`voidsort(int[] a)`:对数组进行排序。

`intbinarySearch(int[] a, intkey)`

arraylist动态方法

```
import java.util.ArrayList; // 引入 ArrayList 类  
  
ArrayList<E> objectName =new ArrayList<>(); // 初始化
```

或者直接补不申明类型

```
ArrayList arrl = new ArrayList();
```

可以往里面加不同的类型，但不同类型不能排序

eg:

```
arrl.add(2);
```

排序 nature 为从小至大，直接sort

```
arrl.sort(Comparator.naturalOrder());
```

访问 ArrayList 中的元素可以使用 get() 方法:

```
System.out.println(sites.get(1)); // 访问第二个元素
```

如果要修改 ArrayList 中的元素可以使用 set() 方法:

```
sites.set(2, "Wiki"); // 第一个参数为索引位置，第二个为要修改的值
```

如果要删除 ArrayList 中的元素可以使用 remove() 方法:

```
sites.remove(3); // 删除第四个元素
```

如果要计算 ArrayList 中的元素数量可以使用 size() 方法:

```
System.out.println(sites.size());
```

也可以使用 for-each 来迭代元素:

```
for (String i : sites) {  
    System.out.println(i);  
}
```

从动态数组中返回指定元素的位置的索引值。如果 obj 元素在动态数组中重复出现，返回在数组中最先出现 obj 的元素索引值。如果动态数组中不存在指定的元素，则该 indexOf() 方法返回 -1。

```
sites.indexOf("Runoob");
```

其他

[lastIndexOf\(\)](#)

返回指定元素在 arraylist 中最后一次出现的位置

[isEmpty\(\)](#)

判断 arraylist 是否为空

截断，在下面的实例中，我们使用该 subList() 方法获取索引 1 到 3 (不包括 3) 元素。

```
sites.subList(1, 3))
```


面向对象

2021年11月28日 17:09

● 面向过程(POP) 与 面向对象(OOP)

- 二者都是一种思想，面向对象是相对于面向过程而言的。面向过程，**强调的是功能行为，以函数为最小单位，考虑怎么做。**面向对象，将功能封装进对象，**强调具备了功能的对象，以类/对象为最小单位，考虑谁来做。**
- 面向对象更加强调运用人类在日常的思维逻辑中采用的思想方法与原则，如抽象、分类、继承、聚合、多态等。

● 面向对象的三大特征

- 封装 (Encapsulation)
- 继承 (Inheritance)
- 多态 (Polymorphism)

面向对象：Object Oriented Programming

面向过程：Procedure Oriented Programming

技术不深勿进请绕行

面向对象的思想概述

- 程序员从面向过程的**执行者**转化成了面向对象的**指挥者**

- 面向对象分析方法分析问题的思路和步骤：

- 根据问题需要，选择问题所针对的**现实世界中的实体**。
- 从实体中寻找解决问题相关的属性和功能，这些属性和功能就形成了**概念世界中的类**。
- 把抽象的实体用计算机语言进行描述，**形成计算机世界中类的定义**。即借助某种程序语言，把类构造成计算机能够识别和处理的数据结构。
- 将**类实例化成计算机世界中的对象**。对象是计算机世界中解决问题的最终工具。

内存分配

JVM内存的划分有五片：

1. 寄存器；
2. 本地方法区；
3. 方法区；
4. 栈内存；
5. 堆内存。

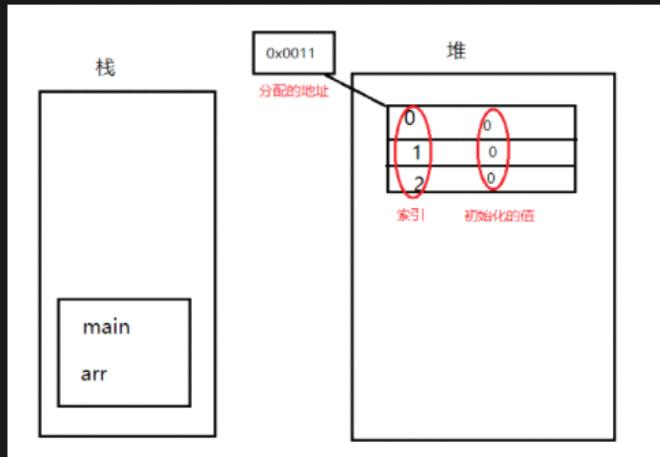
栈内存:栈内存首先是一片内存区域，存储的都是局部变量，凡是定义在方法中的都是局部变量（方法外的是全局变量），for循环内部定义的也是局部变量，是先加载函数才能进行局部变量的定义，所以方法先进栈，然后再定义变量，变量有自己的作用域，一旦离开作用域，变量就会被释放。栈内存的更新速度很快，因为局部变量的生命周期都很短。

堆内存:存储的是数组和对象（其实数组就是对象），凡是new建立的都是在堆中，堆中存放的都是实体（对象），实体用于封装数据，而且是封装多个（实体的多个属性），如果一个数据消失，这个实体也没有消失，还可以用，所以堆是不会随时释放的，但是栈不一样，栈里存放的都是单个变量，变量被释放了，那就没有了。堆里的实体虽然不会被释放，但是会被当成垃圾，Java有垃圾回收机制不定时的收取。

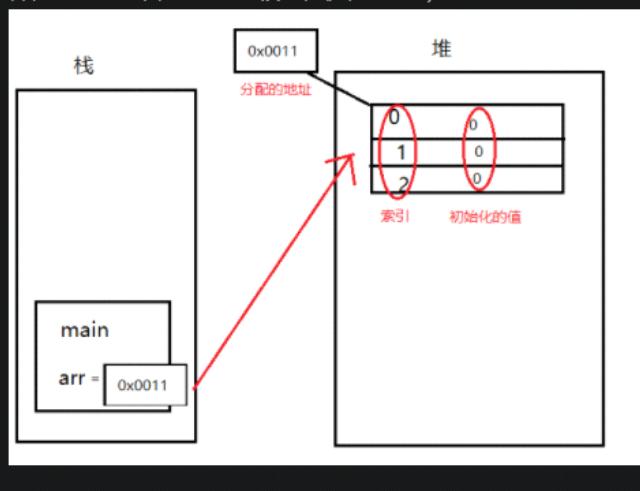
下面我们通过一个图例详细讲一下堆和栈：

比如主函数里的语句 int [] arr=new int [3];在内存中是怎么被定义的：

主函数先进栈，在栈中定义一个变量arr，接下来为arr赋值，但是右边不是一个具体值，是一个实体。实体创建在堆里，在堆里首先通过new关键字开辟一个空间，内存中存储数据的时候都是通过地址来体现的，地址是一块连续的二进制，然后给这个实体分配一个内存地址。数组都是有一个索引，数组这个实体在堆内存中产生之后每一个空间都会进行默认的初始化（这是堆内存的特点，未初始化的数据是不能用的，但在堆里是可以用的，因为初始化过了，但是在栈里没有），不同的类型初始化的值不一样。所以堆和栈里就创建了变量和实体：



我们刚刚说过给堆分配了一个地址，把堆的地址赋给arr，arr就通过地址指向了数组。所以arr想操纵数组时，就通过地址，而不是直接把实体都赋给它。这种我们不再叫他基本数据类型，而叫引用数据类型。称为arr引用了堆内存当中的实体。（可以理解为c或c++的指针，Java成长自c++和c++很像，优化了c++）



垃圾回收

在java中是通过引用来和对象进行关联的，也就是说如果要操作对象，必须通过引用来进行。那么很显然一个简单的办法就是通过引用计数来判断一个

对象是否可以被回收。不失一般性，如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到，那么这个对象就成为可被回收的对象了。这种方式成为引用计数法。

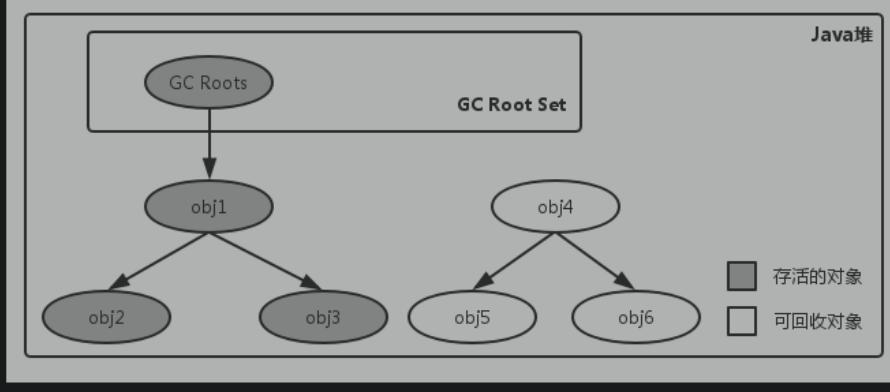
这种方式的特点是实现简单，而且效率较高，但是它无法解决循环引用的问题，因此在Java中并没有采用这种方式（Python采用的是引用计数法）。

1. 引用计数法

给对象添加一引用计数器，被引用一次计数器值就加1；当引用失效时，计数器值就减1；计数器为0时，对象就是不可能再被使用的，简单高效，缺点是无法解决对象之间相互循环引用的问题。

2. 可达性分析算法

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。此算法解决了上述循环引用的问题。



那么c++和c呢

- 1、new/delete是C++的操作符，而malloc/free是C中的函数。
- 2、new做两件事，一是分配内存，二是调用类的构造函数；同样，delete会调用类的析构函数和释放内存。而malloc和free只是分配和释放内存。
- 3、new建立的是一个对象，而malloc分配的是一块内存；new建立的对象可以用成员函数访问，不要直接访问它的地址空间；malloc分配的是一块内存区域，用指针访问，可以在里面移动指针；new出来的指针是带有类型信息的，而malloc返回的是void指针。
- 4、new/delete是保留字，不需要头文件支持；malloc/free需要头文件库函数支持。

- 创建对象语法：类名 对象名 = new 类名();
- 使用“对象名.对象成员”的方式访问对象成员（包括属性和方法）

举例：

```
public class Animal {
    public int legs;
    public void eat(){
        System.out.println("Eating.");
    }
    public void move(){
        System.out.println("Move.");
    }
}
```

```
public class Zoo{
    public static void main(String args[]){
        //创建对象
        Animal xb=new Animal();
        xb.legs=4;//访问属性
        System.out.println(xb.legs);
        xb.eat();//访问方法
        xb.move();//访问方法
    }
}
```

让天下没有难学的技术

- 我们也可以不定义对象的句柄，而直接调用这个对象的方法。这样的对象叫做匿名对象。

➤ 如：**new Person().shout();**

匿名对象特点：

- 1、由于我们没有记录堆内存对象的地址值，所以只能用一次，再次使用就找不到了
- 2、匿名对象的好处就是使用完毕就是垃圾，可以在垃圾回收器空闲时回收，节省内存空间

	成员变量	局部变量
声明的位置	直接声明在类中	方法形参或内部、代码块内、构造器内等
修饰符	private、public、static、final等	不能用权限修饰符修饰，可以用final修饰
初始化值	有默认初始化值	没有默认初始化值，必须显式赋值，方可使用
内存加载位置	堆空间 或 静态域内	栈空间

什么是方法(method、函数)：

- 方法是类或对象行为特征的抽象，用来完成某个功能操作。在某些语言中也称为函数或过程。
- 将功能封装为方法的目的是，可以实现代码重用，简化代码
- Java里的方法不能独立存在，所有的方法必须定义在类里。



方法的声明格式：

修饰符 返回值类型 方法名 (参数类型 形参1, 参数类型 形参2,) {

_{子方法和类体}

方法的声明格式：

```
修饰符 返回值类型 方法名 (参数类型 形参1, 参数类型 形参2, ....) {  
    方法体程序代码  
    return 返回值;  
}
```

其中：

修饰符：**public**,**缺省**,**private**, **protected**等

重载

重载的概念

在同一个类中，允许存在一个以上的同名方法，只要它们的参数个数或者参数类型不同即可。

重载的特点：

与返回值类型无关，只看参数列表，且参数列表必须不同。(参数个数或参数类型)。调用时，根据方法参数列表的不同来区别。

重载示例：

```
//返回两个整数的和  
int add(int x,int y){return x+y;}  
//返回三个整数的和  
int add(int x,int y,int z){return x+y+z;}  
//返回两个小数的和  
double add(double x,double y){return x+y;}
```

传递多个参数

JavaSE 5.0 中提供了**Varargs(variable number of arguments)**机制，允许直接定义能和多个实参相匹配的形参。从而，可以用一种更简单的方式，来传递个数可变的实参。

//JDK 5.0以前：采用数组形参来定义方法，传入多个同一类型变量

public static void test(int a ,String[] books);

//JDK5.0：采用可变个数形参来定义方法，传入多个同一类型变量

public static void test(int a ,String...books);

● Java的实参值如何传入方法呢？

Java里方法的参数传递方式只有一种：**值传递**。 即将实际参数值的副本（复制品）传入方法内，而参数本身不受影响。

- 形参是基本数据类型：将实参基本数据类型变量的“数据值”传递给形参
- 形参是引用数据类型：将实参引用数据类型变量的“地址值”传递给形参

举个例子

```
public static void main(String[] args) {  
    int x = 5;  
  
    System.out.println("修改之前x = " + x); // 5  
  
    // x是实参  
    change(x);  
  
    System.out.println("修改之后x = " + x); // 5  
}  
  
public static void change(int x) {  
    System.out.println("change:修改之前x = " + x);  
    x = 3;  
    System.out.println("change:修改之后x = " + x);  
}
```

这样才可以传应用(借助类)

```
public static void main(String[] args) {  
    Person obj = new Person();  
    obj.age = 5;  
    System.out.println("修改之前age = " + obj.age); // 5  
    // x是实参  
    change(obj);  
    System.out.println("修改之后age = " + obj.age); // 3  
}  
  
public static void change(Person obj) {  
    System.out.println("change:修改之前age = " + obj.age);  
    obj.age = 3;  
    System.out.println("change:修改之后age = " + obj.age);  
}
```

其中Person类定义为：

```
class Person{  
    int age;  
}
```

我们程序设计追求“高内聚，低耦合”。

- 高内聚：类的内部数据操作细节自己完成，不允许外部干涉；
- 低耦合：仅对外暴露少量的方法用于使用。

信息的封装和隐藏

Java中通过将数据声明为私有的(private)，再提供公共的（public）方法:**getXxx()**和**setXxx()**实现对该属性的操作，以实现下述目的：

- 隐藏一个类中不需要对外提供的实现细节；
- 使用者只能通过事先定制好的方法来访问数据，可以方便地加入控制逻辑，限制对属性的不合理操作；
- 便于修改，增强代码的可维护性；

```
private int legs; // 将属性legs定义为private，只能被Animal类内部访问  
public void setLegs(int i) { // 在这里定义方法 eat() 和 move()
```

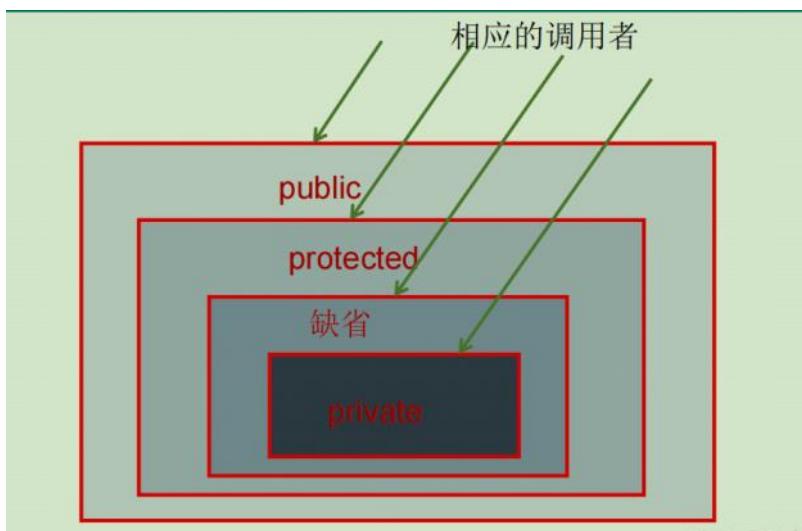
这四种访问修饰符的访问权限不一样。

1.private:及私有的，对访问权限限制最窄的修饰符。被private修饰的属性以及方法只能被该类的对象访问。它的子类也不可以访问，更不支持跨包访问。

2.protected:及保护访问权限，是介于public和private之间的一种访问修饰。被protected修饰的属性及方法只能被类本身的方法和子类访问。（子类在不同的包中也可以访问）

3.public:及共有的，是访问权限限制最宽的修饰符。被public修饰的类、属性、及方法不仅可以跨类访问，而且可以跨包访问。

4.default:及默认的，不加任何访问修饰符。常被叫做“默认访问权限”或者“包访问权限”。无任修饰符时，只支持在同一个包中进行访问。



构造器的作用：创建对象；给对象进行初始化

➤如：Order o = new Order(); Person p = new Person("Peter", 15);

➤如同我们规定每个“人”一出生就必须先洗澡，我们就可以在“人”的构造器中加入完成“洗澡”的程序代码，于是每个“人”一出生就会自动完成“洗澡”，程序就不必再在每个人刚出生时一个一个地告诉他们要“洗澡”了。

```
public class Animal {  
    private int legs;  
    // 构造器  
    public Animal() {  
        legs = 4;  
    }  
}
```

●注 意：

➤Java语言中，每个类都至少有一个构造器

➤默认构造器的修饰符与所属类的修饰符一致

➤一旦显式定义了构造器，则系统不再提供默认构造器

➤一个类可以创建多个重载的构造器

➤父类的构造器不可被子类继承

构造器重载

构造器重载举例

```
public class Person {  
    private String name;  
    private int age;  
    private Date birthDate;  
    public Person(String n, int a, Date d) {  
        name = n;  
        age = a;  
        birthDate = d;  
    }  
    public Person(String n, int a) {  
        name = n;  
        age = a;  
    }  
    public Person(String n, Date d) {  
        name = n;  
        birthDate = d;  
    }  
    public Person(String n) {  
        name = n;  
        age = 30;  
    }  
}
```

让天下

JavaBean

javaBean规范:

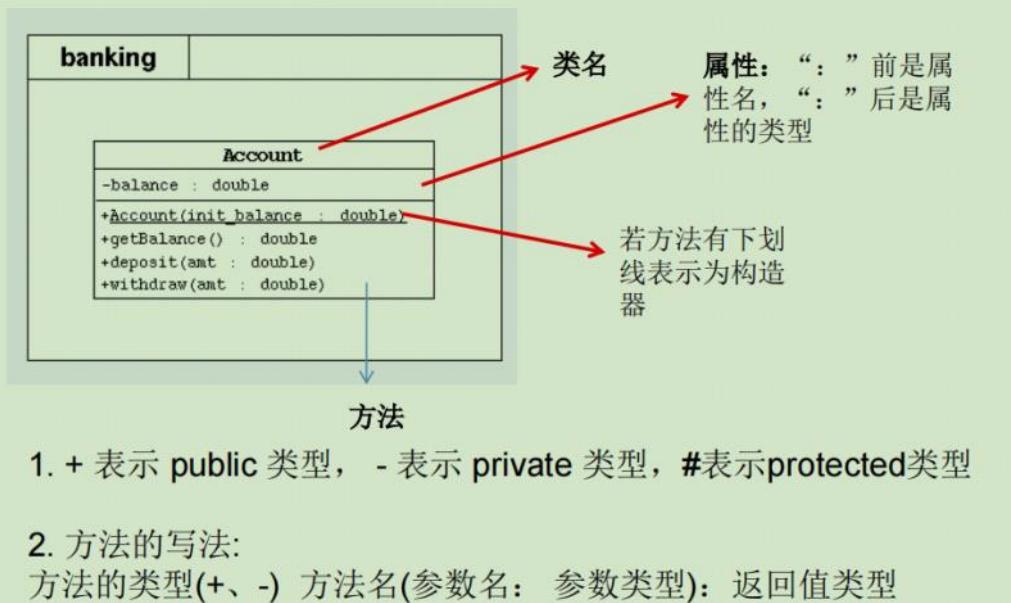
javaBean是一种java语言写成的可重用组件（类）。

必须遵循一定的规范：

- 1、类必须使用public修饰。
- 2、必须保证有公共无参数构造器。
- 3、包含了属性的操作手段（给属性赋值，获取属性值）。

Java语言欠缺属性、事件、多重继承功能。所以，如果要在Java程序中实现一些面向对象编程的常见需求，只能手写大量胶水代码。Java Bean正是编写这套胶水代码^⑨的惯用模式或约定。这些约定包括getXxx、setXxx、isXxx、addXxxListener、XxxEvent等。遵守上述约定的类可以用于若干工具或库。

4.8 拓展知识：UML类图



this

什么时候使用this关键字呢？

➤ 当在方法内需要用到调用该方法的对象时，就用this。

具体的：我们可以用this来区分属性和局部变量。

比如： `this.name = name;`

c++里面this就是指向这个类的指针

import

package语句作为Java源文件的第一条语句，指明该文件中定义的类所在的包。(若缺省该语句，则指定为无名包)。它的格式为：

package 顶层包名.子包名；

举例： pack1\pack2\PackageTest.java

```
package pack1.pack2; //指定类PackageTest属于包pack1.pack2
public class PackageTest{
    public void display(){
        System.out.println("in method display()");
    }
}
```

包对应于文件系统的目录，**package**语句中，用“.”来指明包(目录)的层次；
包通常用小写单词标识。通常使用所在公司域名的倒置：**com.atguigu.xxx**

MVC设计模式

MVC是常用的设计模式之一，将整个程序分为三个层次：视图模型层，控制器层，与数据模型层。这种将程序输入输出、数据处理，以及数据的展示分离开来的设计模式使程序结构变的灵活而且清晰，同时也描述了程序各个对象间的通信方式，降低了程序的耦合性。

模型层 model 主要处理数据

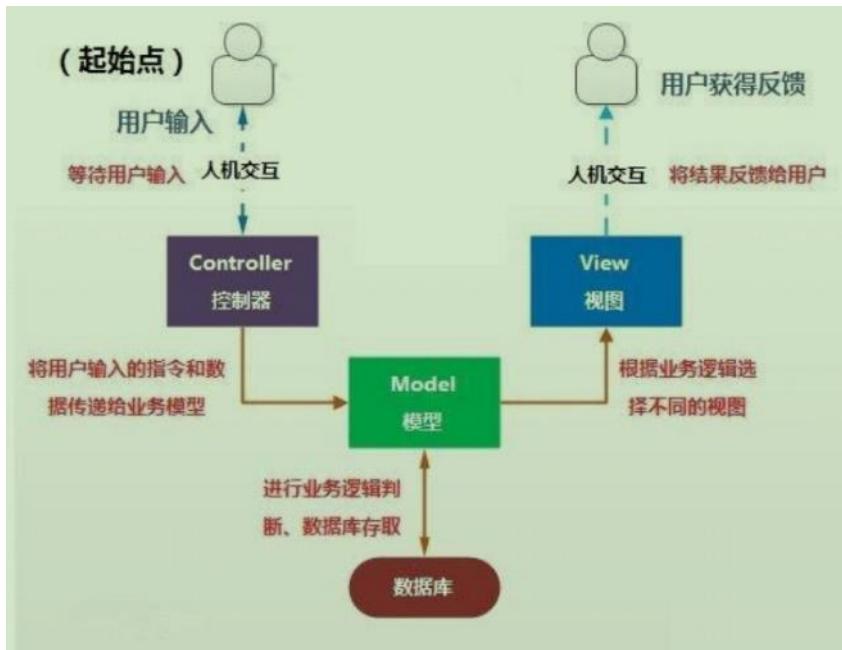
- >数据对象封装 model.bean/domain
- >数据库操作类 model.dao
- >数据库 model.db

控制层 controller 处理业务逻辑

- >应用界面相关 controller.activity
- >存放fragment controller.fragment
- >显示列表的适配器 controller.adapter
- >服务相关的 controller.service
- >抽取的基类 controller.base

视图层 view 显示数据

- >相关工具类 view.utils
- >自定义view view.ui



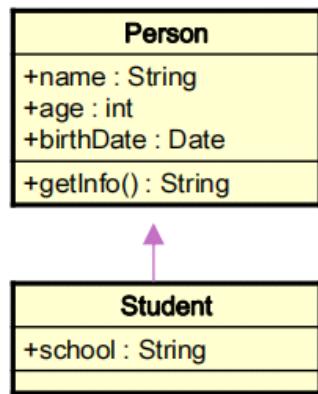
JDK中主要的包介绍

1. **java.lang**----包含一些Java语言的核心类，如String、Math、Integer、System和Thread，提供常用功能
2. **java.net**----包含执行与网络相关的操作的类和接口。
3. **java.io** ----包含能提供多种输入/输出功能的类。
4. **java.util**----包含一些实用工具类，如定义系统特性、接口的集合框架类、使用与日期日历相关的函数。
5. **java.text**----包含了一些java格式化相关的类
6. **java.sql**----包含了java进行JDBC数据库编程的相关类/接口
7. **java.awt**----包含了构成抽象窗口工具集（abstract window toolkits）的多个类，这些类被用来构建和管理应用程序的图形用户界面(GUI)。 B/S C/S

面向对象 (二)

2021年12月17日 22:50

- 通过继承，简化Student类的定义：



```
class Person {
    public String name;
    public int age;
    public Date birthDate;

    public String getInfo() {
        // ...
    }
}

class Student extends Person {
    public String school;
}
```

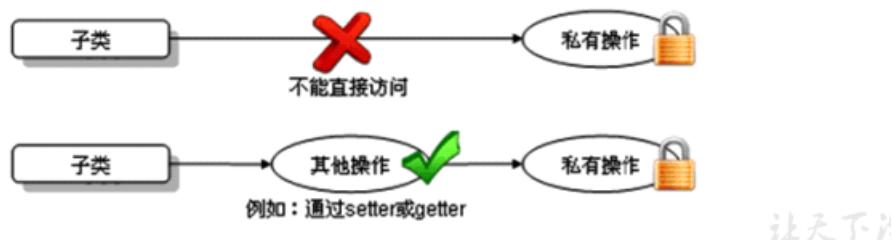
- Student类继承了父类Person的所有属性和方法，并增加了一个属性school。Person中的属性和方法，Student都可以使用。

类继承语法规则：

```
class Subclass extends SuperClass{ }
```

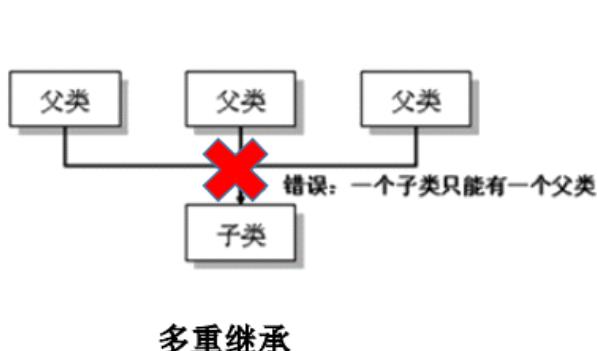
关于继承的规则：

- 子类不能直接访问父类中私有的(private)的成员变量和方法。



- Java只支持单继承和多层继承，不允许多重继承

- 一个子类只能有一个父类
- 一个父类可以派生出多个子类
 - ✓ `class SubDemo extends Demo{ } //ok`
 - ✓ `class SubDemo extends Demo1,Demo2...//error`



- 多层继承

让天下没有难懂的技...

重写



5.2 方法的重写(override/overwrite)



● 定义：在子类中可以根据需要对从父类中继承来的方法进行改造，也称为方法的重置、覆盖。在程序执行时，子类的方法将覆盖父类的方法。

● 要求：

1. 子类重写的方法必须和父类被重写的方法具有相同的方法名称、参数列表
2. 子类重写的方法的返回值类型不能大于父类被重写的方法的返回值类型
3. 子类重写的方法使用的访问权限不能小于父类被重写的方法的访问权限
 - 子类不能重写父类中声明为`private`权限的方法
4. 子类方法抛出的异常不能大于父类被重写方法的异常

● 注意：

子类与父类中同名同参数的方法必须同时声明为非`static`的(即为重写)，或者同时声明为`static`的(不是重写)。因为`static`方法是属于类的，子类无法覆盖父类的方法。

```

public class Person {
    public String name;
    public int age;
    public String getInfo() {
        return "Name: " + name + "\n" + "age: " + age;
    }
}
public class Student extends Person {
    public String school;
    public String getInfo() { //重写方法
        return "Name: " + name + "\nage: " + age
            + "\nschool: " + school;
    }
}
public static void main(String args[]){
    Student s1=new Student();
    s1.name="Bob";
    s1.age=20;
    s1.school="school2";
    System.out.println(s1.getInfo()); //Name:Bob age:20 school:school2
}

```

重写方法举例(1)

```

Person p1=new Person();
//调用Person类的getInfo()方法
p1.getInfo();
Student s1=new Student();
//调用Student类的getInfo()方法
s1.getInfo();

```

这是一种“多态性”：同名的方法，用不同的对象来区分调用的是哪一个方法。

让天下没有难学的技术

```

class Parent {
    public void method1() {}
}

class Child extends Parent {
    //非法，子类中的method1()的访问权限private比被覆盖方法的访问权限public小
    private void method1() {}
}

```

重写方法举例(2)

1. 如果现在父类的一个方法定义成**private**访问权限，在子类中将此方法声明为**default**访问权限，那么这样还叫重写吗？(NO)

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	Yes			
(缺省)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

修饰符	类内部	同一个包	不同包的子类	同一个工程
private	Yes			
(缺省)	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

对于**class**的权限修饰只可以用**public**和**default(缺省)**。

- **public**类可以在任意地方被访问。
- **default**类只可以被同一个包内部的类访问。

在Java类中使用**super**来调用父类中的指定操作：

- **super**可用于访问父类中定义的属性
- **super**可用于调用父类中定义的成员方法
- **super**可用于在子类构造器中调用父类的构造器

注意：

- 尤其当子父类出现同名成员时，可以用**super**表明调用的是父类中的成员
- **super**的追溯不仅限于直接父类
- **super**和**this**的用法相像，**this**代表本类对象的引用，**super**代表父类的内存空间的标识

调用父类的构造器

- 子类中所有的构造器默认都会访问父类中空参数的构造器
- 当父类中没有空参数的构造器时，子类的构造器必须通过**this(参数列表)**或者**super(参数列表)**语句指定调用本类或者父类中相应的构造器。同时，只能“二选一”，且必须放在构造器的首行
- 如果子类构造器中既未显式调用父类或本类的构造器，且父类中又没有无参的构造器，则**编译出错**

调用父类构造器举例

```
public class Student extends Person {  
    private String school;  
    public Student(String name, int age, String s) {  
        super(name, age);  
        school = s;  
    }  
    public Student(String name, String s) {  
        super(name);  
        school = s;  
    }  
    // 编译出错： no super(), 系统将调用父类无参数的构造器。  
    public Student(String s) {  
        school = s;  
    }  
}
```

北京大学出版社教材

this和super的区别

No.	区别点	this	super
1	访问属性	访问本类中的属性，如果本类没有此属性则从父类中继续查找	直接访问父类中的属性
2	调用方法	访问本类中的方法，如果本类没有此方法则从父类中继续查找	直接访问父类中的方法
3	调用构造器	调用本类构造器，必须放在构造器的首行	调用父类构造器，必须放在子类构造器的首行

多态 (父调子)

● 对象的多态 — 在Java中，子类的对象可以替代父类的对象使用

- 一个变量只能有一种确定的数据类型
- 一个引用类型变量可能指向(引用)多种不同类型的对象

`Person p = new Student();`

`Object o = new Person();` // Object类型的变量o，指向Person类型的对象

`o = new Student();` // Object类型的变量o，指向Student类型的对象

◆ 子类可看做是特殊的父类，所以父类类型的引用可以指向子类的对象：向上转型(upcasting)。

一个引用类型变量如果声明为父类的类型，但实际引用的是子类对象，那么该变量就**不能再访问子类中添加的属性和方法**

```
Student m = new Student();
```

```
m.school = "pku"; //合法, Student类有school成员变量
```

```
Person e = new Student();
```

```
e.school = "pku"; //非法, Person类没有school成员变量
```

属性是在编译时确定的，编译时e为**Person**类型，没有**school**成员变量，因而编译错误。

- 正常的方法调用

```
Person e = new Person();
```

```
e.getInfo();
```

```
Student e = new Student();
```

```
e.getInfo();
```

- 虚拟方法调用(多态情况下)

子类中定义了与父类同名同参数的方法，在多态情况下，将此时父类的方法称为虚拟方法，父类根据赋给它的不同子类对象，动态调用属于子类的该方法。这样的方法调用在编译期是无法确定的。

```
Person e = new Student();
```

```
e.getInfo(); //调用Student类的getInfo()方法
```

- 编译时类型和运行时类型

编译时e为**Person**类型，而方法的调用是在运行时确定的，所以调用的是**Student**类的**getInfo()**方法。**——动态绑定**

君子不器者唯学的故

2. 从编译和运行的角度看：

重载，是指允许存在多个同名方法，而这些方法的参数不同。编译器根据方法不同的参数表，对同名方法的名称做修饰。对于编译器而言，这些同名方法就成了不同的方法。**它们的调用地址在编译期就绑定了。**Java的重载是可以包括父类和子类的，即子类可以重载父类的同名不同参数的方法。

所以：对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为“早绑定”或“静态绑定”；

而对于多态，只有等到方法调用的那一刻，解释运行器才会确定所要调用的具体方法，这称为“晚绑定”或“动态绑定”。

引用一句Bruce Eckel的话：“不要犯傻，如果它不是晚绑定，它就不是多态。”

● 多态作用：

- 提高了代码的通用性，常称作接口重用

● 前提：

- 需要存在继承或者实现关系
- 有方法的重写

● 成员方法：

- 编译时：要查看引用变量所声明的类中是否有所调用的方法。
- 运行时：调用实际new的对象所属的类中的重写方法。

● 成员变量：

- 不具备多态性，只看引用变量所声明的类。

x为A类或A的子类， A是爸爸

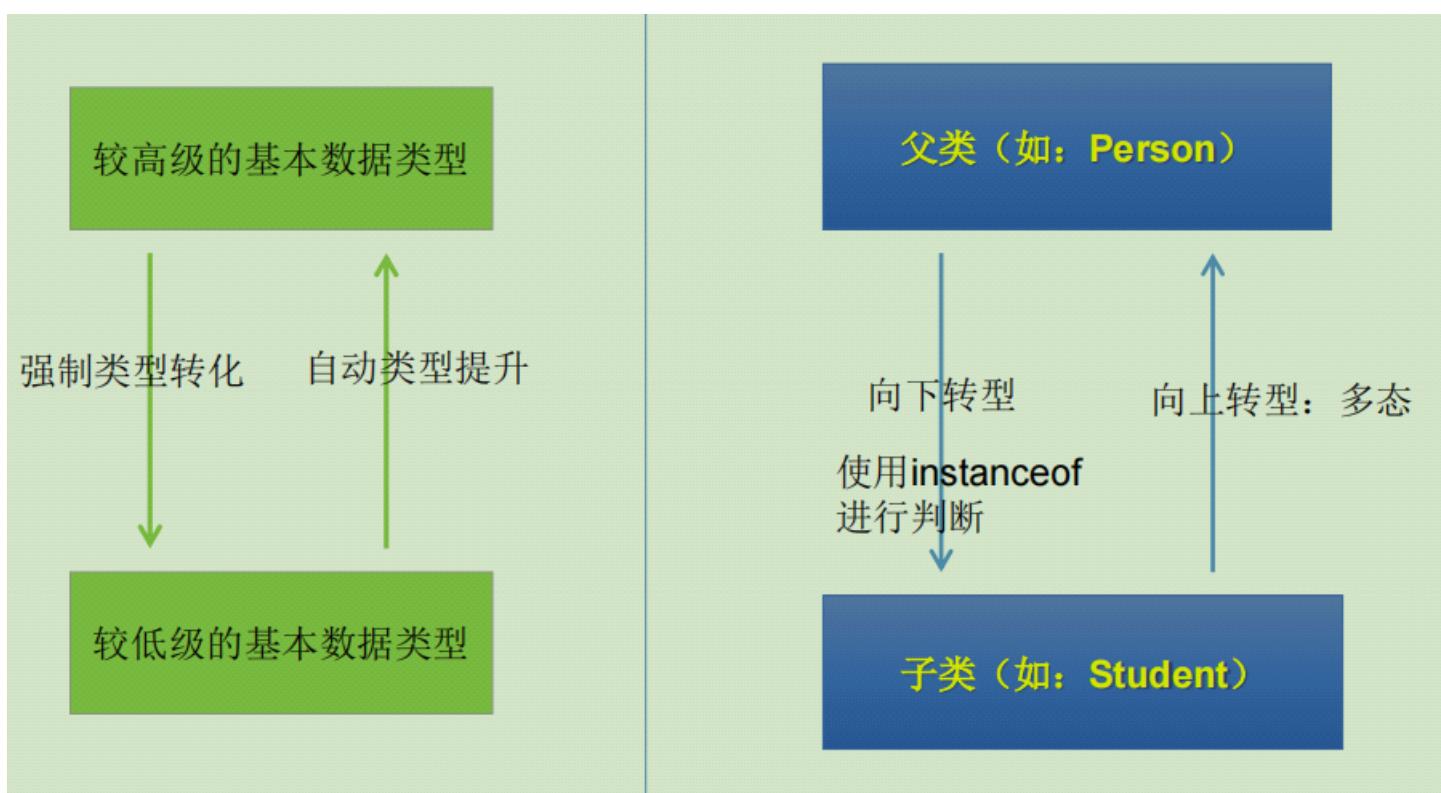
instanceof 操作符

x instanceof A: 检验x是否为类A的对象，返回值为boolean型。

- 要求x所属的类与类A必须是子类和父类的关系，否则编译错误。
- 如果x属于类A的子类B，x instanceof A值也为true。

对象类型转换举例

```
public class Test {  
    public void method(Person e) { // 设Person类中没有getschool() 方法  
        // System.out.println(e.getschool()); //非法,编译时错误  
        if (e instanceof Student) {  
            Student me = (Student) e; // 将e强制转换为Student类型  
            System.out.println(me.getschool());  
        }  
    }  
    public static void main(String[] args){  
        Test t = new Test();  
        Student m = new Student();  
        t.method(m);  
    }  
}
```



object

- Object类是所有Java类的根父类
- 如果在类的声明中未使用extends关键字指明其父类，则默认父类为java.lang.Object类

```
public class Person {  
    ...  
}
```

等价于：

```
public class Person extends Object {  
    ...  
}
```

- 例： method(Object obj){...} //可以接收任何类作为其参数
Person o=new Person();
method(o);

让天下没有难学的技术

Object类中的主要结构

NO.	方法名称	类型	描述
1	public Object()	构造	构造器
2	public boolean equals(Object obj)	普通	对象比较
3	public int hashCode()	普通	取得Hash码
4	public String toString()	普通	对象打印时调用

面试题： == 和 equals 的区别

- 从我面试的反馈，85% 的求职者“理直气壮”的回答错误...

- == 既可以比较基本类型也可以比较引用类型。对于基本类型就是比较值，对于引用类型就是比较内存地址
- equals 的话，它是属于 `java.lang.Object` 类里面的方法，如果该方法没有被重写过默认也是 ==；我们可以看到 `String` 等类的 `equals` 方法是被重写过的，而且 `String` 类在日常开发中用的比较多，久而久之，形成了 `equals` 是比较值的错误观点。
- 具体要看自定义类里有没有重写 `Object` 的 `equals` 方法来判断。
- 通常情况下，重写 `equals` 方法，会比较类中的相应属性是否都相等。

```
String str1 = new String("hello");
String str2 = new String("hello");
System.out.println("str1和str2是否相等？ "+ (str1 == str2));//false
```

```
System.out.println("str1是否equals str2？ "+(str1.equals(str2)));//true
```

toString() 方法

- `toString()` 方法在 `Object` 类中定义，其返回值是 `String` 类型，返回类名和它的引用地址。

- 在进行 `String` 与其它类型数据的连接操作时，自动调用 `toString()` 方法

```
Date now=new Date();
System.out.println("now="+now); 相当于
System.out.println("now="+now.toString());
```

- 可以根据需要在用户自定义类型中重写 `toString()` 方法
如 `String` 类重写了 `toString()` 方法，返回字符串的值。

```
s1="hello";
System.out.println(s1); //相当于 System.out.println(s1.toString());
```

- 基本类型数据转换为 `String` 类型时，调用了对应包装类的 `toString()` 方法
➤ `int a=10; System.out.println("a="+a);`

【面试题】

```
public void test() {  
    char[] arr = new char[] { 'a', 'b', 'c' };  
    System.out.println(arr); // 数据  
  
    int[] arr1 = new int[] { 1, 2, 3 };  
    System.out.println(arr1); // 地址  
  
    double[] arr2 = new double[] { 1.1, 2.2, 3.3 };  
    System.out.println(arr2);  
}
```

包装类

5.8 包装类(Wrapper)的使用



- 针对八种基本数据类型定义相应的引用类型—包装类（封装类）
- 有了类的特点，就可以调用类中的方法，Java才是真正的面向对象

基本数据类型	包装类
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

父类:Number

● 基本数据类型包装成包装类的实例 ---装箱

- 通过包装类的构造器实现:

```
int i = 500; Integer t = new Integer(i);
```

- 还可以通过字符串参数构造包装类对象:

```
Float f = new Float("4.56");
```

```
Long l = new Long("asdf"); //NumberFormatException
```

● 获得包装类对象中包装的基本类型变量 ---拆箱

- 调用包装类的.xxxValue()方法:

```
boolean b = bObj.booleanValue();
```

● JDK1.5之后，支持自动装箱，自动拆箱。但类型必须匹配。

基本
类型



包装类

包装类型的作用:

包装类是为了方便对基本数据类型进行操作,包装类可以解决一些基本类型解决不了的问题:

- 集合只能存放引用类型的数据,不能存放基本数据类型.如add(Object o);
- 基本类型和包装类型之间可以互相转换,自动装箱拆箱.
- 包装类的parse方法可以实现基本数据类型+string类型之间的相互转换
- 函数需要传递进去的参数为Object类型,传入基本数据类型就不可行.

面向对象 (三)

2021年12月19日 22:03

static

6.1 关键字: static

尚硅谷

当我们编写一个类时，其实就是在描述其对象的属性和行为，而并没有产生实质上的对象，只有通过new关键字才会产生出对象，这时系统才会分配内存空间给对象，其方法才可以供外部调用。我们有时候希望无论是否产生了对象或无论产生了多少对象的情况下，**某些特定的数据在内存空间里只有一份**，例如所有的中国人都有个国家名称，每一个中国人都共享这个国家名称，不必在每一个中国人的实例对象中都单独分配一个用于代表国家名称的变量。

```
classDiagram
    class Chinese {
        static String country = "China"
        name
        age
    }
    Chinese <|-- 实例1
    Chinese <|-- 实例2
    ...
    note over Chinese: 中国人的人数 (country="中国")
```

让天下没有难学的技术

● 使用范围：

➤ 在Java类中，可用**static**修饰**属性、方法、代码块、内部类**

● 被修饰后的成员具备以下特点：

➤ 随着类的加载而加载

➤ 优先于对象存在

➤ 修饰的成员，被所有对象所共享

➤ 访问权限允许时，可不创建对象，直接被类调用

类变量应用举例

```
class Person {  
    private int id;  
    public static int total = 0;  
    public Person() {  
        total++;  
        id = total;  
    }  
    public static void main(String args[]){  
        Person Tom=new Person();  
        Tom.id=0;  
        total=100; // 不用创建对象就可以访问静态成员  
    }  
}  
  
public class StaticDemo {  
    public static void main(String args[]) {  
        Person.total = 100; // 不用创建对象就可以访问静态成员  
        //访问方式: 类名.类属性, 类名.类方法  
        System.out.println(Person.total);  
        Person c = new Person();  
        System.out.println(c.total); //输出101  
    }  
}
```

让天下没有难学的书

类方法(class method)



- 没有对象的实例时，可以用**类名.方法名()**的形式访问由**static**修饰的类方法。
- 在**static**方法内部只能访问类的**static**修饰的属性或方法，不能访问类的非**static**的结构。

```
class Person {  
    private int id;  
    private static int total = 0;  
    public static int getTotalPerson() {  
        //id++; //非法  
        return total;}  
    public Person() {  
        total++;  
        id = total;  
    }  
}  
public class PersonTest {  
    public static void main(String[] args) {  
        System.out.println("Number of total is " + Person.getTotalPerson());  
        //没有创建对象也可以访问静态方法  
        Person p1 = new Person();  
        System.out.println("Number of total is " + Person.getTotalPerson());  
    }  
}
```

The output is:
Number of total is 0
Number of total is 1

让天下没有难学的书

- 因为不需要实例就可以访问**static**方法，因此**static**方法内部不能有**this**。（也不能有**super**？YES!）
- **static**修饰的方法不能被重写

单例 (**Singleton**) 设计模式

- **设计模式**是在大量的实践中总结和理论化之后优选的代码结构、编程风格、以及解决问题的思考方式。设计模免去我们自己再思考和摸索。就像是经典的棋谱，不同的棋局，我们用不同的棋谱。”套路”
- 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个取得其对象实例的方法。如果我们要让类在一个虚拟机中只能产生一个对象，我们首先必须将类的构造器的访问权限设置为**private**，这样，就不能用**new**操作符在类的外部产生类的对象了，但在类内部仍可以产生该类的对象。因为在类的外部开始还无法得到类的对象，只能调用该类的某个静态方法以返回类内部创建的对象，静态方法只能访问类中的静态成员变量，所以，指向类内部产生的该类对象的变量也必须定义成静态的。

调用下面的单例：

```
singleton s = singleton.getInstance()
```

```
class Singleton {  
    // 1.私有化构造器  
    private Singleton() {  
    }  
  
    // 2.内部提供一个当前类的实例  
    // 4.此实例也必须静态化  
    private static Singleton single = new Singleton();  
  
    // 3.提供公共的静态的方法，返回当前类的对象  
    public static Singleton getInstance() {  
        return single;  
    }  
}
```

◆ 单例(Singleton)设计模式-懒汉式

```
class Singleton {  
    // 1.私有化构造器  
    private Singleton() {  
    }  
  
    // 2.内部提供一个当前类的实例  
    // 4.此实例也必须静态化  
    private static Singleton single;  
    // 3.提供公共的静态的方法，返回当前类的对象  
    public static Singleton getInstance() {  
        if(single == null) {  
            single = new Singleton();  
        }  
        return single;  
    }  
}
```

懒汉式暂时还存在线程安全问题，讲到多线程时，可修复

● 单例模式的优点：

由于单例模式只生成一个实例，减少了系统性能开销，当一个对象的产生需要比较多的资源时，如读取配置、产生其他依赖对象时，则可以通过在应用启动时直接产生一个单例对象，然后永久驻留内存的方式来解决。

单例(Singleton)设计模式-应用场景



- 网站的计数器，一般也是单例模式实现，否则难以同步。
- 应用程序的日志应用，一般都使用单例模式实现，这一般是由于共享的日志文件一直处于打开状态，因为只能有一个实例去操作，否则内容不好追加。
- 数据库连接池的设计一般也是采用单例模式，因为数据库连接是一种数据库资源。
- 项目中，读取配置文件的类，一般也只有一个对象。没有必要每次使用配置文件数据，都生成一个对象去读取。
- Application 也是单例的典型应用
- Windows 的 Task Manager (任务管理器) 就是很典型的单例模式
- Windows 的 Recycle Bin (回收站) 也是典型的单例应用。在整个系统运行过程中，回收站一直维护着仅有的一一个实例。

main函数

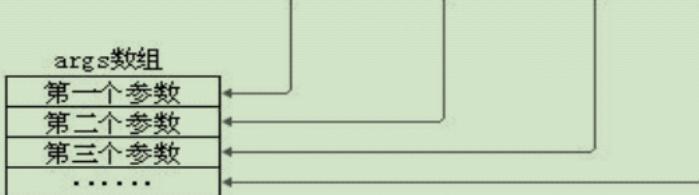
命令行参数用法举例

```
public class CommandPara {  
    public static void main(String[] args) {  
        for (int i = 0; i < args.length; i++) {  
            System.out.println("args[" + i + "] = " + args[i]);  
        }  
    }  
}
```

//运行程序CommandPara.java

java CommandPara "Tom" "Jerry" "Shkstart"

Java 运行的类名 第一个参数 第二个参数 第三个参数



输出结果:

args[0] = Tom
args[1] = Jerry
args[2] = Shkstart

计算机基础与实训教材

```
PS C:\Users\Administrator\Desktop\java\基础入门\src\demo5> java CommandPara.java a b c  
args[0] = a  
args[1] = b  
args[2] = c
```

代码块

- 代码块(或初始化块)的作用:
 - 对Java类或对象进行初始化
- 代码块(或初始化块)的分类:
 - 一个类中代码块若有修饰符，则只能被`static`修饰，称为**静态代码块**(`static block`)，没有使用`static`修饰的，为**非静态代码块**。

**● 静态代码块：用static修饰的代码块**

1. 可以有输出语句。
2. 可以对类的属性、类的声明进行初始化操作。
3. 不可以对非静态的属性初始化。即：不可以调用非静态的属性和方法。
4. 若有多个静态的代码块，那么按照从上到下的顺序依次执行。
5. 静态代码块的执行要先于非静态代码块。
6. 静态代码块随着类的加载而加载，且只执行一次。

● 非静态代码块：没有static修饰的代码块

1. 可以有输出语句。
2. 可以对类的属性、类的声明进行初始化操作。
3. 除了调用非静态的结构外，还可以调用静态的变量或方法。
4. 若有多个非静态的代码块，那么按照从上到下的顺序依次执行。
5. 每次创建对象的时候，都会执行一次。且先于构造器执行。

静态代码块随着对象的加载而运行

Final



● 在Java中声明类、变量和方法时，可使用关键字**final**来修饰，表示“最终的”。

➤ **final**标记的类不能被继承。提高安全性，提高程序的可读性。

✓ String类、System类、StringBuffer类

➤ **final**标记的方法不能被子类重写。

✓ 比如：Object类中的getClass()。

➤ **final**标记的变量(成员变量或局部变量)即称为常量。名称大写，且只能被赋值一次。

✓ **final**标记的成员变量必须在声明时或在每个构造器中或代码块中显式赋值，然后才能使用。

✓ final double MY_PI = 3.14;

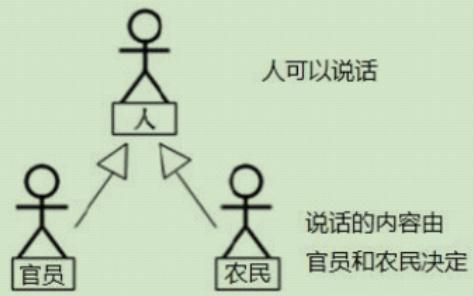
关键字final应用举例

```
public final class Test {  
    public static int totalNumber = 5;  
    public final int ID;  
  
    public Test() {  
        ID = ++totalNumber; // 可在构造器中给final修饰的“变量”赋值  
    }  
    public static void main(String[] args) {  
        Test t = new Test();  
        System.out.println(t.ID);  
        final int I = 10;  
        final int J;  
        J = 20;  
        J = 30; // 非法  
    }  
}
```

2.2.1 深入浅出Java

抽象类

随着继承层次中一个个新子类的定义，类变得越来越具体，而父类则更一般，更通用。类的设计应该保证父类和子类能够共享特征。有时将一个父类设计得非常抽象，以至于它没有具体的实例，这样的类叫做**抽象类**。



- 用 **abstract** 关键字来修饰一个类，这个类叫做**抽象类**。
- 用 **abstract** 来修饰一个方法，该方法叫做**抽象方法**。
 - 抽象方法：只有方法的声明，没有方法的实现。以分号结束：
 - 比如：`public abstract void talk();`
- 含有抽象方法的类必须被声明为抽象类。
- 抽象类不能被实例化。抽象类是用来被继承的，抽象类的子类必须重写父类的抽象方法，并提供方法体。若没有重写全部的抽象方法，仍为抽象类。
- 不能用 **abstract** 修饰变量、代码块、构造器；
- 不能用 **abstract** 修饰私有方法、静态方法、**final** 的方法、**final** 的类。

抽象类应用

● 解决方案

Java 允许类设计者指定：超类声明一个方法但不提供实现，该方法的实现由子类提供。
这样的方法称为**抽象方法**。有一个或更多抽象方法的类称为**抽象类**。

● **Vehicle** 是一个抽象类，有两个抽象方法。

```
public abstract class Vehicle{
    public abstract double calcFuelEfficiency(); //计算燃料效率的抽象方法
    public abstract double calcTripDistance(); //计算行驶距离的抽象方法
}

public class Truck extends Vehicle{
    public double calcFuelEfficiency() { //写出计算卡车的燃料效率的具体方法 }
    public double calcTripDistance() { //写出计算卡车行驶距离的具体方法 }
}

public class RiverBarge extends Vehicle{
    public double calcFuelEfficiency() { //写出计算驳船的燃料效率的具体方法 }
    public double calcTripDistance() { //写出计算驳船行驶距离的具体方法 }
}
```

注意：抽象类不能实例化 `new Vihicle()` 是非法的

让天下没有难学的技

多态的应用：模板方法设计模式(**TemplateMethod**)

抽象类体现的就是一种模板模式的设计，抽象类作为多个子类的通用模板，子类在抽象类的基础上进行扩展、改造，但子类总体上会保留抽象类的行为方式。

解决的问题：

- 当功能内部一部分实现是确定的，一部分实现是不确定的。这时可以把不确定的部分暴露出去，让子类去实现。
- 换句话说，在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。这就是一种模板模式。

接口 (和class是并列关系)

- 一方面，有时必须从几个类中派生出一个子类，继承它们所有的属性和方法。但是，Java不支持多重继承。有了接口，就可以得到多重继承的效果
- 另一方面，有时必须从几个类中抽取出一些共同的行为特征，而它们之间又没有is-a的关系，仅仅是具有相同的行为特征而已。例如：鼠标、键盘、打印机、扫描仪、摄像头、充电器、MP3机、手机、数码相机、移动硬盘等都支持USB连接。
- 接口就是规范，定义的是一组规则，体现了现实世界中“如果你是/要...则必须能...”的思想。**继承是一个"是不是"的关系，而接口实现则是"能不能"的关系。**
- **接口的本质是契约，标准，规范，就像我们的法律一样。制定好后大家都遵守。**

- 接口(interface)是抽象方法和常量值定义的集合。

- 接口的特点：

- 用interface来定义。
- 接口中的所有成员变量都默认是由public static final修饰的。
- 接口中的所有抽象方法都默认是由public abstract修饰的。
- 接口中没有构造器。
- 接口采用多继承机制。

- 接口定义举例

```
public interface Runner {
    int ID = 1;
    void start();
    public void run();
    void stop();
}
```

```
public interface Runner {
    public static final int ID = 1;
    public abstract void start();
    public abstract void run();
    public abstract void stop();
}
```



- 定义Java类的语法格式：先写extends，后写implements

➤ class SubClass extends SuperClass implements InterfaceA{ }

- 一个类可以实现多个接口，接口也可以继承其它接口。

- 实现接口的类中必须提供接口中所有方法的具体实现内容，方可实例化。否则，仍为抽象类。

- 接口的主要用途就是被实现类实现。（面向接口编程）

- 与继承关系类似，接口与实现类之间存在多态性

- 接口和类是并列关系，或者可以理解为一种特殊的类。从本质上讲，接口是一种特殊的抽象类，这种抽象类中只包含常量和方法的定义(JDK7.0及之前)，而没有变量和方法的实现。

- 一个类可以实现多个无关的接口

```
interface Runner { public void run();}
interface Swimmer {public double swim();}
class Creator{public int eat(){...}}
class Man extends Creator implements Runner,Swimmer{
    public void run() {...}
    public double swim() {...}
    public int eat() {...}
}
```

儿子可以代替爸爸

●与继承关系类似，接口与实现类之间存在多态性

```
public class Test{  
    public static void main(String args[]){  
        Test t = new Test();  
        Man m = new Man();  
        t.m1(m);  
        t.m2(m);  
        t.m3(m);  
    }  
    public String m1(Runner f) { f.run(); }  
    public void m2(Swimmer s) {s.swim();}  
    public void m3(Creator a) {a.eat();}  
}
```

接口中的默认方法

- 若一个接口中定义了一个默认方法，而另外一个接口中也定义了一个同名同参数的方法（不管此方法是否是默认方法），在实现类同时实现了这两个接口时，会出现：**接口冲突**。
 - 解决办法：实现类必须覆盖接口中同名同参数的方法，来解决冲突。
- 若一个接口中定义了一个默认方法，而父类中也定义了一个同名同参数的非抽象方法，则不会出现冲突问题。因为此时遵守：**类优先原则**。接口中具有相同名称和参数的默认方法会被忽略。

● 局部内部类的特点

- 内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和\$符号，以及数字编号。
- 只能在声明它的方法或代码块中使用，而且是先声明后使用。除此之外的任何地方都不能使用该类。
- 局部内部类可以使用外部类的成员，包括私有的。
- 局部内部类可以使用外部方法的局部变量，但是必须是final的。由局部内部类和局部变量的声明周期不同所致。
- 局部内部类和局部变量地位类似，不能使用public,protected,缺省,private
- 局部内部类不能使用static修饰，因此也不能包含静态成员

异常处理

2021年12月25日 21:19

常见异常

异常	说明
Exception	异常层次结构的根类
RuntimeException	许多 java.lang 异常的基类
ArithmecticException	算术错误情形，如以零作除数
IllegalArgumentExeption	方法接收到非法参数
ArrayIndexOutOfBoundsException	数组大小小于或大于实际的数组大小
NullPointerException	尝试访问 null 对象成员
ClassNotFoundException	不能加载所需的类
NumberFormatException	数字转化格式异常，比如字符串到 float 型 数字的转换无效
IOException	I/O 异常的根类
FileNotFoundException	找不到文件
EOFException	文件结束
InterruptedException	线程中断

- Java 程序在执行过程中所发生的异常事件可分为两类：

➤ **Error:** Java 虚拟机无法解决的严重问题。如：JVM 系统内部错误、资源耗尽等严重情况。比如：**StackOverflowError** 和 **OOM**。一般不编写针对性的代码进行处理。

➤ **Exception:** 其它因编程错误或偶然的外在因素导致的一般性问题，可以使用针对性的代码进行处理。例如：

- ✓ 空指针访问
- ✓ 试图读取不存在的文件
- ✓ 网络连接中断
- ✓ 数组角标越界

- 对于这些错误，一般有两种**解决方法**：一是遇到错误就终止程序的运行。另一种方法是由程序员在编写程序时，就考虑到错误的检测、错误消息的提示，以及错误的处理。
- 捕获错误最理想的是在**编译期间**，但有的错误只有在**运行时**才会发生。
比如：**除数为0，数组下标越界等**
 - 分类：**编译时异常和运行时异常**

Java异常处理的方式：

方式一：try-catch-finally

方式二：throws + 异常类型

- Java提供的是异常处理的**抓抛模型**。
- Java程序的执行过程中如出现异常，会生成一个**异常类对象**，该异常对象将被提交给Java运行时系统，这个过程称为**抛出 (throw) 异常**。
- 异常对象的生成
 - 由虚拟机**自动生成**：程序运行过程中，虚拟机检测到程序发生了问题，如果在当前代码中没有找到相应的处理程序，就会在后台自动创建一个对应异常类的实例对象并抛出——自动抛出
 - 由开发人员**手动创建**：`Exception exception = new ClassCastException();`——创建好的异常对象不抛出对程序没有任何影响，和创建一个普通对象一样

异常的抛出机制

```
↑ (4 main方法将异常抛至OS并终止程序)  
main(...)  
↓ ↳ (3 methodA再将异常抛至 main )  
methodA()  
↓ ↳ (2 methodB再将异常抛至 methodA )  
methodB()  
↓ ↳ (1 methodC的异常被抛至 methodB )  
methodC()
```

为保证程序正常执行，代码必须对可能出现的异常进行处理。

try-catch-finally

异常处理是通过try-catch-finally语句实现的。

```
try{  
    ..... //可能产生异常的代码  
}  
catch( ExceptionName1 e ){  
    ..... //当产生ExceptionName1型异常时的处置措施  
}  
catch( ExceptionName2 e ){  
    ..... //当产生ExceptionName2型异常时的处置措施  
}  
[ finally{  
    ..... //无论是否发生异常，都无条件执行的语句  
} ]
```

●finally

- 捕获异常的最后一步是通过**finally**语句为异常处理提供一个统一的出口，使得在控制流转到程序的其它部分以前，能够对程序的状态作统一的管理。
- 不论在**try**代码块中是否发生了异常事件，**catch**语句是否执行，**catch**语句是否有异常，**catch**语句中是否有**return**，**finally**块中的语句都会被执行。
- finally**语句和**catch**语句是任选的

不捕获异常时的情况

- 前面使用的异常都是**RuntimeException**类或是它的子类，这些类的异常的特点是：即使没有使用**try**和**catch**捕获，Java自己也能捕获，并且编译通过（但运行时会发生异常使得程序运行终止）。
- 如果抛出的异常是**IOException**等类型的非运行时异常，则必须捕获，否则编译错误。也就是说，我们必须处理编译时异常，将异常进行捕捉，转化为运行时异常

throws

●声明抛出异常是Java中处理异常的第二种方式

- 如果一个方法(中的语句执行时)可能生成某种异常，但是并不能确定如何处理这种异常，则此方法应**显示地**声明抛出异常，表明该方法将不对这些异常进行处理，而由该方法的**调用者**负责处理。
- 在方法声明中用**throws**语句可以声明抛出异常的列表，**throws**后面的异常类型可以是方法中产生的异常类型，也可以是它的父类。

●声明抛出异常举例：

```
public void readFile(String file) throws FileNotFoundException {  
    .....  
    // 读文件的操作可能产生FileNotFoundException类型的异常  
    FileInputStream fis = new FileInputStream(file);  
    .....  
}
```

人工定义异常

- 一般地，用户自定义异常类都是**RuntimeException**的子类。
- 自定义异常类通常需要编写几个重载的构造器。
- 自定义异常需要提供**serialVersionUID**
- 自定义的异常通过**throw**抛出。
- 自定义异常最重要的是异常类的名字，当异常出现时，可以根据名字判断异常类型。

用户自定义异常类**MyException**，用于描述数据取值范围错误信息。用户自己的异常类**必须继承**现有的异常类。

```
class MyException extends Exception {  
    static final long serialVersionUID = 13465653435L;  
    private int idnumber;  
  
    public MyException(String message, int id) {  
        super(message);  
        this.idnumber = id;  
    }  
  
    public int getId() {  
        return idnumber;  
    }  
}
```

多线程

2022年1月8日 17:03

- **程序(program)**是为完成特定任务、用某种语言编写的一组指令的集合。即指一段静态的代码，静态对象。
- **进程(process)**是程序的一次执行过程，或是正在运行的一个程序。是一个动态的过程：有它自身的产生、存在和消亡的过程。——生命周期
 - 如：运行中的QQ，运行中的MP3播放器
 - 程序是静态的，进程是动态的
 - 进程作为资源分配的单位，系统在运行时会为每个进程分配不同的内存区域
- **线程(thread)**，进程可进一步细化为线程，是一个程序内部的一条执行路径。
 - 若一个进程同一时间并行执行多个线程，就是支持多线程的
 - 线程作为调度和执行的单位，每个线程拥有独立的运行栈和程序计数器(pc)，线程切换的开销小
 - 一个进程中的多个线程共享相同的内存单元/内存地址空间→它们从同一堆中分配对象，可以访问相同的变量和对象。这就使得线程间通信更简便、高效。但多个线程操作共享的系统资源可能就会带来安全的隐患。

● 并行与并发

- 并行：多个CPU同时执行多个任务。比如：多人同时做不同的事。
- 并发：一个CPU(采用时间片)同时执行多个任务。比如：秒杀、多人做同一件事。

你吃饭吃到一半，电话来了，你一直到吃完了以后才去接，这就说明你不支持并发也不支持并行。

你吃饭吃到一半，电话来了，你停了下来接了电话，接完后继续吃饭，这说明你支持并发。

你吃饭吃到一半，电话来了，你一边打电话一边吃饭，这说明你支持并行。

并发的关键是你有处理多个任务的能力，不一定要同时。

并行的关键是你有同时处理多个任务的能力。

Thread类

● Thread类的特性

- 每个线程都是通过某个特定Thread对象的run()方法来完成操作的，经常把run()方法的主体称为线程体
- 通过该Thread对象的start()方法来启动这个线程，而非直接调用run()

● 构造器

- **Thread():** 创建新的Thread对象
- **Thread(String threadname):** 创建线程并指定线程实例名
- **Thread(Runnable target):** 指定创建线程的目标对象，它实现了Runnable接口中的run方法
- **Thread(Runnable target, String name):** 创建新的Thread对象

API中创建线程的两种方式

● 方式一：继承Thread类

- 1) 定义子类继承Thread类。
- 2) 子类中重写Thread类中的run方法。
- 3) 创建Thread子类对象，即创建了线程对象。
- 4) 调用线程对象start方法：启动线程，调用run方法。

```
class MyThread extends Thread{  
    public MyThread(){  
        super();  
    }  
    public void run(){  
        for(int i = 0;i<100;i++){  
            System.out.println("子线程：" + i);  
        }  
    }  
}  
public class TestThread {  
    public static void main(String[] args) {  
        //1. 创建线程  
        MyThread mt = new MyThread();  
        //2. 启动线程，并调用当前线程的run()方法。  
        mt.start();  
    }  
}
```

● 注意点：

1. 如果自己手动调用run()方法，那么就只是普通方法，没有启动多线程模式。
2. run()方法由JVM调用，什么时候调用，执行的过程控制都有操作系统的CPU调度决定。
3. 想要启动多线程，必须调用start方法。
4. 一个线程对象只能调用一次start()方法启动，如果重复调用了，则将抛出以上的异常“IllegalThreadStateException”。

● 方式二：实现Runnable接口

- 1) 定义子类，实现Runnable接口。
- 2) 子类中重写Runnable接口中的run方法。
- 3) 通过Thread类含参构造器创建线程对象。
- 4) 将Runnable接口的子类对象作为实际参数传递给Thread类的构造器中。
- 5) 调用Thread类的start方法：开启线程，调用Runnable子类接口的run方法。

由于Java的单继承局限，所以我们推荐使用Runnable接口实现多线程。在Java里面专门提供了Runnable接口，此接口定义如下：

```
1 | @FunctionalInterface  
2 | public interface Runnable{  
3 |     public void run();  
4 | }
```

这是一个函数式接口，规定里面只有一个run方法

那么只需要让一个类实现Runnable接口即可，并且也需要覆写run方法。

与继承Thread类相比，MyThread类在结构上没有区别，除了一点，如果继承了Thread类，同时也直接继承了start方法()，但是如果实现的是我们Runnable接口，并没有start方法可以调用。

不管何种情况下，要想启动多线程，一定依靠Thread类完成，Thread类定义有以下构造方法。

```
1 | public Thread(Runnable target)
```

接收的是Runnable接口对象，自然可以接收Runnable子类。

```
class MyThread implements Runnable{  
    private String name;  
    public MyThread(String name){  
        this.name=name;  
    }  
    @Override  
    public void run() {  
        for (int i = 0; i < 200; i++) {  
            System.out.println(this.name+"==>"+i);  
        }  
    }  
}  
public class TestDemo{  
  
    public static void main(String[] args) {  
        MyThread thread1=new MyThread("ThreadA");  
        MyThread thread2=new MyThread("ThreadB");  
        MyThread thread3=new MyThread("ThreadC");  
  
        new Thread(thread1).start();  
        new Thread(thread2).start();  
        new Thread(thread3).start();  
    }  
}
```

```
new Thread(thread1).start();
new Thread(thread2).start();
new Thread(thread3).start();
```

线程的优先级

- 线程的优先级等级

- MAX_PRIORITY: 10
- MIN_PRIORITY: 1
- NORM_PRIORITY: 5

- 涉及的方法

- **getPriority()** : 返回线程优先值
- **setPriority(int newPriority)** : 改变线程的优先级

- 说明

- 线程创建时继承父线程的优先级
- 低优先级只是获得调度的概率低，并非一定是在高优先级线程之后才被调用

● JDK中用**Thread.State**类定义了线程的几种状态

要想实现多线程，必须在主线程中创建新的线程对象。Java语言使用**Thread**类及其子类的对象来表示线程，在它的一个完整的生命周期中通常要经历如下的五种状态：

- **新建**: 当一个**Thread**类或其子类的对象被声明并创建时，新生的线程对象处于新建状态
- **就绪**: 处于新建状态的线程被**start()**后，将进入线程队列等待CPU时间片，此时它已具备了运行的条件，只是没分配到CPU资源
- **运行**: 当就绪的线程被调度并获得CPU资源时，便进入运行状态，**run()**方法定义了线程的操作和功能
- **阻塞**: 在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时中止自己的执行，进入阻塞状态
- **死亡**: 线程完成了它的全部工作或线程被提前强制性地中止或出现异常导致结束

多线程的安全问题

1. 多线程出现了安全问题

2. 问题的原因：

当多条语句在操作同一个线程共享数据时，一个线程对多条语句只执行了一部分，还没有执行完，另一个线程参与进来执行。导致共享数据的错误。

3. 解决办法：

对多条操作共享数据的语句，只能让一个线程都执行完，在执行过程中，其他线程不可以参与执行。

锁的使用

Synchronized的使用方法

- Java对于多线程的安全问题提供了专业的解决方式：同步机制

1. 同步代码块：

```
synchronized (对象){  
    // 需要被同步的代码;  
}
```

2. synchronized还可以放在方法声明中，表示整个方法为同步方法。

例如：

```
public synchronized void show (String name){  
    ....  
}
```

不会释放锁的操作

- 线程执行同步代码块或同步方法时，程序调用Thread.sleep()、Thread.yield()方法暂停当前线程的执行

- 线程执行同步代码块时，其他线程调用了该线程的suspend()方法将该线程挂起，该线程不会释放锁（同步监视器）。

应尽量避免使用suspend()和resume()来控制线程

或者直接手动lock

3. Lock(锁)

```
class A{  
    private final ReentrantLock lock = new ReentrantLock();  
    public void m(){  
        lock.lock();  
        try{  
            //保证线程安全的代码;  
        }  
        finally{  
            lock.unlock();  
        }  
    }  
}
```

synchronized 与 Lock 的对比

1. Lock是显式锁（手动开启和关闭锁，别忘记关闭锁），synchronized是隐式锁，出了作用域自动释放
2. Lock只有代码块锁，synchronized有代码块锁和方法锁
3. 使用Lock锁，JVM将花费较少的时间来调度线程，性能更好。并且具有更好的扩展性（提供更多的子类）

优先使用顺序：

Lock → 同步代码块（已经进入了方法体，分配了相应资源）→ 同步方法
(在方法体之外)

关键在于有没有**共享数据**

线程的通信

● wait() 与 notify() 和 notifyAll()

- **wait()**: 令当前线程挂起并放弃CPU、同步资源并等待，使别的线程可访问并修改共享资源，而当前线程排队等候其他线程调用**notify()**或**notifyAll()**方法唤醒，唤醒后等待重新获得对监视器的所有权后才能继续执行。
- **notify()**: 唤醒正在排队等待同步资源的线程中优先级最高者结束等待
- **notifyAll ()**: 唤醒正在排队等待资源的所有线程结束等待.

- 这三个方法只有在**synchronized**方法或**synchronized**代码块中才能使用，否则会报 **java.lang.IllegalMonitorStateException** 异常。
- 因为这三个方法必须有锁对象调用，而任意对象都可以作为**synchronized**的同步锁，因此这三个方法只能在**Object**类中声明。

wait()会自动解锁

wait() 方法

- 在当前线程中调用方法： 对象名.wait()
- 使当前线程进入等待（某对象）状态， 直到另一线程对该对象发出 notify (或notifyAll) 为止。
- 调用方法的必要条件： 当前线程必须具有对该对象的监控权（加锁）
- 调用此方法后， 当前线程将释放对象监控权， 然后进入等待
- 在当前线程被notify后， 要重新获得监控权， 然后从断点处继续代码的执行。

JDK5.0新增线程创建方式

实现Callable接口

- 与使用Runnable相比， Callable功能更强大些
 - 相比run()方法， 可以有返回值
 - 方法可以抛出异常
 - 支持泛型的返回值
 - 需要借助FutureTask类， 比如获取返回结果

● Future接口

- 可以对具体Runnable、 Callable任务的执行结果进行取消、 查询是否完成、 获取结果等。
- FutureTask是Future接口的唯一的实现类
- FutureTask 同时实现了Runnable, Future接口。它既可以作为Runnable被线程执行， 又可以作为Future得到Callable的返回值

eg

```
class NumThread implements Callable{  
    @Override  
    public Object call() throws Exception {  
        int sum = 0;  
        for (int i = 1; i <= 100; i++) {  
            if(i % 2 == 0){  
                System.out.println(i);  
                sum += i;  
            }  
        }  
        return sum;  
    }  
}
```

```
public class ThreadNew {  
    public static void main(String[] args) {  
        NumThread numThread = new NumThread();  
  
        FutureTask futureTask = new FutureTask(numThread);  
  
        new Thread(futureTask).start();  
  
        try {  
            //get()返回值即为FutureTask构造器参数Callable实现类重写的call()的返回值。  
            Object sum = futureTask.get();  
            System.out.println("总和为: " + sum);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        } catch (ExecutionException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

原来是 thread 里面装 runnable

现在是 thread 里面装 futuretask , futuretask 里面装 callable

使用线程池

- **背景:** 经常创建和销毁、使用量特别大的资源，比如并发情况下的线程，对性能影响很大。
- **思路:** 提前创建好多个线程，放入线程池中，使用时直接获取，使用完放回池中。可以避免频繁创建销毁、实现重复利用。类似生活中的公共交通工具。
- **好处:**
 - 提高响应速度（减少了创建新线程的时间）
 - 降低资源消耗（重复利用线程池中线程，不需要每次都创建）
 - 便于线程管理
 - ✓ corePoolSize: 核心池的大小
 - ✓ maximumPoolSize: 最大线程数
 - ✓ keepAliveTime: 线程没有任务时最多保持多长时间后会终止
 - ✓ ...

知识点下课有详细的讲解

线程池

```

public class ThreadPool {

    public static void main(String[] args) {
        //1. 提供指定线程数量的线程池
        ExecutorService service = Executors.newFixedThreadPool( nThreads: 10 );
        ThreadPoolExecutor service1 = (ThreadPoolExecutor) service;
        //设置线程池的属性
        System.out.println(service.getClass());
        service1.setCorePoolSize(15);
        service1.setKeepAliveTime();| I

        //2. 执行指定的线程的操作。需要提供实现Runnable接口或Callable接口实现类的对象
        service.execute(new NumberThread());//适合适用于Runnable
        service.execute(new NumberThread1());//适合适用于Runnable

        //3. 提交任务
        service.submit(Callable callable); //适合使用于Callable
        //4. 关闭连接池
        service.shutdown();
    }
}

```

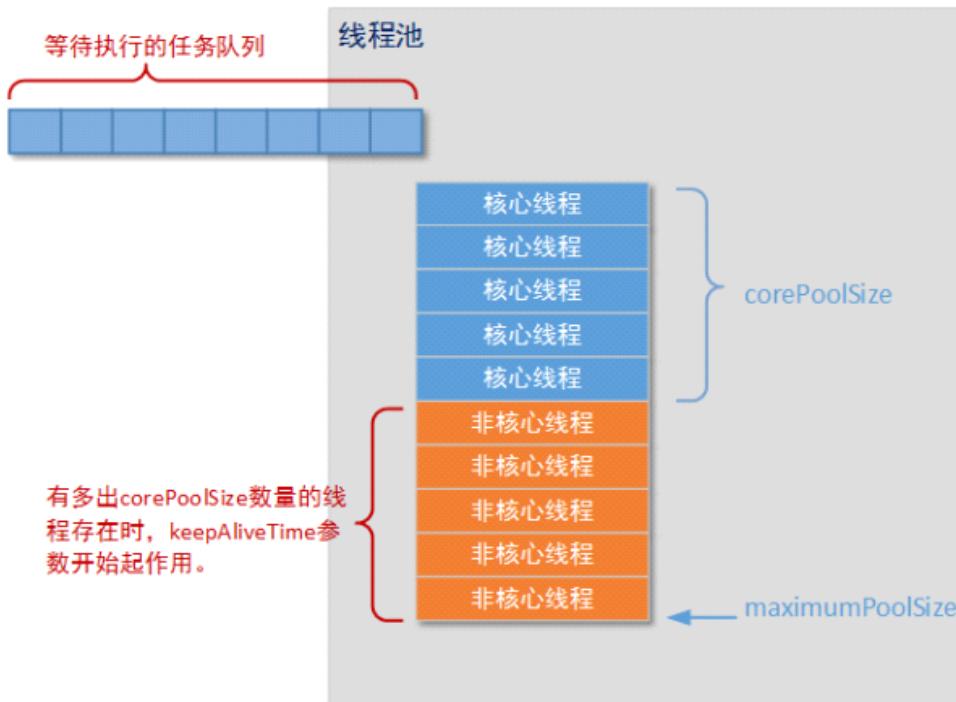
- JDK 5.0起提供了线程池相关API: **ExecutorService** 和 **Executors**
- **ExecutorService**: 真正的线程池接口, 常见子类**ThreadPoolExecutor**
 - `void execute(Runnable command)` : 执行任务/命令, 没有返回值, 一般用来执行 Runnable
 - `<T> Future<T> submit(Callable<T> task)`: 执行任务, 有返回值, 一般又来执行 Callable
 - `void shutdown()` : 关闭连接池
- **Executors**: 工具类、线程池的工厂类, 用于创建并返回不同类型的线程池
 - `Executors.newCachedThreadPool()`: 创建一个可根据需要创建新线程的线程池
 - `Executors.newFixedThreadPool(n)`: 创建一个可重用固定线程数的线程池
 - `Executors.newSingleThreadExecutor()` : 创建一个只有一个线程的线程池
 - `Executors.newScheduledThreadPool(n)`: 创建一个线程池, 它可安排在给定延迟后运行命令或者定期地执行。

让天下没有难学的技术

深入线程池

而我们创建时, 一般使用它的子类: `ThreadPoolExecutor`.

```
1. public ThreadPoolExecutor(int corePoolSize,
2.                             int maximumPoolSize,
3.                             long keepAliveTime,
4.                             TimeUnit unit,
5.                             BlockingQueue<Runnable> workQueue,
6.                             ThreadFactory threadFactory,
7.                             RejectedExecutionHandler handler)
```



Executor是一个顶层接口，在它里面只声明了一个方法execute(Runnable)，返回值为void，参数为Runnable类型，从字面意思可以理解，就是用来执行传进去的任务的；

然后ExecutorService接口继承了Executor接口，并声明了一些方法：submit、invokeAll、invokeAny以及shutDown等；

抽象类AbstractExecutorService实现了ExecutorService接口，基本实现了ExecutorService中声明的所有方法；

然后ThreadPoolExecutor继承了类AbstractExecutorService。

在ThreadPoolExecutor类中有几个非常重要的方法：

```

1 | execute()
2 | submit()
3 | shutdown()
4 | shutdownNow()

```

构造器

在ThreadPoolExecutor类中提供了四个构造方法：

```
1 public class ThreadPoolExecutor extends AbstractExecutorService {  
2     ....  
3     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  
4         BlockingQueue<Runnable> workQueue);  
5  
6     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  
7         BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);  
8  
9     public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  
10        BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);  
11  
12    public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,  
13        BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);  
14    ...  
15 }
```

下面解释下一下构造器中各个参数的含义：

- corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；
- maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；
- keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；
- unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS;	//天
TimeUnit.HOURS;	//小时
TimeUnit.MINUTES;	//分钟
TimeUnit.SECONDS;	//秒
TimeUnit.MILLISECONDS;	//毫秒
TimeUnit.MICROSECONDS;	//微妙
TimeUnit.NANOSECONDS;	//纳秒

- workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，一般来说，这里的阻塞队列有以下几种选择：

ArrayBlockingQueue;
LinkedBlockingQueue;
SynchronousQueue;

ArrayBlockingQueue和PriorityBlockingQueue使用较少，一般使用LinkedBlockingQueue和Synchronous。线程池的排队策略与BlockingQueue有关。

- threadFactory：线程工厂，主要用来创建线程；
- handler：表示当拒绝处理任务时的策略，有以下四种取值：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。
ThreadPoolExecutor.DiscardPolicy: 也是丢弃任务，但是不抛出异常。
ThreadPoolExecutor.DiscardOldestPolicy: 丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）
ThreadPoolExecutor.CallerRunsPolicy: 由调用线程处理该任务

反射

2022年1月25日 17:46

反射小结:

- 我对反射的理解就是通过Class访问class。
- 访问class中的变量使用 Field。
- 访问class中的构造函数使用Constructor。
- 访问class中的方法使用 Method。(方法注解使用setAccessible)

反射相关的主要API

- **java.lang.Class**:代表一个类
- **java.lang.reflect.Method**:代表类的方法
- **java.lang.reflect.Field**:代表类的成员变量
- **java.lang.reflect.Constructor**:代表类的构造器

Class类的常用方法

方法名	功能说明
static Class forName(String name)	返回指定类名 name 的 Class 对象
Object newInstance()	调用缺省构造函数，返回该Class对象的一个实例
getName()	返回此Class对象所表示的实体（类、接口、数组类、基本类型或void）名称
Class getSuperClass()	返回当前Class对象的父类的Class对象
Class [] getInterfaces()	获取当前Class对象的接口
ClassLoader getClassLoader()	返回该类的类加载器
Class getSuperclass()	返回表示此Class所表示的实体的超类的Class
Constructor[] getConstructors()	返回一个包含某些Constructor对象的数组
Field[] getDeclaredFields()	返回Field对象的一个数组
Method getMethod(String name, Class<?>... parameterTypes)	返回一个Method对象，此对象的形参类型为paramType

Method Class.getMethod(String name, Class<?>... parameterTypes)的作用是获得对象所声明的公开方法

该方法的第一个参数name是要获得方法的名字，第二个参数parameterTypes是按声明顺序标识该方法形参类型。

```
person.getClass().getMethod("Speak", null);
```

//获得person对象的Speak方法，因为Speak方法没有形参，所以parameterTypes为null

```
person.getClass().getMethod("run", String.class);
```

//获得person对象的run方法，因为run方法的形参是String类型的，所以parameterTypes为String.class

如果对象内的方法的形参是int类型的，则parameterTypes是int.class

Field主要方法

getName()	获取字段名称
getType()	获取字段类型。例：class java.lang.String , int
setAccessible(true)	设置属性操作。为属性赋值前需要设置可操作，破坏了封装性 可以为私有字段赋值。
set(obj,obj)	为属性设置值，第一个参数为需要设置值的对象实体，第二个为参数
get(obj)	获取属性值

反射的应用举例

```
• String str = "test4.Person";
• Class clazz = Class.forName(str);
• Object obj = clazz.newInstance();
• Field field = clazz.getField("name");
• field.set(obj, "Peter");
• Object name = field.get(obj);
• System.out.println(name);
```

注： test4.Person是test4包下的Person类



Object invoke(Object obj, Object ... args)

说明：

1. Object 对应原方法的返回值，若原方法无返回值，此时返回null
2. 若原方法若为静态方法，此时形参Object obj可为null
3. 若原方法形参列表为空，则Object[] args为null
4. 若原方法声明为private，则需要在调用此invoke()方法前，显式调用方法对象的setAccessible(true)方法，将可访问private的方法。

eg:

```

Class clazz = Person.class; clazz: "class fanshe.java.Person"
//1.通过反射，创建Person类的对象
Constructor cons = clazz.getConstructor(String.class,int.class); cons: "public fanshe.java.Person(java.lang.String,int)"
Object obj = cons.newInstance( ...initargs: "Tom", 12); cons: "public fanshe.java.Person(java.lang.String,int)" obj: "Person{name='Tom', age=10}"
Person p = (Person) obj; obj: "Person{name='Tom', age=10}" p: "Person{name='Tom', age=10}"
System.out.println(p.toString());
//2.通过反射，调用对象指定的属性、方法
//调用属性
Field age = clazz.getDeclaredField( name: "age"); age: "public int fanshe.java.Person.age"
age.set(p,10); age: "public int fanshe.java.Person.age"
System.out.println(p.toString());

//调用方法
Method show = clazz.getDeclaredMethod( name: "show"); show: "public void fanshe.java.Person.show()"
show.invoke(p); p: "Person{name='Tom', age=10}" show: "public void fanshe.java.Person.show()"

System.out.println("*****");
//通过反射，可以调用Person类的私有结构的。比如：私有的构造器、方法、属性
//调用私有的构造器
Constructor cons1 = clazz.getDeclaredConstructor(String.class); cons1: "private fanshe.java.Person(java.lang.String)"
cons1.setAccessible(true);
Person p1 = (Person) cons1.newInstance( ...initargs: "Jerry"); cons1: "private fanshe.java.Person(java.lang.String)" p1: "Person{name='HanMeimei', age=0}"
System.out.println(p1);

//调用私有的属性
Field name = clazz.getDeclaredField( name: "name"); name: "private java.lang.String fanshe.java.Person.name"
name.setAccessible(true);
name.set(p1,"HanMeimei"); name: "private java.lang.String fanshe.java.Person.name"
System.out.println(p1);

//调用私有的方法
Method showNation = clazz.getDeclaredMethod( name: "showNation", String.class); clazz: "class fanshe.java.Person" showNation: "private java.lang.String fanshe.java.Person.showNation(java.lang.String)"
showNation.setAccessible(true);
String nation = (String) showNation.invoke(p1, ...args: "中国");//相当于String nation = p1.showNation("中国") p1: "Person{name='HanMeimei', age=0}"
System.out.println(nation); nation: "中国"

```

```

> clazz = (Class@1115) "class fanshe.java.Person" ...
> cons = (Constructor@1119) "public fanshe.java.Person(java.lang.String,int)"
> obj = (Person@1198) "Person{name='Tom', age=10}"
> p = (Person@1198) "Person{name='Tom', age=10}"
> age = (Field@1225) "public int fanshe.java.Person.age"
> show = (Method@1233) "public void fanshe.java.Person.show()"
> cons1 = (Constructor@1251) "private fanshe.java.Person(java.lang.String)"
> p1 = (Person@1260) "Person{name='HanMeimei', age=0}"
> name = (Field@1265) "private java.lang.String fanshe.java.Person.name"
> showNation = (Method@1275) "private java.lang.String fanshe.java.Person.showNation(java.lang.String)"
> nation = "中国"

```

html初步

2022年1月9日 14:55

Html 的代码注释 <!-- 这是 html 注释，可以在页面右键查看源代码中看到 -->

7、HTML 标签介绍

1.标签的格式:

<标签名>封装的数据</标签名>

2.标签名大小写不敏感。

3.标签拥有自己的属性。

i. 分为基本属性: `bgcolor="red"`

可以修改简单的样式效果

ii. 事件属性: `onclick="alert('你好！');"`

可以直接设置事件响应后的代码。

4.标签又分为, 单标签和双标签。

i. 单标签格式: <标签名 />

`br` 换行 `hr` 水平线

ii. 双标签格式: <标签名> ...封装的数据...</标签名>

```
<font color="red" face="宋体" size="7">我是字体标签</font>
```

最常用的字符实体

显示结果	描述	实体名称	实体编号
	空格	<code>&ampnbsp</code>	<code>&#160;</code>
<	小于号	<code>&amplt</code>	<code>&#60;</code>
>	大于号	<code>&ampgt</code>	<code>&#62;</code>
&	和号	<code>&ampamp</code>	<code>&#38;</code>
"	引号	<code>&ampquot</code>	<code>&#34;</code>
'	撇号	<code>&ampapos</code> (IE不支持)	<code>&#39;</code>

```
<body>
<!-- 标题标签
需求1：演示标题1到标题6的

h1 - h6 都是标题标签
h1 最大
h6 最小

align 属性是对齐属性
    left      左对齐(默认)
    center    剧中
    right    右对齐

-->
<h1 align="left">标题 1</h1>
<h2 align="center">标题 2</h2>
<h3 align="right">标题 3</h3>
<h4>标题 4</h4>
<h5>标题 5</h5>
<h6>标题 6</h6>
<h7>标题 7</h7>
</body>
```

超链接

```
<body>
<!-- a 标签是 超链接
    href 属性设置连接的地址
    target 属性设置哪个目标进行跳转
        _self    表示当前页面(默认值)
        _blank   表示打开新页面来进行跳转
-->
```

```
<a href="http://localhost:8080">百度</a><br/>
<a href="http://localhost:8080" target="_self">百度_self</a><br/>
<a href="http://localhost:8080" target="_blank">百度_blank</a><br/>
</body>
```

列表

```
<body>
    <!-- 需求 1：使用无序列表方式，把东北 F4，赵四，刘能，小沈阳，宋小宝，展示出来
        ul 是无序列表
        type 属性可以修改列表项前面的符号
        li 是列表项
    -->
    <ul type="none">
        <li>赵四</li>
        <li>刘能</li>
        <li>小沈阳</li>
        <li>宋小宝</li>
    </ul>
</body>
```

有序就是ol，前面多个序号

显示图片

```
img 标签是图片标签，用来显示图片
    src 属性可以设置图片的路径
    width 属性设置图片的宽度
    height 属性设置图片的高度
    border 属性设置图片边框大小
    alt 属性设置当指定路径找不到图片时，用来代替显示的文本内容
```

在 JavaSE 中路径也分为相对路径和绝对路径。

相对路径：从工程名开始算

绝对路径：盘符：/目录/文件名

在 web 中路径分为相对路径和绝对路径两种

相对路径：

.	表示当前文件所在的目录
..	表示当前文件所在的上一级目录
文件名	表示当前文件所在目录的文件，相当于 ./ 文件名
	./ 可以省略

绝对路径：

正确格式是： http://ip:port/工程名/资源路径

错误格式是： 盘符:/目录/文件名

```
-->





```

```
修改left 和top的值。  
可以设置图片位置
```

表格标签

```
<body>  
<!--  
    需求 1：做一个 带表头的，三行，三列的表格，并显示边框  
    需求 2：修改表格的宽度，高度，表格的对齐方式，单元格间距。  
  
    table 标签是表格标签  
        border 设置表格标签  
        width 设置表格宽度  
        height 设置表格高度  
        align 设置表格相对于页面的对齐方式
```

```
cellspacing 设置单元格间距  
  
tr 是行标签  
th 是表头标签  
td 是单元格标签  
align 设置单元格文本对齐方式  
  
b 是加粗标签  
-->  
<table align="center" border="1" width="300" height="300" cellspacing="0">  
    <tr>  
        <th>1.1</th>  
        <th>1.2</th>  
        <th>1.3</th>  
    </tr>  
    <tr>  
        <td>2.1</td>  
        <td>2.2</td>  
        <td>2.3</td>
```

跨行表格

```
colspan 属性设置跨列  
rowspan 属性设置跨行
```

```
<td colspan="2">1.1</td>
```

跨两列

```
<td rowspan="2">2.1</td>
```

跨两行

iframe 框架标签

```
<body>
    我是一个单独的完整的页面<br/><br/>
    <!--iframe 标签可以在页面上开辟一个小区域显示一个单独的页面
        iframe 和 a 标签组合使用的步骤：
            1 在 iframe 标签中使用 name 属性定义一个名称
            2 在 a 标签的 target 属性上设置 iframe 的 name 的属性值
    -->
    <iframe src="3.标题标签.html" width="500" height="400" name="abc"></iframe>
    <br/>

    <ul>
        <li><a href="0-标签语法.html" target="abc">0-标签语法.html</a></li>
        <li><a href="1.font 标签.html" target="abc">1.font 标签.html</a></li>
        <li><a href="2.特殊字符.html" target="abc">2.特殊字符.html</a></li>
    </ul>
```

指定网页跳转

表单标签（重点）

```
form 标签就是表单
    input type=text      是文件输入框  value 设置默认显示内容
    input type=password 是密码输入框  value 设置默认显示内容
    input type=radio     是单选框    name 属性可以对其进行分组  checked="checked" 表示默认选中
```

```
input type=checkbox 是复选框 checked="checked"表示默认选中  
input type=reset 是重置按钮 value 属性修改按钮上的文本  
input type=submit 是提交按钮 value 属性修改按钮上的文本  
input type=button 是按钮 value 属性修改按钮上的文本  
input type=file 是文件上传域  
input type=hidden 是隐藏域 当我们要发送某些信息，而这些信息，不需要用户参与，就可以使用隐藏域（提交的时候同时发送给服务器）
```

select 标签是下拉列表框

option 标签是下拉列表框中的选项 selected="selected"设置默认选中

textarea 表示多行文本输入框（起始标签和结束标签中的内容是默认值）

rows 属性设置可以显示几行的高度

cols 属性设置每行可以显示几个字符宽度

-->

```
<form>  
    用户名称: <input type="text" value="默认值"/><br/>  
    用户密码: <input type="password" value="abc"/><br/>  
    确认密码: <input type="password" value="abc"/><br/>  
    性别: <input type="radio" name="sex" />男<input type="radio" name="sex" checked="checked" />女<br/>  
    兴趣爱好: <input type="checkbox" checked="checked" />Java<input type="checkbox" />JavaScript<input  
    type="checkbox" />C++<br/>  
    国籍: <select>  
        <option>--请选择国籍--</option>  
        <option selected="selected" />中国</option>  
        <option>美国</option>  
        <option>小日本</option>  
    </select><br/>  
    自我评价: <textarea rows="10" cols="20">我才是默认值</textarea><br/>  
    <input type="reset" value="abc" />  
    <input type="submit" />  
</form>  
</body>
```

一般将表单放在单元格里面

表单提交

```
<!--  
form 标签是表单标签  
action 属性设置提交的服务器地址  
method 属性设置提交的方式 GET(默认值)或 POST  
  
表单提交的时候，数据没有发送给服务器的三种情况：  
1、表单项没有 name 属性值  
2、单选、复选（下拉列表中的 option 标签）都需要添加 value 属性，以便发送给服务器  
3、表单项不在提交的 form 标签中
```

GET 请求的特点是：

- 1、浏览器地址栏中的地址是：action 属性[+?+请求参数]
请求参数的格式是：name=value&name=value
- 2、不安全
- 3、它有数据长度的限制

POST 请求的特点是：

- 1、浏览器地址栏中只有 action 属性值
- 2、相对于 GET 请求要安全
- 3、理论上没有数据长度的限制

-->

```
<form action="http://localhost:8080" method="post">  
  <input type="hidden" name="action" value="login" />
```

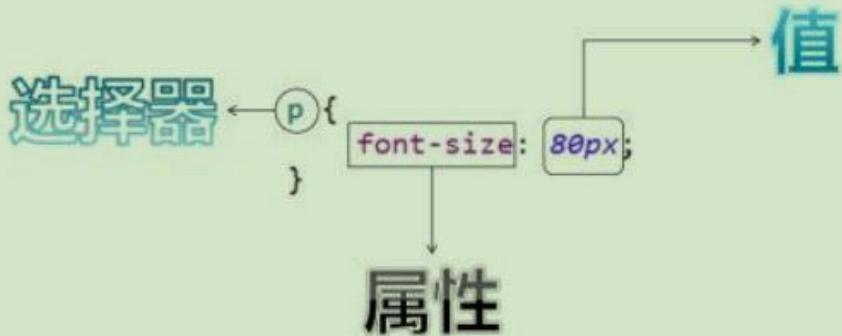
其他标签

```
<!--需求 1: div、span、p 标签的演示  
div 标签      默认独占一行  
span 标签      它的长度是封装数据的长度  
p 段落标签    默认会在段落的上方或下方各空出一行来（如果已有就不再空）
```

CSS初步

2022年1月11日 13:57

2、语法规则



选择器：浏览器根据“选择器”决定受 CSS 样式影响的 HTML 元素（标签）。

CSS 和 HTML 的结合方式

9.3.1、第一种：

在标签的 style 属性上设置“key:value value;”，修改标签样式。

```
<!--需求 1：分别定义两个 div、span 标签，分别修改每个 div 标签的样式为：边框 1 个像素，实线，红色。-->
<div style="border: 1px solid red;">div 标签 1</div>
<div style="border: 1px solid red;">div 标签 2</div>
```

问题：这种方式的缺点？

- 1.如果标签多了。样式多了。代码量非常庞大。
- 2.可读性非常差。
- 3.Css 代码没什么复用性可方言。

9.3.2、第二种：

在 head 标签中，使用 style 标签来定义各种自己需要的 css 样式。
格式如下：

```
xxx {  
    Key : value value;  
}
```

问题：这种方式的缺点。

- 1.只能在同一页面内复用代码，不能在多个页面中复用 css 代码。
- 2.维护起来不方便，实际的项目中会有成千上万的页面，要到每个页面中去修改。工作量太大了。

前两种都是写在html里面，下面单独css，方便复用

9.3.3、第三种：

把 css 样式写成一个单独的 css 文件，再通过 link 标签引入即可复用。

使用 html 的 `<link rel="stylesheet" type="text/css" href=".//styles.css" />` 标签 导入 css 样式文件。

CSS选择器

9.4.1、标签名选择器

标签名选择器的格式是：

```
标签名{  
    属性: 值;  
}
```

标签名选择器，可以决定哪些标签被动的使用这个样式。

9.4.2、id 选择器

id 选择器的格式是：

```
#id 属性值{  
    属性: 值;  
}
```

id 选择器，可以让我们通过 id 属性选择性的去使用这个样式。

```
<div id="id001">div 标签 1</div>  
<div id="id002">div 标签 2</div>
```

在定义html是需要标id

css里面

```
#id001{  
    color: blue;  
    font-size: 30px;  
    border: 1px yellow solid;
```

9.4.3、class 选择器（类选择器）

class 类型选择器的格式是：

```
.class 属性值{  
    属性: 值;  
}
```

class 类型选择器，可以通过 class 属性有效的选择性的去使用这个样式。

```
<div class="class01">div 标签 class01</div>  
<div class="class02">div 标签</div>  
<span class="class01">span 标签 class01</span>  
<span>span 标签 2</span>
```

在定义html是需要写用哪类css格式

CSS里面

```
.class01{  
    color: blue;  
    font-size: 30px;  
    border: 1px solid yellow;  
}
```

9.4.4、组合选择器

组合选择器的格式是：

```
选择器 1, 选择器 2, 选择器 n{  
    属性: 值;  
}
```

组合选择器可以让多个选择器共用同一个 css 样式代码。

```
<div class="class01">div 标签 class01</div> <br />  
<span id="id01">span 标签</span> <br />
```

```
.class01 , #id01{  
    color: blue;  
    font-size: 20px;  
    border: 1px yellow solid;  
}
```

常用样式

1、字体颜色

```
color: red;
```

2、宽度

`width:19px;`

宽度可以写像素值： 19px;

也可以写百分比值： 20%;

3、高度

`height:20px;`

高度可以写像素值： 19px;

也可以写百分比值： 20%;

4、背景颜色

`background-color:#0F2D4C`

4、字体样式：

`color: #FF0000; 字体颜色红色
font-size: 20px; 字体大小`

5、红色 1 像素实线边框

`border: 1px solid red;`

7、DIV 居中

`margin-left: auto;
margin-right: auto;`

8、文本居中：

`text-align: center;`

9、超连接去下划线

```
text-decoration: none;
```

10、表格细线

```
table {  
    border: 1px solid black; /* 设置边框 */  
    border-collapse: collapse; /* 将边框合并 */  
}  
td, th {  
    border: 1px solid black; /* 设置边框 */  
}
```

11、列表去除修饰

```
ul {  
    list-style: none;  
}
```

JavaScript初步

2022年1月11日 20:28

JS 是弱类型， Java 是强类型。

特点：

1. 交互性（它可以做的就是信息的动态交互）
2. 安全性（不允许直接访问本地硬盘）
3. 跨平台性（只要是可以解释 JS 的浏览器都可以执行，和平台无关）

弱类型就是定义变量后，变量类型还可以改变。

2、JavaScript 和 html 代码的结合方式

2.1、第一种方式

只需要在 head 标签中，或者在 body 标签中， 使用 script 标签来书写 JavaScript 代码

```
<script type="text/javascript">
    // alert 是 JavaScript 语言提供的一个警告框函数。
    // 它可以接收任意类型的参数，这个参数就是警告框的提示信息
    alert("hello javaScript!");
</script>
</head>
```

2.2、第二种方式

使用 script 标签引入 单独的 JavaScript 代码文件

```
<!--
```

现在需要使用 `script` 引入外部的 js 文件来执行

`src` 属性专门用来引入 js 文件路径（可以是相对路径，也可以是绝对路径）

`script` 标签可以用来定义 js 代码，也可以用来引入 js 文件

但是，两个功能二选一使用。不能同时使用两个功能

```
-->
```

```
<script type="text/javascript" src="1.js"></script> ↴  
<script type="text/javascript"> X
```

变量

JavaScript 的变量类型：

数值类型：	number
字符串类型：	string
对象类型：	object
布尔类型：	boolean
函数类型：	function

JavaScript 里特殊的值：

<code>undefined</code>	未定义，所有 js 变量未赋于初始值的时候，默认值都是 <code>undefined</code> .
<code>null</code>	空值
<code>NaN</code>	全称是：Not a Number。非数字。非数值。



JS 中的定义变量格式：

```
var 变量名;  
var 变量名 = 值;
```

语法上是不报错的

```
var a = 12;  
var b = "abc";  
  
alert( a * b ); // NaN 是非数字，非数值。
```

等于:	<code>==</code>	等于是简单的做字面值的比较
全等于:	<code>===</code>	除了做字面值的比较之外，还会比较两个变量的数据类型

```
var a = "12";
var b = 12;

alert( a == b ); // true
alert( a === b ); // false
```

且运算:	<code>&&</code>
或运算:	<code> </code>
取反运算:	<code>!</code>

在 JavaScript 语言中，所有的变量，都可以做一个 boolean 类型的变量去使用。

0 、 null、 undefined、 ""(空串) 都认为是 false;

```
/*
&& 且运算。
有两种情况:
第一种: 当表达式全为真的时候。返回最后一个表达式的值。
第二种: 当表达式中, 有一个为假的时候。返回第一个为假的表达式的值
```

```
|| 或运算
第一种情况: 当表达式全为假时, 返回最后一个表达式的值
第二种情况: 只要有一个表达式为真。就会把回第一个为真的表达式的值
```

```
并且 && 与运算 和 ||或运算 有短路。
短路就是说, 当这个&&或||运算有结果了之后 。后面的表达式不再执行
*/
```

数组

JS 中 数组的定义:

格式:

```
var 数组名 = [];// 空数组
```

```
var 数组名 = [1, 'abc', true]; // 定义数组同时赋值元素
```

原来arr长度为2

```
// JavaScript 语言中的数组，只要我们通过数组下标赋值，那么最大的下标值，就会自动的给数组做扩容操作。  
arr[2] = "abc";  
alert(arr.length); //3
```

函数

第一种，可以使用 **function** 关键字来定义函数。

使用的格式如下:

```
function 函数名(形参列表){  
    函数体  
}
```

在 JavaScript 语言中，如何定义带有返回值的函数？

只需要在函数体内直接使用 **return** 语句返回值即可！

```
// 定义一个无参函数
function fun(){
    alert("无参函数 fun()被调用了");
}

// 函数调用==才执行
// fun();

function fun2(a ,b) {
    alert("有参函数 fun2()被调用了 a=>" + a + ",b=>" + b);
}

// fun2(12, "abc");
```

有返回值

```
function sum(num1,num2) {
    var result = num1 + num2;
    return result;
}

alert( sum(100,50) );
```

函数的第二种定义方式，格式如下：

使用格式如下：

```
var 函数名 = function(形参列表){ 函数体 }
```

```
var fun = function () {
    alert("无参函数");
}

// fun();
```

注：在 Java 中函数允许重载。但是在 JS 中函数的重载会直接覆盖掉上一次的定义

8.2、函数的 arguments 隐形参数（只在 function 函数内）

就是在 function 函数中不需要定义，但却可以直接用来获取所有参数的变量。我们管它叫隐形参数。
隐形参数特别像 java 基础的可变长参数一样。

public void fun(Object ... args);

可变长参数其他是一个数组。

那么 js 中的隐形参数也跟 java 的可变长参数一样。操作类似数组。

```
function fun(a) {
    alert( arguments.length );//可看参数个数

    alert( arguments[0] );
    alert( arguments[1] );
    alert( arguments[2] );

    alert("a = " + a);

    for (var i = 0; i < arguments.length; i++){
        alert( arguments[i] );
    }

    alert("无参函数 fun()");
}
```

```
// 需求：要求 编写 一个函数。用于计算所有参数相加的和并返回
function sum(num1,num2) {
    var result = 0;
    for (var i = 0; i < arguments.length; i++) {
        if (typeof(arguments[i]) == "number") {
            result += arguments[i];
        }
    }
    return result;
}
```

Object 形式的自定义对象

对象的定义：

```
var 变量名 = new Object(); // 对象实例（空对象）
变量名.属性名 = 值; // 定义一个属性
变量名.函数名 = function(){}
// 定义一个函数
```

对象的访问：

```
变量名.属性 / 函数名();
```

```
var obj = new Object();
obj.name = "华仔";
obj.age = 18;
obj.fun = function () {
    alert("姓名：" + this.name + "，年龄：" + this.age);
}
// 对象的访问：
//     变量名.属性 / 函数名();
// alert( obj.age );
obj.fun();
```

{}花括号形式的自定义对象

对象的定义：

```
var 变量名 ={
    属性名: 值, // 定义一个属性
    属性名: 值, // 定义一个属性
    函数名: function(){}
}; // 定义一个函数
```

对象的访问：

```
变量名.属性 / 函数名();
```

```

var obj = {
    name:"国哥",
    age:18,
    fun : function () {
        alert("姓名: " + this.name + " , 年龄: " + this.age);
    }
};

// 对象的访问:
// 变量名.属性 / 函数名();
alert(obj.name);
obj.fun();

```

js 中的事件

常用事件:

onload 加载完成事件:	页面加载完成之后，常用于做页面 js 代码初始化操作
onclick 单击事件:	常用于按钮的点击响应操作。
onblur 失去焦点事件:	常用于输入框失去焦点后验证其输入内容是否合法。
onchange 内容发生改变事件:	常用于下拉列表和输入框内容发生改变后操作
onsubmit 表单提交事件:	常用于表单提交前，验证所有表单项是否合法。

事件的注册又分为静态注册和动态注册两种:

什么是事件的注册（绑定）？

其实就是告诉浏览器，当事件响应后要执行哪些操作代码，叫事件注册或事件绑定。

静态注册事件：通过 html 标签的事件属性直接赋予事件响应后的代码，这种方式我们叫静态注册。

动态注册事件：是指先通过 js 代码得到标签的 dom 对象，然后再通过 dom 对象.事件名 = function(){}
这种形式赋予事件响应后的代码，叫动态注册。

动态注册基本步骤：

- 1、获取标签对象
- 2、标签对象.事件名 = function(){}

Onload

动态

```
// onLoad 事件动态注册。是固定写法
window.onload = function () {
    alert("动态注册的 onload 事件");
}

</script>
```

静态

```
function onLoadFun() {
    alert('静态注册 onload 事件，所有代码');
}
```

```
<!-- 静态注册 onLoad 事件
onLoad 事件是浏览器解析完页面之后就会自动触发的事件
<body onLoad="onLoadFun();">
```

Onclick

静态

```
<!-- 静态注册 onClick 事件-->
<button onclick="onClickFun();">按钮 1</button>
<button id="btn01">按钮 2</button>
```

动态

```
<button id="btn01">按钮 2</button>

var btnObj = document.getElementById("btn01");
// alert( btnObj );
// 2 通过标签对象. 事件名 = function(){}
btnObj.onclick = function () {
    alert("动态注册的 onclick 事件");
}
```

onblur 失去焦点事件 (输入截止, 光标移开)

和onclick一样

onchange 内容发生改变事件 ()

onsubmit 表单提交事件

静态, 错误填表, 阻止提交

```
function onsubmitFun(){
    // 要验证所有表单项是否合法, 如果, 有一个不合法就阻止表单提交
    alert("静态注册表单提交事件----发现不合法");

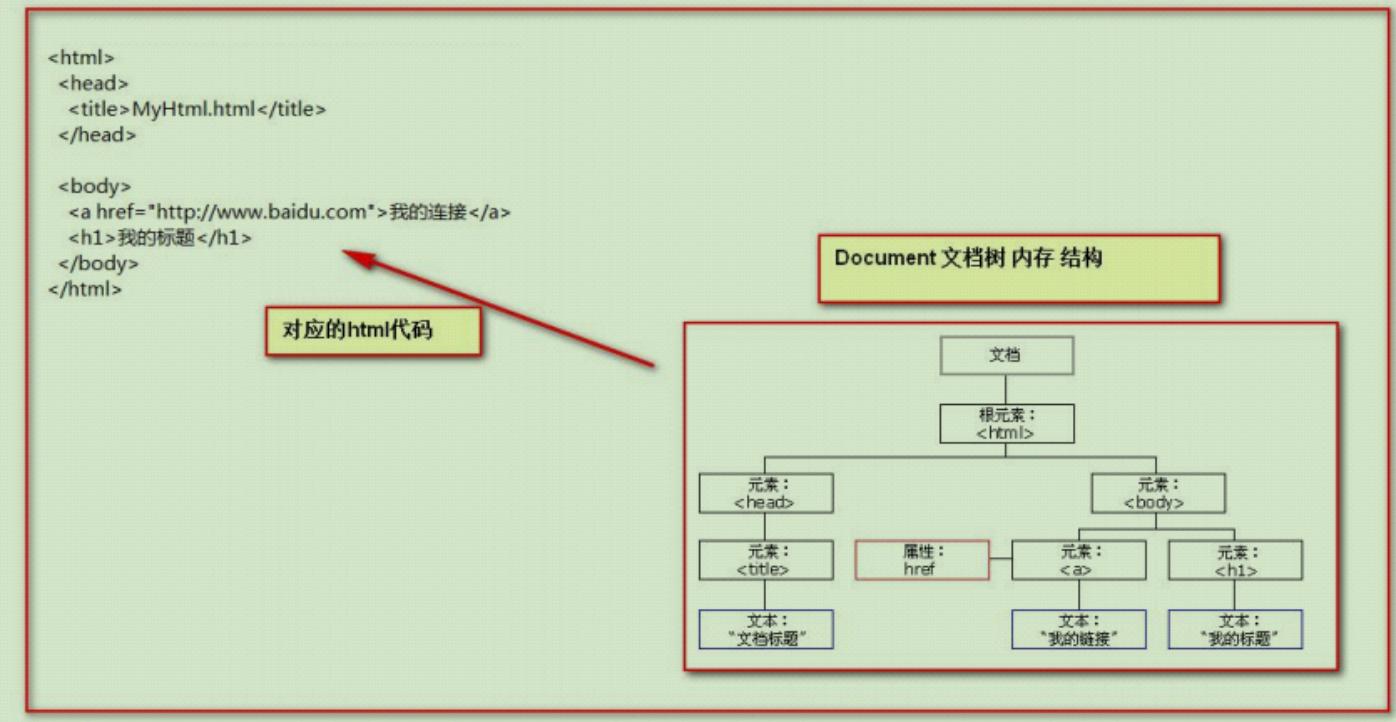
    return flase;
}
```

```
<form action="http://localhost:8080" method="get" onsubmit="return onsubmitFun();">
    <input type="submit" value="静态注册"/>
```

DOM 模型

大白话, 就是把文档中的标签, 属性, 文本, 转换成为对象来管理。

10.1、Document 对象 (*****重点)



那么 html 标签 要 对象化 怎么办？

```
<body>
  <div id="div01">div01</div>
</body>
```

模拟对象化，相当于：

```
class Dom{
    private String id;          // id 属性
    private String tagName; // 表示标签名
    private Dom parentNode; // 父亲
    private List<Dom> children; // 孩子结点
    private String innerHTML; // 起始标签和结束标签中间的内容
}
```

10.4、Document 对象中的方法介绍 (*****重点)

document.getElementById(elementId)

通过标签的 id 属性查找标签 dom 对象， elementId 是标签的 id 属性值

document.getElementsByName(elementName)

通过标签的 name 属性查找标签 dom 对象， elementName 标签的 name 属性值

document.getElementsByTagName(tagname)

通过标签名查找标签 dom 对象。 tagname 是标签名

document.createElement(tagName)

方法，通过给定的标签名，创建一个标签对象。 tagName 是要创建的标签名

验证填表内容

.value方法和 .test测试

```
function onclickFun() {
    // 1 当我们要操作一个标签的时候，一定要先获取这个标签对象。
    var usernameObj = document.getElementById("username");
    // [object HTMLInputElement] 它就是 dom 对象
    var usernameText = usernameObj.value;
    /* 如何 验证 字符串，符合某个规则，需要使用正则表达式技术 */
    var patt = /\w{5,12}$/;
    /*
     * test()方法用于测试某个字符串，是不是匹配我的规则，
     * 匹配就返回 true。不匹配就返回 false.
     */
}
```

```
if (patt.test(usernameText)) {
    // alert("用户名合法！");
    // usernameSpanObj.innerHTML = "用户名合法！";
    usernameSpanObj.innerHTML = "<img src=\"right.png\" width=\"18\" height=\"18\">";
} else {
    // alert("用户名不合法！");
    // usernameSpanObj.innerHTML = "用户名不合法！";
    usernameSpanObj.innerHTML = "<img src=\"wrong.png\" width=\"18\" height=\"18\">";
}
```

在文本框旁显示alert，或者改标签的内容

```
// innerHTML 表示起始标签和结束标签中的内容  
// innerHTML 这个属性可读，可写  
usernameSpanObj.innerHTML = "国哥真可爱！";
```

正则

[语法](#) [详细](#)

语法

/正则表达式主体/g(可选)

在 JavaScript 中，正则表达式通常用于两个字符串方法：search() 和 replace()。

正则表达式修饰符

修饰符 可以在全局搜索中不区分大小写：

修饰符	描述
i	执行对大小写不敏感的匹配。
g	执行全局匹配（查找所有匹配而非在找到第一个匹配后停止）。
m	执行多行匹配。

方括号用于查找某个范围内的字符：

表达式	描述
[abc]	查找方括号之间的任何字符。
[0-9]	查找任何从 0 至 9 的数字。
(x y)	查找任何以 分隔的选项。

元字符是拥有特殊含义的字符:

元字符	描述
\d	查找数字。
\s	查找空白字符。
\b	匹配单词边界。
\uxxxx	查找以十六进制数 xxxx 规定的 Unicode 字符。

量词:

量词	描述
n+	匹配任何包含至少一个 n 的字符串。
n*	匹配任何包含零个或多个 n 的字符串。
n?	匹配任何包含零个或一个 n 的字符串。

使用 exec()

exec() 方法是一个正则表达式方法。

exec() 方法用于检索字符串中的正则表达式的匹配。

该函数返回一个数组，其中存放匹配的结果。如果未找到匹配，则返回值为 null。

以下实例用于搜索字符串中的字母 "e":

实例 1

```
/e/.exec("The best things in life are free!");
```

字符串中含有 "e"，所以该实例输出为:

```
e
```

jQuery初步

2022年1月11日 20:57

文档

```
//使用$()代替window.onload
$(function(){
    //使用选择器获取按钮对象，随后绑定单击响应函数
    $('#btnId').click(function(){
        //弹出Hello
        alert('Hello');
    });
});
```

```
var $btnObj = $("#btnId"); // 表示按 id 查询标签对象
$btnObj.click(function () { // 绑定单击事件
```

查询要加#

其余函数要写在\$()里面

jQuery的目的就是查找和筛选

jQuery速记

jQuery 选择器

选择器	实例	选取
*	\$("")	所有元素
#id	\$("#lastname")	id="lastname" 的元素
.class	\$(".intro")	所有 class="intro" 的元素
element	\$("p")	所有 <p> 元素
.class.class	\$(".intro.demo")	所有 class="intro" 且 class="demo" 的元素
:first	\$("p:first")	第一个 <p> 元素
:last	\$("p:last")	最后一个 <p> 元素
:even	\$("tr:even")	所有偶数 <tr> 元素

选择器		
:even	\$(“tr:even”)	所有偶数 <tr> 元素
:odd	\$(“tr:odd”)	所有奇数 <tr> 元素
:eq(index)	\$(“ul li:eq(3)”)	列表中的第四个元素 (index 从 0 开始)
:gt(no)	\$(“ul li:gt(3)”)	列出 index 大于 3 的元素
:lt(no)	\$(“ul li:lt(3)”)	列出 index 小于 3 的元素
:not(selector)	\$(“input:not(:empty)”)	所有不为空的 input 元素
:header	\$(“:header”)	所有标题元素 <h1> - <h6>
:animated		所有动画元素
:contains(text)	\$(“:contains(‘W3School’)”)	包含指定字符串的所有元素
:empty	\$(“:empty”)	无子（元素）节点的所有元素
:hidden	\$(“p:hidden”)	所有隐藏的 <p> 元素
:visible	\$(“table:visible”)	所有可见的表格
s1,s2,s3	\$(“th,td,.intro”)	所有带有匹配选择的元素
[attribute]	\$(“[href]”)	所有带有 href 属性的元素
[attribute=value]	\$(“[href=‘#’]”)	所有 href 属性的值等于 “#” 的元素
[attribute!=value]	\$(“[href!=‘#’]”)	所有 href 属性的值不等于 “#” 的元素
[attribute\$=value]	\$(“[href\$=‘.jpg’]”)	所有 href 属性的值包含以 “.jpg” 结尾的元素
:input	\$(“:input”)	所有 <input> 元素
:text	\$(“:text”)	所有 type=“text”的 <input> 元素
:password	\$(“:password”)	所有 type=“password”的 <input> 元素
:radio	\$(“:radio”)	所有 type=“radio”的 <input> 元素
:checkbox	\$(“:checkbox”)	所有 type=“checkbox”的 <input> 元素
:submit	\$(“:submit”)	所有 type=“submit”的 <input> 元素
:reset	\$(“:reset”)	所有 type=“reset”的 <input> 元素
:button	\$(“:button”)	所有 type=“button”的 <input> 元素
:image	\$(“:image”)	所有 type=“image”的 <input> 元素
:file	\$(“:file”)	所有 type=“file”的 <input> 元素

<code>:file</code>	<code>\$(":file")</code>	所有 type="file" 的 <code><input></code> 元素
<code>:enabled</code>	<code>\$(":enabled")</code>	所有激活的 <code>input</code> 元素
<code>:disabled</code>	<code>\$(":disabled")</code>	所有禁用的 <code>input</code> 元素
<code>:selected</code>	<code>\$(":selected")</code>	所有被选取的 <code>input</code> 元素
<code>:checked</code>	<code>\$(":checked")</code>	所有被选中的 <code>input</code> 元素

3、jQuery 核心函数

`$` 是 jQuery 的核心函数，能完成 jQuery 的很多功能。`$()`就是调用`$`这个函数

1、传入参数为 [函数] 时：

表示页面加载完成之后。相当于 `window.onload = function(){};`

2、传入参数为 [HTML 字符串] 时：

会对我们创建这个 `html` 标签对象

3、传入参数为 [选择器字符串] 时：

`$("#id 属性值");` id 选择器，根据 id 查询标签对象

`$("标签名");` 标签名选择器，根据指定的标签名查询标签对象

`$(".class 属性值");` 类型选择器，可以根据 class 属性查询标签对象

4、传入参数为 [DOM 对象] 时：

会把这个 `dom` 对象转换为 jQuery 对象

4.2、问题：jQuery 对象的本质是什么？

jQuery 对象是 dom 对象的数组 + jQuery 提供的一系列功能函数。

jQuery 对象不能使用 DOM 对象的属性和方法

DOM 对象也不能使用 jQuery 对象的属性和方法

jQuery 对象不能使用 DOM 对象的属性和方法
DOM 对象也不能使用 jQuery 对象的属性和方法

1、dom 对象转化为 jQuery 对象 (*重点)

- 1、先有 DOM 对象
- 2、\$(DOM 对象) 就可以转换成为 jQuery 对象

2、jQuery 对象转为 dom 对象 (*重点)

- 1、先有 jQuery 对象
- 2、jQuery 对象[下标]取出相应的 DOM 对象

5. 1、基本选择器

#ID 选择器：根据 id 查找标签对象
.class 选择器：根据 class 查找标签对象
element 选择器：根据标签名查找标签对象
* 选择器：表示任意的，所有的元素
selector1, selector2 组合选择器：合并选择器 1, 选择器 2 的结果并返回

eg

```
$(function(){
    //1. 选择 id 为 one 的元素 "background-color", "#bbffaa"
    $("#btn1").click(function () {
        $("#one").css("background-color", "#bbffaa")
    })
})
```

多类之间用，隔开

```
//5. 选择所有的 span 元素和 id 为 two 的元素
$("#btn5").click(function () {
    $("span,#two").css("background-color", "#bbffaa");
});
```

5. 2、层级选择器

ancestor descendant 后代选择器：在给定的祖先元素下匹配所有的后代元素
parent > child 子元素选择器：在给定的父元素下匹配所有的子元素
prev + next 相邻元素选择器：匹配所有紧接在 prev 元素后的 next 元素
prev ~ siblings 之后的兄弟元素选择器：匹配 prev 元素之后的所有 siblings 元素

Eg

```
//1. 选择 body 内的所有 div 元素
```

Eg

```
//1.选择 body 内的所有 div 元素
$("#btn1").click(function(){
    $("body div").css("background", "#dbffaa");
});
```

5.3、过滤选择器

HTML 代码:

```
<ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
</ul>
```

jQuery 代码:

```
$('#li:last')
```

结果:

```
[<li>list item 5</li>]
```

上面为示例

:first	获取第一个元素
:last	获取最后一个元素
:not(selector)	去除所有与给定选择器匹配的元素
:even	匹配所有索引值为偶数的元素，从 0 开始计数
:odd	匹配所有索引值为奇数的元素，从 0 开始计数
:eq(index)	匹配一个给定索引值的元素
:gt(index)	匹配所有大于给定索引值的元素
:lt(index)	匹配所有小于给定索引值的元素
:header	匹配如 h1, h2, h3 之类的标题元素
:animated	匹配所有正在执行动画效果的元素

:contains(text)	匹配包含给定文本的元素
:empty	匹配所有不包含子元素或者文本的空元素
:parent	匹配含有子元素或者文本的元素
:has(selector)	匹配含有选择器所匹配的元素的元素

查找所有含有 id 属性的 div 元素

HTML 代码:

```
<div>
    <p>Hello!</p>
</div>
<div id="test2"></div>
```

jQuery 代码:

```

-----  

jQuery 代码:  

-----  

$("div[id]")
-----  

结果:  

-----  

[ <div id="test2"></div> ]
-----  


```

上为eg

[attribute]	匹配包含给定属性的元素。
[attribute=value]	匹配给定的属性是某个特定值的元素
[attribute!=value]	匹配所有不含有指定的属性，或者属性不等于特定值的元素。
[attribute^=value]	匹配给定的属性是以某些值开始的元素
[attribute\$=value]	匹配给定的属性是以某些值结尾的元素
[attribute*=value]	匹配给定的属性是以包含某些值的元素
[attrSel1][attrSel2][attrSelN]	复合属性选择器，需要同时满足多个条件时使用。

:input	匹配所有 input, textarea, select 和 button 元素
:text	匹配所有 文本输入框
:password	匹配所有的密码输入框
:radio	匹配所有的单选框
:checkbox	匹配所有的复选框
:submit	匹配所有提交按钮
:image	匹配所有 img 标签
:reset	匹配所有重置按钮
:button	匹配所有 input type=button <button>按钮
:file	匹配所有 input type=file 文件上传
:hidden	匹配所有不可见元素 display:none 或 input type=hidden

:enabled	匹配所有可用元素
:disabled	匹配所有不可用元素
:checked	匹配所有选中的单选，复选，和下拉列表中选中的 option 标签对象
:selected	匹配所有选中的 option

6、jQuery 元素筛选

eq()	获取给定索引的元素	功能跟 :eq() 一样
first()	获取第一个元素	功能跟 :first 一样
last()	获取最后一个元素	功能跟 :last 一样
filter(exp)	留下匹配的元素	
is(exp)	判断是否匹配给定的选择器，只要有一个匹配就返回， true	
has(exp)	返回包含有匹配选择器的元素的元素	功能跟 :has 一样
not(exp)	删除匹配选择器的元素	功能跟 :not 一样
children(exp)	返回匹配给定选择器的子元素	功能跟 parent>child 一样
find(exp)	返回匹配给定选择器的后代元素	功能跟 ancestor>descendant 一样

<code>not(exp)</code>	删除匹配选择器的元素	功能跟 :not 一样
<code>children(exp)</code>	返回匹配给定选择器的子元素	功能跟 parent>child 一样
<code>find(exp)</code>	返回匹配给定选择器的后代元素	功能跟 ancestor descendant 一样

<code>next()</code>	返回当前元素的下一个兄弟元素
<code>nextAll()</code>	返回当前元素后面所有的兄弟元素
<code>nextUntil()</code>	返回当前元素到指定匹配的元素为止的后面元素
<code>parent()</code>	返回父元素
<code>prev(exp)</code>	返回当前元素的上一个兄弟元素
<code>prevAll()</code>	返回当前元素前面所有的兄弟元素
<code>prevUnit(exp)</code>	返回当前元素到指定匹配的元素为止的前面元素
<code>siblings(exp)</code>	返回所有兄弟元素
<code>add()</code>	把 add 匹配的选择器的元素添加到当前 jquery 对象中

jQuery 的属性操作

<code>html()</code>	它可以设置和获取起始标签和结束标签中的内容。	跟 dom 属性 innerHTML 一样。
<code>text()</code>	它可以设置和获取起始标签和结束标签中的文本。	跟 dom 属性 innerText 一样。
<code>val()</code>	它可以设置和获取表单项的 value 属性值。	跟 dom 属性 value 一样

Eg

```

$(function () {
/*
    // 批量操作单选
    $(":radio").val(["radio2"]);
    // 批量操作筛选框的选中状态
    $(":checkbox").val(["checkbox3", "checkbox2"]);
    // 批量操作多选的下拉框选中状态
    $("#multiple").val(["mul2", "mul3", "mul4"]);
    // 操作单选的下拉框选中状态
    $("#single").val(["sin2"]);
*/
$("#multiple,#single,:radio,:checkbox").val(["radio2","checkbox1","checkbox3","mul1","mul4","sin3"]);
});
```

`attr()` 可以设置和获取属性的值，不推荐操作 checked、readOnly、selected、disabled 等等

attr 方法还可以操作非标准的属性。比如自定义属性：abc,bbj

`prop()` 可以设置和获取属性的值,只推荐操作 checked、readOnly、selected、disabled 等等

eg

```

// 给全选绑定单击事件
$("#checkedAllBtn").click(function () {
    ":checkbox").prop("checked",true);
});
```

```
$(":checkbox").prop("checked",true);  
});
```

DOM 的增删改

内部插入:

appendTo()	a.appendTo(b)	把 a 插入到 b 子元素末尾，成为最后一个子元素
------------	---------------	---------------------------

prependTo()	a.prependTo(b)	把 a 插到 b 所有子元素前面，成为第一个子元素
-------------	----------------	---------------------------

外部插入:

insertAfter()	a.insertAfter(b)	得到 ba
insertBefore()	a.insertBefore(b)	得到 ab

替换:

replaceWith()	a.replaceWith(b)	用 b 替换掉 a
replaceAll()	a.replaceAll(b)	用 a 替换掉所有 b

删除:

remove()	a.remove();	删除 a 标签
empty()	a.empty();	清空 a 标签里的内容

6、CSS 样式操作。

addClass()	添加样式
removeClass()	删除样式
toggleClass()	有就删除，没有就添加样式。
offset()	获取和设置元素的坐标。

7、jQuery 动画

基本动画

show()	将隐藏的元素显示
hide()	将可见的元素隐藏。
toggle()	可见就隐藏，不可见就显示。

以上动画方法都可以添加参数。

- 1、第一个参数是动画 执行的时长，以毫秒为单位
- 2、第二个参数是动画的回调函数 (动画完成后自动调用的函数)

- 1、第一个参数是动画 执行的时长，以毫秒为单位
 2、第二个参数是动画的回调函数 (动画完成后自动调用的函数)

淡入淡出动画

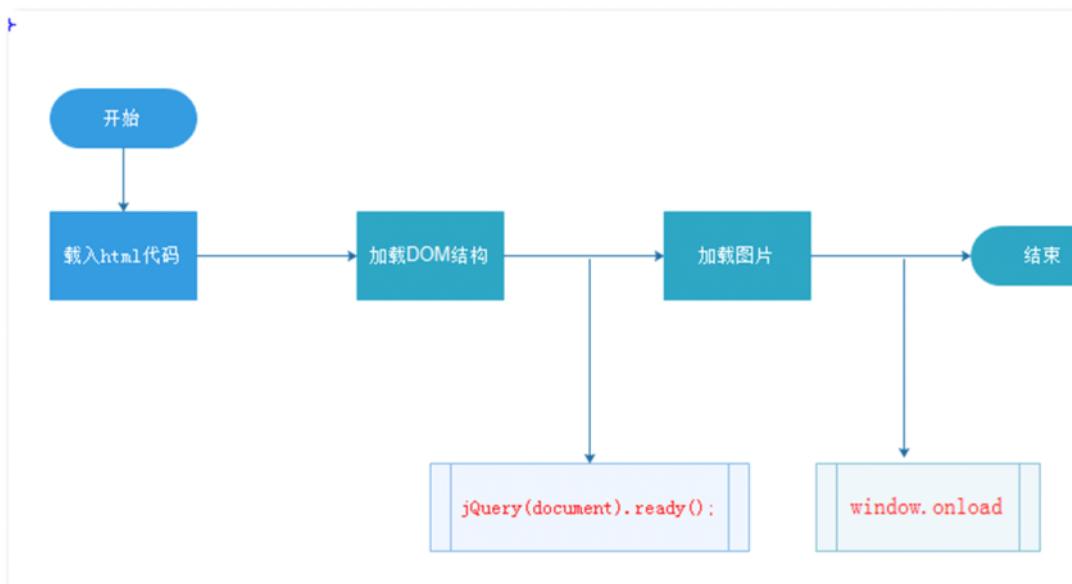
<code>fadeIn()</code>	淡入（慢慢可见）
<code>fadeOut()</code>	淡出（慢慢消失）
<code>fadeTo()</code>	在指定时长内慢慢的将透明度修改到指定的值。0 透明，1 完成可见，0.5 半透明
<code>fadeToggle()</code>	淡入/淡出 切换

```
//显示 show()
$("#btn1").click(function(){
  $("#div1").show(2000, function () {
    alert("show动画完成 ")
  });
});
```

8、jQuery 事件操作

`$(function(){});`
 和
`window.onload = function(){}`
 的区别？

- 1、`window.onload` 不能有多个，后面的功能会覆盖前面。而 `jQuery(document).ready()` 可以存在多个。
 2、`window.onload` 在页面所有元素（包括图片，引用文件）加载完后执行。而 `jQuery(document).ready()` 页面中会在所有HTML DOM, CSS DOM 结构加载完之后就会执行，其他图片可能还没有加载完。



DOM文档加载步骤：

- 1.解析HTML结构
- 2.加载外部的脚本和样式文件
- 3.解析并执行脚本代码

- 2.加载外部的脚本和样式文件
- 3.解析并执行脚本代码
- 4.执行 `$(function() {})` 内对应代码
- 5.加载图片等二进制资源
- 6.页面加载完毕，执行 `window.onload`

jQuery 中其他的事件处理方法：

<code>click()</code>	它可以绑定单击事件，以及触发单击事件
<code>mouseover()</code>	鼠标移入事件
<code>mouseout()</code>	鼠标移出事件
<code>bind()</code>	可以给元素一次性绑定一个或多个事件。
<code>one()</code>	使用上跟 <code>bind</code> 一样。但是 <code>one</code> 方法绑定的事件只会响应一次。
<code>unbind()</code>	跟 <code>bind</code> 方法相反的操作，解除事件的绑定
<code>live()</code>	也是用来绑定事件。它可以用来绑定选择器匹配的所有元素的事件。哪怕这个元素是后面动态创建出来的也有效

```
//绑定单击事件,改变字体颜色
$("#btn1").bind("click",function(){
    alert("事件绑定");
});
```

事件的冒泡

什么是事件的冒泡？

事件的冒泡是指，父子元素同时监听同一个事件。当触发子元素的事件的时候，同一个事件也被传递到了父元素的事件里去响应。

那么如何阻止事件冒泡呢？

在子元素事件函数体内，`return false;` 可以阻止事件的冒泡传递。

通俗讲，就是点击大框里面的小框，大框也会被选择。

javaScript 事件对象

事件对象，是封装有触发的事件信息的一个 javascript 对象。

我们重点关心的是怎么拿到这个 javascript 的事件对象。以及使用。

如何获取呢 javascript 事件对象呢？

在给元素绑定事件的时候，在事件的 `function(event)` 参数列表中添加一个参数，这个参数名，我们习惯取名为 `event`。这个 `event` 就是 javascript 传递参事件处理函数的事件对象。

事件在浏览器中是以对象的形式存在的，即 `event`。触发一个事件，就会产生一个事件对象 `event`，该对象包含着所有与事件有关的信息。包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。

例如：鼠标操作产生的 `event` 中会包含鼠标位置的信息；键盘操作产生的 `event` 中会包含与按下的键有关的信息。

包括导致事件的元素、事件的类型以及其他与特定事件相关的信息。

例如：鼠标操作产生的event中会包含鼠标位置的信息；键盘操作产生的event中会包含与按下的键有关的信息。

```
hello.html?_ijt=k8lq...almbigdat0rreupq:10
jQuery.Event {originalEvent: PointerEvent, type: 'click', timeStamp: 127
  ▾ 97.7999999702, jQuery17207345914805988742: true, isDefaultPrevented:
    f, ...} ⓘ
  altKey: false
  attrChange: undefined
  attrName: undefined
  bubbles: true
  button: 0
  buttons: 0
  cancelable: true
  clientX: 27
  clientY: 27
  ctrlKey: false
  ▶ currentTarget: p
  data: null
  ▶ delegateTarget: p
  eventPhase: 2
  fromElement: null
  ▶ handleObj: {type: 'click', origType: 'click', data: null, guid: 1, han...
  isDefaultPrevented: f returnFalse()
  jQuery17207345914805988742: true
  metaKey: false
  offsetX: 19
  offsetY: 11
  ▶ originalEvent: PointerEvent {isTrusted: true, pointerId: 1, width: 1, ...
  pageX: 27
```

//1.原生 javascript 获取 事件对象

```
window.onload = function () {
  document.getElementById("areaDiv").onclick = function (event) {
    console.log(event);
  }
}
```

//2.jQuery 代码获取 事件对象

```
$(function () {
  $("#areaDiv").click(function (event) {
    console.log(event);
  });
});
```

//3.使用 bind 同时对多个事件绑定同一个函数。怎么获取当前操作是什么事件。

```
$("#areaDiv").bind("mouseover mouseout",function (event) {
  if (event.type == "mouseover") {
    console.log("鼠标移入");
  } else if (event.type == "mouseout") {
    console.log("鼠标移出");
  }
});
```

xml 的作用？

xml 的主要作用有：

- 1、用来保存数据，而且这些数据具有自我描述性
- 2、它还可以做为项目或者模块的配置文件
- 3、还可以做为网络传输数据的格式（现在 JSON 为主）。

XML 被设计为传输和存储数据，其焦点是数据的内容。

HTML 被设计用来显示数据，其焦点是数据的外观。

HTML 旨在显示信息，而 XML 旨在传输信息。

XML允许用户定义自己的标签。XML文件有且仅有一个根标签，其它标签都是这个根标签的子孙标签。XML文件中的标签，分为开始标签和结束标签，如<a>和，成为一对。一对标签内的空格和换行都作为原始内容被处理。下面的 (1) 和 (2) 是不同的

(1) 呵呵

(2) 呵

呵

xml文件中的注释采用：<!--注释--> 格式。

注意：XML声明之前不能有注释；注释不能嵌套。

```
<?xml version="1.0" encoding="UTF-8"?> xml 声明。  
<!-- xml 声明 version 是版本的意思 encoding 是编码 -->  
而且这个<?xml 要连在一起写，否则会有报错
```

属性

version	是版本号
encoding	是 xml 的文件编码
standalone="yes/no"	表示这个 xml 文件是否是独立的 xml 文件

XML 元素必须遵循以下命名规则：

2.1) 名称可以含字母、数字以及其他字符

2.2) 名称不能以数字或者标点符号开始

2.4) 名称不能包含空格

XML 属性

从 HTML，你会回忆起这个：。"src" 属性提供有关 元素的额外信息。

在 HTML 中（以及在 XML 中），属性提供有关元素的额外信息：

```
  
<a href="demo.asp">
```

属性通常提供不属于数据组成部分的信息。在下面的例子中，文件类型与数据无关，但是对需要处理这个元素的软件来说却很重要：

```
<file type="gif">computer.gif</file>
```

XML 属性必须加引号

属性值必须被引号包围，不过单引号和双引号均可使用。比如一个人的性别，person 标签可以这样写：

```
<person sex="female">
```

或者这样也可以：

```
<person sex='female'>
```

注释：如果属性值本身包含双引号，那么有必要使用单引号包围它，就像这个例子：

```
<gangster name='George "Shotgun" Ziegler'>
```

或者可以使用实体引用：

```
<gangster name="George &quot;Shotgun&quot; Ziegler">
```

3.5.4) XML 文档必须有根元素

根元素就是顶级元素，

没有父标签的元素，叫顶级元素。

根元素是没有父标签的顶级元素，而且是唯一一个才行。

所有 XML 文档中的文本均会被解析器解析。

只有 CDATA 区段 (CDATA section) 中的文本会被解析器忽略。

```
<name>java编程思想</name> <!-- name标签描述的是图书的信息 -->
<author>
<![CDATA[
<<<<<华仔>>>>>
]]>
</author> <!-- author单词是作者的意思，描述图书作者 -->
```

CDATA文本区，里面的
内容不会被解
析。只会把它们当
成纯文本

dom4j 解析技术

3.3、dom4j 编程步骤：

第一步：先加载 xml 文件创建 Document 对象

第二步：通过 Document 对象拿到根元素对象

第三步：通过根元素.elements(标签名); 可以返回一个集合，这个集合里放着。所有你指定的标签名的元素对象

第四步：找到你想要修改、删除的子元素，进行相应在的操作

第五步，保存到硬盘上

第一步，通过创建 SAXReader 对象。来读取 xml 文件，获取 Document 对象

```
SAXReader reader = new SAXReader();
Document document = reader.read("src/books.xml");
// 第二步，通过 Document 对象。拿到 XML 的根元素对象
Element root = document.getRootElement();
// 打印测试
// Element.asXML() 它将当前元素转换成为 String 对象
// System.out.println( root.asXML() );
第三步，通过根元素对象。获取所有的 book 标签对象
// Element.elements(标签名)它可以拿到当前元素下的指定的子元素的集合
List<Element> books = root.elements("book");
第四小，遍历每个 book 标签对象。然后获取到 book 标签对象内的每一个元素，
for (Element book : books) {
    // 测试
    // System.out.println(book.asXML());
    // 拿到 book 下面的 name 元素对象
    Element nameElement = book.element("name");
    // 拿到 book 下面的 price 元素对象
    Element priceElement = book.element("price");
    // 拿到 book 下面的 author 元素对象
    Element authorElement = book.element("author");
    // 再通过 getText() 方法拿到起始标签和结束标签之间的文本内容
    System.out.println("书名" + nameElement.getText() + "，价格:"
        + priceElement.getText() + "，作者：" + authorElement.getText());
}
```

a)什么是 JavaWeb

JavaWeb 是指，所有通过 Java 语言编写可以通过浏览器访问的程序的总称，叫 JavaWeb。
JavaWeb 是基于请求和响应来开发的。

b)什么是请求

请求是指客户端给服务器发送数据，叫请求 Request。

c)什么是响应

响应是指服务器给客户端回传数据，叫响应 Response。

3. 常用的 Web 服务器

Tomcat: 由 Apache 组织提供的一种 Web 服务器，提供对 jsp 和 Servlet 的支持。它是一种轻量级的 javaWeb 容器（服务器），也是当前应用最广的 JavaWeb 服务器（免费）。

Jboss: 是一个遵从 JavaEE 规范的、开放源代码的、纯 Java 的 EJB 服务器，它支持所有的 JavaEE 规范（免费）。

GlassFish: 由 Oracle 公司开发的一款 JavaWeb 服务器，是一款强健的商业服务器，达到产品级质量（应用很少）。

Resin: 是 CAUCHO 公司的产品，是一个非常流行的服务器，对 servlet 和 JSP 提供了良好的支持，
性能也比较优良，resin 自身采用 JAVA 语言开发（收费，应用比较多）。

WebLogic: 是 Oracle 公司的产品，是目前应用最广泛的 Web 服务器，支持 JavaEE 规范，
而且不断的完善以适应新的开发要求，适合大型项目（收费，用的不多，适合大公司）。

Tomcat 的使用

b) 目录介绍

bin	专门用来存放 Tomcat 服务器的可执行程序
conf	专门用来存放 Tomcat 服务器的配置文件
lib	专门用来存放 Tomcat 服务器的 jar 包
logs	专门用来存放 Tomcat 服务器运行时输出的日志信息
temp	专门用来存放 Tomcat 运行时产生的临时数据
webapps	专门用来存放部署的 Web 工程。
work	是 Tomcat 工作时的目录，用来存放 Tomcat 运行时 jsp 翻译为 Servlet 的源码，和 Session 钝化的目录。

c) 如何启动 Tomcat 服务器

找到 Tomcat 目录下的 bin 目录下的 startup.bat 文件，双击，就可以启动 Tomcat 服务器。

如何测试 Tomcat 服务器启动成功？？？

打开浏览器，在浏览器地址栏中输入以下地址测试：

- 1、<http://localhost:8080>
- 2、<http://127.0.0.1:8080>
- 3、<http://真实 ip:8080>

f) 如何部署 web 工程到 Tomcat 中

第一种部署方法：只需要把 web 工程的目录拷贝到 Tomcat 的 webapps 目录下即可。

第二种部署方法：

找到 Tomcat 下的 conf 目录\ Catalina\localhost\ 下, 创建如下的配置文件：



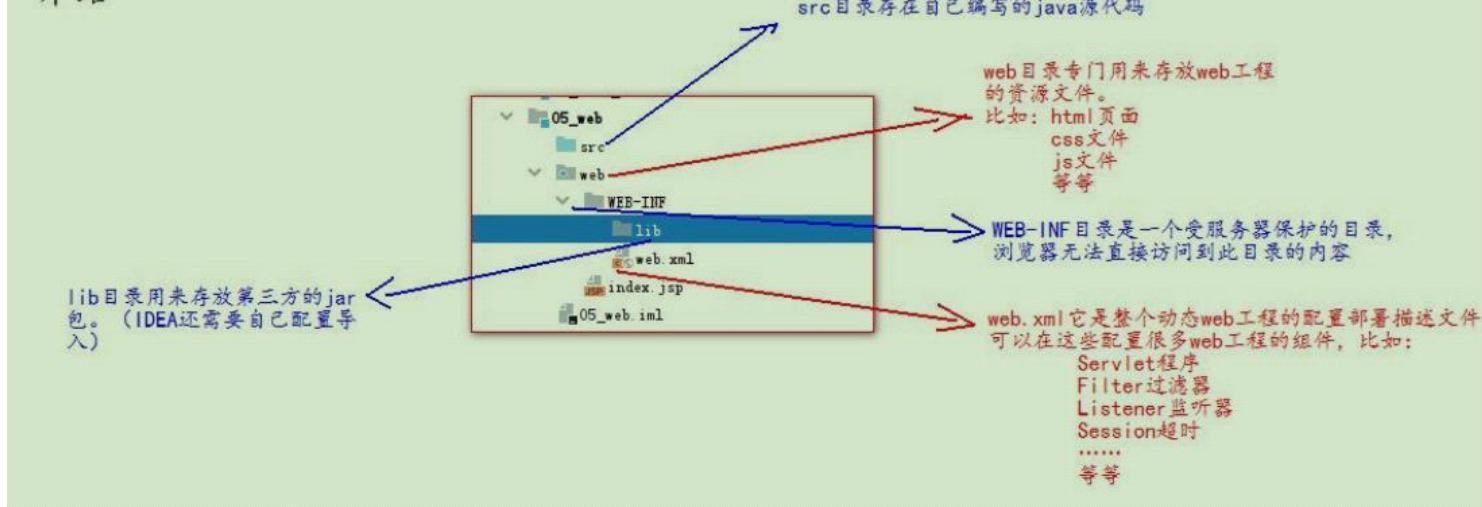
abc.xml 配置文件内容如下：

```
<!-- Context 表示一个工程上下文  
      path 表示工程的访问路径:/abc  
      docBase 表示你的工程目录在哪里  
-->  
<Context path="/abc" docBase="E:\book" />
```

访问这个工程的路径如下:<http://ip:port/abc/> 就表示访问 E:\book 目录

创建动态工程

整个动态WEB工程目录的介绍



servlet技术

2022年1月14日 21:15

a)什么是 Servlet

- 1、Servlet 是 JavaEE 规范之一。规范就是接口
- 2、Servlet 就 JavaWeb 三大组件之一。三大组件分别是：Servlet 程序、Filter 过滤器、Listener 监听器。
- 3、Servlet 是运行在服务器上的一个 java 小程序，它可以接收客户端发送过来的请求，并响应数据给客户端。

b)手动实现 Servlet 程序

- 1、编写一个类去实现 Servlet 接口
- 2、实现 service 方法，处理请求，并响应数据
- 3、到 web.xml 中去配置 servlet 程序的访问地址

接口中实现servlet程序

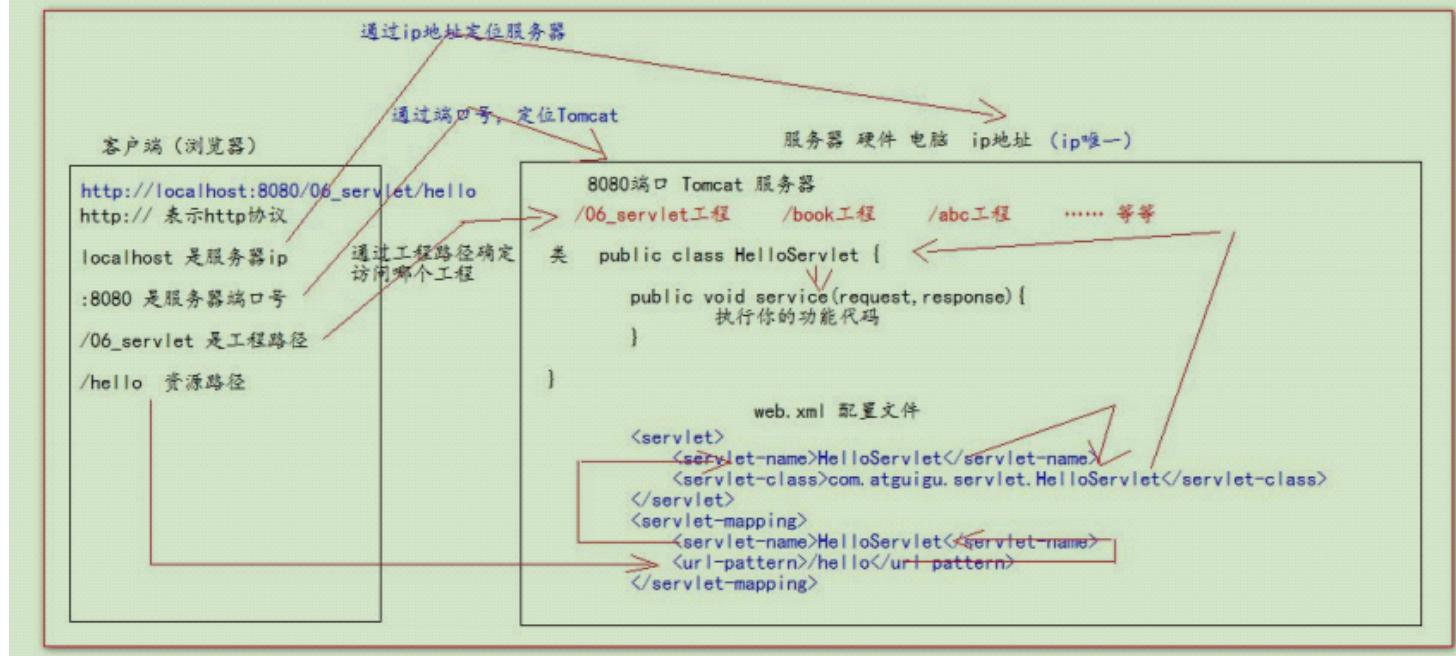
```
public class HelloServlet implements Servlet {
    /**
     * service 方法是专门用来处理请求和响应的
     * @param servletRequest
     * @param servletResponse
     * @throws ServletException
     * @throws IOException
     */
    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
        System.out.println("Hello Servlet 被访问了");
    }
}
```

配置网址

web.xml 中的配置：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
    version="4.0">
    <!-- servlet 标签给 Tomcat 配置 Servlet 程序 -->
    <servlet>
        <!--servlet-name 标签 Servlet 程序起一个别名（一般是类名） -->
        <servlet-name>HelloServlet</servlet-name>
        <!--servlet-class 是 Servlet 程序的全类名-->
        <servlet-class>com.atguigu.servlet.HelloServlet</servlet-class>
    </servlet>
    <!--servlet-mapping 标签给 servlet 程序配置访问地址-->
    <servlet-mapping>
        <!--servlet-name 标签的作用是告诉服务器，我当前配置的地址给哪个 Servlet 程序使用-->
        <servlet-name>HelloServlet</servlet-name>
        <!--url-pattern 标签配置访问地址 <br/>
            / 斜杠在服务器解析的时候，表示地址为：http://ip:port/工程路径 <br/>
            /hello 表示地址为：http://ip:port/工程路径/hello <br/>
        -->
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>
</web-app>
```

c) url 地址到 Servlet 程序的访问



d)Servlet 的生命周期

- 1、执行 Servlet 构造器方法
- 2、执行 init 初始化方法

第一步、二步，是在第一次访问，的时候创建 Servlet 程序会调用。

- 3、执行 service 方法

第三步，每次访问都会调用。

- 4、执行 destroy 销毁方法

第四步，在 web 工程停止的时候调用。

获取请求的类型

```
public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws  
ServletException, IOException {  
    System.out.println("3 service === Hello Servlet 被访问了");  
    // 类型转换 (因为它有getMethod()方法)  
    HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;  
    // 获取请求的方式  
    String method = httpServletRequest.getMethod();  
  
    if ("GET".equals(method)) {  
        doGet();  
    } else if ("POST".equals(method)) {  
        doPost();  
    }  
}
```

f) 通过继承 HttpServlet 实现 Servlet 程序

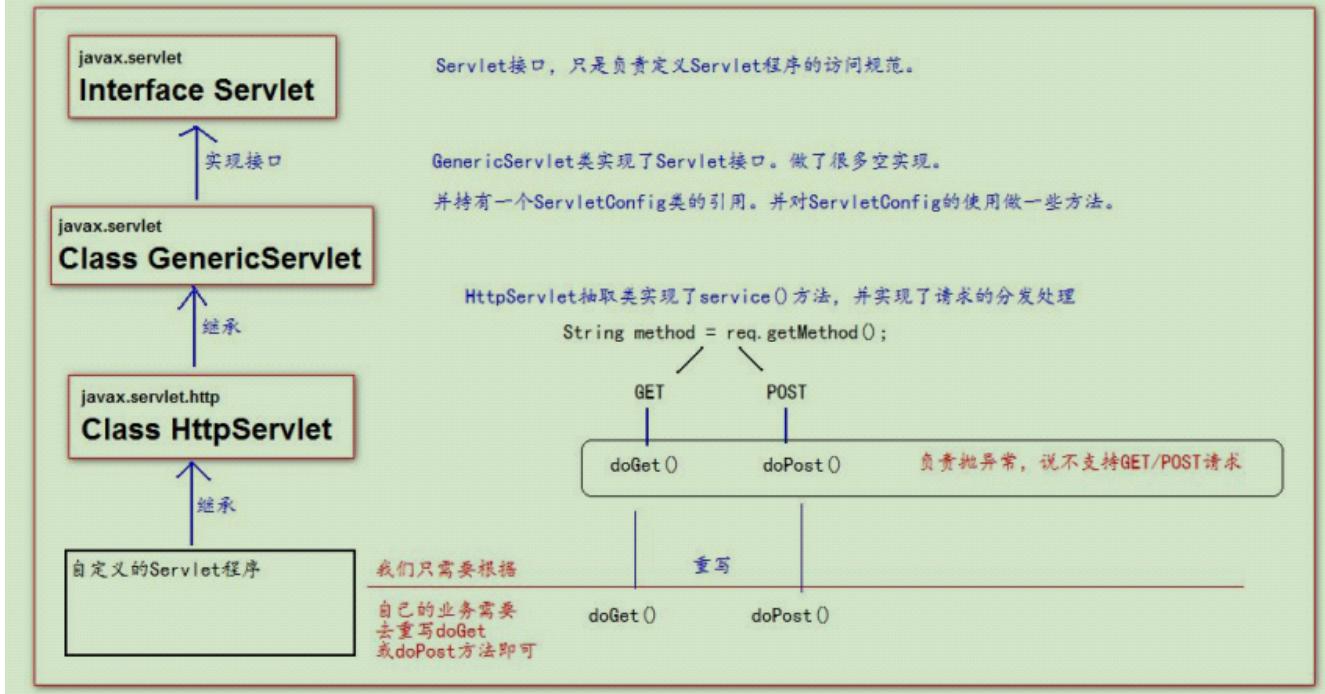
一般在实际项目开发中，都是使用继承 HttpServlet 类的方式去实现 Servlet 程序。

- 1、编写一个类去继承 HttpServlet 类
- 2、根据业务需要重写 doGet 或 doPost 方法
- 3、到 web.xml 中的配置 Servlet 程序的访问地址

g) 使用 IDEA 创建 Servlet 程序

菜单: new ->Servlet 程序

h)Servlet 类的继承体系



2.ServletConfig 类

ServletConfig 类从类名上来看, 就知道是 Servlet 程序的配置信息类。

Servlet 程序和 ServletConfig 对象都是由 Tomcat 负责创建, 我们负责使用。

Servlet 程序默认是第一次访问的时候创建, ServletConfig 是每个 Servlet 程序创建时, 就创建一个对应的 ServletConfig 对象。

a)ServletConfig 类的三大作用

- 1、可以获取 Servlet 程序的别名 servlet-name 的值
- 2、获取初始化参数 init-param
- 3、获取 ServletContext 对象

在web.xml中的配置

```

<!--init-param 是初始化参数-->
<init-param>
    <!-- 是参数名-->
    <param-name>username</param-name>
    <!-- 是参数值-->
    <param-value>root</param-value>
</init-param>
<!--init-param 是初始化参数-->
<init-param>
    <!-- 是参数名-->
    <param-name>url</param-name>
    <!-- 是参数值-->
    <param-value>jdbc:mysql://localhost:3306/test</param-value>
</init-param>

```

一对对配置

1、可以获取 Servlet 程序的别名 `servlet-name` 的值
`System.out.println("HelloServlet 程序的别名是:" + servletConfig.getServletName());`

2、获取初始化参数 `init-param`
`System.out.println("初始化参数 username 的值是;" + servletConfig.getInitParameter("username"))`
`System.out.println("初始化参数 url 的值是;" + servletConfig.getInitParameter("url"));`

3、获取 `ServletContext` 对象
`System.out.println(servletConfig.getServletContext());`

a)什么是 `ServletContext`?

- 1、`ServletContext` 是一个接口，它表示 Servlet 上下文对象
- 2、一个 web 工程，只有一个 `ServletContext` 对象实例。
- 3、`ServletContext` 对象是一个域对象。
- 4、`ServletContext` 是在 web 工程部署启动的时候创建。在 web 工程停止的时候销毁。

什么是域对象？

域对象，是可以像 Map 一样存取数据的对象，叫域对象。
 这里的域指的是存取数据的操作范围，整个 web 工程。

	存数据	取数据	删除 数据
<code>Map</code>	<code>put()</code>	<code>get()</code>	<code>remove()</code>
域对象	<code>setAttribute()</code>	<code>getAttribute()</code>	<code>removeAttribute();</code>

b)`ServletContext` 类的四个作用

- 1、获取 `web.xml` 中配置的上下文参数 `context-param`
- 2、获取当前的工程路径，格式：/工程路径
- 3、获取工程部署后在服务器硬盘上的绝对路径
- 4、像 Map 一样存取数据

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
    // 获取ServletContext 对象  
    ServletContext context = getServletContext();  
    System.out.println(context);  
    System.out.println("保存之前: Context1 获取 key1 的值是:" + context.getAttribute("key1"));  
  
    context.setAttribute("key1", "value1");  
  
    System.out.println("Context1 中获取域数据 key1 的值是:" + context.getAttribute("key1"));  
}
```

HTTP

i. GET 请求

- 1、请求行
(1) 请求的方式 GET
(2) 请求的资源路径 [+?+请求参数]
(3) 请求的协议的版本号 HTTP/1.1

2、请求头
key : value 组成 不同的键值对，表示不同的

下面的内容，就是GET请求的HTTP协议内容

请求行:

1、请求的方式
2、请求的资源路径
3、请求的协议的版本号

GET /06_servlet/a.html HTTP/1.1

Accept: application/x-ms-application, image/jpeg, application/xaml+xml, image/gif, image/pjpeg, application/x-ms-xbap, */*

Accept-Language: zh-CN

User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64; Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0)

UA-CPU: AMD64

Accept-Encoding: gzip, deflate

Host: localhost:8080

Connection: Keep-Alive

请求头

Accept: 告诉服务器，客户端可以接收的数据类型
Accept-Language: 告诉服务器客户端可以接收的语言类型
zh_CN 中文中国
en_US 英文美国

User-Agent: 就是浏览器的信息

Accept-Encoding: 告诉服务器，客户端可以接收的数据编码（压缩）格式

Host: 表示请求的服务器ip和端口号

Connection: 告诉服务器请求连接如何处理
Keep-Alive 告诉服务器回传数据不要马上关闭，保持一小段时间的连接
Closed 马上关闭

c) 响应的 HTTP 协议格式

1、响应行

- (1) 响应的协议和版本号
- (2) 响应状态码
- (3) 响应状态描述符

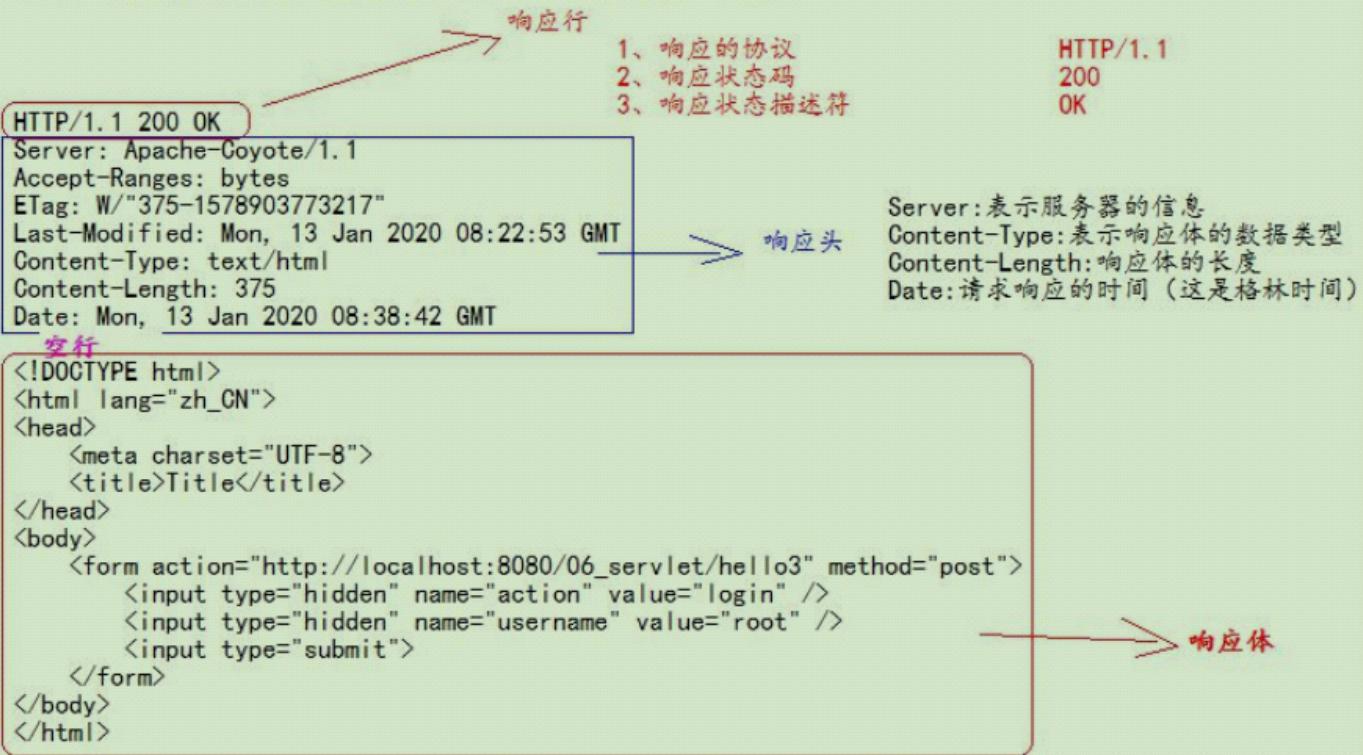
2、响应头

- (1) key : value 不同的响应头，有其不同含义

空行

3、响应体 ---->>> 就是回传给客户端的数据

以下内容就是响应的HTTP协议示例：



d) 常用的响应码说明

200	表示请求成功
302	表示请求重定向（明天讲）
404	表示请求服务器已经收到了，但是你要的数据不存在（请求地址错误）
500	表示服务器已经收到请求，但是服务器内部错误（代码错误）

服务器的接受

1.HttpServletRequest 类

a) HttpServletRequest 类有什么作用。

每次只要有请求进入 Tomcat 服务器，Tomcat 服务器就会把请求过来的 HTTP 协议信息解析好封装到 Request 对象中。然后传递到 service 方法（doGet 和 doPost）中给我们使用。我们可以通过 HttpServletRequest 对象，获取到所有请求的信息。

b) HttpServletRequest 类的常用方法

i. getRequestURI()	获取请求的资源路径
ii. getRequestURL()	获取请求的统一资源定位符（绝对路径）
iii. getRemoteHost()	获取客户端的 ip 地址
iv. getHeader()	获取请求头
v. getParameter()	获取请求的参数
vi. getParameterValues()	获取请求的参数（多个值的时候使用）
vii. getMethod()	获取请求的方式 GET 或 POST
viii. setAttribute(key, value);	设置域数据
ix.getAttribute(key);	获取域数据
x. getRequestDispatcher()	获取请求转发对象

c) 如何获取请求参数

表单：

```
<body>
<form action="http://localhost:8080/07_servlet/parameterServlet" method="get">
    用户名: <input type="text" name="username"><br/>
    密码: <input type="password" name="password"><br/>
    兴趣爱好: <input type="checkbox" name="hobby" value="cpp">C++
              <input type="checkbox" name="hobby" value="java">Java
              <input type="checkbox" name="hobby" value="js">JavaScript<br/>
              <input type="submit">
</form>
</body>
```

```
@Override  
protected void doGet(HttpServletRequest req, HttpServletResponse res)  
throws IOException {  
    // 获取请求参数  
    String username = req.getParameter("username");  
    String password = req.getParameter("password");  
    String[] hobby = req.getParameterValues("hobby");
```

doGet 请求的中文乱码解决：

```
// 获取请求参数  
String username = req.getParameter("username");  
  
//1 先以 iso8859-1 进行编码  
//2 再以 utf-8 进行解码  
username = new String(username.getBytes("iso-8859-1"), "UTF-8");
```

d)POST 请求的中文乱码解决

```
@Override  
protected void doPost(HttpServletRequest req, HttpServletResponse res)  
throws IOException {  
    // 设置请求体的字符集为UTF-8，从而解决post 请求的中文乱码问题  
    req.setCharacterEncoding("UTF-8");  
    System.out.println("-----doPost-----");  
    // 获取请求参数  
    String username = req.getParameter("username");  
    String password = req.getParameter("password");  
    String[] hobby = req.getParameterValues("hobby");
```

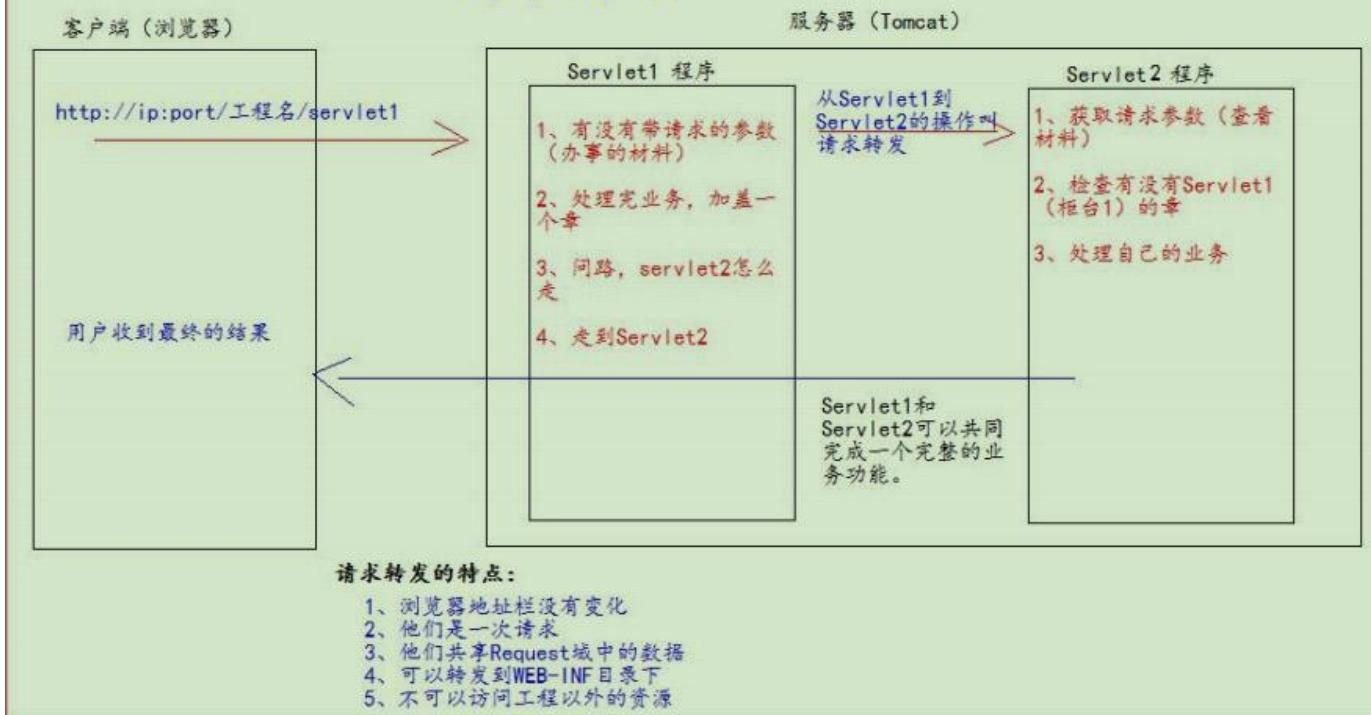
网页转发，跳转

e)请求的转发

什么是请求的转发？

请求转发是指，服务器收到请求后，从一次资源跳转到另一个资源的操作叫请求转发。

请求转发



eg1: (页面跳转)

请求重定向的第一种方案:

```
// 设置响应状态码 302 , 表示重定向, (已搬迁)
resp.setStatus(302);
// 设置响应头, 说明 新的地址在哪里
resp.setHeader("Location", "http://localhost:8080");
```

请求重定向的第二种方案 (推荐使用) :

```
resp.sendRedirect("http://localhost:8080");
```

eg2: (转发)

```
RequestDispatcher requestDispatcher = req.getRequestDispatcher("/servlet2");
// RequestDispatcher requestDispatcher = req.getRequestDispatcher("http://www.baidu.com");

// 走向Servlet2 (柜台 2)
requestDispatcher.forward(req, resp);
```

即把req转到了servlet2

服务器的回应

a) HttpServletResponse 类的作用

HttpServletResponse 类和 HttpServletRequest 类一样。每次请求进来，Tomcat 服务器都会创建一个 Response 对象传递给 Servlet 程序去使用。HttpServletRequest 表示请求过来的信息，HttpServletResponse 表示所有响应的信息，

我们如果需要设置返回给客户端的信息，都可以通过 HttpServletResponse 对象来进行设置

b)两个输出流的说明。

字节流	getOutputStream();	常用于下载（传递二进制数据）
字符流	getWriter();	常用于回传字符串（常用）

两个流同时只能使用一个。

使用了字节流，就不能再使用字符流，反之亦然，否则就会报错。

要求：往客户端回传 字符串 数据。

```
public class ResponseIOServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse res
    IOException {
        // 要求：往客户端回传 字符串 数据。
        PrintWriter writer = resp.getWriter();
        writer.write("response's content!!!");
    }
}
```

解决响应中文乱码方案二（推荐）：

```
// 它会同时设置服务器和客户端都使用UTF-8 字符集，还设置了响应头
// 此方法一定要在获取流对象之前调用才有效
resp.setContentType("text/html; charset=UTF-8");
```

Servlet中的初始化方法有两个

init() , init(config)

其中带参数的方法代码如下：

```
public void init(ServletConfig config) throws ServletException {
    this.config = config;
    init();
}
```

另外一个无参的init方法如下：

```
public void init() throws ServletException{  
}
```

如果我们想要在Servlet初始化时做一些准备工作，那么我们可以重写init方法

我们可以通过如下步骤去获取初始化设置的数据

- 获取config对象： ServletConfig config = getServletConfig();
- 获取初始化参数值： config.getInitParameter(key);

```
public class Demo01Servlet extends HttpServlet {  
    @Override  
    public void init() throws ServletException {  
        ServletConfig config = getServletConfig();  
        String initValue = config.getInitParameter("hello");  
        System.out.println("initValue = " + initValue);  
    }  
}
```

将获取到 web.xml 中

```
<servlet>  
    <servlet-name>Demo01Servlet</servlet-name>  
    <servlet-class>servlet.Demo01Servlet</servlet-class>  
    <init-param>  
        <param-name>hello</param-name>  
        <param-value>world</param-value>  
    </init-param>  
</servlet>
```

```
    ServletContext servletContext = getServletContext();  
    String contextConfigLocation = servletContext.getInitParameter("contextConfigLocation");  
    System.out.println("contextConfigLocation = " + contextConfigLocation);  
}
```

将获取到 web.xml 中

```
<context-param>  
    <param-name>contextConfigLocation</param-name>  
    <param-value>classpath:applicationContext.xml</param-value>  
</context-param>
```

数据库初步

2022年1月18日 16:53

[链接](#)

方式二：通过命令行方式

启动： net start mysql服务名

停止： net stop mysql服务名

net start mysql_eric

◆登录

mysql -h 主机名 -u用户名 -p密码

◆退出

exit

数据类型

MySQL 创建数据表

创建MySQL数据表需要以下信息：

- 表名
- 表字段名
- 定义每个表字段

语法

以下为创建MySQL数据表的SQL通用语法：

```
CREATE TABLE table_name (column_name column_type);
```

以下例子中我们将在 RUNOOB 数据库中创建数据表runoob_tbl：

```
CREATE TABLE IF NOT EXISTS `runoob_tbl`(  
    `runoob_id` INT UNSIGNED AUTO_INCREMENT,  
    `runoob_title` VARCHAR(100) NOT NULL,  
    `runoob_author` VARCHAR(40) NOT NULL,  
    `submission_date` DATE,  
    PRIMARY KEY ( `runoob_id` )  
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

实例解析：

- 如果你不希望字段为 **NULL** 可以设置字段的属性为 **NOT NULL**，在操作数据库时如果输入该字段的数据为**NULL**，就会报错。
- **AUTO_INCREMENT** 定义列为自增的属性，一般用于主键，数值会自动加1。
- **PRIMARY KEY** 关键字用于定义列为主键。您可以使用多列来定义主键，列间以逗号分隔。
- **ENGINE** 设置存储引擎，**CHARSET** 设置编码。

MySQL 删除数据表

MySQL中删除数据表是非常容易操作的，但是你在进行删除表操作时要非常小心。

语法

以下为删除MySQL数据表的通用语法：

```
DROP TABLE table_name ;
```

在命令提示窗口中删除数据表

在mysql>命令提示窗口中删除数据表SQL语句为 DROP TABLE :

实例

以下实例删除了数据表runoob_tbl:

```
root@host# mysql -u root -p
Enter password:*****
mysql> use RUNOOB;
Database changed
mysql> DROP TABLE runoob_tbl;
Query OK, 0 rows affected (0.8 sec)
mysql>
```

MySQL 插入数据

MySQL 表中使用 **INSERT INTO** SQL语句来插入数据。

你可以通过 mysql> 命令提示窗口中向数据表中插入数据，或者通过PHP脚本来插入数据。

语法

以下为向MySQL数据表插入数据通用的 **INSERT INTO** SQL语法:

```
INSERT INTO table_name ( field1, field2,...fieldN )
VALUES
( value1, value2,...valueN );
```

如果数据是字符型，必须使用单引号或者双引号，如: "value"。

通过命令提示窗口插入数据

以下我们将使用 SQL **INSERT INTO** 语句向 MySQL 数据表 runoob_tbl 插入数据

实例

以下实例中我们将向 runoob_tbl 表插入三条数据:

```
root@host# mysql -u root -p password;
Enter password:*****
mysql> use RUNOOB;
Database changed
mysql> INSERT INTO runoob_tbl
-> (runoob_title, runoob_author, submission_date)
-> VALUES
-> ("学习 PHP", "菜鸟教程", NOW());
Query OK, 1 rows affected, 1 warnings (0.01 sec)

mysql> INSERT INTO runoob_tbl
-> (runoob_title, runoob_author, submission_date)
-> VALUES
-> ("学习 MySQL", "菜鸟教程", NOW());
Query OK, 1 rows affected, 1 warnings (0.01 sec)

mysql> INSERT INTO runoob_tbl
-> (runoob_title, runoob_author, submission_date)
-> VALUES
-> ("JAVA 教程", "RUNOOB.COM", '2016-05-06');
Query OK, 1 rows affected (0.00 sec)

mysql>
```

MySQL 查询数据

MySQL 数据库使用SQL SELECT语句来查询数据。

你可以通过 mysql> 命令提示窗口中在数据库中查询数据，或者通过PHP脚本来查询数据。

语法

以下为在MySQL数据库中查询数据通用的 SELECT 语法:

```
SELECT column_name,column_name  
FROM table_name  
[WHERE Clause]  
[LIMIT N][ OFFSET M]
```

- 查询语句中你可以使用一个或者多个表，表之间使用逗号(,)分割，并使用WHERE语句来设定查询条件。
- SELECT 命令可以读取一条或者多条记录。
- 你可以使用星号 (*) 来代替其他字段，SELECT语句会返回表的所有字段数据
- 你可以使用 WHERE 语句来包含任何条件。
- 你可以使用 LIMIT 属性来设定返回的记录数。
- 你可以通过OFFSET指定SELECT语句开始查询的数据偏移量。默认情况下偏移量为0。

MySQL WHERE 子句

我们知道从 MySQL 表中使用 SQL SELECT 语句来读取数据。

如需有条件地从表中选取数据，可将 WHERE 子句添加到 SELECT 语句中。

语法

以下是 SQL SELECT 语句使用 WHERE 子句从数据表中读取数据的通用语法:

```
SELECT field1, field2,...fieldN FROM table_name1, table_name2...  
[WHERE condition1 [AND [OR]] condition2....]
```

eg

```
SELECT last_name, salary  
FROM employees  
WHERE salary <= 3000;
```

LAST_NAME	SALARY
Matos	2600
Vargas	2500

其它比较运算

操作符	含义
BETWEEN ... AND ...	在两个值之间 (包含边界)
IN (set)	等于值列表中的一个
LIKE	模糊查询
IS NULL	空值

BETWEEN

使用 BETWEEN 运算来显示在一个区间内的值

```
SELECT last_name, salary  
FROM employees  
WHERE salary BETWEEN 2500 AND 3500;
```

Lower limit Upper limit

MySQL LIKE 子句

我们知道在 MySQL 中使用 SQL SELECT 命令来读取数据，同时我们可以在 SELECT 语句中使用 WHERE 子句来获取指定的记录。

WHERE 子句中可以使用等号 = 来设定获取数据的条件，如 "runoob_author = 'RUNOOB.COM'"。

但是有时候我们需要获取 runoob_author 字段含有 "COM" 字符的所有记录，这时我们就需要在 WHERE 子句中使用 SQL LIKE 子句。

SQL LIKE 子句中使用百分号 % 字符来表示任意字符，类似于UNIX或正则表达式中的星号 *。

如果没有使用百分号 %, LIKE 子句与等号 = 的效果是一样的。

语法

以下是 SQL SELECT 语句使用 LIKE 子句从数据表中读取数据的通用语法：

```
SELECT field1, field2,...fieldN  
FROM table_name  
WHERE field1 LIKE condition1 [AND [OR]] filed2 = 'somevalue'
```

```
mysql> use RUNOOB;
Database changed
mysql> SELECT * from runoob_tbl WHERE runoob_author LIKE '%COM';
+-----+-----+-----+-----+
| runoob_id | runoob_title | runoob_author | submission_date |
+-----+-----+-----+-----+
| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |
| 4 | 学习 Python | RUNOOB.COM | 2016-03-06 |
+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

MySQL UNION 操作符

本教程为大家介绍 MySQL UNION 操作符的语法和实例。

描述

MySQL UNION 操作符用于连接两个以上的 SELECT 语句的结果组合到一个结果集合中。多个 SELECT 语句会删除重复的数据。

语法

MySQL UNION 操作符语法格式：

```
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions]
UNION [ALL | DISTINCT]
SELECT expression1, expression2, ... expression_n
FROM tables
[WHERE conditions];
```

SQL UNION 实例

下面的 SQL 语句从 "Websites" 和 "apps" 表中选取所有不同的country (只有不同的值) :

实例

```
SELECT country FROM Websites
UNION
SELECT country FROM apps
ORDER BY country;
```

MySQL 排序

我们知道从 MySQL 表中使用 SQL SELECT 语句来读取数据。

如果我们需要对读取的数据进行排序，我们就可以使用 MySQL 的 ORDER BY 子句来设定结果。

语法

以下是 SQL SELECT 语句使用 ORDER BY 子句将查询数据排序后再返回数据：

```
SELECT field1, field2,...fieldN FROM table_name1, table_name2...
ORDER BY field1 [ASC | DESC][默认 ASC], [field2...] [ASC | DESC][默认 ASC]
```

```

mysql> use RUNOOB;
Database changed
mysql> SELECT * from runoob_tbl ORDER BY submission_date ASC;
+-----+-----+-----+-----+
| runoob_id | runoob_title | runoob_author | submission_date |
+-----+-----+-----+-----+
| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |
| 4 | 学习 Python | RUNOOB.COM | 2016-03-06 |
| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 |
| 2 | 学习 MySQL | 菜鸟教程 | 2017-04-12 |
+-----+-----+-----+-----+
4 rows in set (0.01 sec)

mysql> SELECT * from runoob_tbl ORDER BY submission_date DESC;
+-----+-----+-----+-----+
| runoob_id | runoob_title | runoob_author | submission_date |
+-----+-----+-----+-----+
| 1 | 学习 PHP | 菜鸟教程 | 2017-04-12 |
| 2 | 学习 MySQL | 菜鸟教程 | 2017-04-12 |
| 4 | 学习 Python | RUNOOB.COM | 2016-03-06 |
| 3 | 学习 Java | RUNOOB.COM | 2015-05-01 |
+-----+-----+-----+-----+
4 rows in set (0.01 sec)

```

键表约束

--主键约束;
它能够唯一确定一张表中的一条记录，也就是我们通过给某个字段添加约束，就可以使得该字段不重复，且不能为空。
有且只有一个

```

create table user (
id int primary key,
name varchar(20)
);

```

--联合主键;
--只要联合主键值加起来不重复就可以,也不可以为空;

```

mysql> create table user2(
-> id int,
-> name varchar(20),
-> password varchar(20),
-> primary key(id,name)
-> );

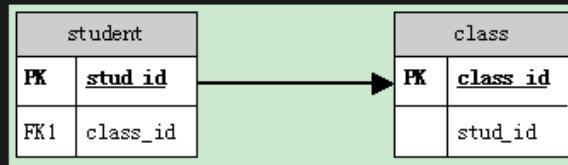
```

```
--自增约束
```

```
mysql> create table user3(
-> id int primary key auto_increment,
-> name varchar(20)
-> );
```

外键

如果一个实体的某个字段指向另一个实体的主键，就称为外键
被指向的实体，称之为**主实体（主表）**，也叫**父实体（父表）**。
负责指向的实体，称之为**从实体（从表）**，也叫**子实体（子表）**



例 1

为了展现表与表之间的外键关系，本例在 test_db 数据库中创建一个部门表 tb_dept1，表结构如下表所示。

字段名称	数据类型	备注
id	INT(11)	部门编号
name	VARCHAR(22)	部门名称
location	VARCHAR(22)	部门位置

创建 tb_dept1 的 SQL 语句和运行结果如下所示。

```
mysql> CREATE TABLE tb_dept1
-> (
-> id INT(11) PRIMARY KEY,
-> name VARCHAR(22) NOT NULL,
-> location VARCHAR(50)
-> );
Query OK, 0 rows affected (0.37 sec)
```

创建数据表 tb_emp6，并在表 tb_emp6 上创建外键约束，让它的键 deptId 作为外键关联到表 tb_dept1 的主键 id，SQL 语句和运行结果如下所示。

```
mysql> CREATE TABLE tb_emp6
-> (
-> id INT(11) PRIMARY KEY,
-> name VARCHAR(25),
-> deptId INT(11),
-> salary FLOAT,
-> CONSTRAINT fk_emp_dept1
-> FOREIGN KEY(deptId) REFERENCES tb_dept1(id)
-> );
```

外键名字

MySQL 索引

MySQL索引的建立对于MySQL的高效运行是很重要的，索引可以大大提高MySQL的检索速度。

打个比方，如果合理的设计且使用索引的MySQL是一辆兰博基尼的话，那么没有设计和使用索引的MySQL就是一个人力三轮车。

拿汉语字典的目录页（索引）打比方，我们可以按拼音、笔画、偏旁部首等排序的目录（索引）快速查找到需要的字。

索引分单列索引和组合索引。单列索引，即一个索引只包含单个列，一个表可以有多个单列索引，但这不是组合索引。组合索引，即一个索引包含多个列。

创建索引时，你需要确保该索引是应用在 SQL 查询语句的条件(一般作为 WHERE 子句的条件)。

实际上，索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录。

上面都在说使用索引的好处，但过多的使用索引将会造成滥用。因此索引也会有它的缺点：虽然索引大大提高了查询速度，同时却会降低更新表的速度，如对表进行INSERT、UPDATE和DELETE。因为更新表时，MySQL不仅要保存数据，还要保存一下索引文件。

建立索引会占用磁盘空间的索引文件。

一个索引是存储的表中一个特定列的值数据结构。索引是在表的列上创建。要记住的关键点是索引包含一个表中列的值，并且这些值存储在一个数据结构中。

```
INDEX `suoyin`(`id`) USING BTREE
```

普通索引

创建索引

这是最基本的索引，它没有任何限制。它有以下几种创建方式：

```
CREATE INDEX indexName ON table_name (column_name)
```

如果是CHAR, VARCHAR类型，length可以小于字段实际长度；如果是BLOB和TEXT类型，必须指定 length。

修改表结构(添加索引)

```
ALTER table tableName ADD INDEX indexName(columnName)
```

创建表的时候直接指定

```
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    INDEX [indexName] (username(length))
);
```

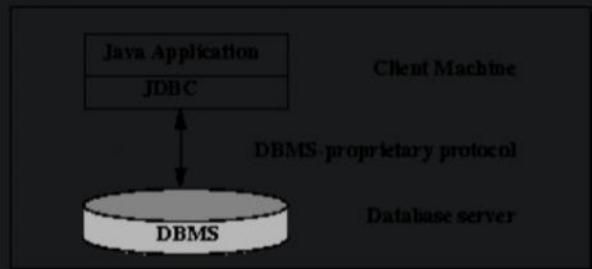
删除索引的语法

```
DROP INDEX [indexName] ON mytable;
```

JDBC 架构

分为双层架构和三层架构。

双层



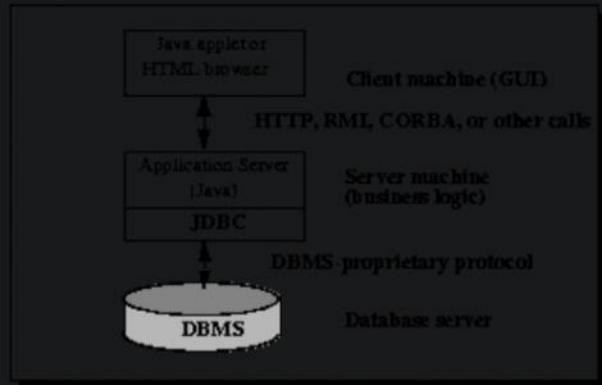
作用：此架构中，Java Applet 或应用直接访问数据源。

条件：要求 Driver 能与访问的数据库交互。

机制：用户命令传给数据库或其他数据源，随之结果被返回。

部署：数据源可以在另一台机器上，用户通过网络连接，称为 C/S 配置（可以是内联网或互联网）。

三层



该架构特殊之处在于，引入中间层服务。

流程：命令和结构都会经过该层。

吸引：可以增加企业数据的访问控制，以及多种类型的更新；另外，也可简化应用的部署，并在多数情况下有性能优势。

历史趋势：以往，因性能问题，中间层都用 C 或 C++ 编写，随着优化编译器（将 Java 字节码 转为 高效的 特定机器码）和技术的发展，如EJB，Java 开始用于中间层的开发这也让 Java 的优势突显出来，使用 Java 作为服务器代码语言，JDBC 随之被重视。

```
public static void main(String[] args) throws Exception {
    //1.加载驱动程序
    Class.forName("com.mysql.jdbc.Driver");
    //2. 获得数据库连接
    Connection conn = DriverManager.getConnection(URL, USER, PASSWORD);
    //3.操作数据库，实现增删改查
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("SELECT user_name, age FROM imooc_goddess");
    //如果有数据，rs.next()返回true
    while(rs.next()){
        System.out.println(rs.getString("user_name")+" 年龄: "+rs.getInt("age"));
    }
}
```

连接数据库写在static里面，对外给出connection方法

```
public class DbUtil {
    public static final String URL = "jdbc:mysql://localhost:3306/imooc";
    public static final String USER = "liulx";
    public static final String PASSWORD = "123456";
    private static Connection conn = null;
    static{
        try {
            //1.加载驱动程序
            Class.forName("com.mysql.jdbc.Driver");
            //2. 获得数据库连接
            conn = DriverManager.getConnection(URL, USER, PASSWORD);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public static Connection getConnection(){
        return conn;
    }
}
```

dao层

先连接

总体的添加方法，将一个 Goddess类加进去（就是一堆 "private 变量" 的类）

```
public void addGoddess(Goddess g) throws SQLException {
    //获取连接
    Connection conn = DbUtil.getConnection();
    //sql
    String sql = "INSERT INTO imooc_goddess(user_name, sex, age, birthday, email, mobile, " +
        "create_user, create_date, update_user, update_date, isdel)" +
        +"values(" + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + "," + "?" + ")";
    //预编译
    PreparedStatement ptmt = conn.prepareStatement(sql); //预编译SQL，减少sql执行

    //传参
    ptmt.setString(1, g.getUser_name());
    ptmt.setInt(2, g.getSex());
    ptmt.setInt(3, g.getAge());
    ptmt.setDate(4, new Date(g.getBirthday().getTime()));
    ptmt.setString(5, g.getEmail());
    ptmt.setString(6, g.getMobile());
    ptmt.setString(7, g.getCreate_user());
    ptmt.setString(8, g.getUpdate_user());
    ptmt.setInt(9, g.getIsDel());

    //执行
    ptmt.execute();
}
```

关于函数 `setString(n,"xxx")`，即把 "xxx" 添加到第 n 个 "?" 处

阿里巴巴的库druid

静态配置

```
static {
    try {
        Properties properties = new Properties();
        // 读取 jdbc.properties属性配置文件
        InputStream inputStream = JdbcUtils.class.getClassLoader().getResourceAsStream("jdbc.properties");
        // 从流中加载数据
        properties.load(inputStream);
        // 创建 数据库连接池
        dataSource = (DruidDataSource) DruidDataSourceFactory.createDataSource(properties);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

连接

```
public static Connection getConnection(){
    Connection conn = null;

    try {
        conn = dataSource.getConnection();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return conn;
}
```

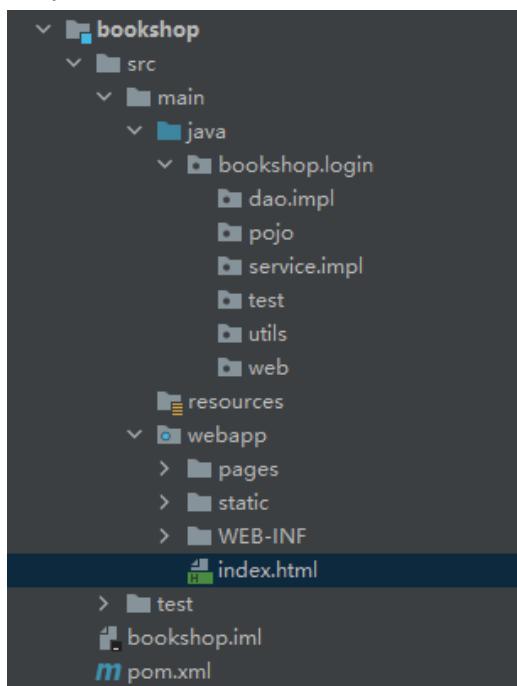
断开

```
public static void close(Connection conn){
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

第一个小项目--书城

2022年1月16日 17:16

框架

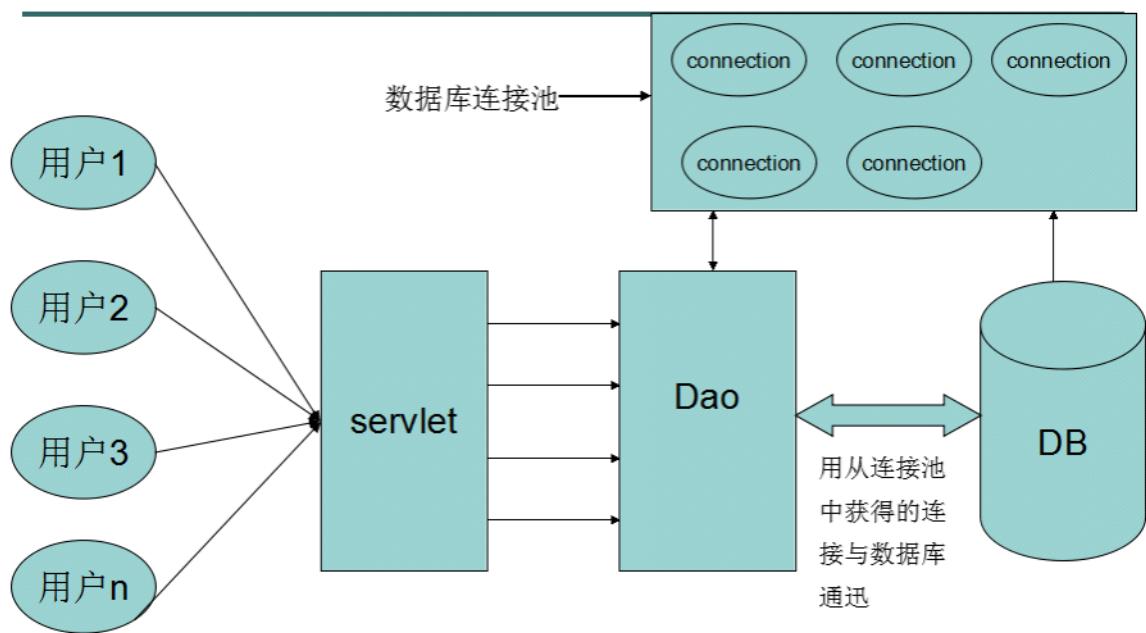


1、先创建书城需要的数据库和表。

```
create table t_user(
    `id` int primary key auto_increment,
    `username` varchar(20) not null unique,
    `password` varchar(32) not null,
    `email` varchar(200)
);
```

2、编写数据库表对应的 **JavaBean** 对象。

```
public class User {
    private Integer id;
    private String username;
    private String password;
    private String email;
```



3、编写工具类 **JdbcUtils**

3.1、导入需要的 jar 包（数据库和连接池需要）：

```
druid-1.1.9.jar
mysql-connector-java-5.1.7-bin.jar
```

目的：连接数据库

```
public class JdbcUtils {

    private static DruidDataSource dataSource;

    static {
        try {
            Properties properties = new Properties();
            // 读取 jdbc.properties 属性配置文件
            InputStream inputStream =
JdbcUtils.class.getClassLoader().getResourceAsStream("jdbc.properties");
            // 从流中加载数据
            properties.load(inputStream);
            // 创建 数据库连接 池
            dataSource = (DruidDataSource) DruidDataSourceFactory.createDataSource(properties);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

}
```

```
public static Connection getConnection(){

    Connection conn = null;

    try {
        conn = dataSource.getConnection();
    } catch (Exception e) {
        e.printStackTrace();
    }

    return conn;
}

/**
 * 关闭连接，放回数据库连接池
 * @param conn
 */
public static void close(Connection conn){
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

4、编写 BaseDao

4.1、导入 DBUtils 的 jar 包

commons-dbutils-1.3.jar

4.2、编写 BaseDao：

目的：实现数据库增删改查的接口

QueryRunner

QueryRunner () 方法的使用和总结

DBUtils包所提供的QueryRunner类，是针对数据库链接池的使用，一方面解决了数据库访问过多时造成数据库承受的压力，另一方面也简化了数据查询。

QueryRunner方法 ()；

QueryRunner中一共有6种方法：

- execute (执行 SQL语句)
- batch (批量处理语句)
- insert (执行INSERT语句)
- insertBatch (批量处理INSERT语句)
- query (SQL中 SELECT语句)
- update (SQL中 INSERT, UPDATE, 或 DELETE 语句) (最为常用)
- **ArrayHandler** : 把结果集中的第一行数据转成对象数组。
- **ArrayListHandler** : 把结果集中的每一行数据都转成一个对象数组，再存放到List中。
- **BeanHandler** : 将结果集中的第一行数据封装到一个对应的JavaBean实例中。
- **BeanListHandler** : 将结果集中的每一行数据都封装到一个对应的JavaBean实例中，存放到List里。//重点
- **MapHandler** : 将结果集中的第一行数据封装到一个Map里，key是列名，value就是对应的值。//重点**
- **MapListHandler** : 将结果集中的每一行数据都封装到一个Map里，然后再存放到List
- **ColumnListHandler** : 将结果集中某一列的数据存放到List中。
- **KeyedHandler(name)** : 将结果集中的每一行数据都封装到一个Map里(List)，再把这些map再存到一个map里，其key为指定的列。
- ScalarHandler**:将结果集第一行的某一列放到某个对象中。//重点

.query(Connection conn, String sql, ResultSetHandler<T>rsh, Object...params)

用于查询

.update(Connection conn, String sql, Object... params) 用于增删改

模板语法

```
public <T> T method(T t){  
    // CODE  
    return t;  
}
```

其中<T>是为了定义当前我有一个 范型变量类型，类型名使用T来表示，而第二部分T，表示method这个函数的返回值类型为T，其中的<T>只是为了在函数声明前，定义一种范型；因此下面的函数也是

update方法

```
/**  
 * update() 方法用来执行: Insert\Update\Delete语句  
 *  
 * @return 如果返回-1, 说明执行失败<br/>返回其他表示影响的行数  
 */  
public int update(String sql, Object... args) {  
    Connection connection = jtbc.getConnection();  
    try {  
        return queryRunner.update(connection, sql, args);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        jtbc.close(connection);  
    }  
    return -1;  
}
```

query方法

```
public <T> T queryForOne(Class<T> type, String sql, Object... args) {  
    Connection con = jtbc.getConnection();  
    try {  
        return queryRunner.query(con, sql, new BeanHandler<T>(type), args);  
    } catch (SQLException e) {  
        e.printStackTrace();  
    } finally {  
        jtbc.close(con);  
    }  
    return null;  
}
```

5、编写 UserDao 和测试

UserDao 接口：

目的：实现上面的接口

注册时查看是否已经注册

```
/**  
 * 根据用户名查询用户信息  
 * @param username 用户名  
 * @return 如果返回null,说明没有这个用户。反之亦然  
 */  
public User queryUserByUsername(String username);
```

注册成功保存

```
/**  
 * 保存用户信息  
 * @param user  
 * @return 返回-1 表示操作失败, 其他是sql语句影响的行数  
 */  
public int saveUser(User user);
```

登录时核对账户密码

```
/**  
 * 根据用户名和密码查询用户信息  
 * @param username  
 * @param password  
 * @return 如果返回null,说明用户名或密码错误,反之亦然  
 */  
public User queryUserByUsernameAndPassword(String username, String password);
```

UserDaoImpl 实现类:

```
public class UserDaoImpl extends BaseDao implements UserDao {  
    @Override  
    public User queryUserByUsername(String username) {  
        String sql = "select `id`, `username`, `password`, `email` from t_user where username = ?";  
        return queryForOne(User.class, sql, username);  
    }  
  
    @Override  
    public User queryUserByUsernameAndPassword(String username, String password) {  
        String sql = "select `id`, `username`, `password`, `email` from t_user where username = ? and  
password = ?";  
        return queryForOne(User.class, sql, username, password);  
    }  
}
```

```
@Test  
public void queryUserByUsername() {  
  
    if (userDao.queryUserByUsername("admin1234") == null){  
        System.out.println("用户名可用!");  
    } else {  
        System.out.println("用户名已存在!");  
    }  
}
```

6、编写 UserService 和测试

UserService 接口:

目的：实现用户登录注册

UserServiceImpl 实现类：

```
public class UserServiceTest {

    UserService userService = new UserServiceImpl();

    @Test
    public void registUser() {
        userService.registUser(new User(null, "bbj168", "666666", "bbj168@qq.com"));
        userService.registUser(new User(null, "abc168", "666666", "abc168@qq.com"));
    }

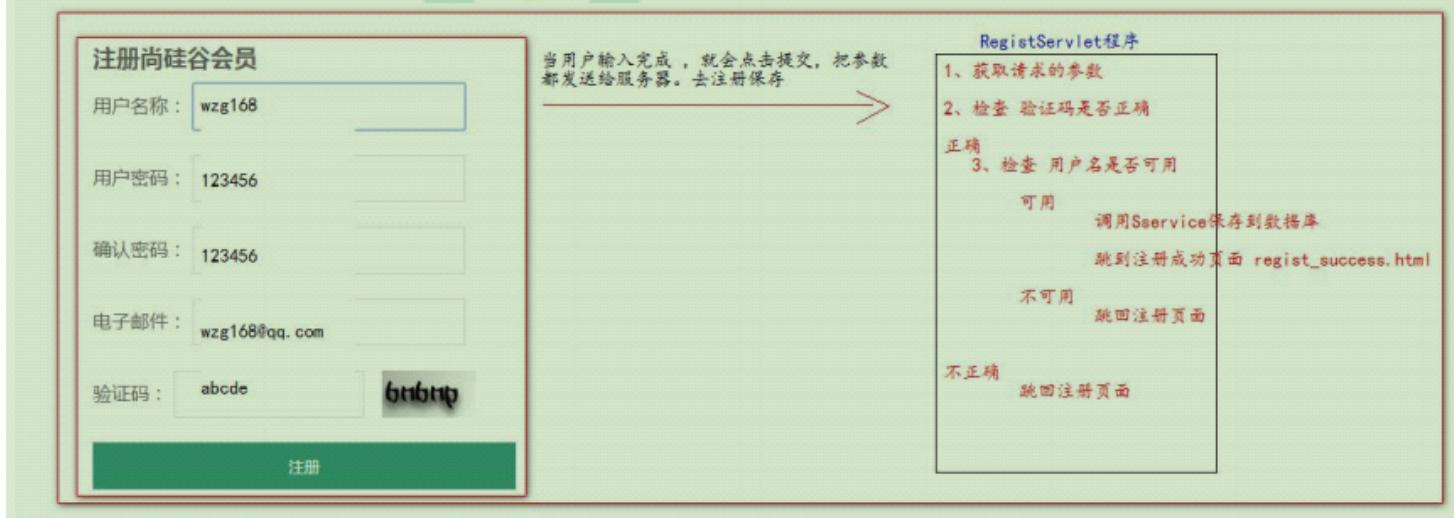
    @Test
    public void login() {
        System.out.println( userService.login(new User(null, "wzg168", "123456", null)) );
    }

    @Test
    public void existsUsername() {
        if (userService.existsUsername("wzg16888")) {
            System.out.println("用户名已存在！");
        } else {
            System.out.println("用户名可用！");
        }
    }
}
```

7、编写 web 层

7.1、实现用户注册的功能

7.1.1、图解用户注册的流程:



7.1.3、编写 RegistServlet 程序

```
public class RegistServlet extends HttpServlet {

    private UserService userService = new UserServiceImpl();

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException,
    IOException {
        // 1. 获取请求的参数
        String username = req.getParameter("username");
        String password = req.getParameter("password");
        String email = req.getParameter("email");
        String code = req.getParameter("code");

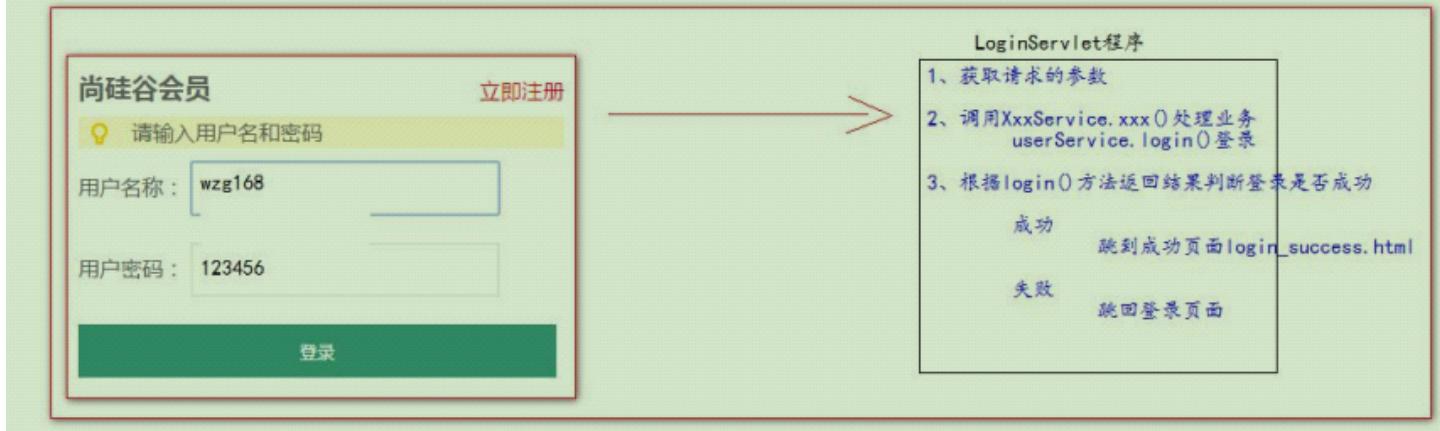
        // 2. 检查 验证码是否正确 === 写死, 要求验证码为: abcde
        if ("abcde".equalsIgnoreCase(code)) {
            // 3. 检查 用户名是否可用
            if (userService.existsUsername(username)) {
                System.out.println("用户名[" + username + "]已存在!");
                // 跳回注册页面
                req.getRequestDispatcher("/pages/user/regist.html").forward(req, resp);
            }
        }
    }
}
```

```
    } else {
        // 可用
        // 调用 Sservice 保存到数据库
        userService.registerUser(new User(null, username, password, email));

        // 跳到注册成功页面 regist_success.html
        req.getRequestDispatcher("/pages/user/regist_success.html").forward(req, resp);
    }
} else {
    System.out.println("验证码[" + code + "]错误");
    req.getRequestDispatcher("/pages/user/regist.html").forward(req, resp);
}
}
}
```

7.3、用户登录功能的实现

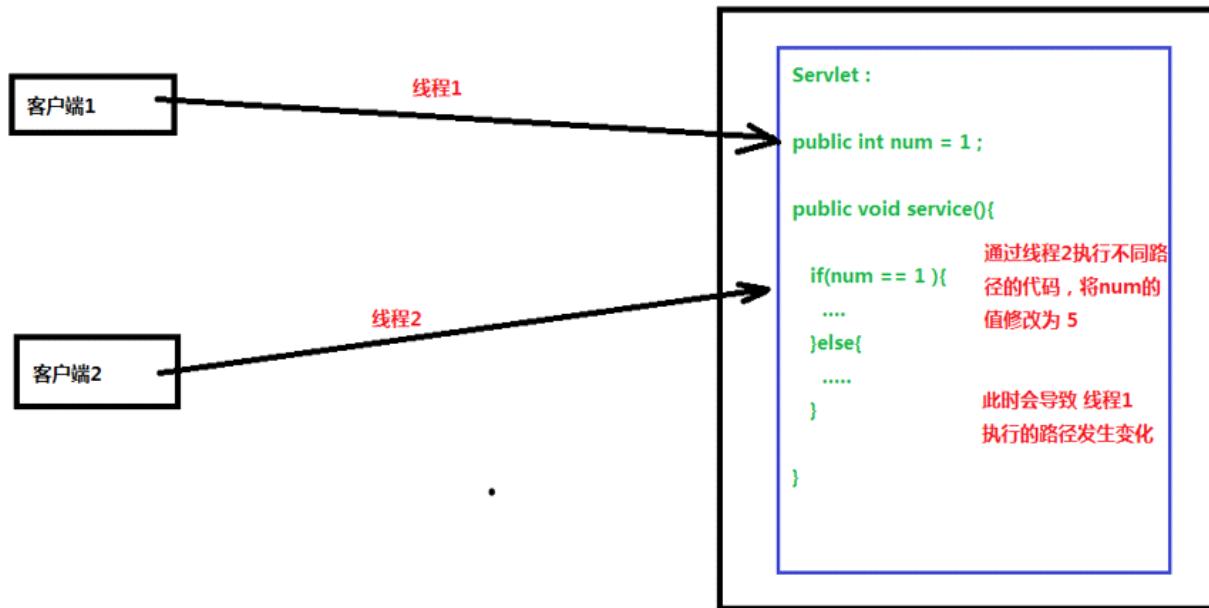
7.3.1、图解用户登录



servlet之session

2022年1月21日 16:29

01.Servlet是线程不安全的



S : 请告诉我你的会话ID
C : 没有！
S : 哦，那我知道了，你是第一次给我发请求，我给你分配一个Session ID : 123

S : 请告诉我你的会话ID
C : 123
S : 哟，我知道了，你是****，上次什么时间访问我的。

S : 请告诉我你的会话ID
C : 345
S : 哟，我知道了，你是***，上次什么时间访问我的

HTTP 无状态：

服务器无法判断这两次请求是同一个客户端发过来的，还是不同的客户端发过来的

通过会话跟踪技术来解决无状态的问题

1 会话跟踪技术

在Servlet规范中，有以下三种机制用于会话跟踪：

1) SSL(安全套接字层)会话：

一种加密技术，主要原理是采用SSL的服务器和客户端之间产生会话密钥，建立一种加密的连接会话。

2) Cookies：

是最常用的跟踪用户会话的方式，Cookie是一种由服务器发送给客户的片段信息，存储在客户端浏览器的内存或硬盘上，客户端在发起请求时携带此信息最为用户的唯一标识。

Cookie以键值对的方式记录会话跟踪的内容，服务器利用响应报头Set-Cookie来发送Cookie信息。在RFC2109中定义的Set-Cookie响应报头的格式为：

Set-Cookie: NAME=VALUE ; Comment=value ; Domain=value ; Max-Age=value ; Path=value ; Secure ; Version=1*DIGIT

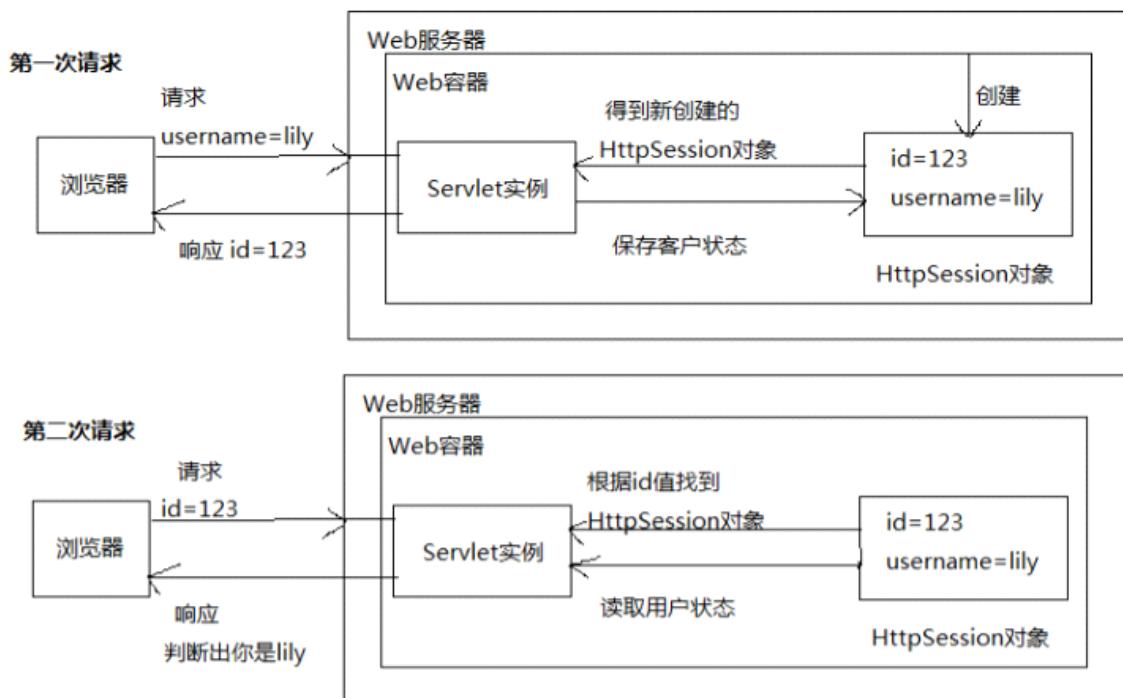
NAME是Cookie的名字，VALUE是Cookie的值。NAME=VALUE 属性-值对必须首先出现，在此之后的属性-值对可以任何顺序出现。在Servlet规范中，用于会话跟踪的Cookie的名字必须是JSESSIONID。

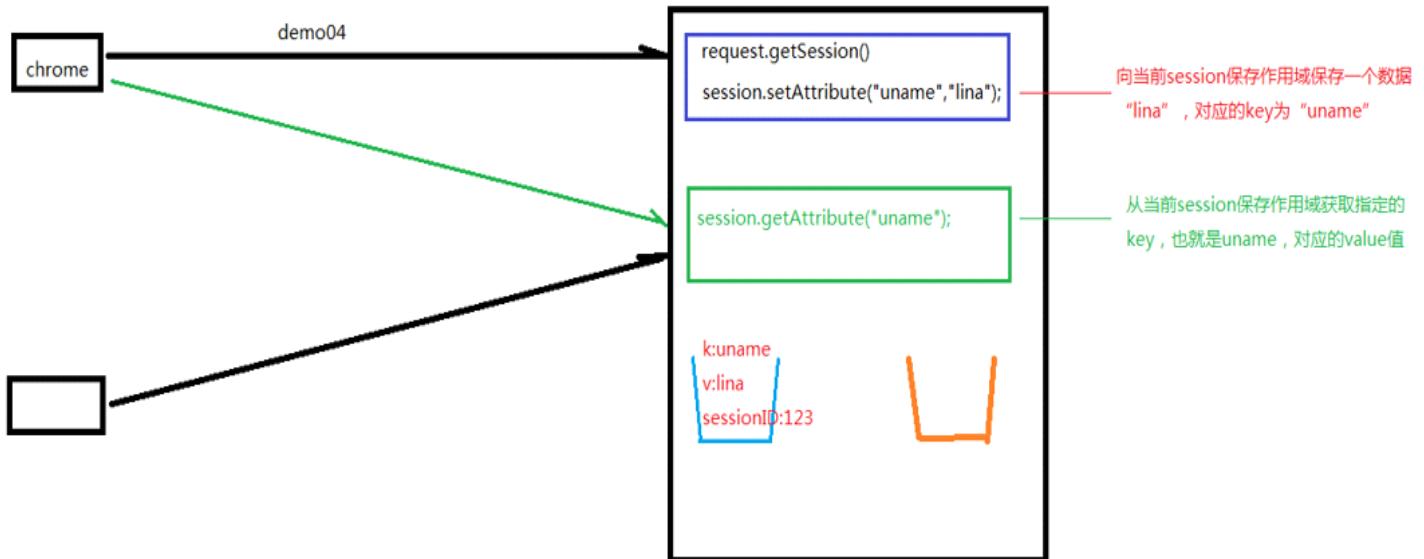
2 Servlet API的会话跟踪

在java Servlet API中，javax.servlet.http.HttpSession接口封装了Session的概念，Servlet容器提供了这个接口的实现。当请求一个会话时，Servlet容器就创建一个 HttpSession 对象，有了这个对象之后，就可以利用这个对象保存客户的状态信息。

比如，购物车，Servlet容器为 HttpSession 对象分配一个唯一的SessionID，将其作为Cookie(或者作为URL的一部分，利用URL重写机制)发送给浏览器，浏览器在内存中保存这个Cookie。当客户再次发送HTTP请求时，浏览器将Cookie随请求一起发送，Servlet容器从请求对象中获取SessionID，然后根据SessionID找到对应的 HttpSession 对象，从而得到客户的状态信息。

整个会话过程对于开发和用户来说都是透明的（由Servlet容器完成），开发人员只需得到 HttpSession 对象，然后调用这个对象的setAttribute()或者getAttribute()方法来保存获取读取客户的状态信息。整个会话跟踪过程如图：





Cookie

```
public void doGet(HttpServletRequest request, HttpServletResponse response) {
```

```
    response.setContentType("text/html;charset=utf-8");
    String id = UUID.randomUUID().toString(); //生成一个随机字符串
    Cookie cookie = new Cookie("id", id); //创建Cookie对象，指定名字和值
    response.addCookie(cookie); //在响应中添加Cookie对象
    response.getWriter().print("已经给你发送了ID");
}
```

```
Cookie[] cs = request.getCookies(); //获取请求中的Cookie
if(cs != null) { //如果请求中存在Cookie
    for(Cookie c : cs) { //遍历所有Cookie
        if(c.getName().equals("id")) { //获取Cookie名字，如果Cookie名字是id
            response.getWriter().print("您的ID是：" + c.getValue()); //打印
        }
    }
}
```

Session

四.cookie 和session 的区别:

- 1、cookie数据存放在客户的浏览器上， session数据放在服务器上。
- 2、cookie不是很安全， 别人可以分析存放在本地的COOKIE并进行COOKIE欺骗
考虑到安全应当使用session。
- 3、session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能
考虑到减轻服务器性能方面，应当使用COOKIE。
- 4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。
- 5、所以个人建议：
将登陆信息等重要信息存放为SESSION
其他信息如果需要保留，可以放在COOKIE中

2) 会话跟踪技术

- 客户端第一次发请求给服务器，服务器获取session，获取不到，则创建新的，然后响应给客户端
- 下次客户端给服务器发请求时，会把sessionId带给服务器，那么服务器就能获取到了，那么服务器就判断这一次请求是老客户了
- 常用的API：
 - request.getSession() -> 获取当前的会话，没有则创建一个新的会话
 - request.getSession(true) -> 效果和不带参数相同
 - request.getSession(false) -> 获得当前会话，没有则返回null，不会创建新的

session.getId() -> 获得sessionId
session.isNew() -> 判断当前session是否是新的
session.getMaxInactiveInterval() -> session的非激活间隔时长，默认1800秒
session.setMaxInactiveInterval()
session.invalidate() -> 强制性让会话立即失效

3) session保存作用域

- session保存作用域是和具体的某一个session对应的
- 常用的API：
 - void session.setAttribute(k,v)
 - Object session.getAttribute(k)
 - void removeAttribute(k)

作用域

javaweb开发中Servlet三大域对象的应用 (request、session、application (ServletContext)) .

1. request

request是表示一个请求，只要发出一个请求就会创建一个request，它的作用域：仅在当前请求中有效。

用处：常用于服务器间同一请求不同页面之间的参数传递，常应用于表单的控件值传递。

方法：request.setAttribute(); request.getAttribute(); request.removeAttribute(); request.getParameter().

2. session

服务器会为每个会话创建一个session对象，所以session中的数据可供当前会话中所有servlet共享。

会话：用户打开浏览器会话开始，直到关闭浏览器会话才会结束。一次会话期间只会创建一个session对象。

用处：常用于web开发中的登陆验证界面（当用户登录成功后浏览器分配其一个session键值对）。

方法：session.setAttribute(); session.getAttribute(); session.removeAttribute();

获得session对象方法：

1. 在Servlet中：HttpSession session = request.getSession();

2. 由于session属于jsp九大内置对象之一，当然可以直接使用。例如：

<%session.setAttribute("name","admin")%>。

session被销毁

- 1)session超时；
- 2)客户端关闭后，再也访问不到和该客户端对应的session了，它会在超时之后被销毁；
- 3)调用session.invalidate();

备注：session是服务器端对象，保存在服务器端。并且服务器可以将创建session后产生的sessionid通过一个cookie返回给客户端，以便下次验证。（session底层依赖于cookie）

3. Application (ServletContext)

作用范围：所有的用户都可以取得此信息，此信息在整个服务器上被保留。Application属性范围值，只要设置一次，则所有的网页窗口都可以取得数据。ServletContext在服务器启动时创建，在服务器关闭时销毁，一个JavaWeb应用只创建一个ServletContext对象，所有的客户端在访问服务器时都共享同一个ServletContext对象；ServletContext对象一般用于在多个客户端间共享数据时使用；

获取Application对象方法（Servlet中）：

```
ServletContext app01 = this.getServletContext();
app01.setAttribute("name", "kaixuan"); //设置一个值进去

ServletContext app02 = this.getServletContext();
app02.getAttribute("name"); //获取键值对
```

ServletContext同属于JSP九大内置对象之一，故可以直接使用

备注：服务器只会创建一个ServletContext 对象，所以app01就是app02，通过app01设置的值当然可以通过app02获取。

request作用域

```
protected void service(HttpServletRequest request, HttpServletResponse response) {
    //1.向request保存作用域保存数据
    request.setAttribute("uname", "lili");
    //2.客户端重定向
    //response.sendRedirect("demo02");

    //3.服务器端转发
    request.getRequestDispatcher("demo02").forward(request, response);
}
```

```
//1.获取request保存作用域保存的数据，key为uname
Object unameObj = request.getAttribute("uname");
System.out.println("unameObj = " + unameObj);
```

session作用域

```
//1.向session保存作用域保存数据
request.getSession().setAttribute("uname", "lili");
//2.客户端重定向
response.sendRedirect("demo04");
```

```
//1.获取session保存作用域保存的数据，key为uname
Object unameObj = request.getSession().getAttribute("uname");
System.out.println("unameObj = " + unameObj);
```

application作用域

```
//1.向application保存作用域保存数据
//ServletContext : Servlet上下文
ServletContext application = request.getServletContext();
application.setAttribute("uname", "lili");
//2.客户端重定向
response.sendRedirect("demo06");
```

```
//1.获取application保存作用域保存的数据，key为uname
ServletContext application = request.getServletContext();
Object unameObj = application.getAttribute("uname");
System.out.println("unameObj = " + unameObj);
```

Thymeleaf初步

2022年1月19日 19:45

1、提出问题

```
// 返回提示消息方案三：确实能在登录页面显示提示消息，但是实现的方式让我想骂人
response.setContentType("text/html;charset=UTF-8");
PrintWriter writer = response.getWriter();
writer.write("<!DOCTYPE html>
<html>
<head>
<base href='/bookstore/' />
<meta charset='UTF-8' />
<title>尚硅谷会员登录页面</title>
<link type='text/css' rel='stylesheet' href='static/css/login.css' />
</head>
<body>
<h1>请输入用户名和密码</h1>
<form action='login' method='post'>
<input type='text' name='username' placeholder='用户名' />
<input type='password' name='password' placeholder='密码' />
<input type='submit' value='登录' />
</form>
</body>
</html>");
```

我们对HTML的新的期待：既能够正常显示页面，又能在页面中包含动态数据部分。而动态数据单靠HTML本身是无法做到的，所以此时我们需要引入服务器端动态视图模板技术。

2、从三层结构到MVC

①MVC概念

M: Model模型

V: View视图

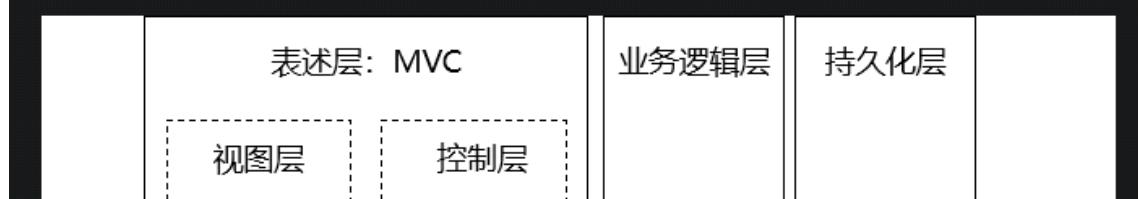
C: Controller控制器

MVC是在表述层开发中运用的一种设计理念。主张把封装数据的『模型』、显示用户界面的『视图』、协调调度的『控制器』分开。

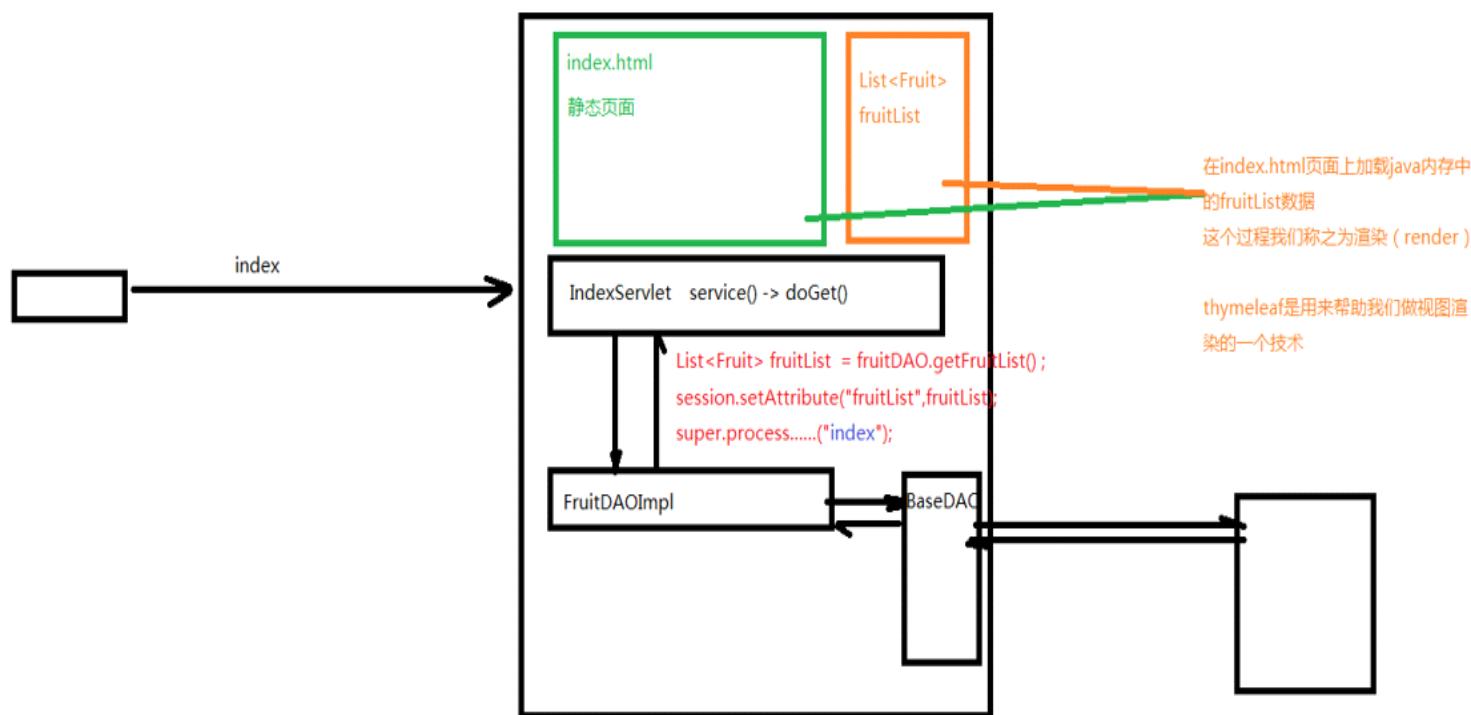
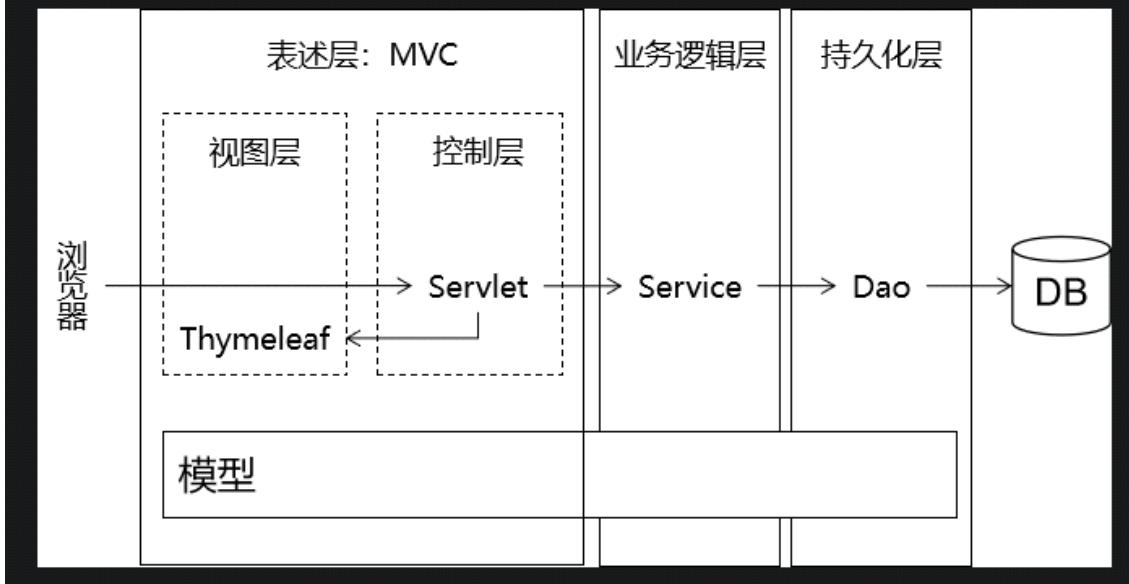
好处：

- 进一步实现各个组件之间的解耦
- 让各个组件可以单独维护
- 将视图分离出来以后，我们后端工程师和前端工程师的对接更方便

②MVC和三层架构之间关系



②MVC和三层架构之间关系



Thymeleaf - 视图模板技术

- 1) 添加thymeleaf的jar包
- 2) 新建一个Servlet类ViewBaseServlet

```
@Override
public void init() throws ServletException {
```

```
@Override
public void init() throws ServletException {
    // 1.获取ServletContext对象
    ServletContext servletContext = this.getServletContext();

    // 2.创建Thymeleaf解析器对象
    ServletContextTemplateResolver templateResolver = new ServletContextTemplateResol
    // 3.给解析器对象设置参数
    // @HTML是默认模式，明确设置是为了代码更容易理解
    templateResolver.setTemplateMode(TemplateMode.HTML);

    // @设置前缀
    String viewPrefix = servletContext.getInitParameter("view-prefix");
    templateResolver.setPrefix(viewPrefix);

    // @设置后缀
    String viewSuffix = servletContext.getInitParameter("view-suffix");
    templateResolver.setSuffix(viewSuffix);

    // @设置缓存过期时间（毫秒）
    templateResolver.setCacheTTLMs(60000L);

    // @设置是否缓存
    templateResolver.setCacheable(true);

    // @设置服务器端编码方式
    templateResolver.setCharacterEncoding("utf-8");

    // 4.创建模板引擎对象
    templateEngine = new TemplateEngine();

    // 5.给模板引擎对象设置模板解析器
    templateEngine.setTemplateResolver(templateResolver);
}

protected void processTemplate(String templateName, HttpServletRequest req, HttpServlet
    // 1.设置响应体内容类型和字符集
    resp.setContentType("text/html;charset=UTF-8");

    // 2.创建WebContext对象
    WebContext webContext = new WebContext(req, resp, getServletContext());

    // 3.处理模板数据
    templateEngine.process(templateName, webContext, resp.getWriter());
}
}
```

3) 在web.xml文件中添加配置

- 配置前缀 view-prefix

- 配置后缀 view-suffix

```
<!-- 在上下文参数中配置视图前缀和视图后缀 -->
<context-param>
    <param-name>view-prefix</param-name>
    <param-value>/WEB-INF/view/</param-value>
</context-param>
<context-param>
    <param-name>view-suffix</param-name>
    <param-value>.html</param-value>
</context-param>
```

4) 使得我们的Servlet继承ViewBaseServlet

5) 根据逻辑视图名称 得到 物理视图名称

```
public class IndexServlet extends ViewBaseServlet {
    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response) {
        FruitDAO fruitDAO = new FruitDAOImpl();
        List<Fruit> fruitList = fruitDAO.getFruitList();

        //保存到session作用域
        HttpSession session = request.getSession();
        session.setAttribute("fruitList", fruitList);
        //此处的视图名称是 index
        //那么thymeleaf会将这个 逻辑视图名称 对应到 物理视图 名称上去
        //逻辑视图名称 : index
        //物理视图名称 : view-prefix + 逻辑视图名称 + view-suffix
        //所以真实的视图名称是: /index.html
        super.processTemplate(templateName: "index", request, response);
    }
}
```

session.setAttribute("fruitList", fruitList);

把数据命名且存到session中（计算机内存里面），方便html里面的 th 引擎调用。

super. processTemplate(viewName, request, response);

是跳转函数，跳到viewName页面

6) 引入thymeleaf的引擎

在html第一行引入

```
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="utf-8">
        <link rel="stylesheet" href="css/index.css">
    </head>
```

如何使用命名空间 xmlns

很简单，使用语法： xmlns:namespace-prefix="namespaceURI",如：

xmlns:abc="http://www.springframework.org".

定义了abc后，要引用" http://www.springframework.org"中的元素，就必须加上abc前缀，如常见的：

```
<context:component-scan base-package="com.taotao.controller" />
```

7) 使用thymeleaf的标签（语法关键字）

th:text

1、修改标签文本值

代码示例：

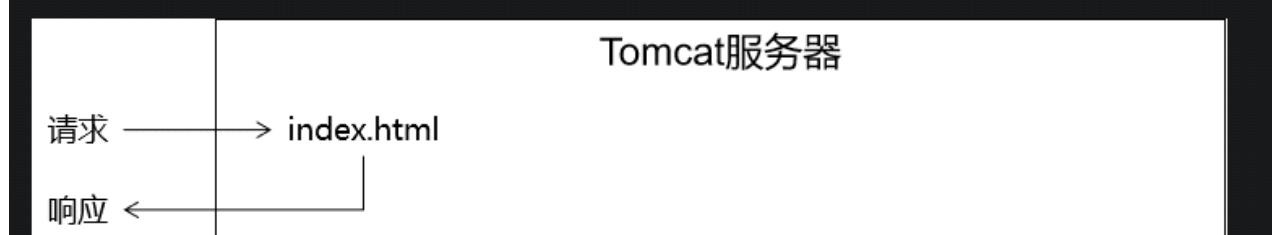
```
1   <p th:text="标签体新值">标签体原始值</p>
```

①th:text作用

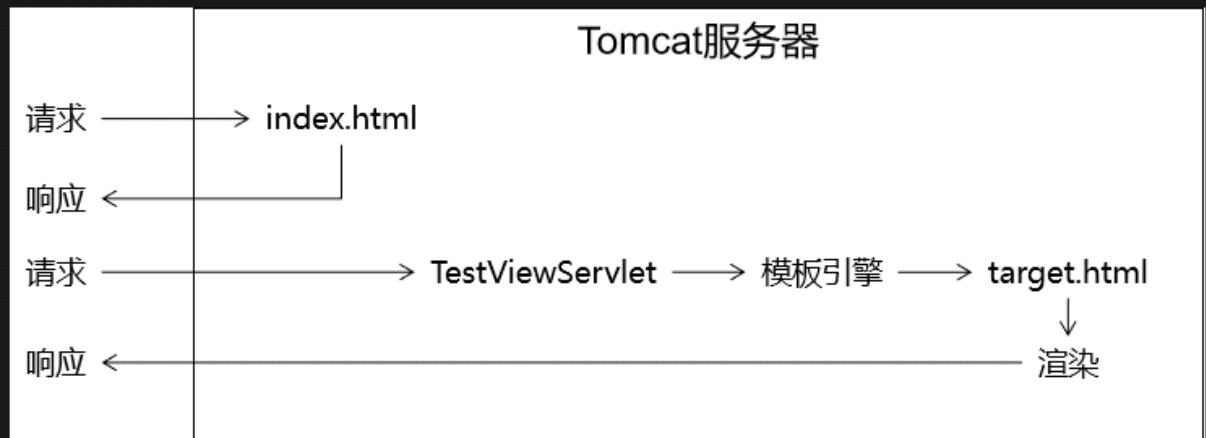
- 不经过服务器解析，直接用浏览器打开HTML文件，看到的是『标签体原始值』
- 经过服务器解析，Thymeleaf引擎根据th:text属性指定的『标签体新值』去替换『标签体原始值』

```
<td th:text="${fruit.fname}">苹果</td>
<td th:text="${fruit.price}">5</td>
<td th:text="${fruit.fcount}">20</td>
```

②首页使用URL地址解析

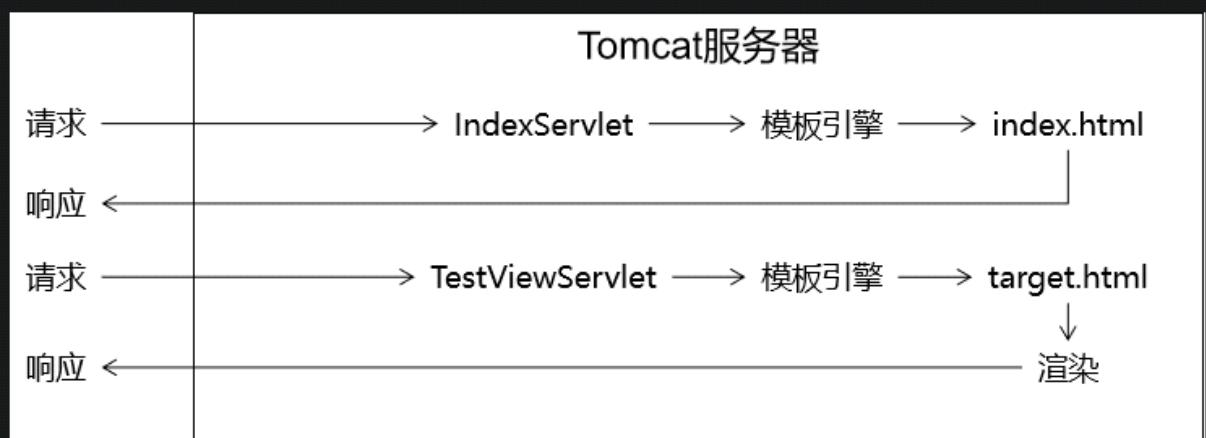


②首页使用URL地址解析



如果我们直接访问index.html本身，那么index.html是不需要通过Servlet，当然也不经过模板引擎，所以index.html上的Thymeleaf的任何表达式都不会被解析。

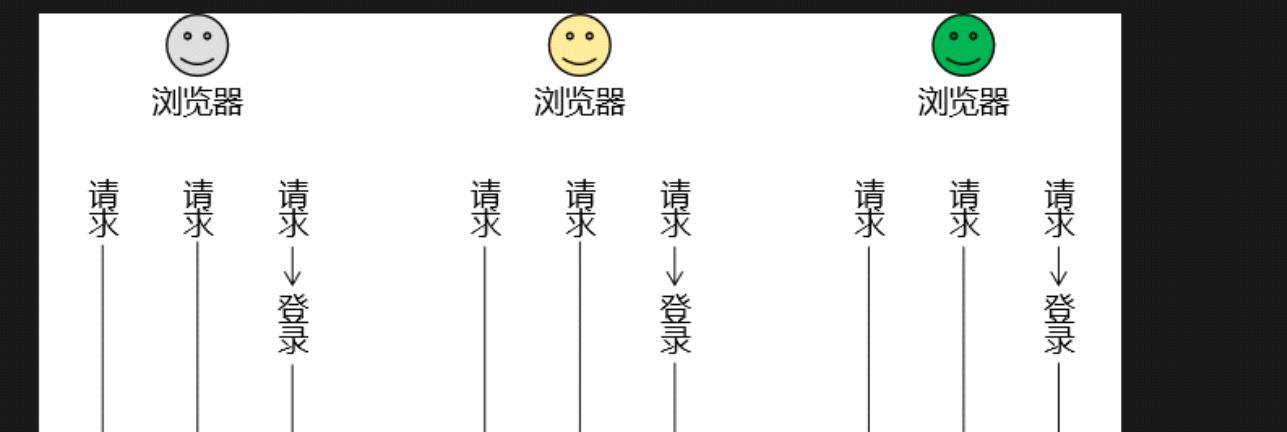
解决办法：通过Servlet访问index.html，这样就可以让模板引擎渲染页面了：

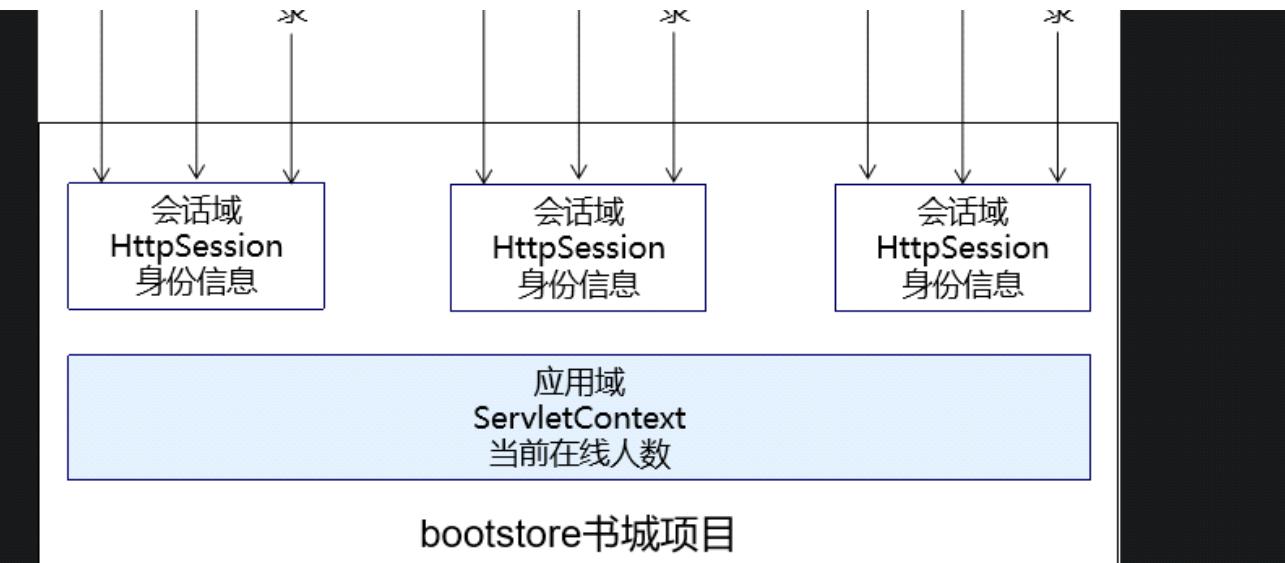


进一步的好处：

通过上面的例子我们看到，所有和业务功能相关的请求都能够确保它们通过Servlet来处理，这样就方便我们统一对这些请求进行特定规则的限定。

③应用域





PS：在我们使用的视图是JSP的时候，域对象有4个

- pageContext
- request：请求域
- session：会话域
- application：应用域

所以在JSP的使用背景下，我们可以说域对象有4个，现在使用Thymeleaf了，没有pageContext。

2、在Servlet中将数据存入属性域

①操作请求域

Servlet中代码：

```

1  String requestAttrName = "helloRequestAttr";
2  String requestAttrValue = "helloRequestAttr-VALUE";
3
4  request.setAttribute(requestAttrName, requestAttrValue);

```

java

Thymeleaf表达式：

```

1  <p th:text="${helloRequestAttr}">request field value</p>

```

html

②操作会话域

Servlet中代码：

```

1  // ①通过request对象获取session对象
2  HttpSession session = request.getSession();
3  session.setAttribute("helloSessionAttr", "helloSessionAttr-VALUE");

```

java

```
1 // ①通过request对象获取session对象  
2 HttpSession session = request.getSession();  
3  
4 // ②存入数据  
5 session.setAttribute("helloSessionAttr", "helloSessionAttr-VALUE");
```

java

Thymeleaf表达式：

```
1 <p th:text="${session.helloSessionAttr}">这里显示会话域数据</p>
```

html

③操作应用域

Servlet中代码：

```
1 // ①通过调用父类的方法获取ServletContext对象  
2 ServletContext servletContext = getServletContext();  
3  
4 // ②存入数据  
5 servletContext.setAttribute("helloAppAttr", "helloAppAttr-VALUE");
```

java

Thymeleaf表达式：

```
1 <p th:text="${application.helloAppAttr}">这里显示应用域数据</p>
```

html

分支if switch

```
6 </tr>  
7 <tr th:if="#{lists.isEmpty(employeeList)}">  
8     <td colspan="3">抱歉！没有查询到你搜索的数据！</td>  
9 </tr>  
10 <tr th:if="not #lists.isEmpty(employeeList)">  
11     <td colspan="3">有数据！</td>  
12 </tr>  
13 <tr th:unless="#{lists.isEmpty(employeeList)}">  
14     <td colspan="3">有数据！</td>  
15 </tr>  
16 </table>
```

if配合not关键词和unless配合原表达式效果是一样的，看自己的喜好。

②switch

②switch

```
1 <h3>测试switch</h3>
2 <div th:switch="${user.memberLevel}">
3     <p th:case="level-1">银牌会员</p>
4     <p th:case="level-2">金牌会员</p>
5     <p th:case="level-3">白金会员</p>
6     <p th:case="level-4">钻石会员</p>
7 </div>
```

迭代

```
<tbody th:if="${not #lists.isEmpty(employeeList)}">
    <!-- 遍历出来的每一个元素的名字 : ${要遍历的集合} --&gt;
    &lt;tr th:each="employee : ${employeeList}"&gt;
        &lt;td th:text="${employee.empId}"&gt;empId&lt;/td&gt;
        &lt;td th:text="${employee.empName}"&gt;empName&lt;/td&gt;
        &lt;td th:text="${employee.empSalary}"&gt;empSalary&lt;/td&gt;
    &lt;/tr&gt;
&lt;/tbody&gt;</pre>
```

路径问题

<base href="http://localhost:8080/pro10/" /> 的作用是：当前页面上的所有的路径都以这个为基础
<link href="css/shopping.css">

```
th:href="@{}"
<link th:href="@{/css/shopping.css}">
```

用@{}来代替base

给URL地址后面附加请求参数

Servlet代码：

```
1 request.setAttribute("reqAttrName", "<span>hello-value</span>");
```

java

```
1 request.setAttribute("reqAttrName", "<span>hello-value</span>");
```

java

页面代码：

```
1 <p>有转义效果：[[${reqAttrName}]]</p>
2 <p>无转义效果：[(${reqAttrName})]</p>
```

html

执行效果：

```
1 <p>有转义效果：&lt;span&gt;hello-value&lt;/span&gt;</p>
2 <p>无转义效果：<span>hello-value</span></p>
```

html

eg

```
<td><a th:text="${fruit.fname}" th:href="@{/edit.do (fid=${fruit.fid})}">苹果
</a></td>
```

字符串拼接

Thymeleaf 字符串拼接

一种是字面量拼接：

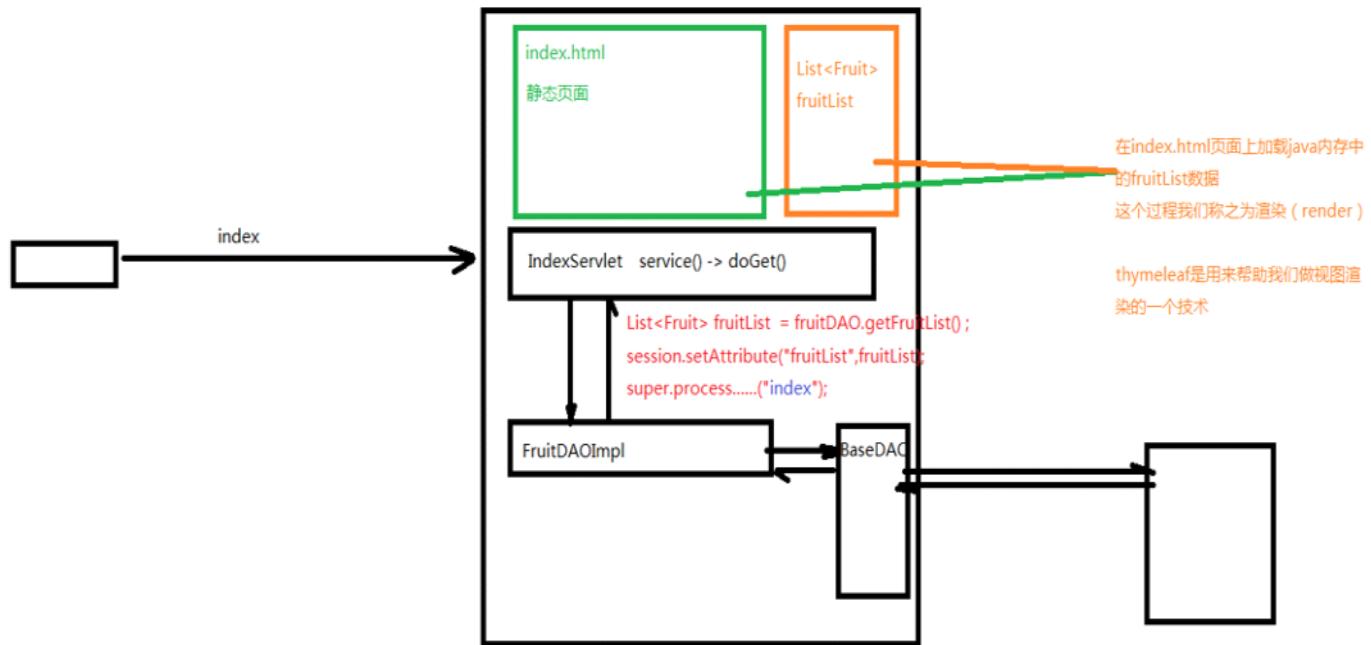
```
<span th:text="当前是第' + ${sex} + '页,共' + ${sex} + '页"> </span>
```

另一种更优雅的方式，使用 “|” 减少了字符串的拼接：

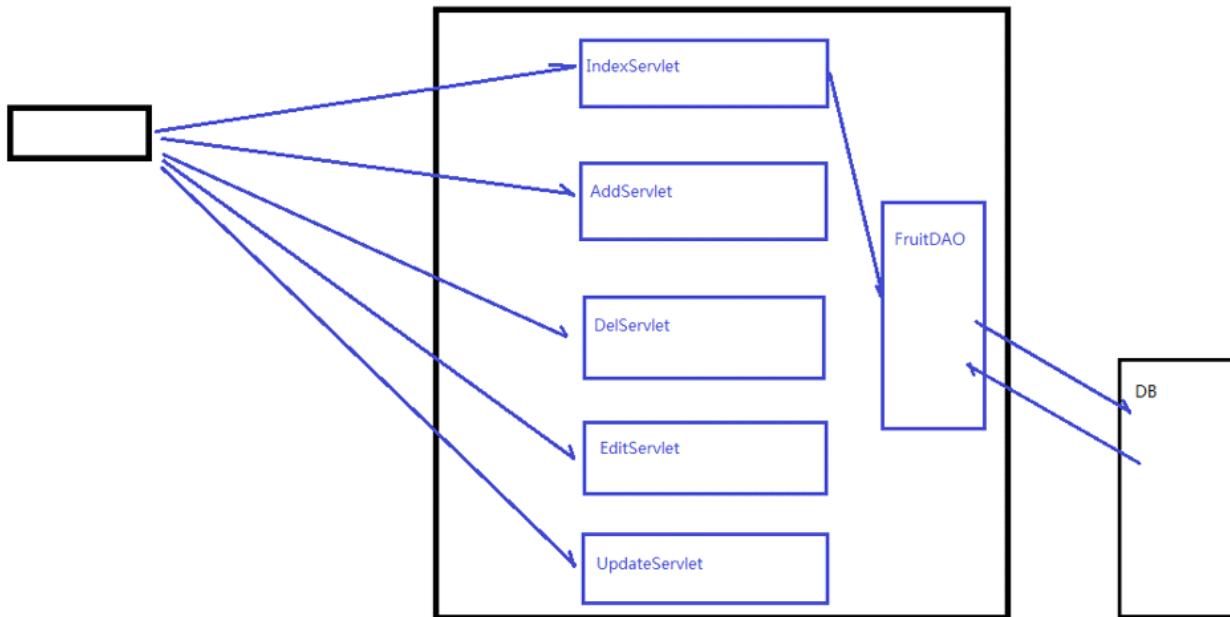
```
<span th:text="|当前是第${sex}页,共${sex}页|"> </span>
```

第一个小项目--水果

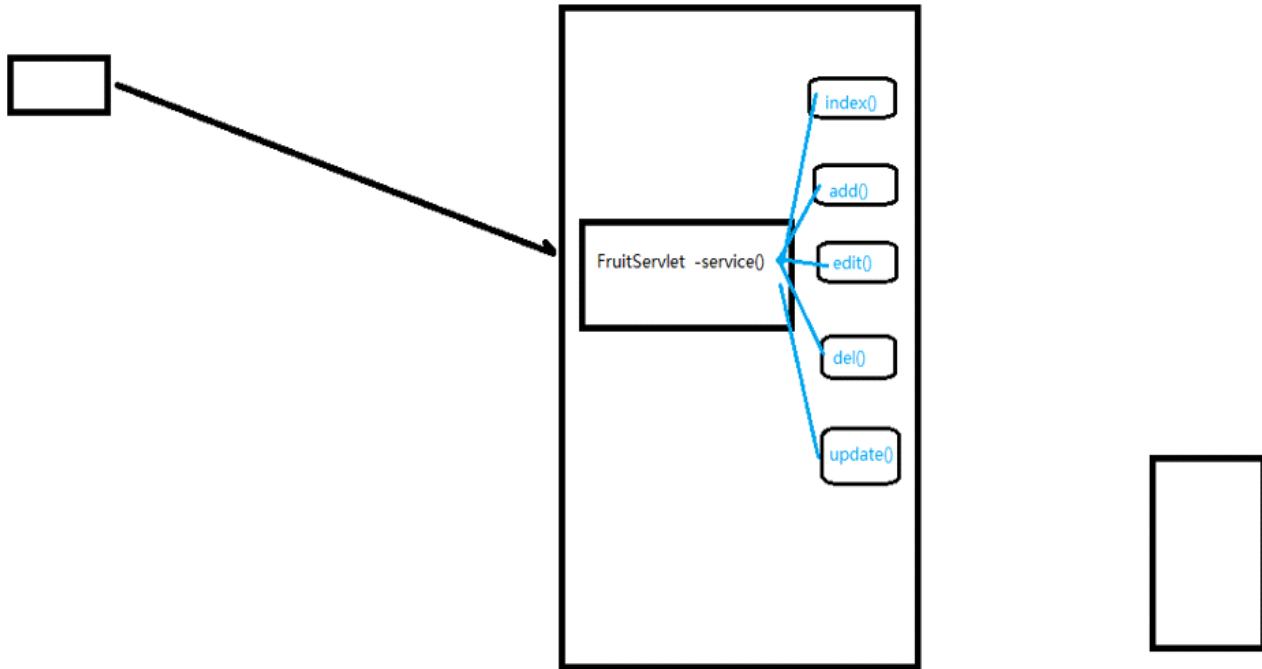
2022年1月21日 17:23



传统设计



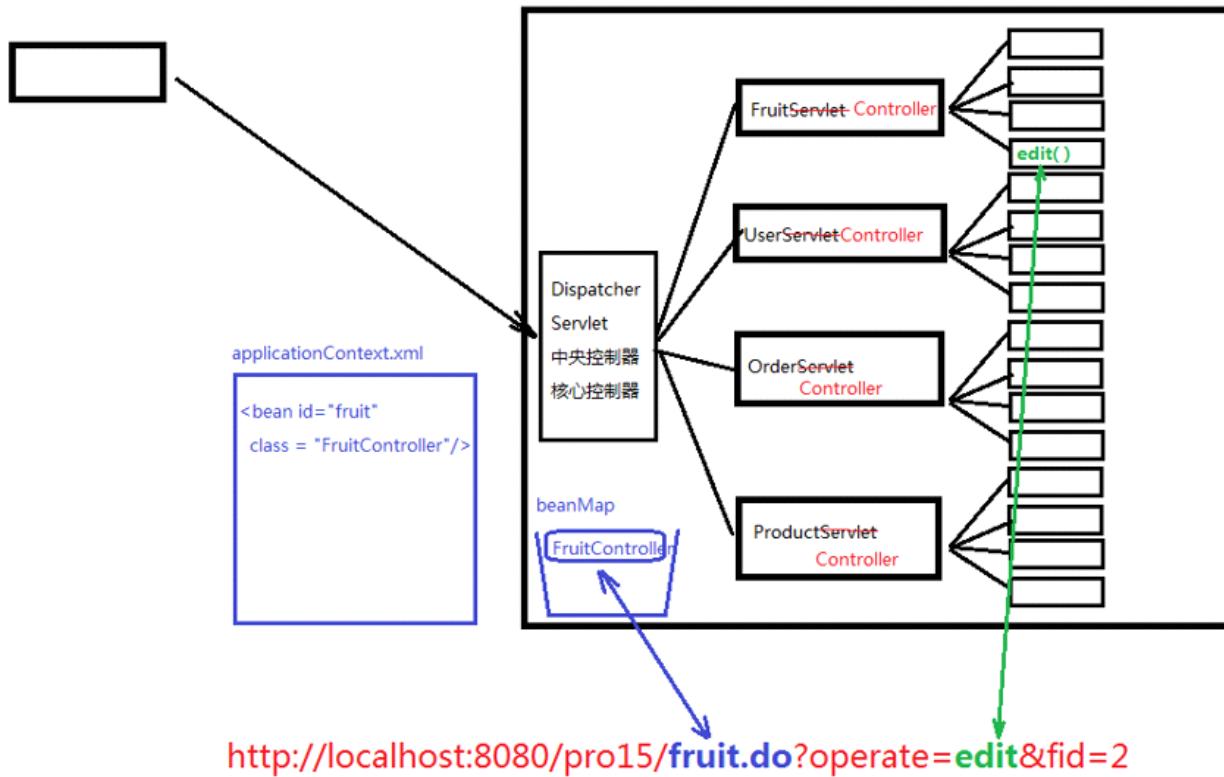
改进后



再升级的大体思路

1. 获取 url 发回的 response , 得到 response 需要的 operation , 截取 operation 需要的方法名。
2. 加载 xml 里面的所有 bean , 使用反射技术 , 全部按 (名字 , 类实例) 存在一个 hashmap 里面。
3. 使用 1 中的方法名去 hashmap 里面找要用的方法。

升级后，使用中央控制器

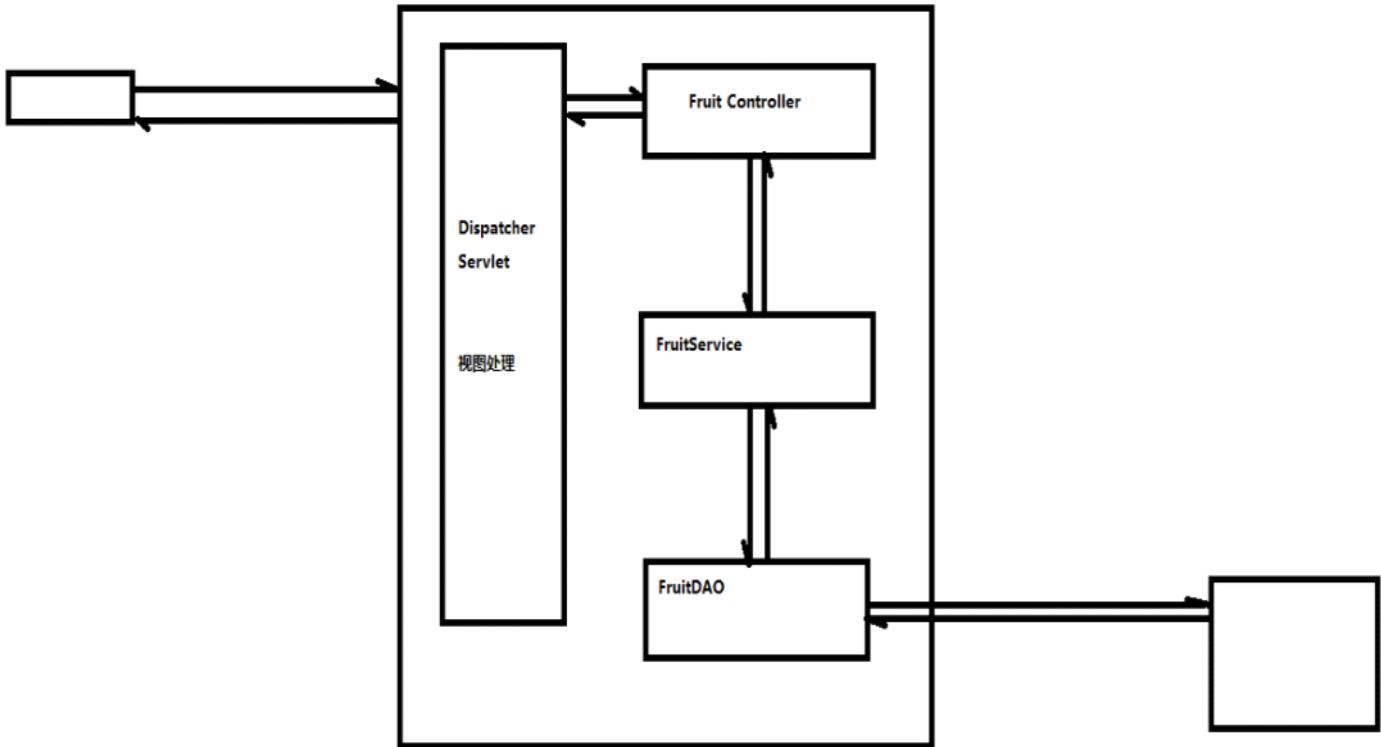


编译java时使用 “-parameters” 可以在编译好的里面保留形参名字
在 maven 里面加

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <compilerArgs>
      <arg>-parameters</arg>
    </compilerArgs>
  </configuration>
</plugin>
    
```

经典



过程review:

- 最初的做法是：一个请求对应一个Servlet，这样存在的问题是servlet太多了
- 把一些列的请求都对应一个Servlet，

IndexServlet/AddServlet/EditServlet/DelServlet/UpdateServlet -> 合并成
FruitServlet

通过一个operate的值来决定调用FruitServlet中的哪一个方法

使用的是switch-case

- 在上一个版本中，Servlet中充斥着大量的switch-case，试想一下，随着我们的项目的业务规模扩大，那么会有很多的Servlet，也就意味着会有很多的switch-case，这是一种代码冗余

因此，我们在servlet中使用了反射技术，我们规定operate的值和方法名一致，那么接收到operate的值是什么就表明我们需要调用对应的方法进行响应，如果找不到对应的方法，则抛异常

- 在上一个版本中我们使用了反射技术，但是其实还是存在一定的问题：每一个servlet中都有类似的反射技术的代码。因此继续抽取，设计了中央控制器类：DispatcherServlet

DispatcherServlet这个类的工作分为两大部分：

- 根据url定位到能够处理这个请求的controller组件：

1)从url中提取servletPath : /fruit.do -> fruit

2)根据fruit找到对应的组件:FruitController，这个对应的依据我们存储在

applicationContext.xml中

<bean id="fruit" class="com.atguigu.fruit.controllers.FruitController"/>

通过DOM技术我们去解析XML文件，在中央控制器中形成一个beanMap容器，用来存放所有的Controller组件

3)根据获取到的operate的值定位到我们FruitController中需要调用的方法

- 调用Controller组件中的方法：

1) 获取参数

获取即将要调用的方法的参数签名信息：Parameter[] parameters =

```

method.getParameters();
    通过parameter.getName()获取参数的名称;
    准备了Object[] parameterValues 这个数组用来存放对应参数的参数值
    另外，我们需要考虑参数的类型问题，需要做类型转化的工作。通过
parameter.getType()获取参数的类型
2) 执行方法
    Object returnObj = method.invoke(controllerBean , parameterValues);
3) 视图处理
    String returnStr = (String)returnObj;
    if(returnStr.startsWith("redirect:")){
        ...
    }else if.....

```

什么是业务层

1) Model1和Model2

MVC : Model (模型) 、 View (视图) 、 Controller (控制器)

视图层：用于做数据展示以及和用户交互的一个界面

控制层：能够接受客户端的请求，具体的业务功能还是需要借助于模型组件来完成

模型层：模型分为很多种：有比较简单的pojo/vo(value object)，有业务模型组件，有
数据访问层组件

1) pojo/vo : 值对象

2) DAO : 数据访问对象

3) BO : 业务对象

2) 区分业务对象和数据访问对象：

1) DAO中的方法都是单精度方法或者称之为细粒度方法。什么叫单精度？一个方法只考虑一个操作，比如添加，那就是insert操作、查询那就是select操作....

2) BO中的方法属于业务方法，也实际的业务是比较复杂的，因此业务方法的粒度是比较粗的

注册这个功能属于业务功能，也就是说注册这个方法属于业务方法。

那么这个业务方法中包含了多个DAO方法。也就是说注册这个业务功能需要通过多个
DAO方法的组合调用，从而完成注册功能的开发。

注册：

1. 检查用户名是否已经被注册 - DAO中的select操作
2. 向用户表新增一条新用户记录 - DAO中的insert操作
3. 向用户积分表新增一条记录（新用户默认初始化积分100分） - DAO中的
insert操作

4. 向系统消息表新增一条记录（某某某新用户注册了，需要根据通讯录信息
向他的联系人推送消息） - DAO中的insert操作

5. 向系统日志表新增一条记录（某用户在某IP在某年某月某日某时某分某
秒某毫秒注册） - DAO中的insert操作

6.

3) 在库存系统中添加业务层组件

IOC

IOC - 控制反转 / DI - 依赖注入

控制反转：

1) 之前在Servlet中，我们创建service对象， FruitService fruitService = new FruitServiceImpl();

这句话如果出现在servlet中的某个方法内部，那么这个fruitService的作用域（生命周期）应该就是这个方法级别；

如果这句话出现在servlet的类中，也就是说fruitService是一个成员变量，那么这个fruitService的作用域（生命周期）应该就是这个servlet实例级别

2) 之后我们在applicationContext.xml中定义了这个fruitService。然后通过解析XML，产生fruitService实例，存放在beanMap中，这个beanMap在一个BeanFactory中

因此，我们转移（改变）了之前的service实例、dao实例等等他们的生命周期。控制权从程序员转移到BeanFactory。这个现象我们称之为控制反转

依赖注入：

1) 之前我们在控制层出现代码：FruitService fruitService = new FruitServiceImpl();

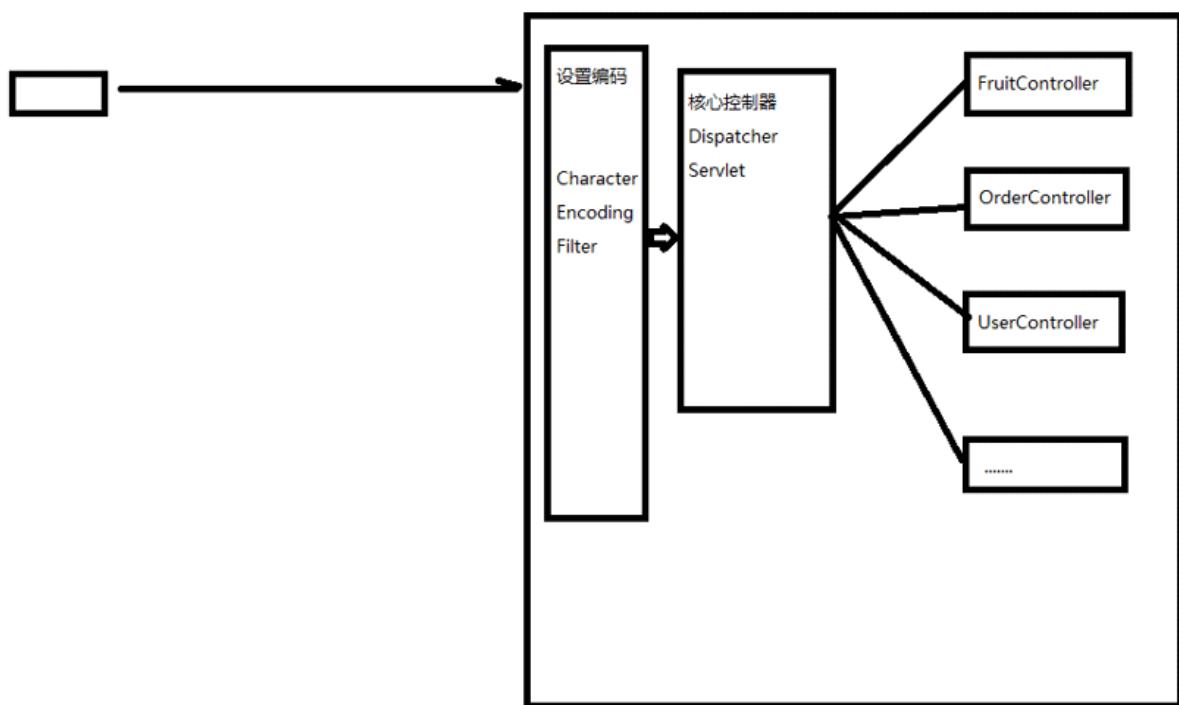
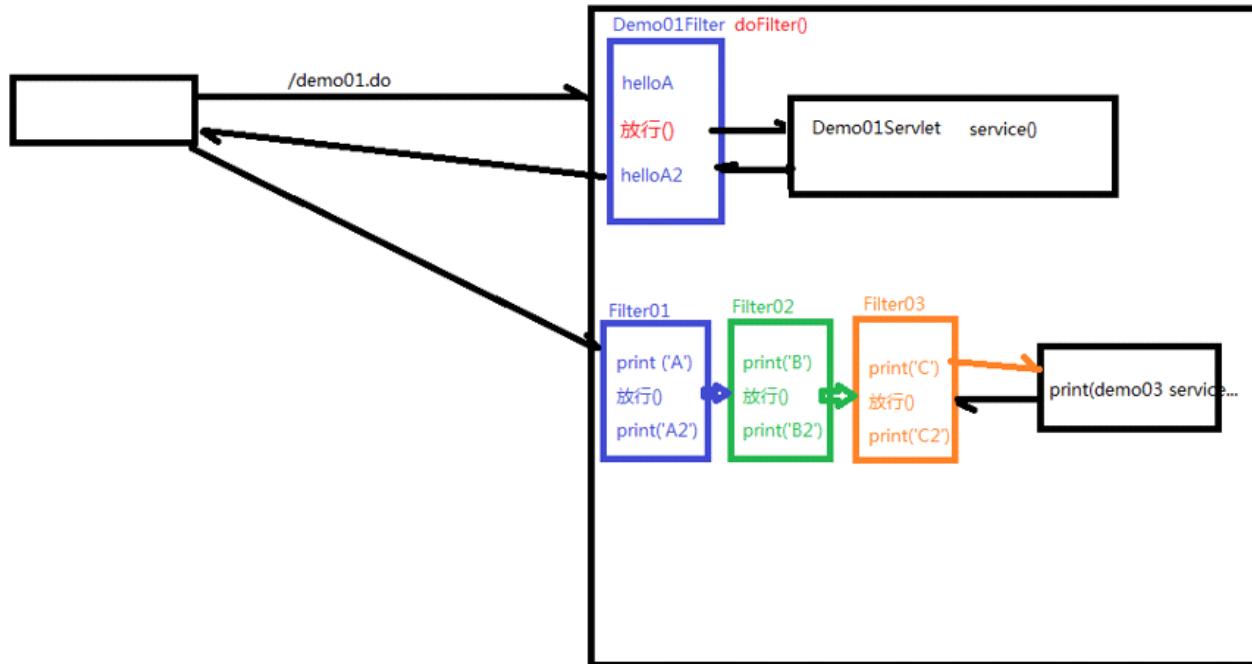
那么，控制层和service层存在耦合。

2) 之后，我们将代码修改成FruitService fruitService = null；

然后，在配置文件中配置：

```
<bean id="fruit" class="FruitController">
    <property name="fruitService" ref="fruitService"/>
</bean>
```

Filter



1) Filter也属于Servlet规范

2) Filter开发步骤：新建类实现Filter接口，然后实现其中的三个方法：

`init`、`doFilter`、`destroy`

配置Filter，可以用注解@WebFilter，也可以使用xml文件<filter><filter-mapping>

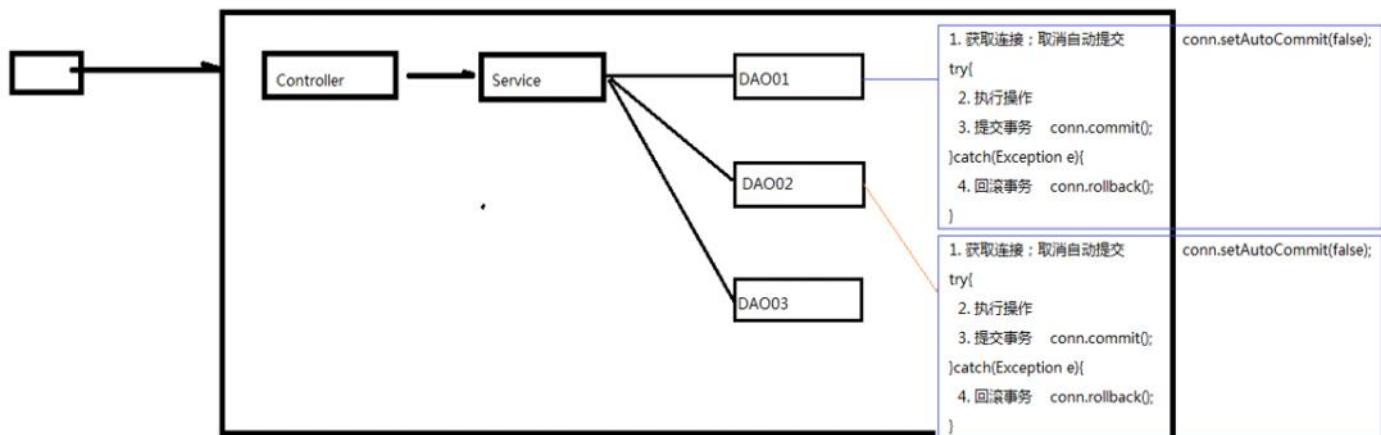
3) Filter在配置时，和servlet一样，也可以配置通配符，例如

`@WebFilter("*.do")`表示拦截所有以.do结尾的请求

4) 过滤器链

- 1) 执行的顺序依次是： A B C demo03 C2 B2 A2
- 2) 如果采取的是注解的方式进行配置，那么过滤器链的拦截顺序是按照全类名的先后顺序排序的
- 3) 如果采取的是xml的方式进行配置，那么按照配置的先后顺序进行排序

事务管理



当前Service 中包含了三个DAO操作。

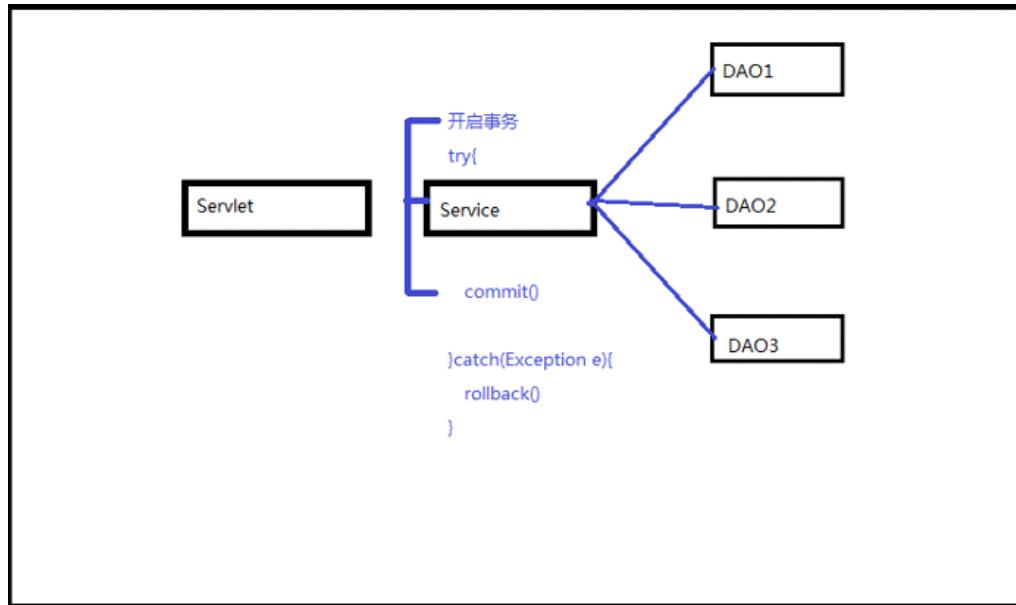
之前的DAO的事务管理的基本API是右边写的方式。

这样方式带来的问题是： DAO01执行成功-提交， DAO02执行失败-回滚， DAO03执行成功-提交

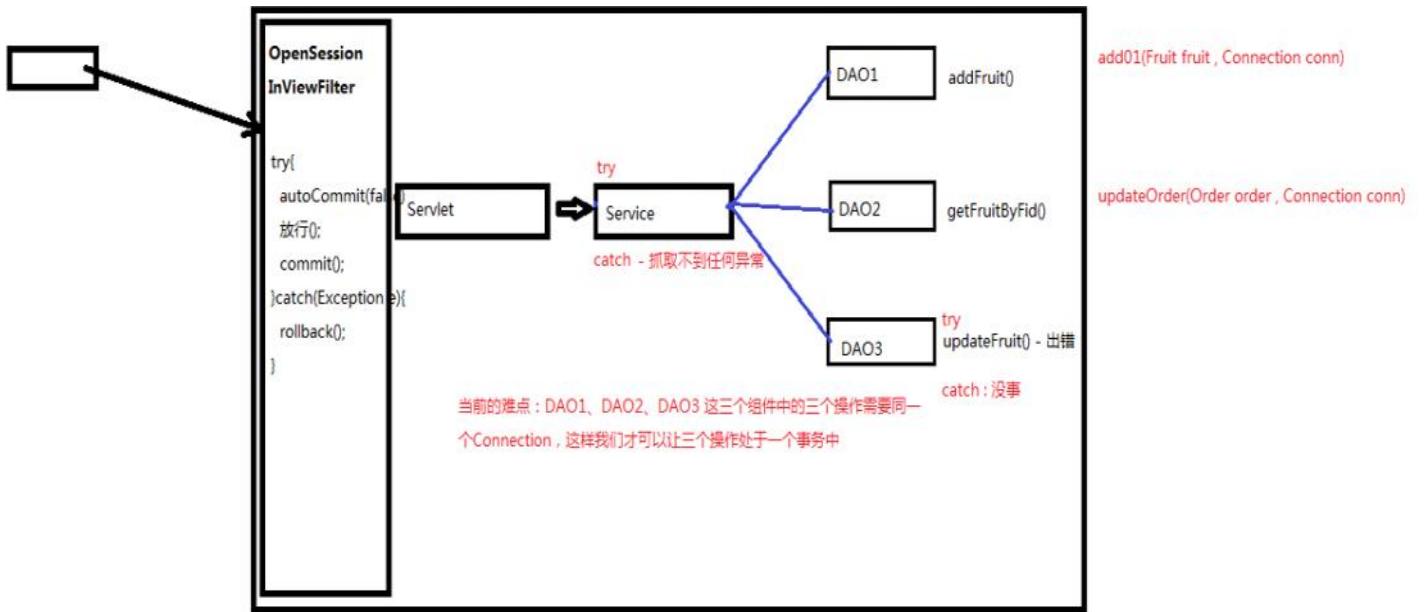
此时service操作是成功还是失败？？？ service的操作应该是一个整体，不能部分成功部分失败。比如两个人完成转账的操作。张三扣款成功了，李四账户增加金额失败了。这个肯定是不允许出现的。

因此，service是一个整体，要么都成功，要么都失败 -- 得出结论：事务管理不能以DAO层的单精度方法为单位，而应该以业务层的方法为单位

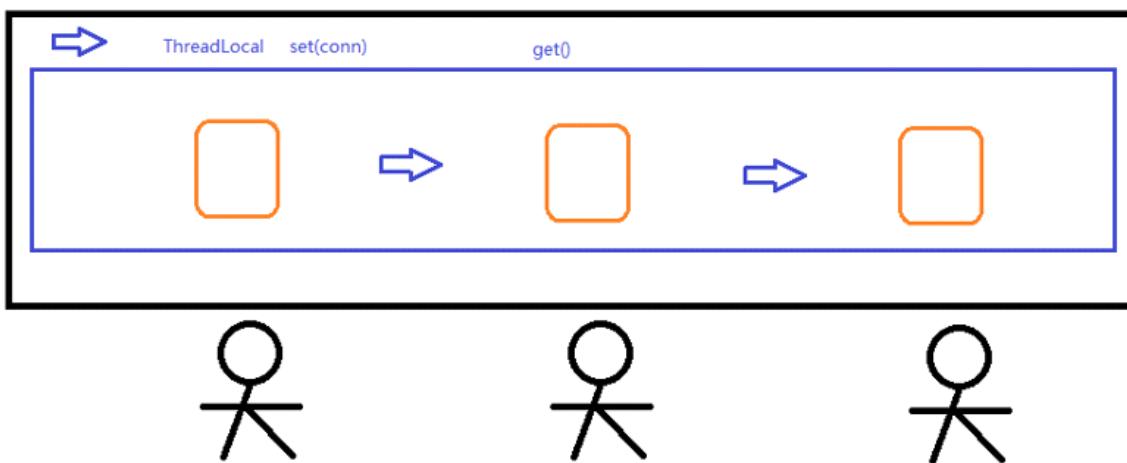
解决方法



服务前置



- ThreadLocal



1) 涉及到的组件：

- OpenSessionInViewFilter
- TransactionManager
- ThreadLocal
- ConnUtil
- BaseDAO

2) ThreadLocal

- get(), set(obj)
- ThreadLocal称之为本地线程。 我们可以通过set方法在当前线程上存储数据、通过get方法在当前线程上获取数据

- set方法源码分析：

```

public void set(T value) {
    Thread t = Thread.currentThread(); //获取当前的线程
    ThreadLocalMap map = getMap(t);    //每一个线程都维护各自的一个容器（ThreadLocalMap）
    if (map != null)
        map.set(this, value);          //这里的key对应的是ThreadLocal，因为我们的组件中需要传输（共享）的对象可能会有多个
    else
        createMap(t, value);         //默认情况下map是没有初始化的，那么第一次往其中添加数据时，会去初始化
}

```

-get方法源码分析：

```

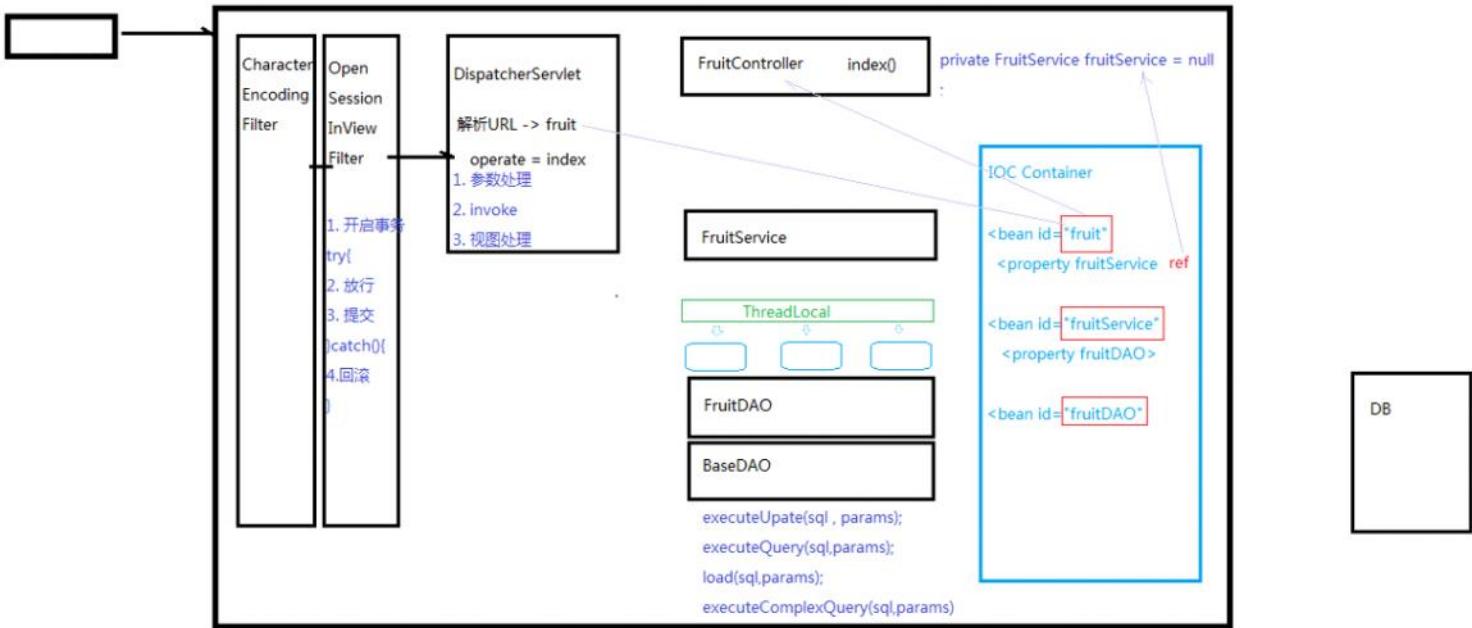
public T get() {
    Thread t = Thread.currentThread(); //获取当前的线程
    ThreadLocalMap map = getMap(t);    //获取和这个线程（企业）相关的ThreadLocalMap（也就是工作纽带的集合）
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);    //this指的是ThreadLocal对象，通过它才能知道是哪一个工作纽带
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;      //entry.value就可以获取到工具箱了
            return result;
        }
    }
    return setInitialValue();
}

```

监听器

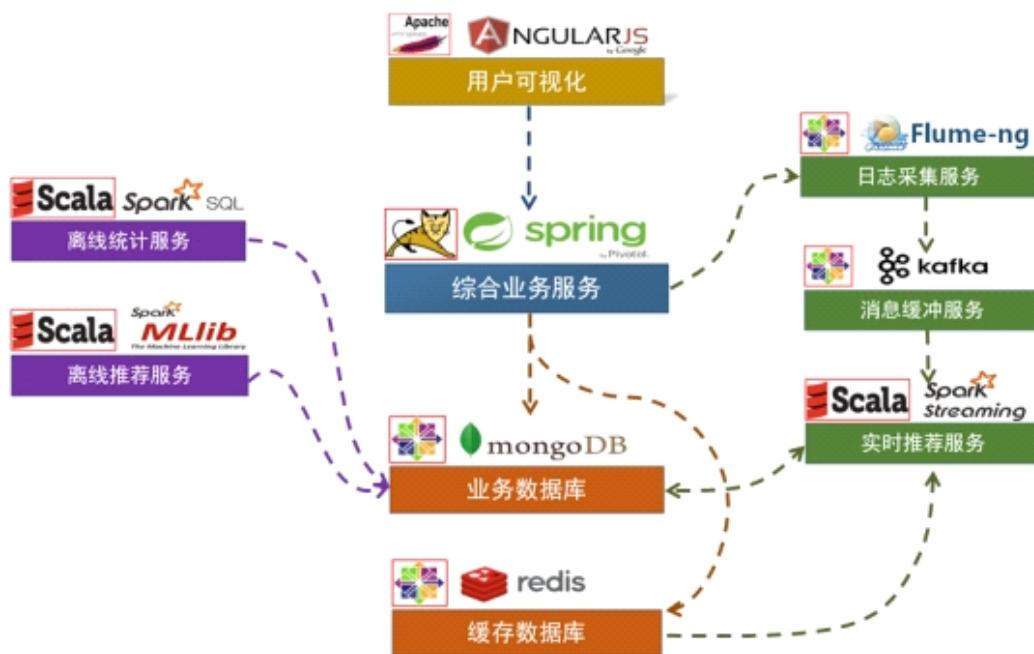
- 1) ServletContextListener - 监听ServletContext对象的创建和销毁的过程
- 2) HttpSessionListener - 监听HttpSession对象的创建和销毁的过程
- 3) ServletRequestListener - 监听ServletRequest对象的创建和销毁的过程
- 4) ServletContextAttributeListener - 监听ServletContext的保存作用域的改动
(add,remove,replace)
- 5) HttpSessionAttributeListener - 监听HttpSession的保存作用域的改动
(add,remove,replace)
- 6) ServletRequestAttributeListener - 监听ServletRequest的保存作用域的改动
(add,remove,replace)
- 7) HttpSessionBindingListener - 监听某个对象在Session域中的创建与移除
- 8) HttpSessionActivationListener - 监听某个对象在Session域中的序列化和反序列化

最终效果



项目体系架构设计

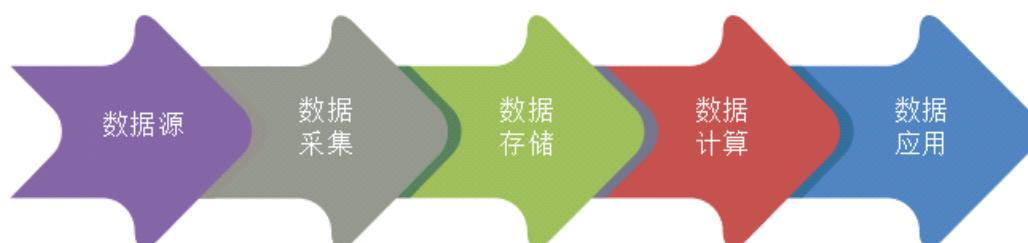
2022年1月21日 17:38



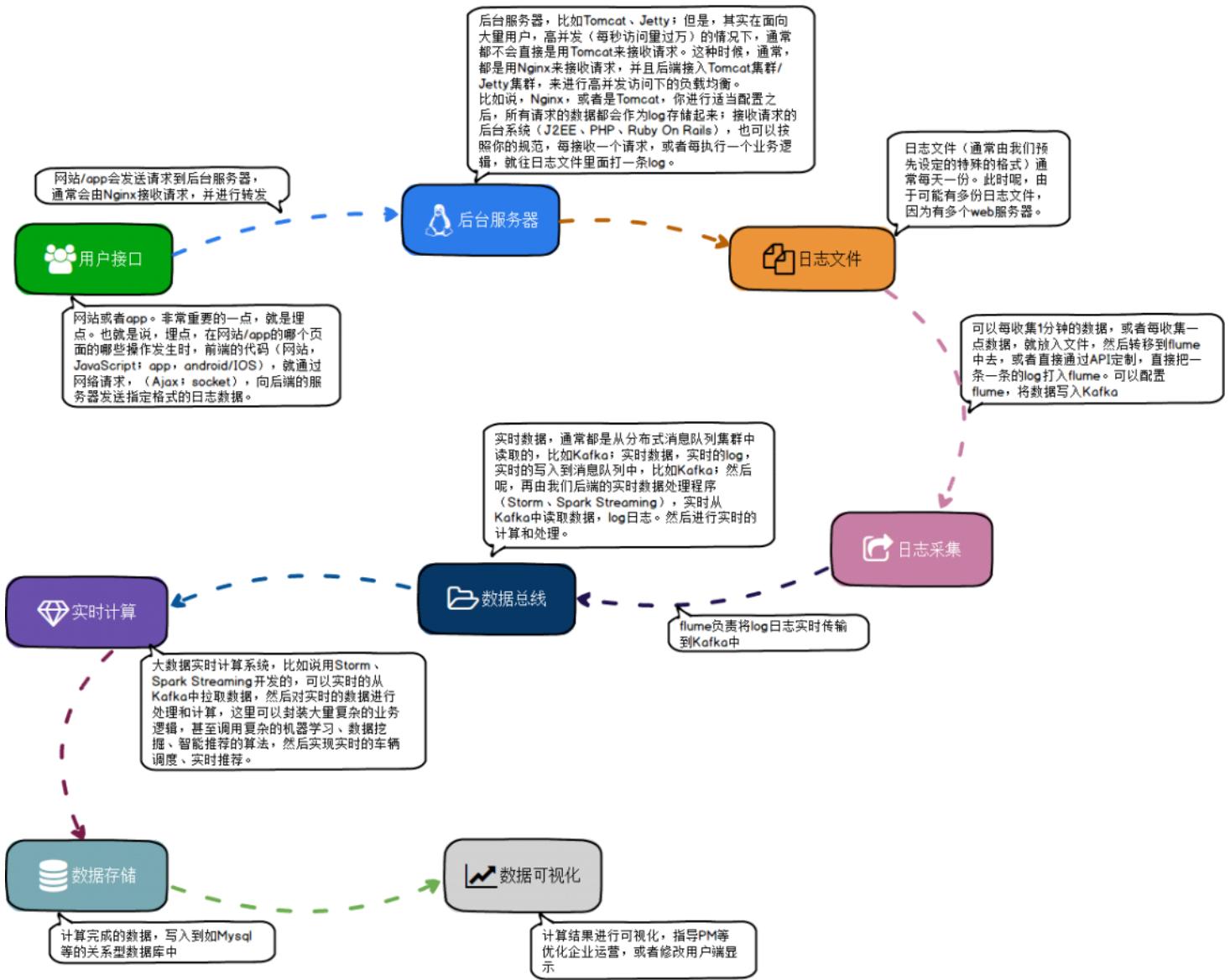
数据生命周期

非结构化数据	图片视频	ETL工具	Oracle	Mahout	业务应用
半结构化数据	日志数据	Scribe	GreenPlum	Storm	Tableau
结构化数据	关系数据	Flume	Cassandra	Flink	BI分析

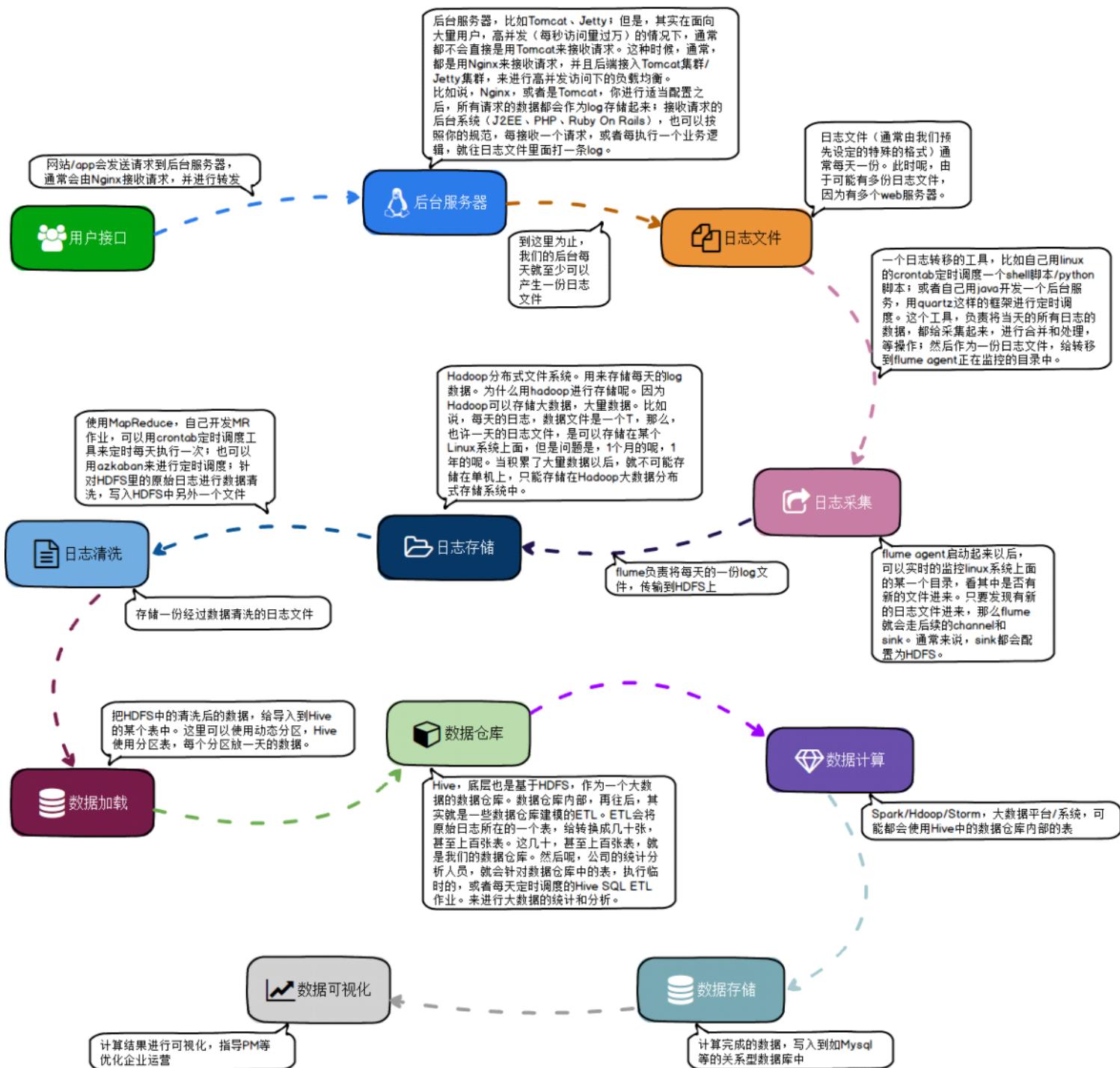
非结构化数据	图片视频	ETL工具	Oracle	Mahout	业务应用
半结构化数据	日志数据	Scribe	GreenPlum	Storm	Tableau
结构化数据	关系数据	Flume	Cassandra	Flink	BI分析



在线模式



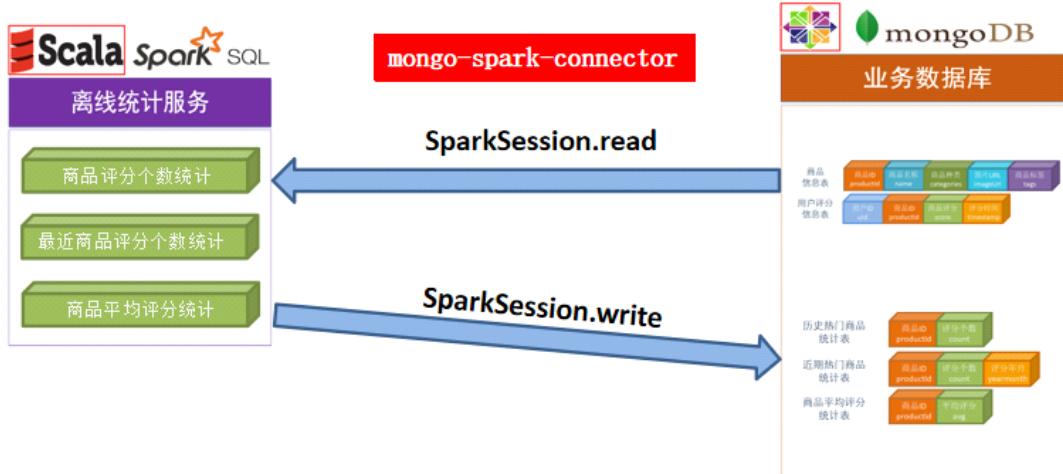
离线模式



主要数据模型



统计推荐模块



历史热门商品统计

- 统计所有历史数据中每个商品的评分数
- `select productId, count(productId) as count from ratings group by productId order by count desc`

→ RateMoreProducts

- RateMoreProducts 数据结构: productId, count

近期热门商品统计

- 统计每月的商品评分个数，就代表了商品近期的热门度
- `select productId, score, changeDate(timestamp) as yearmonth from ratings`
→ ratingOfMonth
- `select productId, count(productId) as count ,yearmonth from ratingOfMonth group by yearmonth, productId order by yearmonth desc,count desc`

→ RateMoreRecentlyProducts

- `changeDate` : UDF函数，使用 `SimpleDateFormat` 对 `Date` 进行格式转化，转化格式为“`yyyyMM`”
- RateMoreRecentlyProducts 数据结构: productId, count, yearmonth

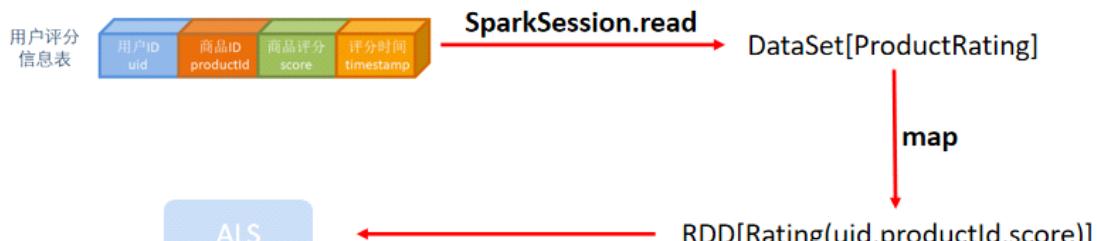
商品平均评分统计

- select productId, avg(score) as avg from ratings group by productId order by avg desc
- AverageProducts
- AverageProducts 数据结构: productId, avg

基于LFM的离线推荐模块

- 用ALS算法训练隐语义模型 (LFM)
- 计算用户推荐矩阵
- 计算商品相似度矩阵

ALS算法进行隐语义模型训练



```
val model = ALS.train(trainData,rank,iterations,lambda)
```

- RMSE

均方根误差：均方误差的算术平方根，预测值与真实值之间的误差

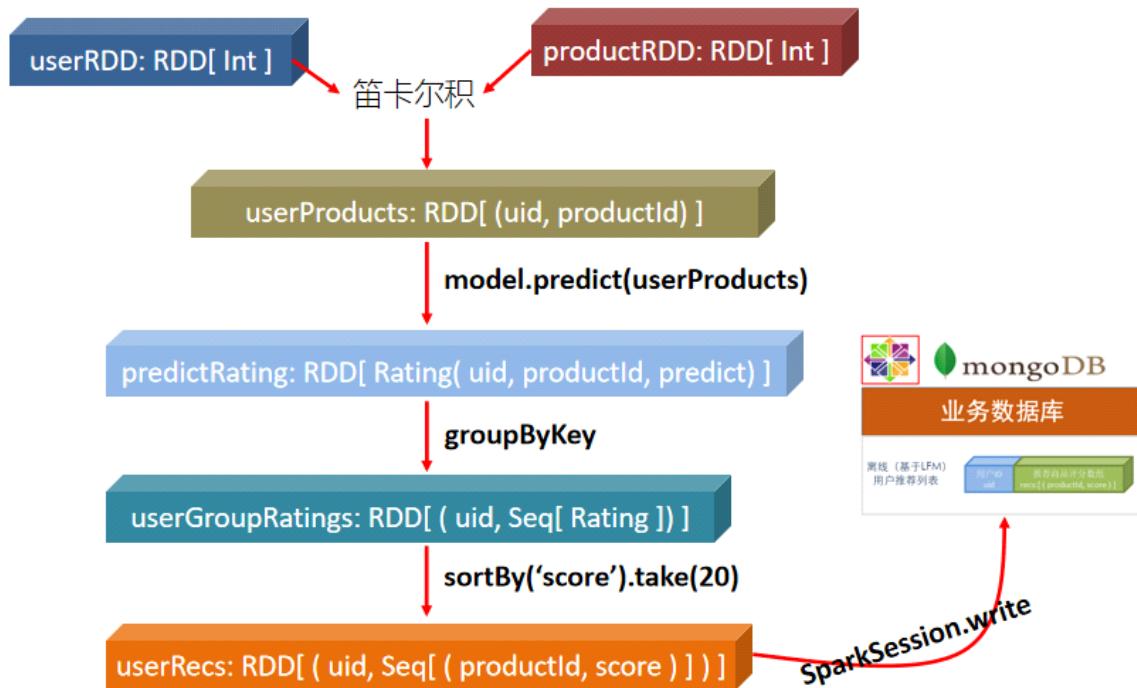
$$RMSE = \sqrt{\frac{1}{N} \sum_{t=1}^N (observed_t - predicted_t)^2}$$

- 参数调整

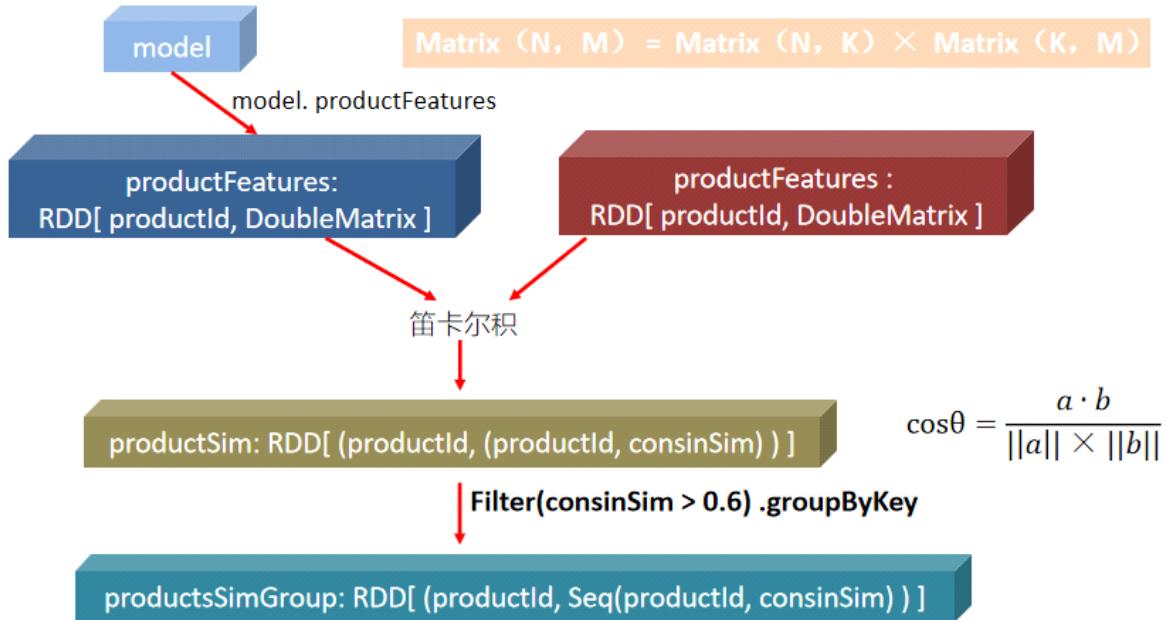
可以通过均方根误差，来多次调整参数值，选择RMSE最小的一组参数值

- rank, iterations, lambda

计算用户推荐矩阵



计算商品相似度矩阵



基于模型的实时推荐模块

- 计算速度要快
- 结果可以不是特别精确
- 有预先设计好的推荐模型

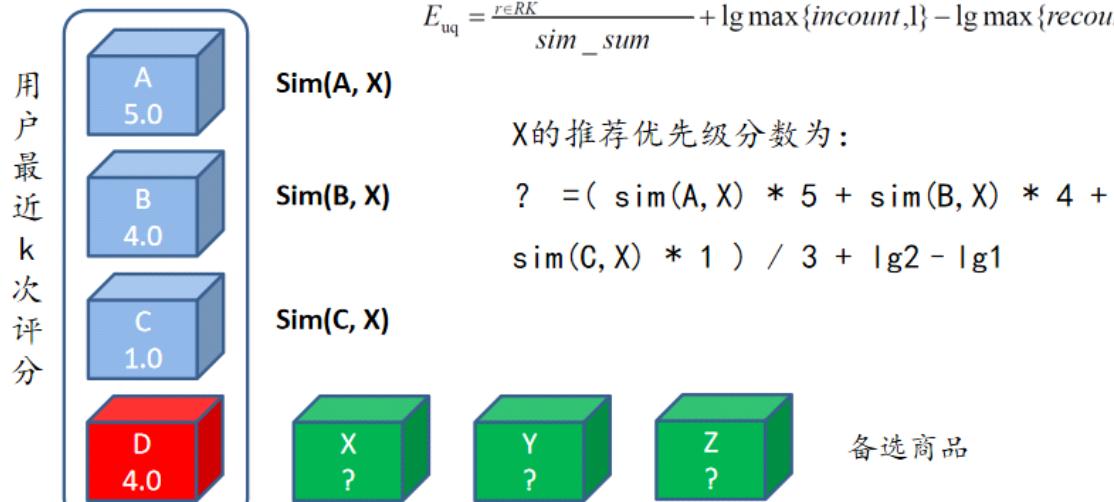


推荐优先级计算

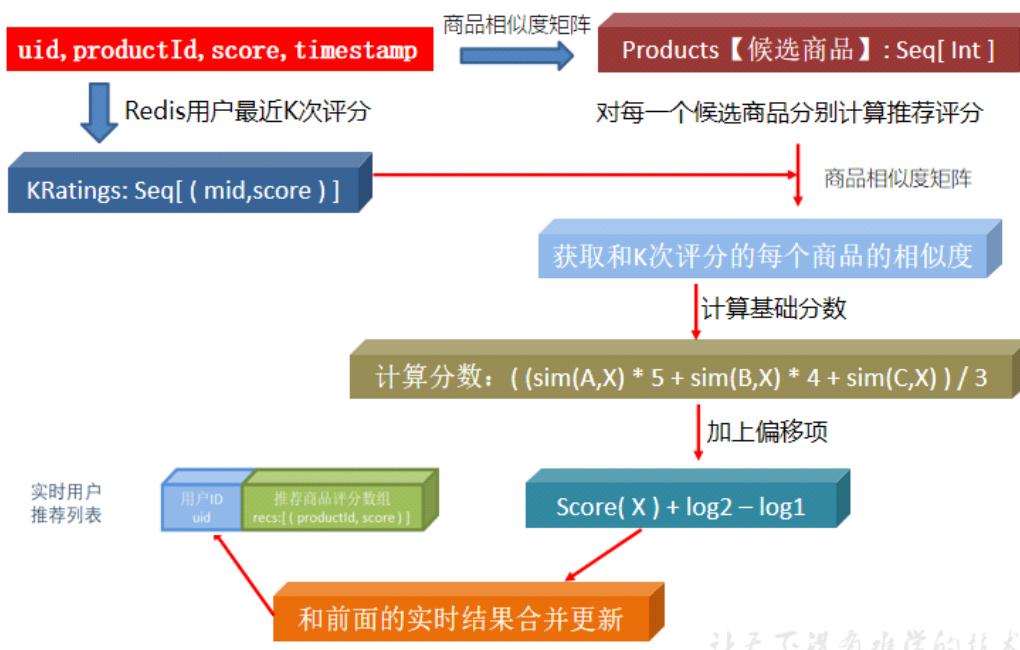
基本原理：用户最近一段时间的口味是相似的

备选商品推荐优先级：

$$E_{uq} = \frac{\sum_{r \in RK} sim(q, r) \times R_r}{sim_sum} + \lg \max\{incount, l\} - \lg \max\{recount, l\}$$



推荐优先级计算



基于内容的推荐

- 基于商品的用户标签信息，用TF-IDF算法提取特征向量

$$TF_{i,j} = \frac{n_{i,j}}{n_{*,j}} \quad IDF_i = \log\left(\frac{N+1}{N_i+1}\right)$$

- 计算特征向量的余弦相似度，从而得到商品的相似列表

$$\cos\theta = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \times \|\mathbf{b}\|} = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2} \times \sqrt{\sum_i y_i^2}}$$

- 在实际应用中，一般会在商品详情页、或商品购买页将相似商品推荐出来

基于物品的协同过滤推荐

- 基于物品的协同过滤 (Item-CF)，只需收集用户的常规行为数据（比如点击、收藏、购买）就可以得到商品间的相似度，在实际项目中应用很广
- “同现相似度”——利用行为数据计算不同商品间的相似度

$$w_{ij} = \frac{|N_i \cap N_j|}{\sqrt{|N_i||N_j|}}$$

其中， N_i 是购买商品 i (或对商品 i 评分) 的用户列表， N_j 是购买商品 j 的用户列表

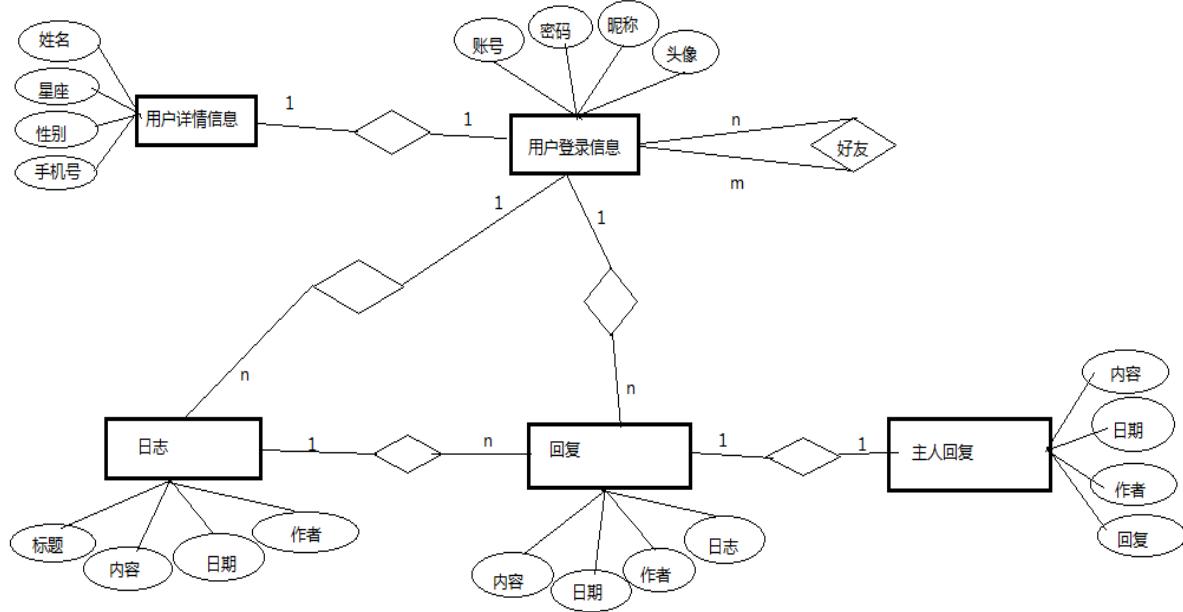
第二个小项目--QQZone

2022年1月28日 21:06

步骤：（有静态网页框架的情况下）

先复制 myssm 包（就是手写的spring） -> 建立数据库 -> 写 pojo 包（与数据库表对应的 class 类） -> 写 dao 包（操作数据库） -> 写 service 包（面向业务，调用 dao，例如登录……） -> 写 Controller 包（调用 service，返回 index 等，转到 servlet 去） -> 配置 html（配置 action 为 user.do，增加 hidden 提交 operate 方法给 Controller） -> 配置 Thymeleaf 进行交互（each 表格等等） -> 整理业务逻辑（session 里面东西的管理）

数据库设计



1) 抽取实体: 用户、日志、回贴

2) 分析其中的属性:

- 用户登录信息: 账号、密码、头像、昵称
- 用户详情信息: 真实姓名、星座、血型、邮箱、手机号……
- 日志: 标题、内容、日期、作者
- 回复: 内容、日期、作者、回复谁的日志
- 主人回复: 内容、日期、作者、回复准

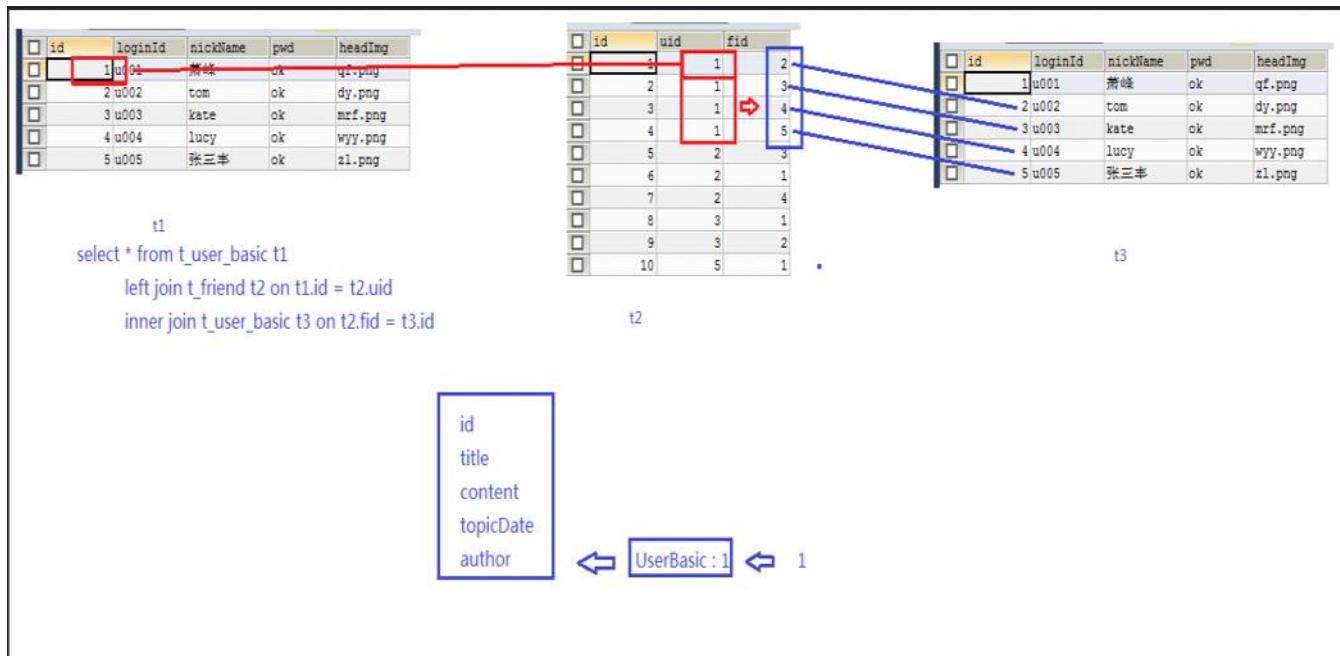
3) 分析实体之间的关系

- 用户登录信息 : 用户详情信息 1: 1 PK
- 用户 : 日志 1: N
- 日志 : 回复 1: N
- 回复 : 主人回复 1: 1 UK
- 用户 : 好友 M : N

数据库的范式：

- 1) 第一范式：列不可再分
- 2) 第二范式：一张表只表达一层含义（只描述一件事情）
- 3) 第三范式：表中的每一列和主键都是直接依赖关系，而不是间接依赖

朋友关系效果图



SQL的查询

2、left join

left join(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录。

2.1、sql语句

```
select * from app01_publisher left join app01_book on app01_publisher.id = app01_book.publish_id
```

3、right join

right join(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录。

3.1、sql语句

```
select * from app01_publisher right join app01_book on app01_publisher.id = app01_book.publish_id
```

4、inner join

inner join(等值连接) 只返回两个表中联结字段相等的行。

4.1、sql语句

```
select * from app01_publisher inner join app01_book on app01_publisher.id = app01_book.publish_id
```

展示好友列表， session 中是 userbasic

```
<li th:if="#lists.isEmpty(session.userBasic.friendList)">一个好友也没有</li>
```

```
<li th:unless="#lists.isEmpty(session.userBasic.friendList)" th:each="friend: ${session.userBasic.friendList}" th:text="${friend.nickName}"></li>
```

关于 thymeleaf 不能渲染

thymeleaf 是在服务器端渲染，如果直接请求页面或者直接在页面中跳转，没有经过服务器端，就没有生效。



解决办法

```
<iframe th:src="@{/page.do?operate=page&page=frames/left}" width="100%" height="100%>
<iframe th:src="@{/page.do?operate=page&page=frames/main}" scrolling="no">
```

先写一个 PageController，用来接受 operate = page 的反射

```
public class PageController {
    public String page(String page) { return page; } // frames/left
}
```

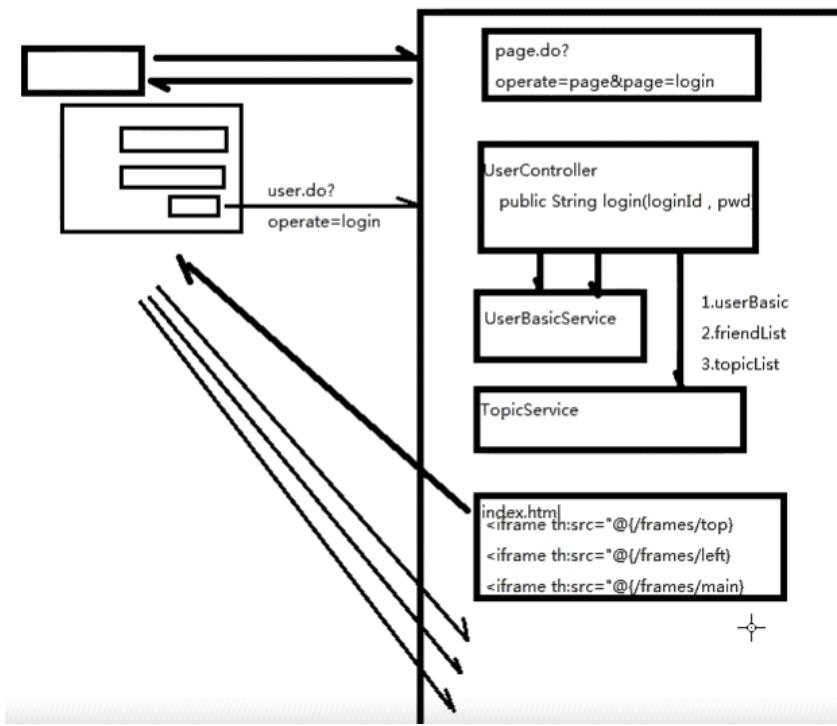
再利用中央控制器的 processTemplate 重定向到 xx 页面

```
//3. 视图处理
String methodReturnStr = (String) returnObj ;
if(methodReturnStr.startsWith("redirect:")){
    String redirectStr = methodReturnStr.substring("redirect:".length());
    response.sendRedirect(redirectStr);
} else{
    super.processTemplate(methodReturnStr, request, response); // 比如： "edit"
}
```

同理在 tomcat 里面更改首页

URL: http://localhost:8080/pro21/page.do?operate=page&page=login

目前架构



进入好友空间

获取当前人的信息保存到 session map 的 friend 里面

```

public String friend(Integer id, HttpSession session){
    //1.根据id获取指定的用户信息
    UserBasic currFriend = userBasicService.getUserBasicById(id);

    List<Topic> topicList = topicService.getTopicList(currFriend);

    currFriend.setTopicList(topicList);

    session.setAttribute("friend", currFriend);

    return "index";
}

```

Html 里面 ,调用 usercontroller 里面的 friend 函数获取朋友空间 , target 到 top (本窗口)

```

<li th:unless="#lists.isEmpty(session.userBasic.friendList)" th:each="friend : ${session.userBasic.friendList}">
    <a th:href="@{/user.do?operate=friend&id=${friend.id}}" th:text="${friend.nickName}" target="top">
</li>

```

获取朋友空间: friend 函数覆盖了 session map 里面的 friend , 窗口一直展示的是 friend 里面的内容

具体思路

1. top.html页面显示登录者昵称、判断是否是自己的空间
 - 1)显示登录者昵称: \${session.userBasic.nickName}
 - 2)判断是否是自己的空间 : \${session.userBasic.id!=session.friend.id}
2. 点击左侧的好友链接，进入好友空间

- 1) 根据id获取指定userBasic信息，查询这个userBasic的topicList，然后覆盖friend对应的value
- 2) main页面应该展示friend中的topicList，而不是userBasic中的topicList
- 3) 跳转后，在左侧(left)中显示整个index页面
 - 问题：在left页面显示整个index布局
 - 解决：给超链接添加target属性： target="_top" 保证在顶层窗口显示整个index页面

- 4) top.html页面需要修改： "欢迎进入\${session.friend}"
- top.html页面的返回自己空间的超链接需要修改：
- ```
<a th:href="@{|/user.do?operate=friend&id=${session.userBasic.id}|}" target="_top">
```

## 点击日志查看详情

### 任务

- 1) 已知topic的id，需要根据topic的id获取特定topic
- 2) 获取这个topic关联的所有回复
- 3) 如果某个回复有主人回复，需要查询出来
  - 在TopicController中获取指定的topic
  - 具体这个topic中关联多少个Reply，由ReplyService内部实现
- 4) 获取到的topic中的author只有id，那么需要在topicService的getTopic方法中封装，在查询topic本身信息时，同时调用userBasicService中的获取userBasic方法，给author属性赋值
- 5) 同理，在reply类中也有author，而且这个author也是只有id，那么我们也要根据id查询得到author，最后设置关联。

首先是点击标题

```
><a th:href="@{|/topic.do?operate=topicDetail&id=${topic.id}|}" th:text="${topic.title}">我乔峰要
```

转到 topiccontroller 的 topicdetail 函数 ----- 第 1 步

```
public String topicDetail(Integer id , HttpSession session){
 Topic topic = topicService.getTopicById(id);

 session.setAttribute("topic", topic);
 return "frames/detail";
}
```

获取 topic 到 session，再跳转到 detail 页面显示

上面的 getTopicByid 函数，需要获取 topic 的回复列表 ----- 第 2 步

```
@Override
public Topic getTopicById(Integer id) {
 Topic topic = getTopic(id);
 List<Reply> replyList = replyService.getReplyListByTopicId(topic.getId());
 topic.setReplyList(replyList);

 return topic ;
}
```

上面的 getReply... 函数获取回复列表，并根据每个回复列表获取 主人回复 ----- 第 3 步

```
@Override
public List<Reply> getReplyListByTopicId(Integer topicId) {
 List<Reply> replyList = replyDAO.getReplyList(new Topic(topicId));
 for (int i = 0; i < replyList.size(); i++) {
 Reply reply = replyList.get(i);
 //1. 将关联的作者设置进去
 UserBasic author = userBasicService.getUserBasicById(reply.getAuthor().getId());
 reply.setAuthor(author);

 //2. 将关联的HostReply 设置进去
 HostReply hostReply = hostReplyService.getHostReplyByReplyId(reply.getId());
 reply.setHostReply(hostReply);
 }
 return replyList;
}
```

### 添加回复 (操作完后要重定向刷新)

网页，通过 name + value 传进 controller

```
<form action="reply.do" method="post">
 <input type="hidden" name="operate" value="addReply"/>
 <input type="hidden" name="topicId" th:value="${session.topic.id}" />
 <table>
 <tr>
 <th style="...">回复日志: </th>
 <td><input type="text" th:value="| ${session.topic.title} |" value="『萧某今天就和天下群雄决" />
 </tr>
 <tr>
 <th>回复内容1: </th>
 <td><textarea name="content" rows="3">这里是另一个回复! </textarea></td>
 </tr>
 <tr>
 <th colspan="2">
 <input type="submit" value="回 复 "/>
 <input type="reset" value="重 置 "/>
 </th>
 </tr>
 </table>
</form>
```

ReplyController里面

```
public String addReply(String content ,Integer topicId , HttpSession session){
 UserBasic author = (UserBasic)session.getAttribute("userBasic");
 Reply reply = new Reply(content , new Date() , author , new Topic(topicId));
 replyService.addReply(reply);
 return "redirect:topic.do?operate=topicDetail&id="+topicId;
 // detail.html
}
```

Replyserver里面

```
@Override
public void addReply(Reply reply) {
 replyDAO.addReply(reply);
}
```

## 删除回复

网页 转到 JavaScript

```
th:onclick="|delReply(${reply.id} , ${session.topic.id})|"/>
```

```
function delReply(replyId , topicId){
 if(window.confirm("是否确认删除? ")){
 window.location.href='reply.do?operate=delReply&replyId='+replyId+'&topicId='+topicId;
 }
}
```

ReplyController里面

```
public String delReply(Integer replyId , Integer topicId){
 replyService.delReply(replyId);
 return "redirect:topic.do?operate=topicDetail&id="+topicId;
}
```

时间字符串转换

```
#{dates.format(topic.topicDate , 'yyyy-MM-dd HH:mm:ss')}
```

## 项目回顾

1.系统启动时，我们访问的页面是：<http://localhost:8080/pro23/page.do?operate=page&page=login>  
为什么不是：<http://localhost:8080/pro23/login.html>？

答：如果是后者，那么属于直接访问静态页面。那么页面上的thymeleaf表达式（标签）浏览器是不能识别的  
我们访问前者的目的其实就是要执行 ViewBaseServlet中的processTemplate()

2.<http://localhost:8080/pro23/page.do?operate=page&page=login> 访问这个URL，执行的过程是什么样的？

答：

```
http:// localhost :8080 /pro23 /page.do ?operate=page&page=login
协议 ServerIP port context root request.getServletPath() query string
1) DispatcherServlet -> urlPattern : *.do 拦截/page.do
2) request.getServletPath() -> /page.do
3) 解析处理字符串，将/page.do -> page
4) 拿到page这个字符串，然后去IOC容器（BeanFactory）中寻找id=page的那个bean对象 -> PageController.java
5) 获取operate的值 -> page 因此得知，应该执行 PageController中的page()方法
6) PageController中的page方法定义如下：
```

```

public String page(String page){
 return page;
}

7) 在queryString: ?operate=page&page=login 中 获取请求参数，参数名是page，参数值是login
因此page方法的参数page值会被赋上"login"
然后return "login" , return 给 谁？？

8) 因为PageController的page方法是DispatcherServlet通过反射调用的
method.invoke(...);
因此，字符串"login"返回给DispatcherServlet

9) DispatcherServlet接收到返回值，然后处理视图
目前处理视图的方式有两种： 1.带前缀redirect: 2.不带前缀
当前，返回"login"，不带前缀
那么执行 super.processTemplate("login",request,response);

10) 此时ViewBaseServlet中的processTemplate方法会执行，效果是：
在"login"这个字符串前面拼接 "/" (其实就是配置文件中view-prefix配置的值)
在"login"这个字符串后面拼接 ".html" (其实就是配置文件中view-suffix配置的值)
最后进行服务器转发

```

## 目前我们进行javaweb项目开发的“套路”是这样的：

1. 拷贝 myssm包
2. 新建配置文件applicationContext.xml或者可以不叫这个名字，在web.xml中指定文件名
3. 在web.xml文件中配置：
  - 1) 配置前缀和后缀，这样thymeleaf引擎就可以根据我们返回的字符串进行拼接，再跳转
 

```

<context-param>
 <param-name>view-prefix</param-name>
 <param-value>/</param-value>
</context-param>
<context-param>
 <param-name>view-suffix</param-name>
 <param-value>.html</param-value>
</context-param>
```
  - 2) 配置监听器要读取的参数，目的是加载IOC容器的配置文件（也就是applicationContext.xml）
 

```

<context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>applicationContext.xml</param-value>
</context-param>
```
4. 开发具体的业务模块：
  - 1) 一个具体的业务模块纵向上由几个部分组成：
    - html页面
    - POJO类
    - DAO接口和实现类
    - Service接口和实现类
    - Controller控制器组件
  - 2) 如果html页面有thymeleaf表达式，一定不能够直接访问，必须要经过PageController
  - 3) 在applicationContext.xml中配置 DAO、Service、Controller，以及三者之间的依赖关系
  - 4) DAO实现类中，继承BaseDAO，然后实现具体的接口，需要注意，BaseDAO后面的泛型不能写错。

例如：

```
public class UserDAOImpl extends BaseDAO<User> implements UserDAO{}
```
- 5) Service是业务控制类，这一层我们只需要记住一点：
  - 业务逻辑我们都封装在service这一层，不要分散在Controller层。也不要出现在DAO层（我们需要保证DAO方法的单精度特性）
  - 当某一个业务功能需要使用其他模块的业务功能时，尽量的调用别人的service，而不是深入到其他模块的DAO细节
- 6) Controller类的编写规则
  - ① 在applicationContext.xml中配置Controller
 

```
<bean id="user" class="com.atguigu.qqzone.controllers.UserController">
```

 那么，用户在前端发请求时，对应的servletpath就是 /user.do，其中的“user”就是对应此处的bean的id值
  - ② 在Controller中设计的方法名需要和operate的值一致

```
public String login(String loginId , String pwd , HttpSession session){
 return "index";
}
```

因此，我们的登录验证的表单如下：

```
<form th:action="@{/user.do}" method="post">
<input type="hidden" name="operate" value="login"/>
</form>
```

③ 在表单中，组件的name属性和Controller中方法的参数名一致

```
<input type="text" name="loginId" />
```

```
public String login(String loginId , String pwd , HttpSession session){
```

④ 另外，需要注意的是：Controller中的方法中的参数不一定都是通过请求参数获取的

```
if("request".equals...) else if("response".equals...) else if("session".equals...){
```

直接赋值

```
}else{
```

此处才是从request的请求参数中获取

```
request.getParameter("loginId")
```

```
}
```

7) DispatcherServlet中步骤大致分为：

0. 从application作用域获取IOC容器

1. 解析servletPath，在IOC容器中寻找对应的Controller组件

2. 准备operate指定的方法所要求的参数

3. 调用operate指定的方法

4. 接收到执行operate指定的方法的返回值，对返回值进行处理- 视图处理

8) 为什么DispatcherServlet能够从application作用域获取到IOC容器？

ContextLoaderListener在容器启动时会执行初始化任务，而它的操作就是：

1. 解析IOC的配置文件，创建一个一个的组件，并完成组件之间依赖关系的注入

2. 将IOC容器保存到application作用域

6. 修改BaseDAO，让其支持properties文件以及druid数据源连接池

讲解了两种方式：

1) 直接自己配置properties，然后读取，然后加载驱动.....

2) 使用druid连接池技术，那么properties中的key是有要求的

# 数据加载

2022年2月2日 14:10

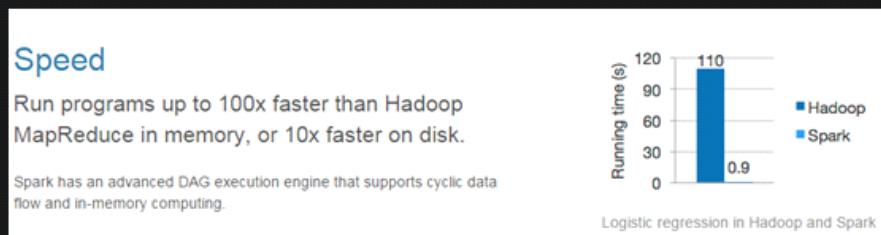
## 什么是spark

Spark是一种快速、通用、可扩展的大数据分析引擎

### Spark特点

#### 快

- 与Hadoop的MapReduce相比，Spark基于内存的运算要快100倍以上，基于硬盘的运算也要快10倍以上。Spark实现了高效的DAG(有向无环图)执行引擎，可以通过基于内存来高效处理数据流。



#### 易用

- Spark支持Java、Python和Scala的API，还支持超过80种高级算法，使用户可以快速构建不同的应用。而且Spark支持交互式的Python和Scala的shell，可以非常方便地在这些shell中使用Spark集群来验证解决问题的方法。

#### Ease of Use

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

```
text_file = spark.textFile("hdfs://...")
text_file.flatMap(lambda line: line.split())
 .map(lambda word: (word, 1))
 .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

#### 通用

- Spark提供了统一的解决方案。Spark可以用于批处理、交互式查询（Spark SQL）、实时流处理（Spark Streaming）、机器学习（Spark MLlib）和图计算（GraphX）。这些不同类型的处理都可以在同一个应用中无缝使用。Spark统一的解决方案非常具有吸引力，毕竟任何公司都想要用统一的平台去处理遇到的问题，减少开发和维护的人力成本和部署平台的物力成本。

### 3.3 数据初始化到 MongoDB

#### 3.3.1 启动 MongoDB 数据库（略）

#### 3.3.2 数据加载程序主体实现

我们会为原始数据定义几个样例类，通过 `SparkContext` 的 `textFile` 方法从文件中读取数据，并转换成 `DataFrame`，再利用 Spark SQL 提供的 `write` 方法进行数据的分布式插入。

在 `DataLoader/src/main/scala` 下新建 package，命名为 `com.atguigu.recommender`，新建名为 `DataLoader` 的 `scala class` 文件。

### 创建 spark session

```
// 创建一个spark config
val sparkConf = new SparkConf().setMaster(config("spark.cores")).setAppName("DataLoader")
// 创建spark session
val spark = SparkSession.builder().config(sparkConf).getOrCreate()
```

### 从文本读取数据

#### 一、RDD的概述

##### 1.1 什么是RDD？

RDD (Resilient Distributed Dataset) 叫做弹性分布式数据集，是Spark中最基本的数据抽象，它代表一个不可变、可分区、里面的元素可并行计算的集合。RDD具有数据流模型的特点：自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将工作集缓存在内存中，后续的查询能够重用工作集，这极大地提升了查询速度。

##### 1.2 RDD的属性

(1) 一组分片 (Partition)，即数据集的基本组成单位。对于RDD来说，每个分片都会被一个计算任务处理，并决定并行计算的粒度。用户可以在创建RDD时指定RDD的分片个数，如果没有指定，那么就会采用默认值。默认值就是程序所分配到的CPU Core的数目。

(2) 一个计算每个分区的函数。Spark中RDD的计算是以分片为单位的，每个RDD都会实现compute函数以达到这个目的。compute函数会对迭代器进行复合，不需要保存每次计算的结果。

(3) RDD之间的依赖关系。RDD的每次转换都会生成一个新的RDD，所以RDD之间就会形成类似于流水线一样的前后依赖关系。在部分分区数据丢失时，Spark可以通过这个依赖关系重新计算丢失的分区数据，而不是对RDD的所有分区进行重新计算。

(4) 一个Partitioner，即RDD的分片函数。当前Spark中实现了两种类型的分片函数，一个是基于哈希的HashPartitioner，另外一个是基于范围的RangePartitioner。只有对于key-value的RDD，才会有Partitioner，非key-value的RDD的Partitioner的值是None。Partitioner函数不但决定了RDD本身的分片数量，也决定了parent RDD Shuffle输出时的分片数量。

(5) 一个列表，存储存取每个Partition的优先位置 (preferred location)。对于一个HDFS文件来说，这个列表保存的就是每个Partition所在的块的位置。按照“移动数据不如移动计算”的理念，Spark在进行任务调度的时候，会尽可能地将计算任务分配到其所要处理数据块的存储位置。

## 二、RDD的创建方式

### 2.1 通过读取文件生成的

由外部存储系统的数据集创建，包括本地的文件系统，还有所有Hadoop支持的数据集，比如HDFS、Cassandra、HBase等

```
scala> val file = sc.textFile("/spark/hello.txt")
file: org.apache.spark.rdd.RDD[String] = /spark/hello.txt MapPartitionsRDD[26] at textFile at <console>:24
scala> [REDACTED] 返回类型
```

对其每个 item 分割后转成 DataFrame

```
// 加载数据
val productRDD = spark.sparkContext.textFile(PRODUCT_DATA_PATH)
val productDF = productRDD.map(item => {
 // product数据通过^分隔，切分出来
 val attr = item.split(regex = "\\^")
 // 转换成Product
 // trim 去首尾空格
 Product(attr(0).toInt, attr(1).trim, attr(4).trim, attr(5).trim, attr(6).trim)
}).toDF()

val ratingRDD = spark.sparkContext.textFile(RATING_DATA_PATH)
val ratingDF = ratingRDD.map(item => {
 val attr = item.split(regex = ",")
 Rating(attr(0).toInt, attr(1).toInt, attr(2).toDouble, attr(3).toInt)
}).toDF()
```

## 2. 什么是 Spark SQL DataFrame? ⚡

从Spark1.3.0版本开始，DF开始被定义为指定到列的数据集（Dataset）。DFS类似于关系型数据库中的表或者像R/Python 中的data frame。可以说是一个具有良好优化技术的关系表。DataFrame背后的思想是允许处理大量结构化数据。DataFrame包含带schema的行。schema是数据结构的说明。

在Apache Spark里面DF优于RDD，但也包含了RDD的特性。RDD和DataFrame的共同特征是不可变性、内存运行、弹性、分布式计算能力。它允许用户将结构强加到分布式数据集合上。因此提供了更高层次的抽象。我们可以从不同的数据源构建DataFrame。例如结构化数据文件、Hive中的表、外部数据仓库或现有的RDDs。DataFrame的应用程序编程接口(api)可以在各种语言中使用。示例包括Scala、Java、Python和R。在Scala和Java中，我们都将DataFrame表示为行数据集。在Scala API中，DataFrames是Dataset[Row]的类型别名。在Java API中，用户使用数据集<Row>来表示数据流。

## 3. 为什么要用 DataFrame?

DataFrame优于RDD，因为它提供了内存管理和优化的执行计划。总结为一下两点：

- a.自定义内存管理：当数据以二进制格式存储在堆外内存时，会节省大量内存。除此之外，没有垃圾回收（GC）开销。还避免了昂贵的Java序列化。因为数据是以二进制格式存储的，并且内存的schema是已知的。
- b.优化执行计划：这也称为查询优化器。可以为查询的执行创建一个优化的执行计划。优化执行计划完成后最终将在RDD上运行执行。

隐式配置 MongoDB 连接，读完数据关闭 spark

```
implicit val mongoConfig = MongoConfig(config("mongo.uri"), config("mongo.db"))
storeDataInMongoDB(productDF, ratingDF)

spark.stop()
```



## MongoDB 的连接与写入

```
// 定义mongodb中存储的表名
val MONGODB_PRODUCT_COLLECTION = "Product"
val MONGODB_RATING_COLLECTION = "Rating"
```

```
def storeDataInMongoDB(productDF: DataFrame, ratingDF: DataFrame)(implicit mongoConfig: MongoConfig): Unit = {
```

### 函数内部

```
// 新建一个mongodb的连接，客户端
val mongoClient = MongoClient(MongoClientURI(mongoConfig.uri))
// 定义要操作的mongodb表，可以理解为 db.Product
val productCollection = mongoClient(mongoConfig.db)(MONGODB_PRODUCT_COLLECTION)
val ratingCollection = mongoClient(mongoConfig.db)(MONGODB_RATING_COLLECTION)
```

```
// 将当前数据存入对应的表中
productDF.write
 .option("uri", mongoConfig.uri)
 .option("collection", MONGODB_PRODUCT_COLLECTION)
 .mode(saveMode = "overwrite")
 .format(source = "com.mongodb.spark.sql")
 .save()

ratingDF.write
 .option("uri", mongoConfig.uri)
 .option("collection", MONGODB_RATING_COLLECTION)
 .mode(saveMode = "overwrite")
 .format(source = "com.mongodb.spark.sql")
 .save()
```

```
// 对表创建索引
productCollection.createIndex(MongoDBObject("productId" -> 1))
ratingCollection.createIndex(MongoDBObject("productId" -> 1))
ratingCollection.createIndex(MongoDBObject("userId" -> 1))

mongoClient.close()
```

# Vue 和 Axios 初步

2022年2月5日 14:47

## 导入vue



```
</body>
<script src="/pro03-vue/script/vue.js" type="text/javascript" charset="utf-8"></script>
<script type="text/javascript">

</script>
```

## 第三节 Vue.js基本语法：声明式渲染

### # 1、概念

#### ①声明式

『声明式』是相对于『编程式』而言的。

- 声明式：告诉框架做什么，具体操作由框架完成
- 编程式：自己编写代码完成具体操作

```
<!-- 使用{{}}格式，指定要被渲染的数据 -->
<div id="app">{{message}}</div>
```

+

```
// 1.创建一个JSON对象，作为new Vue时要使用的参数
var argumentJson = {

 // el用于指定Vue对象要关联的HTML元素。el就是element的缩写
 // 通过id属性值指定HTML元素时，语法格式是：#id
 "el": "#app",

 // data属性设置了Vue对象中保存的数据
 "data": {
 "message": "Hello Vue!"
 }
};

// 2.创建Vue对象，传入上面准备好的参数
var app = new Vue(argumentJson);
```

## 第四节 Vue.js基本语法：绑定元素属性

### 1、基本语法

v-bind:HTML标签的原始属性名

## ①HTML代码

```
1 <div id="app">
2 <!-- v-bind:value表示将value属性交给Vue来进行管理，也就是绑定到Vue对象 -->
3 <!-- vueValue是一个用来渲染属性值的表达式，相当于标签体中加{{}}的表达式 -->
4 <input type="text" v-bind:value="vueValue" />
5
6 <!-- 同样的表达式，在标签体内通过{{}}告诉Vue这里需要渲染； -->
7 <!-- 在HTML标签的属性中，通过v-bind:属性名="表达式"的方式告诉Vue这里要渲染 -->
8 <p>{{vueValue}}</p>
9
10 </div></pre>
```

## ②Vue代码

```
1 // 创建Vue对象，挂载#app这个div标签
2 var app = new Vue({
3 "el": "#app",
4 "data": {
5 "vueValue": "太阳当空照"
6 }
})
```

## 双向绑定

使用了双向绑定后，就可以实现：页面上数据被修改后，Vue对象中的数据属性也跟着被修改。

## ①HTML代码

```
1 <div id="app">
2 <!-- v-bind:属性名 效果是从Vue对象渲染到页面 -->
3 <!-- v-model:属性名 效果不仅是从Vue对象渲染到页面，而且能够在页面上数据修改后反向修改Vue对
4 <input type="text" v-model:value="vueValue" />
5
6 <p>{{vueValue}}</p>
7 </div>
```

html

## ②Vue代码

```
1 // 创建Vue对象，挂载#app这个div标签
2 var app = new Vue({
3 "el": "#app",
4 "data": {
5 "vueValue": "太阳当空照"
6 }
7});
```

js

## # ③页面效果

p标签内的数据能够和文本框中的数据实现同步修改：

  
太阳当空照11333555

### 3、去除前后空格

#### ①:value可以省略

```
1 <input type="text" v-model="vueValue" />
```

html

#### ②.trim修饰符

实际开发中，要考虑到用户在输入数据时，有可能会包含前后空格。而这些前后的空格对我们程序运行来说都是干扰因素，要去掉。在v-model后面加上.trim修饰符即可实现。

```
1 <input type="text" v-model.trim="vueValue" />
```

html

Vue会帮助我们在文本框失去焦点时自动去除前后空格。

if-else

## # 2、v-if和v-else

### ①HTML代码

```
1 <div id="app02">
2 <h3>if/else</h3>
3
4
5 </div>
```

### ②Vue代码

```
1 var app02 = new Vue({
2 "el": "#app02",
3 "data": {
4 "flag": true
5 }
6 });
```

## 3、v-show

### ①HTML代码

```
1 <div id="app03">
2 <h3>v-show</h3>
3
4 </div>
```

v-if 是直接注释掉整个标签，id 还在，show 只是 disable 了显示。

## 迭代

### ①HTML代码

```
1 <div id="app01">
2
3 <!-- 使用v-for语法遍历数组 -->
4 <!-- v-for的值是语法格式是：引用数组元素的变量名 in Vue对象中的数组属性名 -->
5 <!-- 在文本标签体中使用{{引用数组元素的变量名}}渲染每一个数组元素 -->
6 <li v-for="fruit in fruitList">{{fruit}}
7
8 </div>
```

### ②Vue代码

```
1 var app01 = new Vue({
2 "el": "#app01",
3 "data": {
4 "fruitList": [
5 "apple",
6 "banana",
7 "orange",
8 "grape",
9 "dragonfruit"
10]
11 }
12});
```

## 事件驱动

### ①HTML代码

```
1 <div id="app">
2 <p>{{message}}</p>
3
4 <!-- v-on:事件类型="事件响应函数的函数名" -->
5 <button v-on:click="reverseMessage">Click me, reverse message</button>
6 </div>
```

```
4 <!-- v-on:事件类型="事件响应函数的函数名" -->
5 <button v-on:click="reverseMessage">Click me, reverse message</button>
6 </div>
```

不加括号

```
1 var app = new Vue({
2 "el": "#app",
3 "data": {
4 "message": "Hello Vue!"
5 },
6 "methods": {
7 "reverseMessage": function(){
8 this.message = this.message.split("").reverse().join("");
9 }
10 }
11 });
```

## 2、demo：获取鼠标移动时的坐标信息

### ①HTML代码

```
1 <div id="app">
2 <div id="area" v-on:mousemove="recordPosition"></div>
3 <p id="showPosition">{{position}}</p>
4 </div>
```

### ②Vue代码

```
1 var app = new Vue({
2 "el": "#app",
3 "data": {
4 "position": "暂时没有获取到鼠标的位置信息"
5 },
6 "methods": {
7 "recordPosition": function(event){
8 this.position = event.clientX + " " + event.clientY;
9 }
10 }
11});
```

## 侦听属性

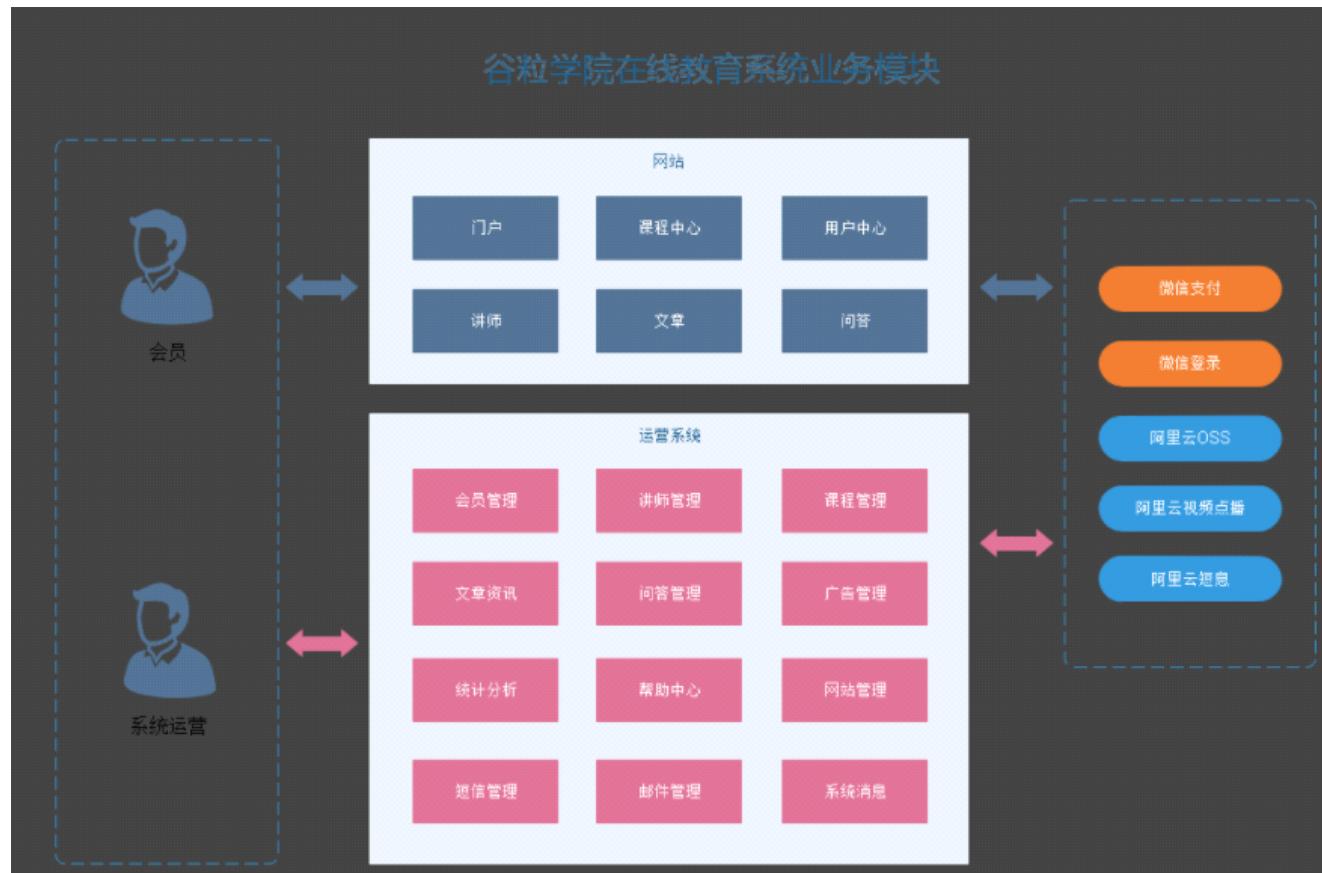
## 2、Vue代码

在watch属性中声明对firstName和lastName属性进行『侦听』的函数：

```
1 var app = new Vue({
2 "el": "#app",
3 "data": {
4 "firstName": "jim",
5 "lastName": "green",
6 "fullName": "jim green"
7 },
8 "watch": {
9 "firstName": function(inputValue){
10 this.fullName = inputValue + " " + this.lastName;
11 },
12 "lastName": function(inputValue){
13 this.fullName = this.firstName + " " + inputValue;
14 }
15 }
16 });
17
```

# 设计思路

2022年2月5日 21:26



# 架构





# MyBatis初步

2022年2月5日 21:53

## Spring 基于注解的配置

2019-05-11 17:49 更新

### 基于注解的配置

从 Spring 2.5 开始就可以使用注解来配置依赖注入。而不是采用 XML 来描述一个 bean 连线，你可以使用相关类，方法或字段声明的注解，将 bean 配置移动到组件类本身。

在 XML 注入之前进行注解注入，因此后者的配置将通过两种方式的属性连线被前者重写。

序号	注解 & 描述
1	<a href="#">@Required</a> @Required 注解应用于 bean 属性的 setter 方法。
2	<a href="#">@Autowired</a> @Autowired 注解可以应用到 bean 属性的 setter 方法，非 setter 方法，构造函数和属性。
3	<a href="#">@Qualifier</a> 通过指定确切的将被连线的 bean，@Autowired 和 @Qualifier 注解可以用来删除混乱。
4	<a href="#">JSR-250 Annotations</a> Spring 支持 JSR-250 的基础的注解，其中包括了 @Resource，@PostConstruct 和 @PreDestroy 注解。

### Spring @Required 注解

**@Required** 注解应用于 bean 属性的 setter 方法，它表明受影响的 bean 属性在配置时必须放在 XML 配置文件中，否则容器就会抛出一个 BeanInitializationException 异常。下面显示的是一个使用 @Required 注解的示例。

# Spring @Autowired 注解

2021-08-17 14:28 更新

使用 Spring 开发时，进行配置主要有两种方式，一是 xml 的方式，二是 java config 的方式。Spring 技术自身也在不断的发展和改变，从当前 Springboot 的火热程度来看，java config 的应用是越来越广泛了，在使用 java config 的过程当中，我们不可避免的会有各种各样的注解打交道，其中，我们使用最多的注解应该就是 @Autowired 注解了。这个注解的功能就是为我们注入一个定义好的 bean。

## @Autowired 注解用法

在分析这个注解的实现原理之前，我们不妨先来回顾一下 @Autowired 注解的用法。

将 @Autowired 注解应用于构造函数，如以下示例所示

```
public class MovieRecommender {

 private final CustomerPreferenceDao customerPreferenceDao;

 @Autowired
 public MovieRecommender(CustomerPreferenceDao customerPreferenceDao) {
 this.customerPreferenceDao = customerPreferenceDao;
 }

 // ...
}
```

将 @Autowired 注解应用于 setter 方法

```
public class SimpleMovieLister {

 private MovieFinder movieFinder;

 @Autowired
 public void setMovieFinder(MovieFinder movieFinder) {
 this.movieFinder = movieFinder;
 }

 // ...
}
```

# Spring @Qualifier 注解

2021-08-17 14:38 更新

## Spring @Qualifier 注解

可能会有这样一种情况，当你创建多个具有相同类型的 bean 时，并且想要用一个属性只为它们其中的一个进行装配，在这种情况下，你可以使用 **@Qualifier** 注解和 **@Autowired** 注解通过指定哪一个真正的 bean 将会被装配来消除混乱。下面显示的是使用 **@Qualifier** 注解的一个示例。

```
public class Profile {
 @Autowired
 @Qualifier("student1")
 private Student student;

<!-- Definition for student1 bean -->
<bean id="student1" class="com.tutorialspoint.Student">
 <property name="name" value="Zara" />
 <property name="age" value="11"/>
</bean>
```

## @Repository的作用

Spring的注解形式：@Repository、@Service、@Controller，它们分别对应存储层Bean，业务层Bean，和展示层Bean。

@Repository、@Service、@Controller 和 @Component 将类标识为Bean

Spring 自 2.0 版本开始，陆续引入了一些注解用于简化 Spring 的开发。@Repository注解便属于最先引入的一批，它用于将数据访问层（DAO 层）的类标识为 Spring Bean。具体只需将该注解标注在 DAO类上即可。同时，为了让 Spring 能够扫描类路径中的类并识别出 @Repository 注解，需要在 XML 配置文件中启用Bean 的自动扫描功能，这可以通过<context:component-scan/>实现。如下所示：

```
// 首先使用 @Repository 将 DAO 类声明为 Bean
package bookstore.dao;

// 其次，在 XML 配置文件中启动 Spring 的自动扫描功能
<beans ... >
 ...
<context:component-scan base-package="bookstore.dao" />
 ...
</beans>
```

[https://blog.csdn.net/qq\\_40943786](https://blog.csdn.net/qq_40943786)

为什么 @Repository 只能标注在 DAO 类上呢？这是因为该注解的作用不只是将类识别为Bean，同时它还能将所标注的类中抛出的数据访问异常封装为 Spring 的数据访问异常类型。Spring本身提供了一个丰富的并且是与具体的数据访问技术无关的数据访问异常结构，用于封装不同的持久层框架抛出的异常，使得异常独立于底层的框架。

## @Component注解的含义

### @Component注解的含义

来源

#### 一、注解分类



1. @controller: controller控制器层（注入服务）
  2. @service : service服务层（注入dao）
  3. @repository : dao持久层（实现dao访问）
  4. @component: 标注一个类为Spring容器的Bean，（把普通pojo实例化到spring容器中，相当于配置文件中的）
- 
1. @Service用于标注业务层组件
  2. @Controller用于标注控制层组件(如struts中的action)
  3. @Repository用于标注数据访问组件，即DAO组件.
  4. @Component泛指组件，当组件不好归类的时候，我们可以使用这个注解进行标注，标识为一个Bean。

## Mybatis



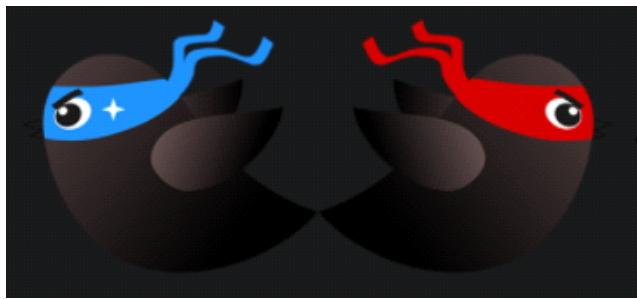
### 为什么使用 MyBatis

在我们传统的 JDBC 中，我们除了需要自己提供 SQL 外，还必须操作 Connection、Statement、ResultSet，不仅如此，为了访问不同的表，不同字段的数据，我们需要些很多雷同模板化的代码，闲的繁琐又枯燥。

而我们在使用了 MyBatis 之后，只需要提供 SQL 语句就好了，其余的诸如：建立连接、操作 Statement、ResultSet，处理 JDBC 相关异常等等都可以交给 MyBatis 去处理，我们的关注点于是可以就此集中在 SQL 语句上，关注在增删改查这些操作层面上。

并且 MyBatis 支持使用简单的 XML 或注解来配置和映射原生信息，将接口和 Java 的 POJOs(Plain Old Java Objects, 普通的 Java 对象)映射成数据库中的记录。

## Mybatis-plus 初步



## @MapperScan注解使用

Mybatis是一个优秀的持久层框架，什么是持久层呢？就是可以长时间保存数据到数据库或者硬盘当中，不会像放在内存中，一断电就丢失了。

@Mapper注解是由Mybatis框架中定义的一个描述数据层接口的注解，注解往往起到的都是一个描述性作用，用于告诉spring框架此接口的实现类由Mybatis负责创建，并将其实现类对象存储到spring容器中。

创建一个接口类，类的上面需要写上@Mapper注释用来表示该接口类的实现类对象交给mybatis底层创建，然后交由Spring框架管理。

1、@Mapper注解：

作用：在接口类上添加了@Mapper，在编译之后会生成相应的接口实现类

添加位置：接口类上面

```
@Mapper
public interface UserDAO {
 //代码
}
```

如果想要每个接口都要变成实现类，那么需要在每个接口类上加上@Mapper注解，比较麻烦，解决这个问题用@MapperScan

## 2、@MapperScan

作用：指定要变成实现类的接口所在的包，然后包下面的所有接口在编译之后都会生成相应的实现类  
添加位置：是在Springboot启动类上面添加，

```
@SpringBootApplication
@MapperScan("com.winter.dao")
public class SpringbootMybatisDemoApplication {

 public static void main(String[] args) {
 SpringApplication.run(SpringbootMybatisDemoApplication.class, args);
 }
}
```

添加@MapperScan("com.winter.dao")注解以后，com.winter.dao包下面的接口类，在编译之后都会生成相应的实现类

## BaseMapper

```
@Repository
public interface UserMapper extends BaseMapper<User> {
}
```

BaseMapper 里面自动实现了数据库操作方法

```
public interface BaseMapper<T> {
 int insert(T var1);

 int deleteById(Serializable var1);

 int deleteByMap(@Param("cm") Map<String, Object> var1);

 int delete(@Param("ew") Wrapper<T> var1);

 int deleteBatchIds(@Param("coll") Collection<? extends Serializable> var1);

 int updateById(@Param("et") T var1);

 int update(@Param("et") T var1, @Param("ew") Wrapper<T> var2);
```

## @Data注解与lombok

用@Data的写法：

```
1 @Data
2 @AllArgsConstructor
3 @NoArgsConstructor
4 public class Person {
5 private String name;
6 private String address;
7 private Integer age;
8 private String hobbit;
9 private String phone;
10 }
```

自动生成相关的方法：



操作数据库例子（全靠 BaseMapper）

```
@Repository
public interface UserMapper extends BaseMapper<User> {
}
```

```
@Autowired
private UserMapper userMapper;

//查询user表所有数据
@Test
public void findAll() {
 List<User> users = userMapper.selectList(wrapper: null);
 System.out.println(users);
}
```

```
//添加操作
@Test
public void addUser() {
 User user = new User();
 user.setName("岳不群1");
 user.setAge(70);
 user.setEmail("lucy@qq.com");
 int insert = userMapper.insert(user);
 System.out.println("insert:" + insert);
}
```

Id 是乱的

5	Billie
1490271160196157441	岳不群1

Id 策略 (在定义 user pojo 类时注解, 前提数据库要有自增)

## (2) 自增策略

- 要想主键自增需要配置如下主键策略
  - 需要在创建数据表的时候设置主键自增
  - 实体字段中配置 @TableId(type = IdType.AUTO)

```
1 @TableId(type = IdType.AUTO)
2 private Long id;
```

要想影响所有实体的配置，可以设置全局主键配置

```
1 #全局设置主键生成策略
2 mybatis-plus.global-config.db-config.id-type=auto
```

```
//修改操作
@Test
public void updateUser() {

 User user = new User();
 user.setId(1231103936770154497L);
 user.setAge(120);

 int row = userMapper.updateById(user);
 System.out.println(row);
}
```

加上这个可以详细看到 mybatis 是怎么生成 sql 语句的

```
mybatis日志
mybatis-plus.configuration.log-impl=org.apache.ibatis.logging.stdout.StdoutImpl
```

自动填充

实体上添加注解

回调函数

```
//create_time
@TableField(fill = FieldFill.INSERT)
private Date createTime;

//update_time
@TableField(fill = FieldFill.INSERT_UPDATE)
private Date updateTime;
```

实现元对象处理器接口，不要忘记添加 @Component 注解

一般写在 handle 包里面

```

@Component
public class MyMetaObjectHandler implements MetaObjectHandler {

 // 使用mp实现添加操作，这个方法执行
 @Override
 public void insertFill(MetaObject metaObject) {
 this.setFieldValByName(fieldName: "createTime", new Date(), metaObject);
 this.setFieldValByName(fieldName: "updateTime", new Date(), metaObject);

 this.setFieldValByName(fieldName: "version", fieldVal: 1, metaObject);
 }

 // 使用mp实现修改操作，这个方法执行
 @Override
 public void updateFill(MetaObject metaObject) { this.setFieldValByName(fieldName: "update"
}

```

## 乐观锁

**主要适用场景：**当要更新一条记录的时候，希望这条记录没有被别人更新，也就是说实现线程安全的数据更新

### 乐观锁实现方式：

- 取出记录时，获取当前version
- 更新时，带上这个version
- 执行更新时，`set version = newVersion where version = oldVersion`
- 如果version不对，就更新失败

添加 @Version 注解

```

@Version
@TableField(fill = FieldFill.INSERT)
private Integer version; // 版本号

```

### (3) 元对象处理器接口添加version的insert默认值

```

1 @Override
2 public void insertFill(MetaObject metaObject) {
3
4 this.setFieldValByName("version", 1, metaObject);

```

```
2 public void insertSelective(InsertObjectMeta object) {
3
4 this.setFieldValByName("version", 1, metaObject);
5 }
```

#### 特别说明:

- 支持的数据类型只有 int, Integer, long, Long, Date, Timestamp, LocalDateTime
- 整数类型下 newVersion = oldVersion + 1
- newVersion 会回写到 entity 中
- 仅支持 updateById(id) 与 update(entity, wrapper) 方法
- 在 update(entity, wrapper) 方法下, wrapper 不能复用!!!

配置乐观锁插件，配在 config 包里面

```
@Configuration
@MapperScan("eric/gulischool/mapper")
public class MpConfig {

 //乐观锁插件
 @Bean
 public OptimisticLockerInterceptor optimisticLockerInterceptor() { return new OptimisticLockerInterce
```

多 id 查询

```
//多个id批量查询
@Test
public void testSelectDemo1() {
 List<User> users = userMapper.selectBatchIds(Arrays.asList(1L, 2L, 3L));
 System.out.println(users);
}
```

## 4、分页

MyBatis Plus自带分页插件，只要简单的配置即可实现分页功能

### (1) 创建配置类

此时可以删除主类中的 `@MapperScan` 扫描注解

```
@Bean
public PaginationInterceptor paginationInterceptor() {
 return new PaginationInterceptor();
}
```

```
//分页查询
@Test
public void testPage() {
 //1 创建page对象
 //传入两个参数：当前页 和 每页显示记录数
 Page<User> page = new Page<>(current: 1, size: 3);
 //调用mp分页查询的方法
 //调用mp分页查询过程中，底层封装
 //把分页所有数据封装到page对象里面
 userMapper.selectPage(page, wrapper: null);

 //通过page对象获取分页数据
 System.out.println(page.getCurrent()); //当前页
 System.out.println(page.getRecords()); //每页数据list集合
 System.out.println(page.getSize()); //每页显示记录数
 System.out.println(page.getTotal()); //总记录数
 System.out.println(page.getPages()); //总页数

 System.out.println(page.hasNext()); //下一页
 System.out.println(page.hasPrevious()); //上一页
}
```

## 逻辑删除

```
@TableLogic
private Integer deleted;
```

```
//逻辑删除插件
@Bean
public ISqlInjector sqlInjector() { return new LogicSqlInjector(); }
```

#### (4) application.properties 加入配置

此为默认值，如果你的默认值和mp默认的一样,该配置可无

```
1 mybatis-plus.global-config.db-config.logic-delete-value=1
2 mybatis-plus.global-config.db-config.logic-not-delete-value=0
```

## 条件查询

## 二、AbstractWrapper

注意：以下条件构造器的方法入参中的 column 均表示数据库字段

### 1、ge、gt、le、lt、isNull、isNotNull

```
@Test
public void testDelete() {

 QueryWrapper<User> queryWrapper = new QueryWrapper<>();
 queryWrapper
 .isNull("name")
 .ge("age", 12)
 .isNotNull("email");
 int result = userMapper.delete(queryWrapper);
 System.out.println("delete return count = " + result);
}
```

ge : >=      gt : >  
le : <=      lt : <

```
queryWrapper.eq("name", "Tom");
```

eq : ==      ne : !=

## 2.1 allEq

```
1 | allEq(Map<R, V> params)
2 | allEq(Map<R, V> params, boolean null2IsNull)
3 | allEq(boolean condition, Map<R, V> params, boolean null2IsNull)
```

复制

- 全部eq(或个别isNull)

个别参数说明:

params : key为数据库字段名,value为字段值

null2IsNull : 为true则在map的value为null时调用 **isNull** 方法,为false时则忽略value为null的

- 例1: allEq({id:1, name:"老王", age:null})—>id = 1 and name = '老王' and age is null
- 例2: allEq({id:1, name:"老王", age:null}, false)—>id = 1 and name = '老王'

# 数据库与代码生成器

2022年2月6日 13:45

## 数据库设计规约

以下规约只针对本模块，更全面的文档参考《阿里巴巴Java开发手册》：五、MySQL数据库

- 1、库名与应用名称尽量一致
- 2、表名、字段名必须使用小写字母或数字，禁止出现数字开头，
- 3、表名不使用复数名词
- 4、表的命名最好是加上“业务名称\_表的作用”。如，edu\_teacher
- 5、表必备三字段：id, gmt\_create, gmt\_modified

说明：

其中 id 必为主键，类型为 bigint unsigned、单表时自增、步长为 1。

(如果使用分库分表集群部署，则id类型为varchar，非自增，业务中使用分布式id生成器)

gmt\_create, gmt\_modified 的类型均为 datetime 类型，前者现在时表示主动创建，后者过去分词表示被动更新。

6、单表行数超过 500 万行或者单表容量超过 2GB，才推荐进行分库分表。说明：如果预计三年后的数据量根本达不到这个级别，请不要在创建表时就分库分表。

7、表达是与否概念的字段，必须使用 is\_xxx 的方式命名，数据类型是 unsigned tinyint（1 表示是，0 表示否）。

说明：任何字段如果为非负数，必须是 unsigned。

注意：POJO 类中的任何布尔类型的变量，都不要加 is 前缀。数据库表示是与否的值，使用 tinyint 类型，坚持 is\_xxx 的命名方式是为了明确其取值含义与取值范围。

正例：表达逻辑删除的字段名 is\_deleted，1 表示删除，0 表示未删除。

8、小数类型为 decimal，禁止使用 float 和 double。说明：float 和 double 在存储的时候，存在精度损失的问题，很可能在值的比较时，得到不正确的结果。如果存储的数据范围超过 decimal 的范围，建议将数据拆成整数和小数分开存储。

9、如果存储的字符串长度几乎相等，使用 char 定长字符串类型。

10、varchar 是可变长字符串，不预先分配存储空间，长度不要超过 5000，如果存储长度大于此值，定义字段类型为 text，独立出来一张表，用主键来对应，避免影响其它字段索引效率。

11、唯一索引名为 uk\_字段名；普通索引名则为 idx\_字段名。

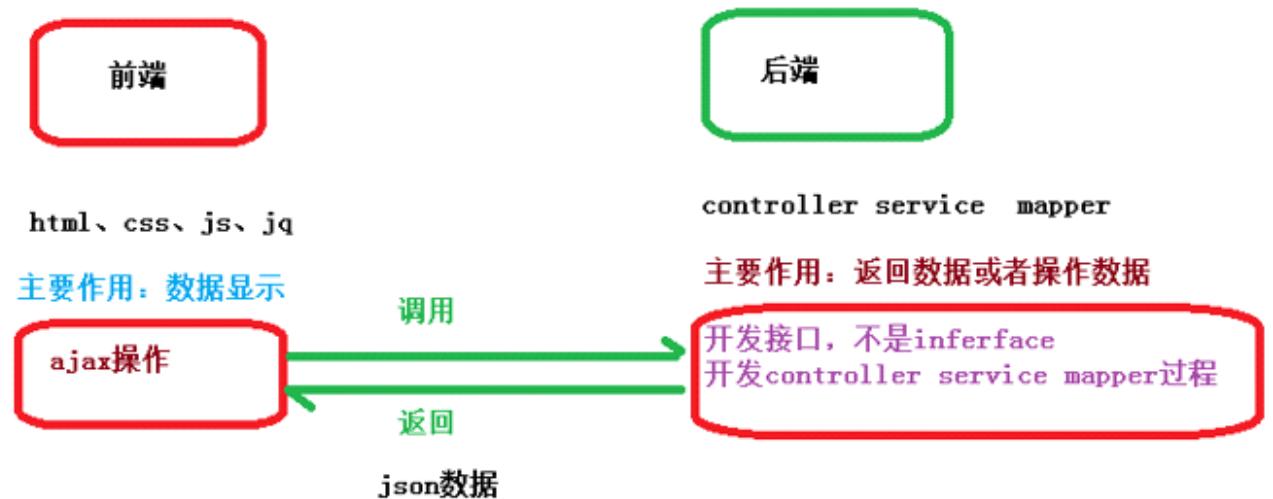
说明：uk\_ 即 unique key；idx\_ 即 index 的简称

12、不得使用外键与级联，一切外键概念必须在应用层解决。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

## 前后端分离

前后端分离开发

开发文档

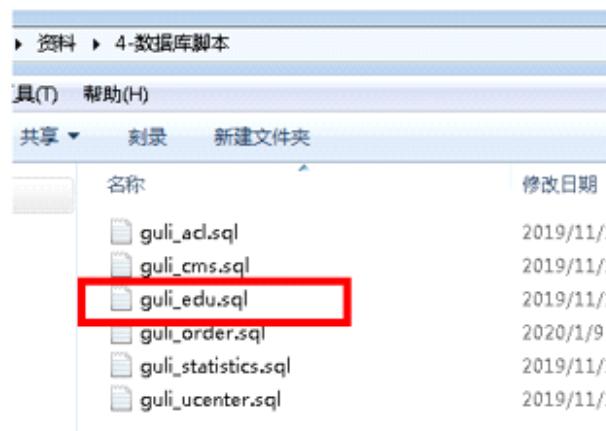


讲师模块

## 开发讲师管理模块后端

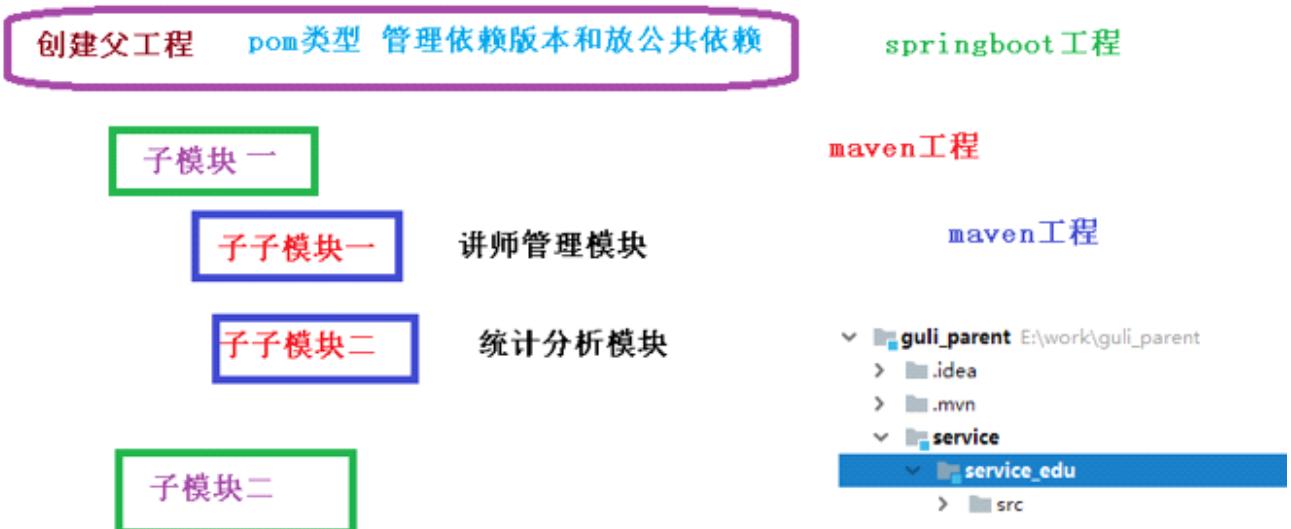
### 准备工作

#### 1、创建数据库，创建讲师数据库表



CREATE TABLE

#### 2、创建项目结构

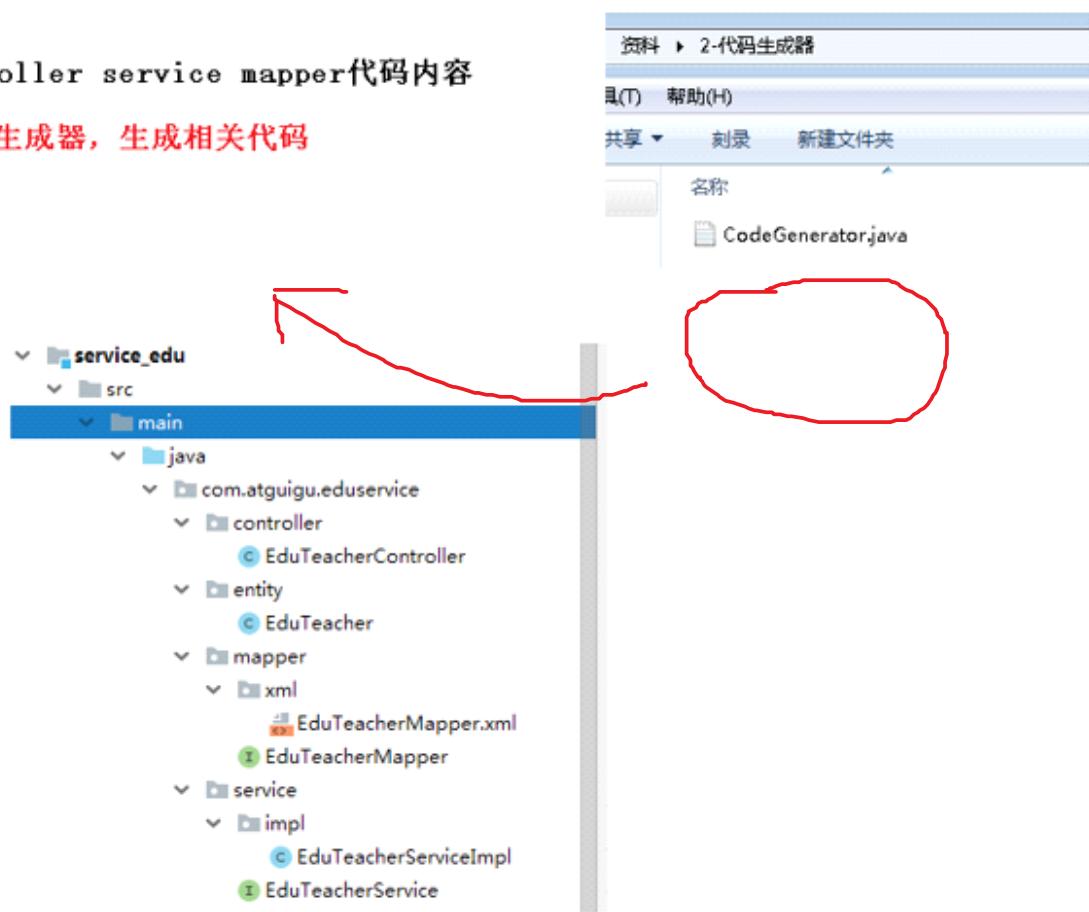


## 开发讲师管理模块

1、创建application.properties配置文件

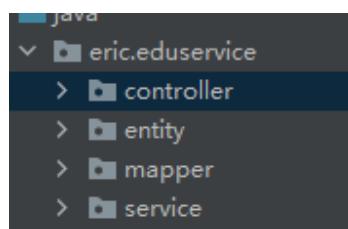
2、编写controller service mapper代码内容

mp提供代码生成器，生成相关代码



## 代码生成器

他会根据数据库里面的表自动生成数据库操作的代码



## @RequestMapping

### 一、@RequestMapping 简介

在Spring MVC 中使用 @RequestMapping 来映射请求，也就是通过它来指定控制器可以处理哪些 URL请求，相当于Servlet中在web.xml中配置

```
1 <servlet>
2 <servlet-name>serviceName</servlet-name>
3 <servlet-class>ServletClass</servlet-class>
4 </servlet>
5 <servlet-mapping>
6 <servlet-name>serviceName</servlet-name>
7 <url-pattern>url</url-pattern>
8 </servlet-mapping>
```

### 在生成代码下

查询所有老师就一行代码

```
//访问地址: http://localhost:8001/eduservice/teacher/findAll
//把service注入
@Autowired
private EduTeacherService teacherService;

@GetMapping("findAll")
public List<EduTeacher> findAllTeacher(){
 //调用service的方法实现查询所有的操作
 List<EduTeacher> list = teacherService.list(wrapper: null);
 return list;
}
```

在浏览器可以访问到 Json 格式的数据

于是有了这样一套开发流程

## 1、编写controller代码

```
@Autowired
private TeacherService teacherService;

@GetMapping
public List<Teacher>list(){
 return teacherService.list(null);
}
```

## 2、创建SpringBoot配置类

在edu包下创建config包，创建MyBatisPlusConfig.java

```
@Configuration
@EnableTransactionManagement
@MapperScan("com.atguigu.eduservice.mapper")
public class MyBatisPlusConfig{

}
```

## 3、创建SpringBoot启动类

创建启动类 EduApplication.java，注意启动类的创建位置

```
@SpringBootApplication
public class EduApplication{
 public static void main(String[] args) {
 SpringApplication.run(EduApplication.class, args);
 }
}
```

}

# swagger测试器

2022年2月7日 14:28

## 一、Swagger2介绍

前后端分离开发模式中，api文档是最好的沟通方式。

Swagger 是一个规范和完整的框架，用于生成、描述、调用和可视化 RESTful 风格的 Web 服务。

1. 及时性 (接口变更后，能够及时准确地通知相关前后端开发人员)
2. 规范性 (并且保证接口的规范性，如接口的地址，请求方式，参数及响应格式和错误信息)
3. 一致性 (接口信息一致，不会出现因开发人员拿到的文档版本不一致，而出现分歧)
4. 可测性 (直接在接口文档上进行测试，以方便理解业务)

## 配置方法

### 1、创建common模块（固定操作）

在总项目下创建模块common

在common下面创建子模块service-base

在模块service-base中，创建swagger的配置类

### 2、在模块service模块中引入service-base

内包引外包

```
<dependency>
 <groupId>com.atguigu</groupId>
```

```
<artifactId>service-base</artifactId>
<version>0.0.1-SNAPSHOT</version>
</dependency>
```

### 3、定义接口说明和参数说明

定义在类上：@Api

定义在方法上：@ApiOperation

定义在参数上：@ApiParam

```
@Api(description="讲师管理")
@RestController
@RequestMapping("/admin/edu/teacher")
public class TeacherAdminController{

 @Autowired
 private TeacherService teacherService;

 @ApiOperation(value="所有讲师列表")
 @GetMapping
 public List<Teacher> list(){
 return teacherService.list(null);
 }

 @ApiOperation(value="根据ID删除讲师")
 @DeleteMapping("{id}")
 public boolean removeById(
 @ApiParam(name="id", value="讲师ID", required=true)
 @PathVariable String id){
 return teacherService.removeById(id);
 }
}
```

### 4、访问<http://localhost:8001/swagger-ui.html>

# 同一数据格式 Json

2022年2月7日 19:01

## 定义统一结果

```
{
 "success": 布尔, //响应是否成功
 "code": 数字, //响应码
 "message": 字符串, //返回消息
 "data": HashMap//返回数据, 放在键值对中
}
```

## 创建统一结果返回类

### 1、在common模块下创建子模块common-utils

### 2、创建接口定义返回码

创建包 commonutils, 创建接口 ResultCode.java

```
Public interface ResultCode{
 public static Integer SUCCESS=20000;
 public static Integer ERROR=20001;
}
```

### 3、创建结果类 (链式编程)

创建类 R.java

```
@Data
Public class R{
 @ApiModelProperty(value="是否成功")
 private Boolean success;

 @ApiModelProperty(value="返回码")
 private Integer code;
```

```
.....
 public R success(Boolean success){
 this.setSuccess(success);
 return this;
 }

}
```

## 4、在service模块中添加依赖

```
<dependency>
 <groupId>eric</groupId>
 <artifactId>common_utils</artifactId>
 <version>0.0.1-SNAPSHOT</version>
</dependency>
```

## 5、修改Controller中的返回结果

```
@ApiOperation(value="所有讲师列表")
@GetMapping
public R list(){
 List<Teacher>list=teacherService.list(null);
 return R.ok().data("items", list);
}
```

返回这种格式数据

```
[
 {
 id: 1,
 label: '一级分类名称',

 children: [
 {
 id: 4,
 label: '二级分类1',
 }
]
 }
]
```

第一步 针对返回数据创建对应的实体类 两个实体类 一级和二级分类

第二步 在两个实体类之间表示关系 (一个一级分类有多个二级分类)

```
@Data
public class OneSubject {
 private String id;
 private String title;

 //一个一级分类有多个二级分类
 private List<TwoSubject> children = new ArrayList<>()
}
```

第三步 编写具体封装代码

List<EduSubject> oneSubjectList

List<OneSubject> finalSubjectList

# 讲师后台搭建

2022年2月8日 14:38

## 一、分页

### 1、MyBatisPlus Config 中配置分页插件

```
/**
 * 分页插件
 */
@Bean
public PaginationInterceptor paginationInterceptor() {
 return new PaginationInterceptor();
}
```

### 2、分页Controller方法

TeacherAdminController中添加分页方法

```
@ApiOperation(value = "分页讲师列表")
@GetMapping("{page}/{limit}")
public R pageList(
 @ApiParam(name = "page", value = "当前页码", required = true)
 @PathVariable Long page,

 @ApiParam(name = "limit", value = "每页记录数", required = true)
 @PathVariable Long limit){

 Page<Teacher> pageParam = new Page<>(page, limit);

 teacherService.page(pageParam, null);
 List<Teacher> records = pageParam.getRecords();
 long total = pageParam.getTotal();

 return R.ok().data("total", total).data("rows", records);
}
```

## 二、条件查询

### 1、创建查询对象

在 实体里面 创建 query 包，创建 TeacherQuery.java查询对象，即条件的entity

```
@Data
public class TeacherQuery{
 @ApiModelProperty(value="教师名称,模糊查询")
 private String name;
 @ApiModelProperty(value="头衔1高级讲师2首席讲师")
 private Integer level;
 @ApiModelProperty(value="查询开始时间",example="2019-01-01 10:10:10")
 private String begin;//注意，这里使用的是String类型，前端传过来的数据无需进行
 类型转换
 @ApiModelProperty(value="查询结束时间",example="2019-12-01 10:10:10")
 private String end;
}
```

### 2、分页Controller方法

TeacherAdminController中添加分页方法

```
@PostMapping("pageTeacherCondition/{current}/{limit}")
public R pageTeacherCondition(@PathVariable long current,@PathVariable long
limit,
 @RequestBody(required=false)
 TeacherQuery teacherQuery)
{
 //创建page对象
 Page<EduTeacher> pageTeacher = new Page<>(current,limit);
 //构建条件
 QueryWrapper<EduTeacher> wrapper=new QueryWrapper<>();
 //多条件组合查询
 //mybatis学过动态sql
```

```

String name=teacherQuery.getName();

//判断条件值是否为空，如果不为空拼接条件
if(!StringUtil.isEmpty(name)){
 //构建条件
 wrapper.like("name",name);
}

.....
//调用方法实现条件查询分页
teacherService.page(pageTeacher,wrapper);

long total=pageTeacher.getTotal(); //总记录数
List<EduTeacher> records = pageTeacher.getRecords(); //数据list集合
returnR.ok().data("total",total).data("rows",records);
}

```

### 三、新增和修改讲师

#### 1、在service-base模块中添加自动填充封装

创建包handler， 创建自动填充类 MyMetaObjectHandler

```

@Component
public class MyMetaObjectHandler implements MetaObjectHandler{
 @Override
 public void insertFill(MetaObject metaObject) {
 this.setFieldValByName("gmtCreate", new Date(), metaObject);
 this.setFieldValByName("gmtModified", new Date(), metaObject);
 }

 @Override
 public void updateFill(MetaObject metaObject) {
 this.setFieldValByName("gmtModified", new Date(), metaObject);
 }
}

```

#### 2、controller方法定义

```

//添加讲师接口的方法
@PostMapping("addTeacher")
public R addTeacher(@RequestBody EduTeacher eduTeacher){
 boolean save = teacherService.save(eduTeacher);
 if(save){
 return R.ok();
 }else{
 return R.error();
 }
}

//讲师修改功能
@PostMapping("updateTeacher")
public R updateTeacher(@RequestBody EduTeacher eduTeacher){
 boolean flag = teacherService.updateById(eduTeacher);
 if(flag){
 return R.ok();
 }else{
 return R.error();
 }
}

```

## 四、统一异常处理

### 1、创建统一异常处理器

在service-base中创建统一异常处理类GlobalExceptionHandler.java:

```

/**
 * 统一异常处理类
 */
@ControllerAdvice
public class GlobalExceptionHandler{

 @ExceptionHandler(Exception.class)
 @ResponseBody
 public R error(Excepione){
 e.printStackTrace();
 return R.error();
 }
}

```

## 2、添加特定异常处理方法

```
@ExceptionHandler(ArithmetcException.class)
@ResponseBody
public R error(ArithmetcException e){
 e.printStackTrace();
 return R.error().message("执行了自定义异常---算数异常...");
}
```

## 3、创建自定义异常类

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class GuliException extends RuntimeException{

 @ApiModelProperty(value = "状态码")
 private Integer code;

 private String msg;
}
```

## 五、统一日志处理

### 1、将日志输出到文件

先注释掉原有的日志配置  
再resources 中创建 logback-spring.xml

### 2、将错误日志输出到文件

GlobalExceptionHandler.java 中

类上添加注解

@Slf4j

异常输出语句

```
log.error(e.getMessage());
```

# ES6, Vue和Node

2022年2月8日 15:14

## 一、ECMAScript 6 简介

ECMAScript 6.0 (以下简称 ES6) 是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

### 1、let 声明变量

```
// var 声明的变量没有局部作用域
// let 声明的变量 有局部作用域
{
 var a=0
 let b=1
}
console.log(a) // 0
console.log(b) // ReferenceError: b is not defined

// var 可以声明多次
// let 只能声明一次
var m=1
var m=2
let n=3
let n=4
console.log(m) // 2
console.log(n) // Identifier 'n' has already been declared
```

### 2、const 声明常量（只读变量）

```
// 1、声明之后不允许改变
constPI="3.1415926"
PI=3// TypeError: Assignment to constant variable.

// 2、一但声明必须初始化，否则会报错
constMY_AGE// SyntaxError: Missing initializer in const declaration
```

## 3、解构赋值

解构赋值是对赋值运算符的扩展。

他是一种针对数组或者对象进行模式匹配，然后对其中的变量进行赋值。

在代码书写上简洁且易读，语义更加清晰明了；也方便了复杂对象中数据字段获取。

```
//1、数组解构
// 传统
let a=1, b=2, c=3
console.log(a, b, c)
// ES6
let [x, y, z]=[1, 2, 3]
console.log(x, y, z)

//2、对象解构
let user={name: 'Helen', age: 18}
// 传统
let name1=user.name
let age1=user.age
console.log(name1, age1)
// ES6
let {name, age} = user//注意：结构的变量必须是user中的属性
console.log(name, age)
```

## 4、模板字符串

模板字符串相当于加强版的字符串，用反引号`除了作为普通字符串，还可以用来定义多行字符串，还可以在字符串中加入变量和表达式。

```
// 1、多行字符串
let string1=`Hey,
can you stop angry now?
console.log(string1)
// Hey,
```

```

// can you stop angry now?

// 2、字符串插入变量和表达式。变量名写在 ${} 中，${} 中可以放入 JavaScript 表达式。
let name="Mike"
let age=27
let info=`My Name is ${name}, I am ${age+1} years old next year.`
console.log(info)
// My Name is Mike, I am 28 years old next year.

// 3、字符串中调用函数
function f(){
 return "have fun!"
}
let string2=`Game start, ${f()}`;
console.log(string2); // Game start, have fun!

```

## 5、声明对象简写

```

const age=12
const name="Amy"

// 传统
const person1={age: age, name: name}
console.log(person1)

// ES6
const person2={age, name}
console.log(person2) // {age: 12, name: "Amy"}

```

## 6、定义方法简写

```

// 传统
const person1={
 sayHi:function(){
 console.log("Hi")
 }
}
person1.sayHi(); // "Hi"

// ES6
const person2={
 sayHi(){

```

```
 console.log("Hi")
 }
}
person2.sayHi() // "Hi"
```

## 7、对象拓展运算符

拓展运算符 (...) 用于取出参数对象所有可遍历属性然后拷贝到当前对象。

```
// 1、拷贝对象
let person1={name: "Amy", age: 15}
let someone={ ...person1}
console.log(someone) // {name: "Amy", age: 15}
```

```
// 2、合并对象
let age={age: 15}
let name={name: "Amy"}
let person2={...age, ...name}
console.log(person2) // {age: 15, name: "Amy"}
```

## 8、箭头函数

箭头函数提供了一种更加简洁的函数书写方式。基本语法是：

参数 => 函数体

```
// 传统
var f1=function(a){
 return a
}
console.log(f1(1))
```

```
// ES6
var f2 = a=>a
console.log(f2(1))
```

// 当箭头函数没有参数或者有多个参数，要用 () 括起来。

// 当箭头函数函数体有多行语句，用 {} 包裹起来，表示代码块，

// 当只有一行语句，并且需要返回结果时，可以省略 {}，结果会自动返回。

```
var f3=(a,b) =>{
 let result=a+b
```

```
 return result
 }
console.log(f3(6,2)) // 8

// 前面代码相当于:
var f4=(a,b) =>a+b
```

箭头函数多用于匿名函数的定义

## 二、Vue.js

### 1、Vue.js 是什么

Vue 是一套用于构建用户界面的渐进式框架。

Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

官方网站：<https://cn.vuejs.org>

### 2、初始Vue.js [前面的链接](#)

### 3、修饰符

修饰符 (Modifiers) 是以半角句号 (.) 指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。

例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()：即阻止事件原本的默认行为

```
data: {
 user: {}
}
```

<!-- 修饰符用于指出一个指令应该以特殊方式绑定。

这里的 .prevent 修饰符告诉 v-on 指令对于触发的事件调用 js 的 event.preventDefault()：

即阻止表单提交的默认行为 -->

```

<form action="save" v-on:submit.prevent="onSubmit">
 <label for="username">
 <input type="text" id="username" v-model="user.username">
 <button type="submit">保存</button>
 </label>
</form>

methods: {
 onSubmit() {
 if(this.user.username) {
 console.log('提交表单')
 } else{
 alert('请输入用户名')
 }
 }
}

```

## 4、组件

### 1、局部组件

定义组件

```

var app=new Vue({
 el: '#app',
 // 定义局部组件，这里可以定义多个局部组件
 components: {
 //组件的名字
 'Navbar': {
 //组件的内容
 template: '首页学员管理'
 }
 }
})

```

使用组件

```

<div id="app">
 <Navbar></Navbar>
</div>

```

### 2、全局组件

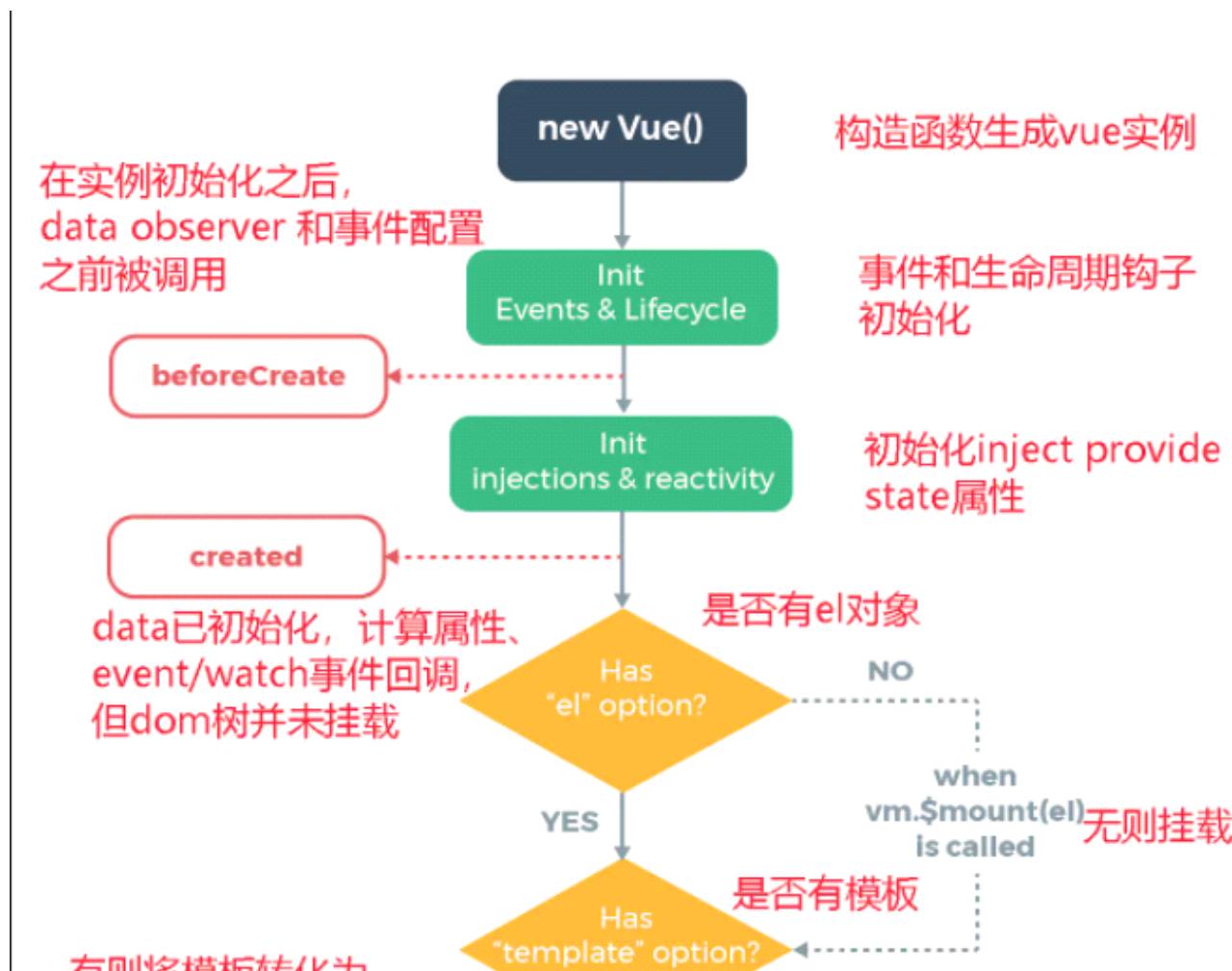
定义全局组件: components/Navbar.js

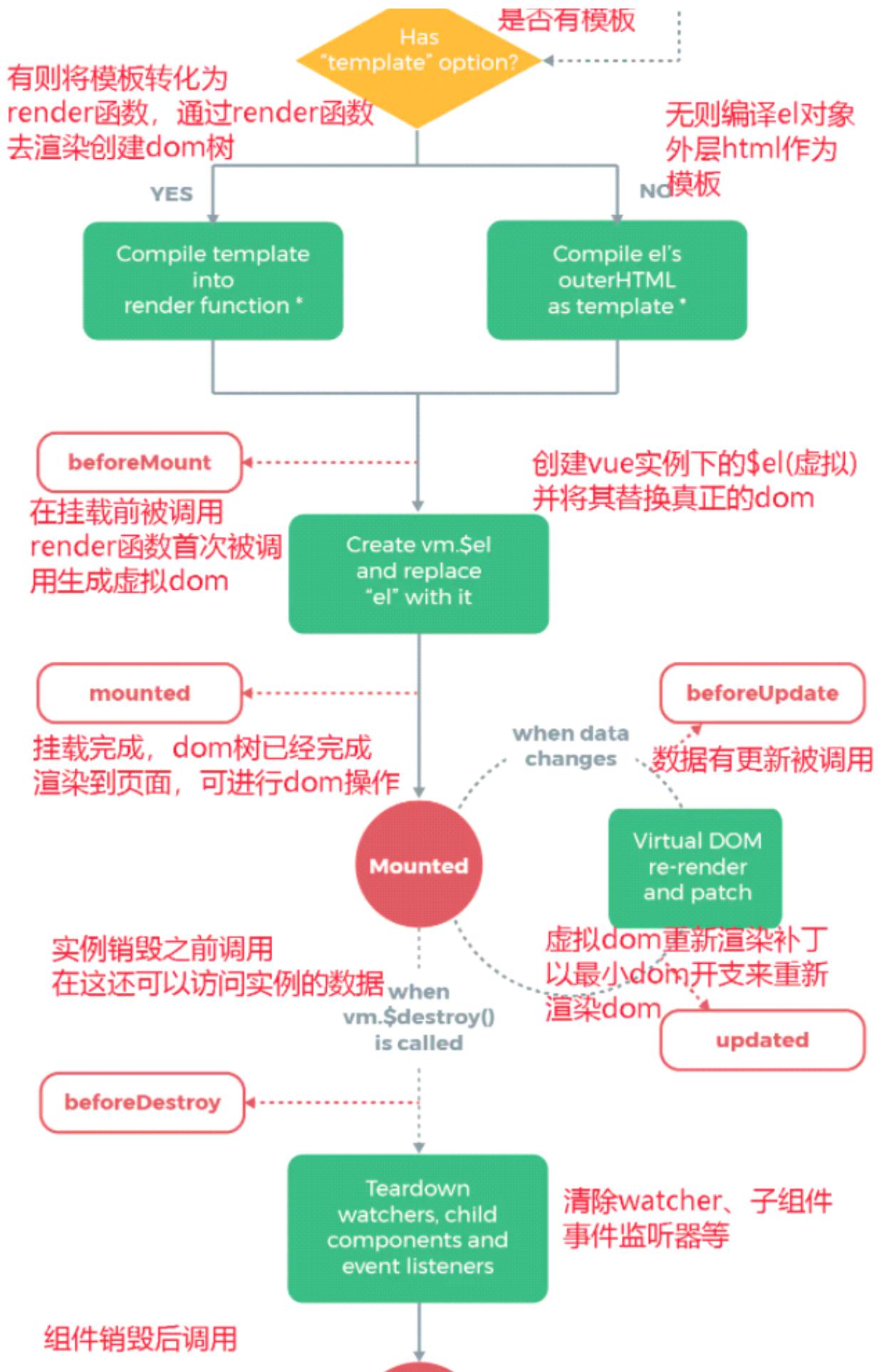
```
// 定义全局组件
Vue.component('Navbar', {
 template: '首页学员管理讲师管理'
})
```

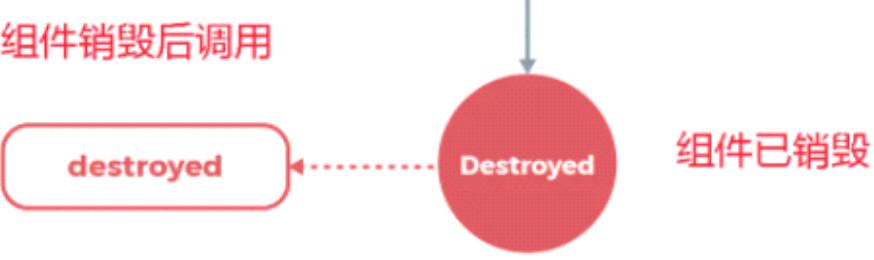
使用组件

```
<div id="app">
 <Navbar></Navbar>
</div>
<script src="vue.min.js"></script>
<script src="components/Navbar.js"></script>
<script>
 var app=new Vue({
 el: '#app'
 })
</script>
```

## 5、生命周期







\* template compilation is performed ahead-of-time if using  
a build step, e.g. single-file components

## 6、路由（超链接）

Vue.js 路由允许我们通过不同的 URL 访问不同的内容。

通过 Vue.js 可以实现多视图的单页Web应用 (single page web application, SPA) 。

Vue.js 路由需要载入 vue-router 库

### 1、引入js

```
<script src="vue.min.js"></script>
<script src="vue-router.min.js"></script>
```

### 2、编写html

```
<div id="app">
 <h1>Hello App!</h1>
 <p>
 <!-- 使用 router-link 组件来导航. -->
 <!-- 通过传入 `to` 属性指定链接. -->
 <!-- <router-link> 默认会被渲染成一个 `<a>` 标签 -->
 <router-link to="/">首页</router-link>
 <router-link to="/student">会员管理</router-link>
 <router-link to="/teacher">讲师管理</router-link>
 </p>
 . . .
```

```

<router-link to="/teacher">讲师管理</router-link>
</p>
<!-- 路由出口 -->
<!-- 路由匹配到的组件将渲染在这里 -->
<router-view></router-view>
</div>

```

### 3、编写js

```

<script>
 // 1. 定义（路由）组件。
 // 可以从其他文件 import 进来
 const Welcome={ template: '<div>欢迎</div>' }
 const Student={ template: '<div>student list</div>' }
 const Teacher={ template: '<div>teacher list</div>' }

 // 2. 定义路由
 // 每个路由应该映射一个组件。
 const routes=[
 { path: '/', redirect: '/welcome' }, //设置默认指向的路径
 { path: '/welcome', component: Welcome },
 { path: '/student', component: Student },
 { path: '/teacher', component: Teacher }
]

 // 3. 创建 router 实例，然后传 `routes` 配置
 const router=new VueRouter({
 routes// (缩写) 相当于 routes: routes
 })

 // 4. 创建和挂载根实例。
 // 从而让整个应用都有路由功能
 const app = new Vue({
 el: '#app',
 router
 })

 // 现在，应用已经启动了!
</script>

```

### 7、xios

## 7、xios

axios是独立于vue的一个项目，基于promise用于浏览器和node.js的http客户端

- 在浏览器中可以帮助我们完成 ajax 请求的发送
- 在node.js中可以向远程接口发送请求

### 1. 导包

```
<script src="vue.min.js"></script>
<script src="axios.min.js"></script>
```

注意：测试时需要开启后端服务器，并且后端开启跨域访问权限

```
var app = new Vue({
 el: '#app',
 data: {
 memberList: [] // 数组
 },
 created() {
 this.getList()
 },
 methods: {
 getList(id) {
 // vm = this
 axios.get('http://localhost:8081/admin/ucenter/member')
 .then(response => {
 console.log(response)
 this.memberList = response.data.data.items
 })
 .catch(error => {
 console.log(error)
 })
 }
 }
})
```

控制台查看输出

### 2、显示数据

```
<div id="app">
 <table border="1">
 <tr>
 <td>id</td>
```

```

<table border="1">
 <tr>
 <td>id</td>
 <td>姓名</td>
 </tr>
 <tr v-for="item in memberList">
 <td>{{item.memberId}}</td>
 <td>{{item.nickname}}</td>
 </td>
 </tr>
</table>
</div>

```

## 8、element-ui:

element-ui 是饿了么前端出品的基于 Vue.js 的后台组件库，方便程序员进行页面快速布局和构建

官网：<http://element-cn.eleme.io/#/zh-CN>

将element-ui引入到项目



### 1、引入css

```

<!-- import CSS -->
<link rel="stylesheet" href="element-ui/lib/theme-chalk/index.css">

```

### 2、引入js

```

<!-- import Vue before Element -->
<script src="vue.min.js"></script>
<!-- import JavaScript -->
<script src="element-ui/lib/index.js"></script>

```

### 3、编写html

```

<div id="app">
 <el-button @click="visible = true">Button</el-button>
 <el-dialog:visible.sync="visible" title="Hello world">
 <p>Try Element</p>
 </el-dialog>

```

```
<el-dialog:visible.sync="visible" title="Hello world">
 <p>Try Element</p>
</el-dialog>
</div>
```

关于.sync的扩展阅读

<https://www.jianshu.com/p/d42c508ea9de>

## 4、编写js

```
<script>
new Vue({
 el: '#app',
 data: function() {//定义Vue中data的另一种方式
 return{ visible: false}
 }
})
</script>
```

# 一、Bode.js 简介

## 1、什么是Node.js

简单的说 Node.js 就是运行在服务端的 JavaScript。

Node.js是一个事件驱动I/O服务端JavaScript环境，基于Google的V8引擎，V8引擎执行 Javascript的速度非常快，性能非常好。

## 2、Node.js有什么用

如果你是一个前端程序员，你不懂得像PHP、Python或Ruby等动态编程语言，然后你想创建自己的服务，那么Node.js是一个非常好的选择。

Node.js 是运行在服务端的 JavaScript，如果你熟悉Javascript，那么你将会很容易的学会 Node.js。

当然，如果你是后端程序员，想部署一些高性能的服务，那么学习Node.js也是一个非常好的选择。

## 3、服务器端应用开发（了解）

创建 02-server-app.js

## 2、服务器端应用开发（J脚本）

创建 02-server-app.js

```
const http=require('http');
http.createServer(function(request, response) {
 // 发送 HTTP 头部
 // HTTP 状态值: 200 : OK
 // 内容类型: text/plain
 response.writeHead(200, {'Content-Type': 'text/plain'});
 // 发送响应数据 "Hello World"
 response.end('Hello Server');
}).listen(8888);
// 终端打印如下信息
console.log('Server running at http://127.0.0.1:8888/');
```

运行服务器程序

node 02-server-app.js

服务器启动成功后，在浏览器中输入：<http://localhost:8888/> 查看webserver成功运行，  
并输出html页面

## 4、什么是NPM

NPM全称Node Package Manager，是Node.js包管理工具，是全球最大的模块生态系统，  
里面所有的模块都是开源免费的；也是Node.js的包管理工具，相当于前端的Maven。

## 5、使用npm管理项目

### 1、创建文件夹npm

### 2、项目初始化

```
#建立一个空文件夹，在命令提示符进入该文件夹 执行命令初始化
npm init
#按照提示输入相关信息。如果是用默认值则直接回车即可。
```

```
#建立一个空文件，输入提示符进入该文件 执行命令初始化
npm init
#按照提示输入相关信息，如果是用默认值则直接回车即可。
#name: 项目名称
#version: 项目版本号
#description: 项目描述
#keywords: {Array}关键词，便于用户搜索到我们的项目
#最后会生成package.json文件，这个是包的配置文件，相当于maven的pom.xml
#我们之后也可以根据需要进行修改。

#如果想直接生成 package.json 文件，那么可以使用命令
npm init -y
```

## 2、修改npm镜像

NPM官方的管理的包都是从 <http://npmjs.com>下载的，但是这个网站在国内速度很慢。  
这里推荐使用淘宝 NPM 镜像 <http://npm.taobao.org/>，淘宝 NPM 镜像是一个完整  
npmjs.com 镜像，同步频率目前为 10分钟一次，以保证尽量与官方服务同步。

设置镜像地址：

```
#经过下面的配置，以后所有的 npm install 都会经过淘宝的镜像地址下载
npm config setregistry https://registry.npm.taobao.org

#查看npm配置信息
npm config list
```

## 3、npm install命令的使用

```
#使用 npm install 安装依赖包的最新版，
#模块安装的位置：项目目录\node_modules
#安装会自动在项目目录下添加 package-lock.json文件，这个文件帮助锁定安装包的版
本
#同时package.json 文件中，依赖包会被添加到dependencies节点下，类似maven中的
<dependencies>
npm install jquery

#npm管理的项目在备份和传输的时候一般不携带node_modules文件夹
npm install #根据package.json中的配置下载依赖，初始化项目
```

```
#如果安装时想指定特定的版本
npm install jquery@2.1.x

#devDependencies节点：开发时的依赖包，项目打包到生产环境的时候不包含的依赖
#使用 -D参数将依赖添加到devDependencies节点
npm install --save-dev eslint
#或
npm install -D eslint

#全局安装
#Node.js全局安装的npm包和工具的位置：用户目录\AppData\Roaming\npm
\node_modules
#一些命令行工具常使用全局安装的方式
npm install -g webpack
```

## 4、其它命令

#更新包（更新到最新版本）

```
npm update 包名
```

#全局更新

```
npm update -g 包名
```

#卸载包

```
npm uninstall 包名
```

#全局卸载

```
npm uninstall -g 包名
```

## 5、Bable简介

Babel是一个广泛使用的转码器，可以将ES6代码转为ES5代码，从而在现有环境执行。这意味着，你可以现在就用ES6编写程序，而不用担心现有环境是否支持。

## 安装命令行转码工具

# 安装命令行转码工具

Babel提供babel-cli工具，用于命令行转码。它的安装命令如下：

```
npm install --global babel-cli
```

```
#查看是否安装成功
babel --version
```

## 三、Babel的使用

### 1、初始化项目

```
npm init -y
```

### 2、创建文件

下面是一段ES6代码：

```
// 转码前
// 定义数据
let input=[1, 2, 3]
// 将数组的每个元素 +1
input=input.map(item=>item+1)
console.log(input)
```

### 2、配置.babelrc

Babel的配置文件是.babelrc，存放在项目的根目录下，该文件用来设置转码规则和插件，基本格式如下。

```
{
 "presets": [],
 "plugins": []
}
```

presets字段设定转码规则，将es2015规则加入 .babelrc：

```
,
```

```
{
 "presets": ["es2015"],
 "plugins": []
}
```

### 3、安装转码器

在项目中安装

```
npm install --save-dev babel-preset-es2015
```

### 4、转码

```
转码结果写入一个文件
mkdir dist1
--out-file 或 -o 参数指定输出文件
babel src/example.js --out-file dist1/compiled.js
或者
babel src/example.js -o dist1/compiled.js
```

```
整个目录转码
mkdir dist2
--out-dir 或 -d 参数指定输出目录
babel src --out-dir dist2
或者
babel src -d dist2
```

# 后台的前端页面

2022年2月8日 17:22

## CommonJS模块规范

每个文件就是一个模块，有自己的作用域。在一个文件里面定义的变量、函数、类，都是私有的，对其他文件不可见。

### 1、创建“module”文件夹

### 2、导出模块

创建 common-js模块化/四则运算.js

```
// 定义成员:
const sum=function(a,b){
 return parseInt(a) +parseInt(b)
}
const subtract=function(a,b){
 return parseInt(a) -parseInt(b)
}
const multiply=function(a,b){
 return parseInt(a) *parseInt(b)
}
const divide=function(a,b){
 return parseInt(a) /parseInt(b)
}
```

导出模块中的成员

```
// 导出成员:
module.exports={
 sum: sum,
 subtract: subtract,
 multiply: multiply,
 divide: divide
}
```

简写

```
//简写
module.exports={
 sum,
 subtract,
 multiply,
 divide
}
```

### 3、导入模块

创建 common-js模块化/引入模块.js

```
//引入模块，注意：当前路径必须写 ./
const m=require('./四则运算.js')
console.log(m)

const result1 = m.sum(1, 2)
const result2 = m.subtract(1, 2)
console.log(result1, result2)
```

# ES6模块化规范

ES6使用 export 和 import 来导出、导入模块。

## 1、创建模块

创建 es6模块化/userApi.js

```
export function getList() {
 console.log('获取数据列表')
}

export function save() {
 console.log('保存数据')
}
```

## 2、导入模块

创建 es6模块化/userComponent.js

```
//只取需要的方法即可，多个方法用逗号分隔
import{ getList, save} from"./userApi.js"
getList()
save()
```

注意：这时的程序无法运行的，因为ES6的模块化无法在Node.js中执行，需要用Babel编辑成ES5后再执行。

# ES6模块化的另一种写法

## 导入模块

创建 es6模块化/userComponent2.js

```
import user from"./userApi2.js"
user.getList()
user.save()
```

# Webpack

## 1、创建webpack文件夹

进入webpack目录，执行命令

```
npm init-y
```

## 2、创建src文件夹

## 3、src下创建common.js

```
exports.info=function(str) {
 document.write(str);
}
```

## 4、src下创建utils.js

```
exports.add=function(a, b) {
 return a+b;
}
```

## 5、src下创建main.js

```
const common=require('./common');
const utils=require('./utils');

common.info('Hello world!'+utils.add(100, 200));
```

# JS打包

## 1、webpack目录下创建配置文件webpack.config.js

以下配置的意思是：读取当前项目目录下src文件夹中的main.js（入口文件）内容，分析资源依赖，把相关的js文件打包，打包后的文件放入当前目录的dist文件夹下，打包后的js文件名为bundle.js

```
const path=require("path"); //Node.js内置模块
module.exports={
 entry: './src/main.js', //配置入口文件
 output: {
 path: path.resolve(__dirname, './dist'), //输出路径, __dirname: 当前文件所在路径
 filename: 'bundle.js'//输出文件
 }
}
```

## 2、命令行执行编译命令

```
webpack #有黄色警告
webpack --mode=development #没有警告
#执行后查看bundle.js 里面包含了上面两个js文件的内容并有了代码压缩
也可以配置项目的npm运行命令，修改package.json文件
```

```
"scripts": {
 //...
 "dev": "webpack --mode=development"
}
```

运行npm命令执行打包

```
npm run dev
```

## 3、webpack目录下创建index.html

引用bundle.js

```
<body>
 <script src="dist/bundle.js"></script>
</body>
```

## 4、浏览器中查看index.html

## CSS打包

### 1、安装style-loader和 css-loader

Webpack 本身只能处理 JavaScript 模块，如果要处理其他类型的文件，就需要使用 loader 进行转换。

Loader 可以理解为是模块和资源的转换器。

首先我们需要安装相关Loader插件，css-loader 是将 css 装载到 javascript； style-loader 是让 javascript 认识css

```
npm install --save-dev style-loader css-loader
```

### 2、修改webpack.config.js

```
const path = require("path"); //Node.js内置模块
module.exports = {
 //...
 output: {},
 module: {
 rules: [
 {
 test: /\.css$/,
 use: ['style-loader', 'css-loader']
 }
]
 }
}
```

## vue 框架的二开

### 1. API里面写接口，访问后端地址

```
//添加讲师
addTeacher(teacher) {
 return request({
 url: `/eduservice/teacher/addTeacher`,
 method: 'post',
 data: teacher
 })
},
```

## 2.view里面写页面，页面函数调用API接口

```
//添加讲师的方法
saveTeacher() {
 teacherApi.addTeacher(this.teacher)
 .then(response => {//添加成功
 //提示信息
 this.$message({
 type: 'success',
 message: '添加成功!'
 });
 //回到列表页面 路由跳转
 this.$router.push({path: '/teacher/table'})
 })
}
```

## 3.router里面配置路由，实现页面跳转

```
{
 path: 'save',
 name: '添加讲师',
 component: () => import('@/views/edu/teacher/save'),
 meta: { title: '添加讲师', icon: 'tree' }
},
```

# vue-admin-template 模板

## 1、简介

vueAdmin-template是基于vue-element-admin的一套后台管理系统基础模板（最少精简版），可作为模板进行二次开发。

GitHub地址：<https://github.com/PanJiaChen/vue-admin-template>

建议：你可以在 vue-admin-template 的基础上进行二次开发，把 vue-element-admin当做工具箱，想要什么功能或者组件就去 vue-element-admin 那里复制过来。

## 2、安装

```
解压压缩包
进入目录
cd vue-admin-template-master

安装依赖
npm install

启动。执行后，浏览器自动弹出并访问http://localhost:9528/
```

```
npm run dev
```

### 3、修改

把系统登录功能改造本地 模拟登录

#### 1、系统登录默认使用这个地址

```
: https://easy-mock.com/mock/5950a2419adc231f356a6636/vue-admin/user/login
```

把登录请求地址改造本地 <http://localhost:8001>

#### 2、修改配置文件请求地址

在config文件夹里面有dev.env.js

```
'use strict'
const merge = require('webpack-merge')
const prodEnv = require('./prod.env')

module.exports = merge(prodEnv, {
 NODE_ENV: '"development"',
 // BASE_API: '"https://easy-mock.com/mock/5950a2419adc231f356a6636/vue-admin/user/login"',
 BASE_API: '"http://localhost:8001"',
})
```

3、进行登录调用两个方法，login登录操作方法，和info登录之后获取用户信息的方法。所以，创建接口两个方法实现登录

(1) login 返回token值

(2) info 返回roles name avatar

#### 4、开发接口

```
@RestController
@RequestMapping("/eduservice/user")
public class EduLoginController {
 //login
 @PostMapping("login")
 public R login() {
 return R.ok().data("token", "admin");
 }
 //info
 @GetMapping("info")
 public R info() {
 return R.ok().data("roles", "[admin]").data("name", "admin");
 }
}
```

#### 5、修改api文件夹login.js修改本地接口路径

```
export function login(username, password) {
 return request({
 url: '/eduservice/user/login',
 ...
 })
}

export function getInfo(token) {
 return request({
 url: '/eduservice/user/info',
 method: 'get',
 })
}
```

## 6、最终测试，出现问题

```
XHRHttpRequest cannot load http://localhost:8001/eduservice/user/login. Response to preflight request doesn't pass access control check: No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin 'http://localhost:9528' is therefore not allowed ac...
```

跨域问题

通过一个地址去访问另外一个地址，这个过程中如果有三个地方任何一个不一样

访问协议	http	https
IP地址	192.168.1.1	172.11.11.11
端口号	9528	8001

http://localhost:9528/

不一样

http://localhost:8001/

## 7、跨域解决方式

(1) 在后端接口controller添加注解（常用）

```
@CrossOrigin //解决跨域
public class EduLoginController {
```

(2) 使用网关解决（后面讲到）

## 根据路由更改页面

### 框架使用过程

#### 第一步 添加路由

```
src
 api
 assets
 components
 icons
 router
 index.js
```

#### 第二步 点击某个路由，显示路由对应页面内容

在views创建vue页面

```
{
 path: 'tree', 路由对应页面
 name: 'Tree',
 component: () => import('@/views/tree/index'),
 meta: { title: 'Tree', icon: 'tree' }
}
```

```
VIEWS
 dashboard
 form
 layout
 login
 nested
 table
 tree
 index.vue
```

#### 第三步 在api文件夹创建js文件，定义接口地址和参数

```
import request from '@/utils/request'

export function getList(params) {
 return request({
 url: '/table/list',
 method: 'get',
 params
 })
```

#### 第四步 在创建vue页面引入js文件，调用方法实现功能

引入 import user from ...'

```
data: [
],
created() {
},
methods: [
]
}
最后使用element-ui显示数据内容
```

#### 第五步 把请求接口获取数据在页面进行显示

使用组件 element-ui实现

Data

Table 表格

## 实现查询讲师列表

### 1、讲师列表添加分页实现

```
<!-- 分页 -->
<el-pagination
 :current-page="page" 当前页
 :page-size="limit" 没有记录数
 :total="total" 总计记录数
 style="padding: 30px 0; text-align: center;"
 layout="total, prev, pager, next, jumper"
 @current-change="getList" 分页的方法
/>
```

分页的方法修改，添加页码参数

```
//讲师列表的方法
getList(page =1) {
 this.page = page
}
```

Total 16 < 1 2 > Go to

### 2、添加条件查询

张 高级 2019-1-1 2019-1-31  
名称  级别  开始  结束



使用element-ui组件实现出来

在列表上面添加条件输入表单，使用v-model数据绑定

```
<!--讲师列表-->
<el-form :inline="true" class="demo-form-inline">
 <el-form-item>
 <el-input v-model="teacherQuery.name" placeholder="讲师名"/>
 </el-form-item>

 <!-- @click="getList()">查询</el-button>
```

(1) 清空表单输入条件数据

(2) 查询所有的数据

```
resetData() //清空的方法
//表单输入项数据清空
this.teacherQuery = {}
//查询所有讲师数据
this.getList()
```

## 实现讲师列表的添加，删除和修改

## 讲师删除功能

1、在每条记录后面添加 删除按钮

**删除**

2、在按钮绑定事件

`@click="removeDataById"`

3、在绑定事件的方法传递删除讲师的id值

`@click="removeDataById(scope.row.id)">`

```
//删除讲师的方法
removeDataById(id) {
 this.$confirm('此操作将永久删除讲师记录, 是否继续?', '提示', {
 confirmButtonText: '确定',
 cancelButtonText: '取消',
 type: 'warning'
 }).then(() => {
 //点击确定, 执行then方法
 //调用删除的方法
 teacher.deleteTeacherId(id)
 .then(response => {//删除成功
 //提示信息
 this.$message({
 type: 'success',
 message: '删除成功!'
 });
 //回到列表页面
 this.getList()
 })
 //点击取消, 执行catch方法
 })
}
```

4、在api文件夹teacher.js定义删除接口的地址

```
//删除讲师
deleteTeacherId(id) {
 return request({
 url: `/eduservice/teacher/${id}`,
 method: 'delete'
 })
}
```

5、页面调用 方法实现删除



## 添加讲师功能



点击 添加讲师 按钮  
进入表单页面，输入讲师信息

讲师名称	张老师
讲师排序	0
讲师头衔	首席讲师
讲师资历	444
讲师简介	333

在表单页面点击 保存 ，提交接口，添加数据库

(1) api定义接口地址

```
//添加讲师
addTeacher(teacher) {
 return request({
 url: `/eduservice/teacher/addTeacher`,
 method: 'post',
 data: teacher
 })
}
```

(2) 在页面实现调用

```
//添加讲师的方法
saveTeacher() {
 teacherApi.addTeacher(this.teacher)
 .then(response => {//添加成功
 //提示信息
 this.$message({
 type: 'success',
 message: '添加成功!'
 });
 //回到列表页面 路由跳转
 this.$router.push({path: '/teacher/table'})
 })
}
```

## 讲师修改功能

1、每条记录后面添加 修改 按钮

[修改](#)

2、点击修改按钮 进入表单页面，进行数据回显

根据讲师id查询数据显示

## 3、通过路由跳转进入数据回显页面，在路由index页面添加路由

```
{
 path: 'save/:id',
 name: 'EduTeacherEdit',
 component: () => import('@/views/edu/teacher/save'),
 meta: { title: '编辑讲师', noCache: true },
 hidden: true
}

<template slot-scope="scope">
 <router-link :to="'/teacher/edit/'+scope.row.id">
 <el-button type="primary" size="mini" icon="el-icon-edit">修改</el-button>
 </router-link>
</template>
```

## 4、在表单页面实现数据回显

(1) 在teacher.js定义根据id查询接口

```
getTeacherInfo(id) {
 return request({
 url:
 `/eduservice/teacher/getTeacher/${id}`,
 method: 'get'
 })
}
```

(3) 调用 根据id查询的方法

因为添加和修改使用save页面  
区别添加还是修改，只有修改时候查询数据回显

判断路径里面是否有讲师id值，如果有id  
值修改，没有id值直接添加

(2) 在页面调用接口实现数据回显

```
//根据讲师id查询的方法
 getInfo(id) {
 teacherApi.getTeacherInfo(id)
 .then(response => {
 this.teacher = response.data.teacher
 })
},
```

created() { //页面渲染之前执行

```
//判断路径是否有id值
if(this.$route.params && this.$route.params.id) {
 //从路径获取id值
 const id = this.$route.params.id
 //调用根据id查询的方法
 this.getInfo(id)
},
```

## 一个神奇的bug

### 遇到问题

1、第一次点击修改 进行数据回显

第二次再去点击 添加讲师，进入表单页面，但是  
问题：表单页面还是显示修改回显数据，正确效果  
应该是表单数据清空

### 解决方式：

做添加讲师时候，表单数据清空就可以了

```
created() { //页面渲染之前执行
 //判断路径有id值,做修改
 if(this.$route.params && this.$route.params.id) {
 //从路径获取id值
 const id = this.$route.params.id
 //调用根据id查询的方法
 this.getInfo(id)
 } else { //路径没有id值, 做添加
 //清空表单
 this.teacher = {}
 }
},
```

上面代码没有解决问题，为什么？

多次路由跳转到同一个页面，在页面中  
created方法只会执行第一次，后面在进行  
跳转不会执行的

最终解决：使用vue监听

```
watch: { //监听
 $route(to, from) { //路由变化方式，路由发生变化，方
法就会执行
 this.init()
 }
},
```

汽车 报警器

# 阿里云OSS与easyExcel

2022年2月9日 16:41

## Access key

LTAI5tRk7Vs9pWw9rJGvpEA9  
SLRnnuGVjhLOPPLeGYaxc1pmzhgKRV

### 1、在service创建子模块 service\_oss

```
└─ service
 ├─ service_edu
 └─ service_oss
```

### 2、在service\_oss引入相关OSS依赖

```
<dependencies>
 <!-- 阿里云OSS依赖 -->
 <dependency>
 <groupId>com.aliyun.oss</groupId>
 <artifactId>aliyun-sdk-oss</artifactId>
 </dependency>
 <!-- 日期工具类依赖 -->
 <dependency>
 <groupId>joda-time</groupId>
 <artifactId>joda-time</artifactId>
 </dependency>
</dependencies>
```

### 3、创建配置文件

```
server.port=8002
#服务名
spring.application.name=service-oss

#环境设置: dev, test, prod
spring.profiles.active=dev

#阿里云OSS
#不同的服务器, 地址不同
aliyun.oss.file.endpoint=oss-cn-beijing.aliyuncs.com
aliyun.oss.file.keyid=LTAI4FvvVEWiTJ3GNJJqJnk7
aliyun.oss.file.keysecret=
9st82dv7EvFk9mTjYO1XXbM632fRbG
#bucket可以在控制台创建, 也可以使用java代码创建
aliyun.oss.file.bucketname=edu-guli-1010
```

### 4. 创建启动类, 启动报错了

Description:

Failed to configure a DataSource: 'url' attribute is not specified!

Reason: Failed to determine a suitable driver class

启动时候, 找数据库配置, 但是现在模块因为不需要操作数据库, 只是做上传到OSS功能, 没有配置数据库

解决方式: 1. 添加上数据库配置

2. 在启动类添加属性, 默认不去加载数据库配置

使用这种方式

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@ComponentScan(basePackages = {"com.atguigu"})
public class OssApplication {
```

## value注入 (读取配置文件)

### 1、创建常量类, 读取配置文件内容

```
//当项目已启动, spring接口, spring加载之后, 执行接口一个方法
@Component
public class ConstantPropertiesUtils implements InitializingBean {

 //读取配置文件内容
 @Value("${aliyun.oss.file.endpoint}")
 private String endpoint;
```

[返回图片网址](#)

## 2、创建controller，创建service

```
@RestController
@RequestMapping("/eduoss/fileoss")
public class OssController {
 @Autowired
 private OssService ossService;
 //上传头像的方法
 @PostMapping
 public R uploadOssFile(MultipartFile file) {
 //获取上传文件 MultipartFile
 //返回上传到oss的路径
 String url = ossService.uploadFileAvatar(file);
 return R.ok().data("url",url);
 }
}
```

用官方例程上传，根据日期分文件夹

## 3、在service实现上传文件到oss过程

```
public String uploadFileAvatar(MultipartFile file) {
 // 工具类获取值
 String endpoint = ConstantPropertiesUtils.END_POINT;
 String accessKeyId = ConstantPropertiesUtils.ACCESS_KEY_ID;
 String accessKeySecret = ConstantPropertiesUtils.ACCESS_KEY_SECRET;
 String bucketName = ConstantPropertiesUtils.BUCKET_NAME;

 try {
 // 创建OSS实例。
 OSS ossClient = new OSSClientBuilder().build(endpoint, accessKeyId,
 accessKeySecret);

 // 获取上传文件输入流
 InputStream inputStream = file.getInputStream();
 // 获取文件名称
 String fileName = file.getOriginalFilename();
 }
```

```
// 调用oss方法实现上传
// 第一个参数 Bucket名称
// 第二个参数 上传到oss文件路径和文件名称 /aa/bb/1.jpg
// 第三个参数 上传文件输入流
ossClient.putObject(bucketName,fileName , inputStream);

// 关闭OSSClient。
ossClient.shutdown();

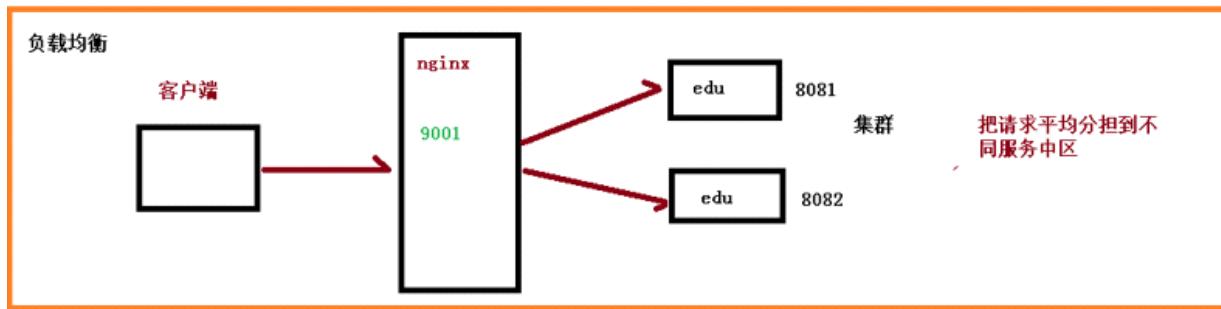
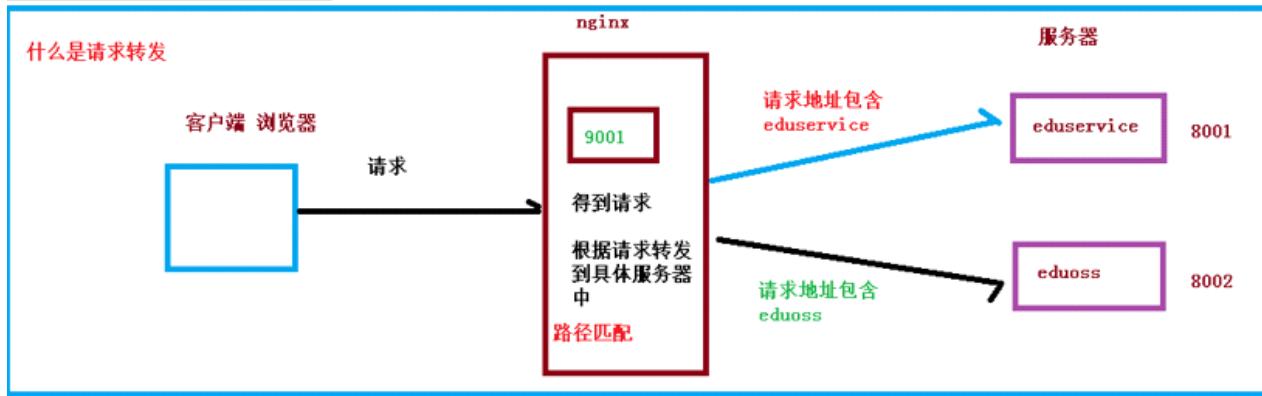
// 把上传之后文件路径返回
// 需要把上传到阿里云oss路径手动拼接出来
// https://edu-guli-1010.oss-cn-beijing.aliyuncs.com/01.jpg
String url = "https://" +bucketName+ "/" +endpoint+ "/" +fileName;
return url;
} catch(Exception e) {
 e.printStackTrace();
 return null;
}
```

```
//2 把文件按照日期进行分类
//获取当前日期
// 2019/11/12
String datePath = new DateTime().toString(s: "yyyy/MM/dd");
//拼接
// 2019/11/12/ewtqr313401.jpg
fileName = datePath + "/" +fileName;
```

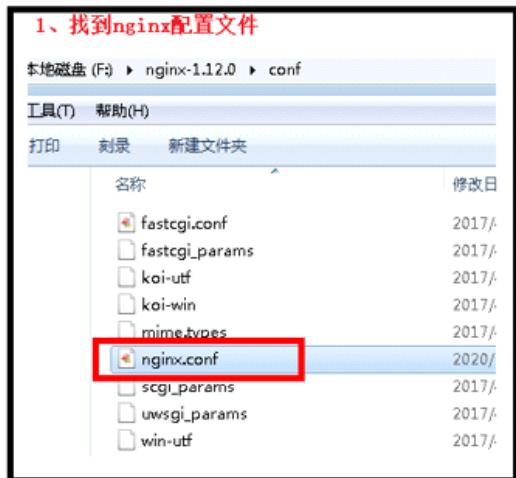
Nginx

nginx 反向代理服务器

- 1、请求转发
- 2、负载均衡
- 3、动静分离



### 配置nginx实现请求转发的功能



修改前端请求地址  
改为nginx地址

JS dev.env.js ×

```

1 'use strict'
2 const merge = require('webpack-merge')
3 const prodEnv = require('./prod.env')
4
5 module.exports = merge(prodEnv, {
6 NODE_ENV: '"development"',
7 // BASE_API: '"https://easy-mock.com/mock/5950a2419ad'
8 BASE_API: '"http://localhost:9001"', nginx地址
9 })

```

### 2、在nginx.conf进行配置

(1) 修改nginx默认端口 把80 修改 81

```

server {
 listen 81;
 server_name localhost;
}

```

#### (2) 配置nginx转发规则

在http 里面创建配置

```

server {
 listen 9001; 监听端口
 server_name localhost; 主机
 location ~ /eduservice/ { 匹配路径
 proxy_pass http://localhost:8001; 转发服务器地址
 }
 location ~ /eduoss/ {
 proxy_pass http://localhost:8002;
 }
}

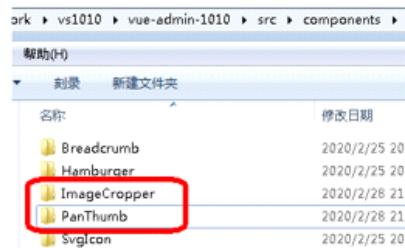
```

## 头像上传前端

### 1、添加讲师实现上传头像前端整合

(1) 在添加讲师页面，创建上传组件，实现上传

使用element-ui组件实现



(2) 添加讲师页面使用这个复制上传组件  
从课件复制上传组件代码

(3) 使用组件  
\* data() 定义变量和初始值

```
//上传弹框组件是否显示
imagecropperShow:false,
imagecropperKey:0,//上传组件key值
BASE_API:process.env.BASE_API, //获取dev.env.js里面地
```

(4) 引入组件和声明组件

```
import ImageCropper from '@/components/ImageCropper'
import PanThumb from '@/components/PanThumb' 引入
export default {
 components: { ImageCropper, PanThumb }, 声明
```

(5) 修改上传接口地址

(6) 编写close方法和上传成功的方法

```
:image-cropper
 v-show="imagecropperShow"
 :width="300"
 :height="300"
 :key="imagecropperKey"
 :url="BASE_API+'/eduoss/fileoss'"
 field="file"
```

## EasyExcel

### 二、EasyExcel简介

#### 1、EasyExcel特点

- Java领域解析、生成Excel比较有名的框架有Apache poi、jxl等。但他们都存在一个严重的问题就是非常的耗内存。如果你的系统并发量不大的话可能还行，但是一旦并发上来后一定会OOM或者JVM频繁的full gc。
- EasyExcel是阿里巴巴开源的一个excel处理框架，以使用简单、节省内存著称。EasyExcel能大大减少占用内存的主要原因是在解析Excel时没有将文件数据一次性全部加载到内存中，而是从磁盘上一行行读取数据，逐个解析。
- EasyExcel采用一行一行的解析模式，并将一行的解析结果以观察者的模式通知处理(AnalysisEventListener)。

## EasyExcel操作excel进行读和写操作

### 使用EasyExcel进行写操作

#### 第一步 引入easyexcel依赖

```
<dependencies>
<dependency>
<groupId>com.alibaba</groupId>
<artifactId>easyexcel</artifactId>
<version>2.1.1</version>
</dependency>
</dependencies> 需要poi依赖
```

#### 第二步 创建实体类，和excel数据对应

```
@Data
public class DemoData {
 //设置excel表头名称
 @ExcelProperty("学生编号")
 private Integer sno;

 @ExcelProperty("学生姓名")
 private String sname;
}
```

```
//实现excel写的操作
//1 设置写入文件夹地址和excel文件名称
String filename = "F:\\write.xlsx";
//2 调用easyexcel里面的方法实现写操作
//write方法两个参数：第一个参数文件路径名称，第二个参数实体类class
EasyExcel.write(filename, DemoData.class).sheet(sheetName: "学生列表").doWrite(getData());
```

#### 第一步 创建和excel对应实体类，标记对应列关系

```
@Data
public class DemoData {
 //设置excel表头名称
 @ExcelProperty(value = "学生编号", index = 0)
 private Integer sno;
 @ExcelProperty(value = "学生姓名", index = 1)
 private String sname;
}
```

#### 第二步 创建监听进行excel文件读取

```
//一行一行读取excel内容
@Override
public void invoke(DemoData data, AnalysisContext analysisContext) {
 System.out.println("===="+data);
}

//读取表头内容
@Override
public void invokeHeadMap(Map<Integer, String> headMap, AnalysisContext context) {
 System.out.println("表头: "+headMap);
}

//读取完成之后
@Override
public void doAfterAllAnalysed(AnalysisContext analysisContext) {}
```

#### 第三步 最终方法调用

```
//实现excel读操作
String filename = "F:\\write.xlsx";
EasyExcel.read(filename, DemoData.class, new ExcelListener()).sheet().doRead();
```

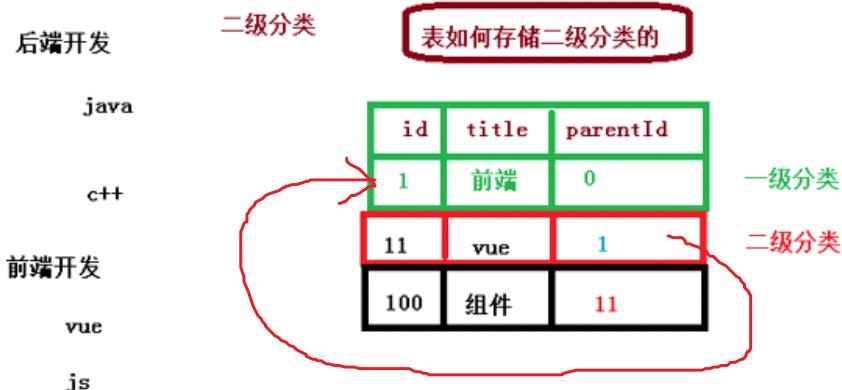
# 课程管理系统

2022年2月10日 20:25

## 链表结构建库

### 课程分类管理

课程名称: java基础开发课程	分类 后端开发
课程名称: vue高级开发课程	分类 前端开发



## 课程添加功能 (读取用户上传的excel)

### 添加课程分类前端

第一步 添加课程分类路由

- router
- index.js

课程分类管理

课程分类列表

添加课程分类

### 第二步 创建课程分类页面，修改路由对应的页面路径

- views
- dashboard
- edu
- subject
  - list.vue
  - save.vue

### 第三步 在添加课程分类页面 实现效果

#### 添加上传组件实现

信息描述 [excel模版说明](#) [点击下载模版](#)

选择Excel

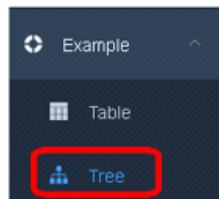
## 显示树状课程

课程列表功能

树形结构显示

```
‐ Level one 1
 ‐ Level two 1-1
‐ Level one 2
 Level two 2-1
 Level two 2-2
‐ Level one 3
 Level two 3-1
```

1、参考 tree 模块把前端整合出来



需要做的事情：创建接口，把分类按照要求的格式返回数据就可以了

返回这种格式数据

第一步 针对返回数据创建对应的实体类 两个实体类 一级和二级分类

```
[{
 id: 1,
 label: '一级分类名称',

 children: [
 {
 id: 4,
 label: '二级分类1',
 }
]
 }]
```

第二步 在两个实体类之间表示关系（一个一级分类有多个二级分类）

```
@Data
public class OneSubject {
 private String id;
 private String title;

 //一个一级分类有多个二级分类
 private List<TwoSubject> children = new ArrayList<>();
}
```

第三步 编写具体封装代码

List<EduSubject> oneSubjectList

List<OneSubject> finalSubjectList

## 后端开发步骤

1. 建表，代码生成器

2. 写 controller，配路由（请求地址），里面调用 service 方法

3. 写 service 方法接口，接口直接继承 IService<T>（IService里面集合了数据库操作）

4. 实现 service 接口

## 前端开发步骤

1. `import('@/.../xxx')` `xxx` `②`

2.在 xxx 页面中添加 <template> 和 <script>

3.<script> 如下

```
export default {
 data() { // 变量初始值
 return {
 }
 },
 created() { // 初始化方法
 },
 methods() { // 实现组件中调用的方法
 },
 component() { // 组件
 }
}
```

4.根据前端接口数据格式建类，写后端接口

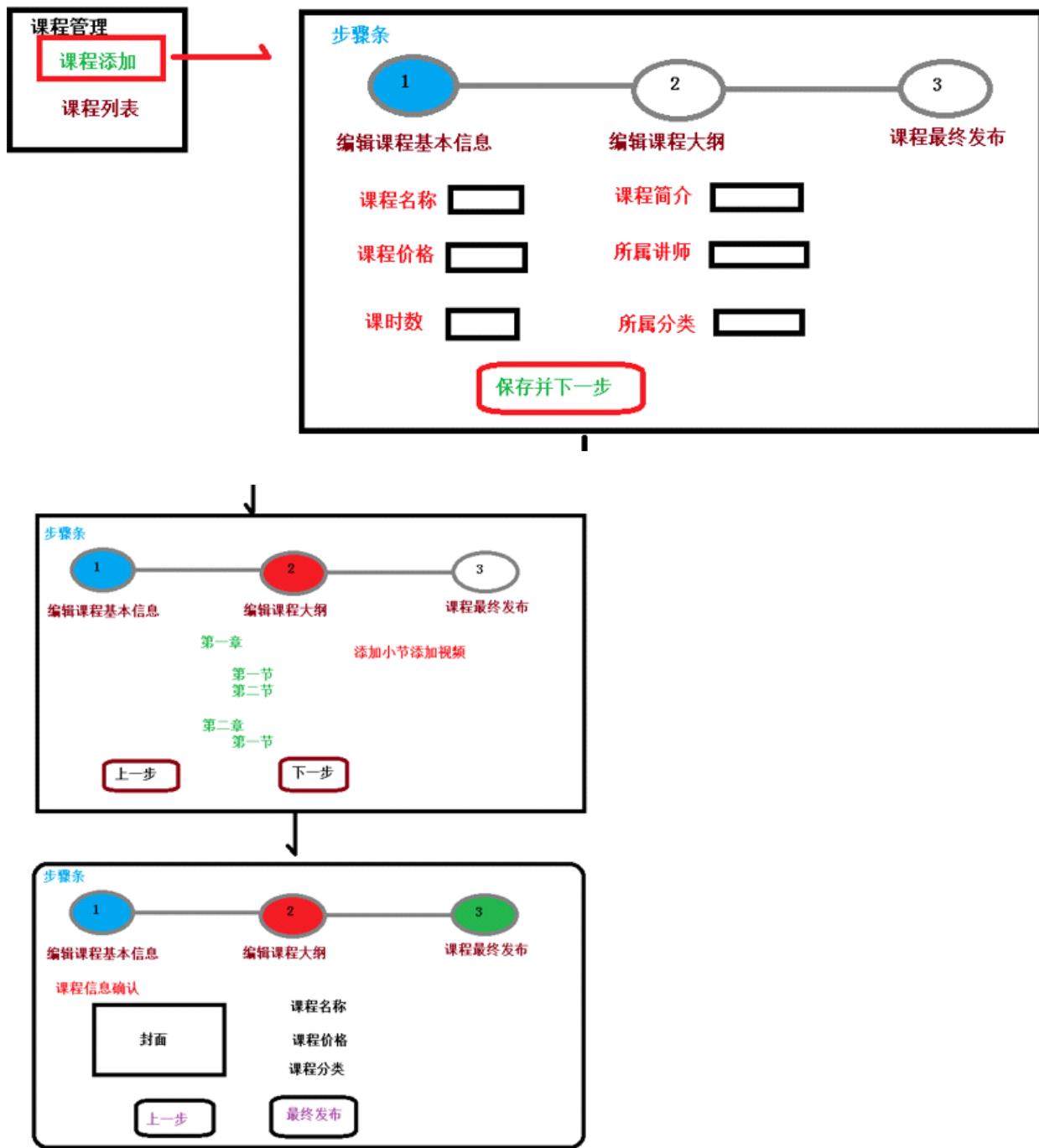
5.在 API 写接口，请求后端地址和数据传输，如下

```
import request from '@/utils/request'
export default {
 //1 添加课程信息
 addCourseInfo(courseInfo) {
 return request({
 url: '/eduservice/course/addCourseInfo',
 method: 'post',
 data: courseInfo
 })
 },
}
```

6.最后回到 xxx 页面，在 methods 函数里面调用 API 完成数据显示

```
saveOrUpdate() {
 course.addCourseInfo(this.courseInfo)
 .then(response => {
 //提示
 this.$message({
 type: 'success',
 message: '添加课程信息成功!'
 });
 //跳转到第二步
 this.$router.push({path: '/course/chapter/' + response.data.courseId})
 })
},
```

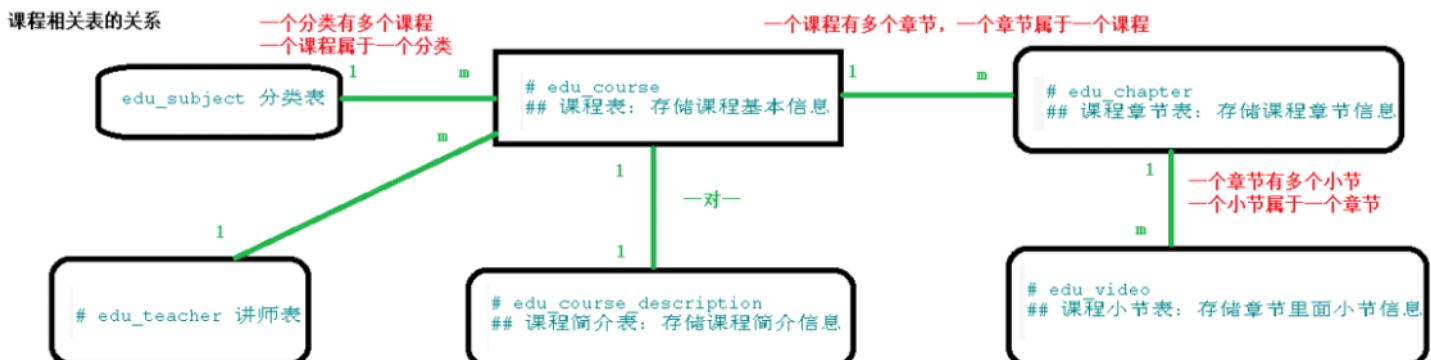
课程添加（步骤条功能）



### 1、细节问题

- (1) 创建vo实体类用于表单数据封装
- (2) 把表单提交过来的数据添加数据库  
向两张表添加数据：课程表 和 课程描述表
- (3) 把讲师 和 分类使用下列列表显示  
课程分类 做成二级联动效果

# 建表



一对多建表

一对一键表

课程管理-添加课程基本信息

第一步 使用代码生成器生成课程相关的代码

第二步 创建vo类封装表单提交的数据

第三步 编写controller和service部分

课程 和 描述是一对一关系，添加之后，id值一样的

```

//获取添加之后课程id
String cid = eduCourse.getId();

//2 向课程简介表添加课程简介
//edu_course_description
EduCourseDescription courseDescription = new EduCourseDescription()
courseDescription.setDescription(courseInfoVo.getDescription());
//设置描述id就是课程id
courseDescription.setId(cid);
courseDescriptionService.save(courseDescription);

```

修改描述实体类id生成策略

```

@ApiModelProperty(value = "课程ID")
@TableId(value = "id", type = IdType.INPUT)
private String id;

```

手动设置

baseMapper 直接操作对应的传入的数据的表

```
int insert = baseMapper.insert(eduCourse);
```

### 第一步 添加课程管理路由

添加隐藏路由，做页面跳转



### 第三步 添加之后，返回课程id

```
//添加课程基本信息的方法
@PostMapping("addCourseInfo")
public R addCourseInfo(@RequestBody CourseInfoVo courseInfoVo)
 //返回添加之后课程id，为了后面添加大纲使用
 String id = courseService.saveCourseInfo(courseInfoVo);
 return R.ok().data("courseId", id);
```

### 第二步 编写表单页面，实现接口调用

```
saveOrUpdate() {
 course.addCourseInfo(this.courseInfo)
 .then(response => {
 //提示
 this.$message({
 type: 'success',
 message: '添加课程信息成功!'
 });
 //跳转到第二步
 this.$router.push({path: '/course/chapter/' + response.data.courseId})
 })
}
```

下列列表显示所有讲师

```
<!-- 课程讲师 -->
<el-form-item label="课程讲师">
<el-select
 v-model="courseInfo.teacherId"
 placeholder="请选择">

 <el-option
 v-for="teacher in teacherList"
 :key="teacher.id"
 :label="teacher.name"
 :value="teacher.id"/>
</el-select>
</el-form-item>
```

```
created() {
 //初始化所有讲师
 this.getListTeacher()
},
methods:{
 //查询所有的讲师
 getListTeacher() {
 course.getListTeacher()
 .then(response => {
 this.teacherList = response.data.items
 })
 },
 saveOrUpdate() {
```

一级分类



所有一级都显示

二级分类



为空

选择了某个一级分类

显示选择的一级分类里  
面对应的二级分类

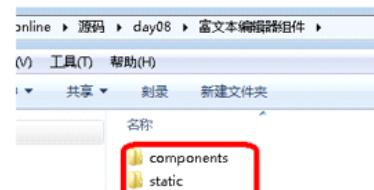
```
//点击某个一级分类，触发change，显示对应二级分类
subjectLevelOneChanged(value) {
```

```
//value就是一级分类id值
//遍历所有的分类，包含一级和二级
for(var i=0;i<this.subjectOneList.length;i++) {
 //每个一级分类
 var oneSubject = this.subjectOneList[i]
 //判断：所有一级分类id 和 点击一级分类id是否一样
 if(value === oneSubject.id) {
 //从一级分类获取里面所有的二级分类
 this.subjectTwoList = oneSubject.children
 //把二级分类id值清空
 this.courseInfo.subjectId = ''
 }
},
```

富文本编辑器

## 整合文本编辑器

### 第一步 复制文本编辑器组件 相关组件进行复制到项目里面



### 第二步 找到build/webpack.dev.conf.js 中添加配置

```
new HtmlWebpackPlugin({
 filename: 'index.html',
 template: 'index.html',
 inject: true,
 favicon: resolve('favicon.ico'),
 title: 'vue-admin-template',
 templateParameters: {
 BASE_URL: config.dev.assetsPublicPath + config.dev.assetsSubDirectory
 }
})
```

### 第三步 找到index.html引入脚本文件

```
<script src="<%= BASE_URL %>/tinymce4.7.5/tinymce.min.js></script>
<script src="<%= BASE_URL %>/tinymce4.7.5/langs/zh_CN.js></script>
```

### 第四步 页面使用文本编辑器组件

```
import Tinymce from '@/components/Tinymce' 引入组件

export default { 声明组件
 components: { Tinymce },

}
```

#### 页面中使用标签实现 文本编辑器组件

```
<!-- 课程简介-->
<el-form-item label="课程简介">
 <tinymce :height="300" v-model="courseInfo.description"/>
</el-form-item>
```

点击上一步 (有id, 而新创建的没有)

### 修改课程基本信息

第一步 点击 上一步时候

上一步

回到第一步

把课程基本信息数据回显

第二步 在数据回显页面，修改内容，保存，修改数据库内容

### 开发后端接口

1、根据课程id查询课程基本信息接口

//根据课程id查询课程基本信息

```
@GetMapping("getCourseInfo/{courseId}")
public R getCourseInfo(@PathVariable String courseId) {
 CourseInfoVo courseInfoVo = courseService.getCourseInfo(courseId);
 return R.ok().data("courseInfoVo", courseInfoVo);
}
```

2、修改课程信息接口

//修改课程信息

```
@PostMapping("updateCourseInfo")
public R updateCourseInfo(@RequestBody CourseInfoVo courseInfoVo) {
 courseService.updateCourseInfo(courseInfoVo);
 return R.ok();
}
```

### 开发前端

第一步 在api里面course.js定义接口两个方法

第二步修改chapter页面，跳转路径

课程id

```
previous() {
 this.$router.push({path: '/course/info' + this.courseId})
},
```

第三步 在info页面实现数据回显

获取路由课程id，调用根据id查询接口，数据显示

```
created() {
 //获取路由id值
 if(this.$route.params && this.$route.params.id) {
 this.courseId = this.$route.params.id
 //调用根据id查询课程的方法
 this.getInfo()
 }
}
```

```
methods: {
 //根据课程id查询
 getInfo() {
 course.getCourseInfoId(this.courseId)
 .then(response => {
 this.courseInfo = response.data.courseInfoVo
 })
 },
}

```

### 删除

**删除章节：**

**如果章节里面没有小节，直接删除**

**如果章节里面有小节，如何删除？**

**第一种 删除章节时候，把章节里面所有小节都删除**

**第二种 如果删除的章节下面有小节，不让进行删除**

////删除章节的方法

```
@Override
public boolean deleteChapter(String chapterId) {
 //根据chapterId章节id 查询小节表，如果查询数据，不进行删除
 QueryWrapper<EduVideo> wrapper = new QueryWrapper<>();
 wrapper.eq("chapter_id",chapterId);
 int count = videoService.count(wrapper);
 //判断
 if(count>0){//查询出小节，不进行删除
 throw new GuliException(20001,"不能删除");
 } else { //不能查询数据，进行删除
 //删除章节
 int result = baseMapper.deleteById(chapterId);
 //成功 1>0 0>0
 return result>0;
 }
}
```

## 修改时的弹窗

```
<!-- 添加和修改章节表单 -->
<el-dialog :visible.sync="dialogChapterFormVisible" title="添加章节">
 <el-form :model="chapter" label-width="120px">
 <el-form-item label="章节标题">
 <el-input v-model="chapter.title"/>
 </el-form-item>
 <el-form-item label="章节排序">
 <el-input-number v-model="chapter.sort" :min="0" controls-position="right"/>
 </el-form-item>
 </el-form>
 <div slot="footer" class="dialog-footer">
 <el-button @click="dialogChapterFormVisible = false">取消</el-button>
 <el-button type="primary" @click="saveOrUpdate">确定</el-button>
 </div>
</el-dialog>
```

## 自定义 mapper 函数（sql）（当查询多张表时）

## 课程信息确认



课程名称  
课程价格 课程简介  
课程分类 课程讲师

这些数据查询多张表才可以得到，一般编写sql语句实现

### 多表连接查询

# 内连接  
# 左外连接  
# 右外连接

1	前端
2	后端
3	运维

11	java	2
12	vue	1
13	mysql	null

```
SELECT ec.id, ec.title, ec.price, ec.lesson_num,
 ecd.description,
 et.name,
 es1.title AS oneSubject,
 es2.title AS twoSubject
 FROM edu_course ec LEFT OUTER JOIN edu_course_description ecd ON ec.id=ecd.id
 LEFT OUTER JOIN edu_teacher et ON ec.teacher_id=et.id
 LEFT OUTER JOIN edu_subject es1 ON ec.subject_parent_id=es1.id
 LEFT OUTER JOIN edu_subject es2 ON ec.subject_id=es2.id
 WHERE ec.id='1234747900958183425'
```

项目中创建Mapper接口，编写xml文件sql语句，执行出现错误

ibatis.binding.BindingException: Invalid bound statement (not found): com.atguigu.eduservice.mapper.

这个错误是有maven默认加载机制造成问题

maven加载时候，把java文件夹里面.java类型文件进行编译，如果其他类型文件，不会加载

解决方式：

1、复制xml到target目录中

2、把xml文件放到resources目录中

3、推荐使用：通过配置实现

(1) pom.xml

(2) 项目application.properties

## 课程的删除

从小到大的删除，小的表里面要设置大的id，方便操作

## 课程删除

1、课程里面 有 课程描述 章节 小节 视频  
(1) 删除课程 把视频 小节 章节 描述 课程本身都删除

外键 一般不建议声明出来

课程 1

1	java入门
2	vue高级
3	mysql开发

章节 m 外键

11	1 java语句	1
12	2 java语法	1
13	1 vue指令	2
14	1 mysql安装	3

在多的那一方创建  
字段，作为外键，  
指向一的那一方  
主键

```
@Override
public void removeCourse(String courseId) {
 //1 根据课程id删除小节
 eduVideoService.removeVideoByCourseId(courseId);

 //2 根据课程id删除章节
 chapterService.removeChapterByCourseId(courseId);

 //3 根据课程id删除描述
 courseDescriptionService.removeById(courseId);

 //4 根据课程id删除课程本身
 int result = baseMapper.deleteById(courseId);
 if(result == 0){ //失败返回
 throw new GuliException(20001,"删除失败");
 }
}
```

# 阿里云视频

2022年2月13日 15:32

## 阿里云视频点播



开通视频点播服务，选择按照流量计费

1、看到上传到阿里云视频点播视频文件  
2、上传视频文件

媒体处理配置  
转码模板组

1 服务端：后端接口  
客户端：浏览器、安卓、ios

httpclient技术可以调用api地址  
?Action=GetPlayInfo&VideoId=1234  
http://vod.cn-shanghai.aliyuncs.com/

2 API : 阿里云提供固定的地址，只需要调用这个固定的地址，向地址传递参数，实现功能  
SDK : sdk对api方式进行封装，更方便使用。之前使用EasyExcel  
调用阿里云提供类或者接口里面的方法实现视频功能

1、获取视频播放地址 根据视频id获取到

video\_source\_id varchar(100) NULL 云端视频资源

因为上传视频可以进行加密，加密之后，使用加密之后地址不能进行视频播放，在数据库存储不存地址，而是存储 视频id

2、获取视频播放凭证 根据视频id获取到

3、上传视频到阿里云视频点播服务

### 3、上传视频到阿里云视频点播服务

1、在service创建子模块service\_vod  
引入相关依赖

到01-视频点播微服务的创建笔记 进行复制

2、初始化操作，创建DefaultAcsClient 对象

```
public class InitObject {
 public static DefaultAcsClient initVodClient(String accessKeyId, String
accessKeySecret) throws ClientException {
 String regionId = "cn-shanghai"; // 点播服务接入区域
 DefaultProfile profile = DefaultProfile.getProfile(regionId, accessKeyId,
accessKeySecret);
 DefaultAcsClient client = new DefaultAcsClient(profile);
 return client;
 }
}
```

### 3、实现根据视频id获取视频播放地址

```
//1 根据视频ID获取视频播放地址
//创建初始化对象
DefaultAcsClient client = InitObject.initVodClient("LTAI4FvvVEW", "345");

//创建获取视频地址request和response
GetPlayInfoRequest request = new GetPlayInfoRequest();
GetPlayInfoResponse response = new GetPlayInfoResponse();

//向request对象里面设置视频id
request.setVideoId("474be24d43ad4f76af344b9f4daab1");

//调用初始化对象里面的方法，传递request，获取数据
response = client.getAcsResponse(request);

List<GetPlayInfoResponse.PlayInfo> playInfoList = response.getPlayInfoList();
//播放地址
for (GetPlayInfoResponse.PlayInfo playInfo : playInfoList) {
 System.out.print("PlayInfo.PlayURL = " + playInfo.getPlayURL() + "\n");
}
```

4、实现根据视频id获取播放凭证

```
//1 根据视频ID获取视频播放凭证
DefaultAcsClient client = InitObject.initVodClient("LTAI4FvvVEW",
"9st82dv7EvF");

GetVideoPlayAuthRequest request = new GetVideoPlayAuthRequest();
GetVideoPlayAuthResponse response = new
GetVideoPlayAuthResponse();

request.setVideoId("474be24d43ad4f76af344b9f4daab1");

response = client.getAcsResponse(request);
System.out.println("playAuth:" + response.getPlayAuth());
```

添加小节 上传视频

添加课时

课时标题 第一节 java入门语法

课时排序 1

是否免费  免费  默认

上传视频

第一步 引入依赖

第二步 创建application配置文件

第三步 创建启动类

```
@SpringBootApplication(exclude = DataSourceAutoConfiguration.class)
@ComponentScan(basePackages = {"com.atguigu"})
public class VodApplication {
 public static void main(String[] args) {
 SpringApplication.run(VodApplication.class, args);
 }
}
```

上面代码运行出现错误，上传大小问题

```
Caused by: org.apache.tomcat.util.http.fileupload.
FileUploadBase$FileSizeLimitExceededException:
The field file exceeds its maximum permitted size of 1048576 bytes.
```

第四步 创建controller service

```
InputStream inputStream = file.getInputStream();
UploadStreamRequest request = new UploadStreamRequest(
ConstantVodUtils.ACCESS_KEY_ID, ConstantVodUtils.ACCESS_KEY_SECRET, title, fileName,
inputStream);

UploadVideoImpl uploader = new UploadVideoImpl();
UploadStreamResponse response = uploader.uploadStream(request);

String videoId = null;
if (response.isSuccess()) {
 videoId = response.getVideoId();
} else { //如果设置回调url不正确，不能响应视频上传，可以返回VideoId同时会返回错误码。其他情况下
//失败时，VideoId为空，此时需要根据返回错误码分析具体错误原因
 videoId = response.getVideoId();
}
return videoId;
```

在application进行文件大小设置

```
最大上传单个文件大小：默认1M
spring.servlet.multipart.max-file-size=1024MB
最大总上传的数据大小：默认10M
spring.servlet.multipart.max-request-size=1024MB
```



# Javaweb开发步骤（一）

2022年2月11日 14:08

## 后端开发步骤

- 1.建表，代码生成器
- 2.写 controller，配路由（请求地址），里面调用 service 方法
- 3.写 service 方法接口，接口直接继承 IService<T>（IService里面集合了数据库操作）
- 4.实现 service 接口，继承 mapper 和接口，如下  

```
public class EduSubjectServiceImpl extends ServiceImpl<EduSubjectMapper, EduSubject>
implements EduSubjectService
```
- 5.关于mapper（持久层），在 xml 里面定义好 sql 语句，就可以调用 mapper
- 6.swagger 测试接口

## 前端开发步骤

- 1.写路由，import('@/.../xxx')，跳转到 xxx 页面
- 2.写 xxx 页面，先复制组件（<template>标签），再写函数（<script>标签）
- 3.<script>标签写法如下

```
export default {
 data() { // 变量初始值
 return {

```

```

 }
 },
 created() { // 初始化方法
 },
 methods() { // 实现组件中调用的方法
 },
 component() { // 组件
 }
}

```

4.根据前端接口数据格式建类，写后端接口

5.在 API 写接口，请求后端地址和数据传输，如下

```

import request from '@/utils/request'
export default {
 //1 添加课程信息
 addCourseInfo(courseInfo) {
 return request({
 url: '/eduservice/course/addCourseInfo',
 method: 'post',
 data:courseInfo
 })
 },
}

```

6.最后回到 xxx 页面，在 methods 函数里面调用 API 完成数据显示

```

saveOrUpdate() {
 course.addCourseInfo(this.courseInfo)
 .then(response => {
 //提示
 this.$message({
 type: 'success',
 message: '添加课程信息成功!'
 });
 //跳转到第二步
 this.$router.push({path: '/course/chapter/' + response.data.courseId})
 })
 ,
}

```

# 微服务架构

2022年2月14日 16:52

## 一、什么是微服务

### 1、微服务的由来

微服务最早由Martin Fowler与James Lewis于2014年共同提出，微服务架构风格是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，并使用轻量级机制通信，通常是HTTP API，这些服务基于业务能力构建，并能够通过自动化部署机制来独立部署，这些服务使用不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理。

### 2、为什么需要微服务

在传统的IT行业软件大多都是各种独立系统的堆砌，这些系统的问题总结来说就是扩展性差，可靠性不高，维护成本高。到后面引入了SOA服务化，但是，由于 SOA 早期均使用了总线模式，这种总线模式是与某种技术栈强绑定的，比如：J2EE。这导致很多企业的遗留系统很难对接，切换时间太长，成本太高，新系统稳定性的收敛也需要一些时间。

### 3、微服务与单体架构区别

(1) 单体架构所有的模块全都耦合在一块，代码量大，维护困难。

微服务每个模块就相当于一个单独的项目，代码量明显减少，遇到问题也相对来说比较好解决。

(2) 单体架构所有的模块都共用一个数据库，存储方式比较单一。

微服务每个模块都可以使用不同的存储方式（比如有的用redis，有的用mysql等），数据库也是单个模块对自己的数据库。

(3) 单体架构所有的模块开发所使用的技术一样。

微服务每个模块都可以使用不同的开发技术，开发模式更灵活。

### 4、微服务本质

(1) 微服务，关键其实不仅仅是微服务本身，而是系统要提供一套基础的架构，这种架构使得微服务可以独立的部署、运行、升级，不仅如此，这个系统架构还让微服务与微服务之间在结构上“松耦合”，而在功能上则表现为一个统一的整体。这种所谓的“统一的整体”表现出来的是统一风格的界面，统一的权限管理，统一的安全策略，统一的上线过程，统一的日志和审计方法，统一的调度方式，统一的访问入口等等。

(2) 微服务的目的是有效的拆分应用，实现敏捷开发和部署。

(3) 微服务提倡的理念团队间应该是 inter-operate, not integrate 。inter-operate是定义好系统的边界和接口，在一个团队内全栈，让团队自治，原因就是因为如果团队按照这样的方式组建，将沟通的成本维持在系统内部，每个子系统就会更加内聚，彼此的依赖耦合能变弱，跨系统的沟通成本也就能降低。

### 5、什么样的项目适合微服务

微服务可以按照业务功能本身的独立性来划分，如果系统提供的业务是非常底层的，如：操作系统内核、存储系统、网络系统、数据库系统等等，这类系统都偏底层，功能和功能之间有着紧密的配合关系，如果强制拆分为较小的服务单元，会让集成工作量急剧上升，并且这种人为的切割无法带来业务上的真正的隔离，所以无法做到独立部署和运行，也就不适合做成微服务了。

## 6、微服务开发框架

目前微服务的开发框架，最常用的有以下四个：

Spring Cloud: <http://projects.spring.io/spring-cloud> (现在非常流行的微服务架构)

Dubbo: <http://dubbo.io>

Dropwizard: <http://www.dropwizard.io> (关注单个微服务的开发)

Consul、etcd&etc. (微服务的模块)

## 7、什么是Spring Cloud

Spring Cloud是一系列框架的集合。它利用Spring Boot的开发便利性简化了分布式系统基础设施的开发，如服务发现、服务注册、配置中心、消息总线、负载均衡、熔断器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过SpringBoot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留下了一套简单易懂、易部署和易维护的分布式系统开发工具包。

## 8、Spring Cloud和Spring Boot是什么关系

Spring Boot 是 Spring 的一套快速配置脚手架，可以基于Spring Boot 快速开发单个微服务，Spring Cloud是一个基于Spring Boot实现的开发工具；Spring Boot专注于快速、方便集成的单个微服务个体，Spring Cloud关注全局的服务治理框架；Spring Boot使用了默认大于配置的理念，很多集成方案已经帮你选择好了，能不配置就不配置，Spring Cloud很大的一部分是基于Spring Boot来实现，必须基于Spring Boot开发。可以单独使用Spring Boot开发项目，但是Spring Cloud离不开 Spring Boot。

## 9、Spring Cloud相关基础服务组件

服务发现——Netflix Eureka (Nacos)

服务调用——Netflix Feign

熔断器——Netflix Hystrix

服务网关——Spring Cloud GateWay

分布式配置——Spring Cloud Config (Nacos)

消息总线 —— Spring Cloud Bus (Nacos)

## 10、Spring Cloud的版本

Spring Cloud并没有熟悉的数字版本号，而是对应一个开发代号。

Cloud代号	Boot版本(train)	Boot版本(tested)	lifecycle
Angle	1.2.x	incompatible with 1.3	EOL in July 2017
Brixton	1.3.x	1.4.x	2017-07卒
Camden	1.4.x	1.5.x	-
Dalston	1.5.x	not expected 2.x	-
Edgware	1.5.x	not expected 2.x	-
Finchley	2.0.x	not expected 1.5.x	-
Greenwich	<b>2.1.x</b>		

开发代号看似没有什么规律，但实际上首字母是有顺序的，比如：Dalston版本，我们可以简称D版本，对应的Edgware版本我们可以简称E版本。

### 小版本

Spring Cloud 小版本分为：

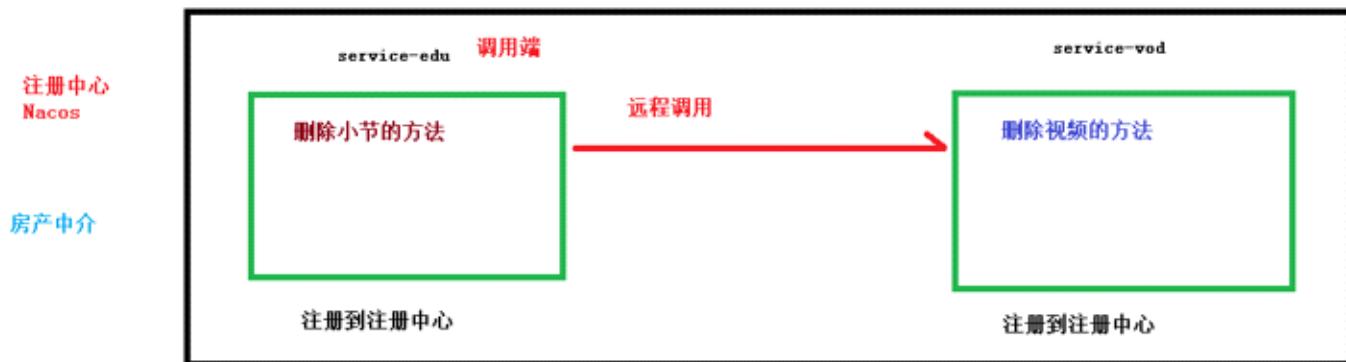
SNAPSHOT：快照版本，随时可能修改

M： MileStone，M1表示第1个里程碑版本，一般同时标注PRE，表示预览版版。

SR：Service Release，SR1表示第1个正式版本，一般同时标注GA：(Generally Available)，表示稳定版本。

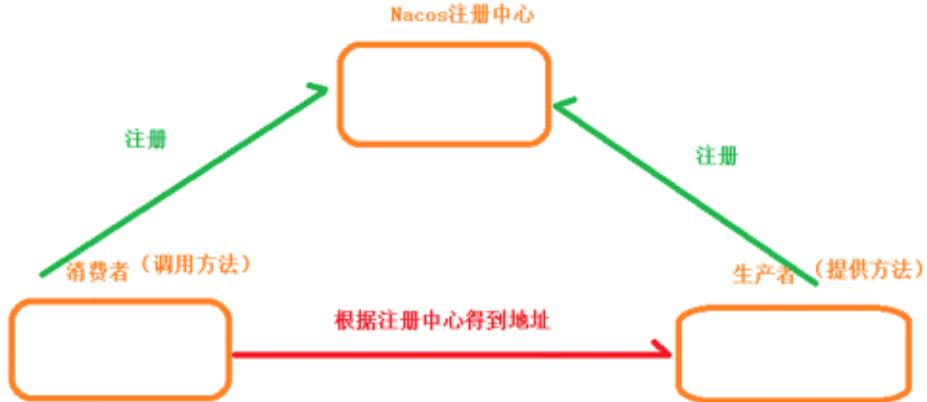
## 模块间调用，Nacos

~~删除小节~~  
~~删除阿里云视频~~



实现不同的微服务模块之间调用，把这些模块在注册中心进行注册，注册之后，实现互相调用

### Nacos流程



## 服务注册

### 把service-edu服务在Nacos进行注册

#### 第一步 引入依赖在service的pom文件中

```
<!--服务注册-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

#### 第三步 在启动类添加注解

```
@SpringBootApplication
@EnableDiscoveryClient //nacos注册
@ComponentScan(basePackages = { com.atguigu })
public class EduApplication {
```

#### 第二步 在要注册的服务的配置文件中 application进行配置 配置Nacos地址

```
nacos服务地址
spring.cloud.nacos.discovery.server-addr=
127.0.0.1:8848
```



## 服务调用

### 一、Feign

#### 1、基本概念

Feign是Netflix开发的声明式、模板化的HTTP客户端，Feign可以帮助我们更快捷、优雅地调用HTTP API。

Feign支持多种注解，例如Feign自带的注解或者JAX-RS注解等。

Spring Cloud对Feign进行了增强，使Feign支持了Spring MVC注解，并整合了Ribbon和Eureka，从而让Feign的使用更加方便。

Spring Cloud Feign是基于Netflix feign实现，整合了Spring Cloud Ribbon和Spring Cloud Hystrix，除了提供这两者的强大功能外，还提供了一种声明式的Web服务客户端定义的方式。

Spring Cloud Feign帮助我们定义和实现依赖服务接口的定义。在Spring Cloud feign的实现下，只需要创建一个接口并用注解方式配置它，即可完成服务提供方的接口绑定，简化了在使用Spring Cloud Ribbon时自行封装服务调用客户端的开发量。

#### 二、实现服务调用

前提条件：把互相调用服务在Nacos进行注册

#### 第一步 引入依赖在service模块

```
<!--服务调用-->
<dependency>
 <groupId>org.springframework.cloud</groupId>
 <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

#### 第三步 在调用端 创建interface，使用注解指定调用服务名称，定义调用的方法路径

```
@FeignClient("service-vod")
@Component
public interface VodClient {

 //定义调用的方法路径
 //根据视频id删除阿里云视频
 //@PathVariable注解一定要指定参数名称，否则出错
 @DeleteMapping("/eduvod/video/removeAlyVideo/{id}")
 public R removeAlyVideo(@PathVariable("id") String id);
}
```

#### 第二步 在调用端 service-edu服务启动类添加注解

```
@SpringBootApplication
@EnableDiscoveryClient //nacos注册
@EnableFeignClients
@ComponentScan(basePackages = {"com.atguigu"})
public class EduApplication {
```

#### 第四步 实现代码删除小节删除阿里云视频

```
//删除小节，删除对应阿里云视频
@DeleteMapping("{id}")
public R deleteVideo(@PathVariable String id) {
 //根据小节id获取视频id，调用方法实现视频删除
 EduVideo eduVideo = videoService.getById(id);
 String videoSourceId = eduVideo.getVideoSourceId();
 //判断小节里面是否有视频id
 if(StringUtils.isNotEmpty(videoSourceId)) {
 //根据视频id，远程调用实现视频删除
 vodClient.removeAlyVideo(videoSourceId);
 }
 //删除小节
 videoService.removeById(id);
 return R.ok();
}
```

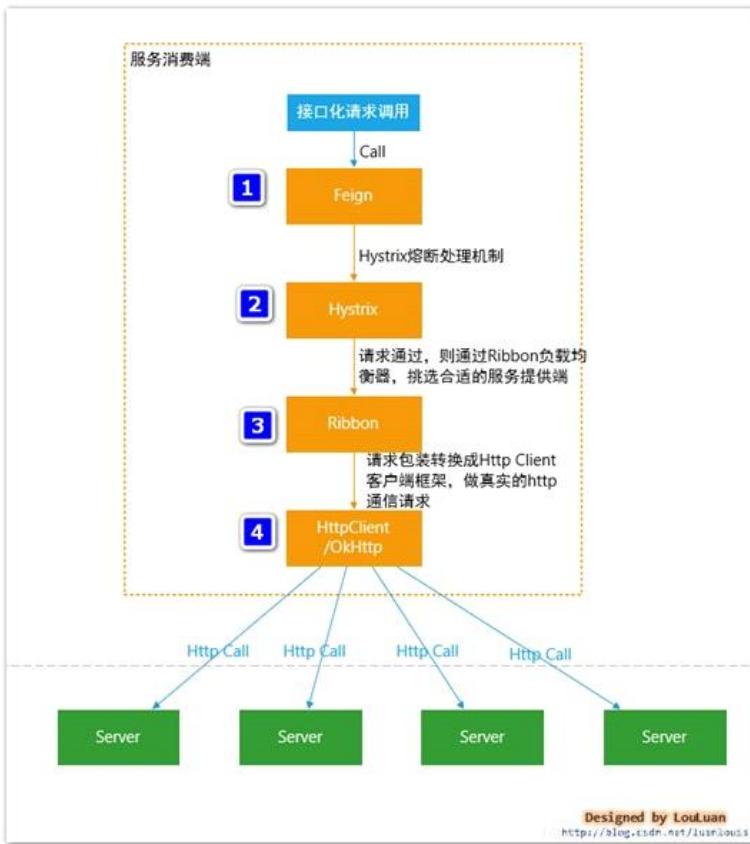
## 熔断器

### 一、Hystrix基本概念

#### 1、Spring Cloud调用接口过程

Spring Cloud 在接口调用上，大致会经过如下几个组件配合：

Feign -----> Hystrix —> Ribbon —> Http Client (apache http components 或者 Okhttp) 具体交互流程上，如下图所示：



(1) 接口化请求调用当调用被@FeignClient注解修饰的接口时，在框架内部，将请求转换成Feign的请求实例feign.Request，交由Feign框架处理。

2 Feign：转化请求Feign是一个http请求调用的轻量级框架，可以以Java接口注解的方式调用Http请求，封装了Http调用流程。

3 Hystrix：熔断处理机制 Feign的调用关系，会被Hystrix代理拦截，对每一个Feign调用请求，Hystrix都会将其包装成HystrixCommand,参与Hystrix的流控和熔断规则。如果请求判断需要熔断，则Hystrix直接熔断，抛出异常或者使用FallbackFactory返回熔断Fallback结果；如果通过，则将调用请求传递给Ribbon组件。

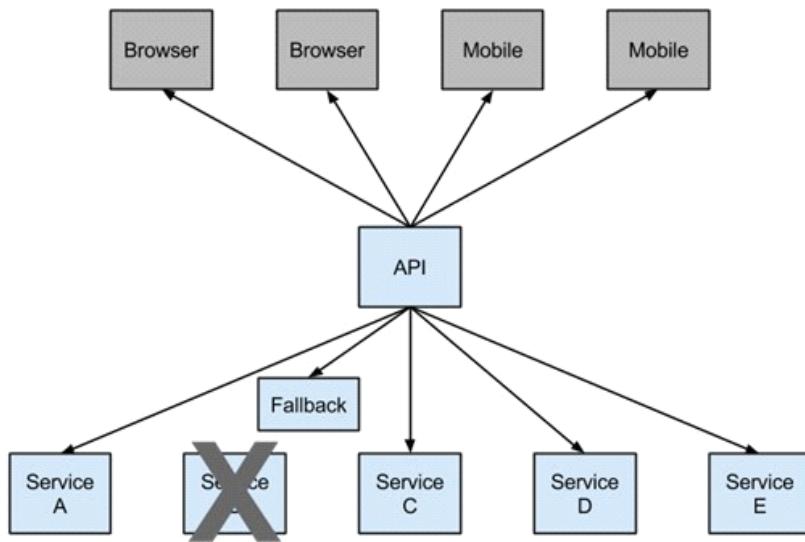
4 Ribbon：服务地址选择 当请求传递到Ribbon之后,Ribbon会根据自身维护的服务列表，根据服务的服务质量，如平均响应时间，Load等，结合特定的规则，从列表中挑选合适的服务实例，选择好机器之后，然后将机器实例的信息请求传递给Http Client客户端，HttpClient客户端来执行真正的Http接口调用；

5 HttpClient：Http客户端，真正执行Http调用根据上层Ribbon传递过来的请求，已经指定了服务地址，则HttpClient开始执行真正的Http请求

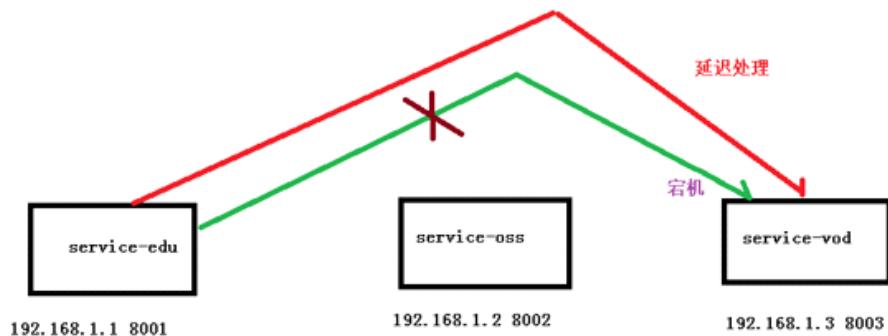
## 2、Hystrix概念

Hystrix是一个供分布式系统使用，提供延迟和容错功能，保证复杂的分布系统在面临不可避免的失败时，仍能有其弹性。

比如系统中有很多服务，当某些服务不稳定的时候，使用这些服务的用户线程将会阻塞，如果没有隔离机制，系统随时就有可能会挂掉，从而带来很大的风险。SpringCloud使用Hystrix组件提供断路器、资源隔离与自我修复功能。下图表示服务B触发了断路器，阻止了级联失败



## 加入熔断器



第一步 添加熔断器依赖

第二步 在调用端配置文件中开启熔断器

```
#开启熔断机制
feign.hystrix.enabled=true
```

第三步 在创建interface之后，还需要创建interface 对应实现类，在实现类实现方法，出错了输出内容

```
@Component
public class VodFileDegradeFeignClient implements VodClient {
 //出错之后会执行
 @Override
 public R removeAlyVideo(String id) {
 return R.error().message("删除视频出错了");
 }

 @Override
 public R deleteBatch(List<String> videoIdList) {
 return R.error().message("删除多个视频出错了");
 }
}
```

第四步 在interface上面添加注解和属性

```
@FeignClient(name = "service-vod", fallback = VodFileDegradeFeignClient.class)
@Component
public interface VodClient {
```

# MUXT前台

2022年2月14日 16:59

## 一、服务端渲染技术NUXT

### 1、什么是服务端渲染

服务端渲染又称SSR (Server Side Render)是在服务端完成页面的内容，而不是在客户端通过AJAX获取数据。

服务器端渲染(SSR)的优势主要在于：更好的 SEO，由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面。

如果你的应用程序初始展示 loading 图，然后通过 Ajax 获取内容，抓取工具并不会等待异步完成后再进行页面内容的抓取。也就是说，如果 SEO 对你的站点至关重要，而你的页面又是异步获取内容，则你可能需要服务器端渲染(SSR)解决此问题。

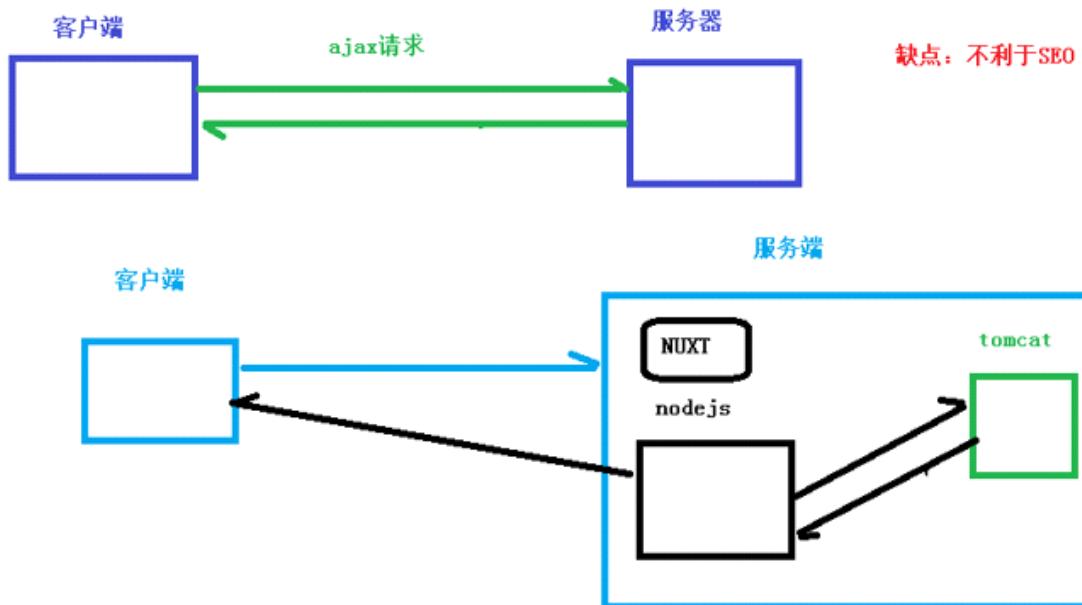
另外，使用服务器端渲染，我们可以获得更快的内容到达时间(time-to-content)，无需等待所有的 JavaScript 都完成下载并执行，产生更好的用户体验，对于那些「内容到达时间(time-to-content)与转化率直接相关」的应用程序而言，服务器端渲染(SSR)至关重要。

### 2、什么是NUXT

Nuxt.js 是一个基于 Vue.js 的轻量级应用框架,可用来创建服务端渲染 (SSR) 应用,也可充当静态站点引擎生成静态站点应用,具有优雅的代码结构分层和热加载等特性。

#### 搭建项目前台系统环境

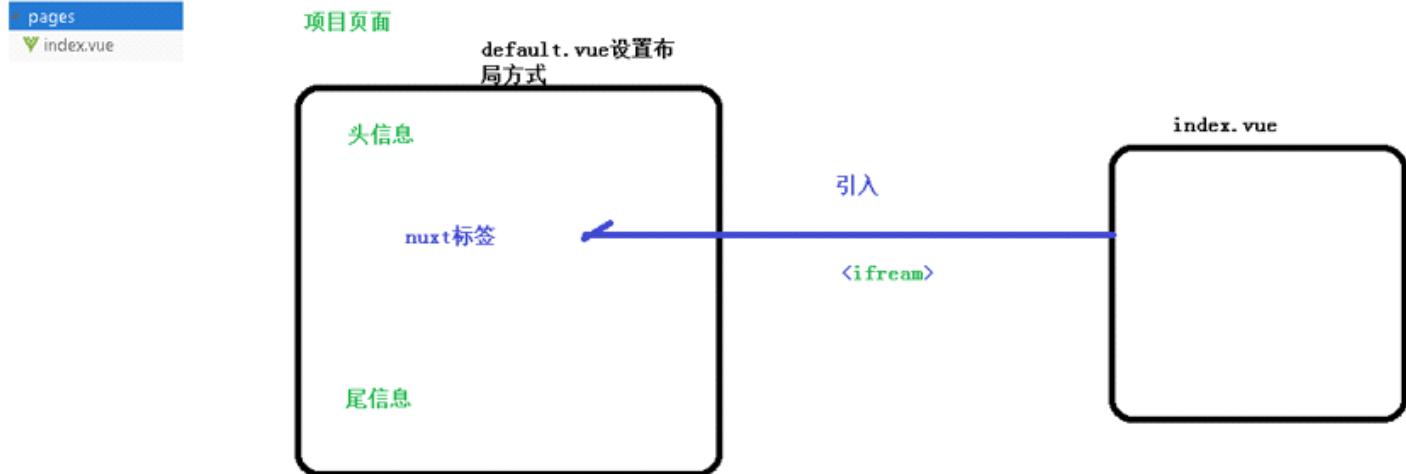
##### 使用NUXT框架搭建前台环境 服务端渲染技术



## nuxt环境目录结构



JS nuxt.config.js  
nuxt框架核心配置文件



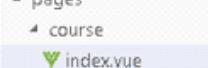
## nuxt路由

第一种 固定路由 路径是固定地址，不发生变化 课程 名师

```
<router-link to="/course" tag="li" active-class="current">
 <a>课程
</router-link>
```

to属性设置路由跳转地址 /course

做法：在pages里面创建文件夹 course  
在course文件夹 创建 index.vue



## 第二种 动态路由

每次生成路由地址不一样，比如课程详情页面，每个课程id不一样

NUXT的动态路由是以下划线开头的vue文件，参数名为下划线后边的文件名



## banner 设置

首页数据banner显示（幻灯片或者轮播图）

1、在service创建子模块service\_cms



2、创建配置文件

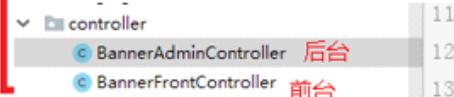
3、创建数据库表，根据 表使用代码生成器

(1) 到资料找到数据库脚本创建



4、后台对banner管理接口 crud操作

前台对banner显示接口



```
//查询所有banner
@Override
public List<CrmBanner> selectAllBanner() {
 //根据id进行降序排列，显示排列之后前两条记录
 QueryWrapper<CrmBanner> wrapper = new QueryWrapper<>();
 wrapper.orderByDesc("id");
 //last方法，拼接sql语句
 wrapper.last("lastSql: 'limit 2'");
 List<CrmBanner> list = baseMapper.selectList(wrapper);
 return list;
}
```

# redis 缓存

2022年2月15日 20:34



## 一、Redis介绍

Redis是当前比较热门的NOSQL系统之一，它是一个开源的使用ANSI c语言编写的key-value存储系统（区别于MySQL的二维表格的形式存储。）。和Memcache类似，但很大程度补偿了Memcache的不足。和Memcache一样，Redis数据都是缓存在计算机内存中，不同的是，Memcache只能将数据缓存到内存中，无法自动定期写入硬盘，这就表示，一断电或重启，内存清空，数据丢失。所以Memcache的应用场景适用于缓存无需持久化的数据。而Redis不同的是它会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，实现数据的持久化。

Redis的特点：

- 1, Redis读取的速度是110000次/s，写的速度是81000次/s；
- 2, 原子。Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。
- 3, 支持多种数据结构：string（字符串）；list（列表）；hash（哈希），set（集合）；zset（有序集合）
- 4, 持久化，集群部署
- 5, 支持过期时间，支持事务，消息订阅

第一步 创建redis配置类 写到common里面

- (1) 引入springboot整合redis相关依赖
- (2) 创建redis缓存配置类，配置插件

common的pom文件引入

```
<!-- redis -->
<dependency>
 <groupId>org.springframework.boot</groupId>
 <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

```
<!-- spring2.X集成redis所需common-pool2-->
<dependency>
 <groupId>org.apache.commons</groupId>
 <artifactId>commons-pool2</artifactId>
 <version>2.6.0</version>
</dependency>
```

第二步 在查询所有banner的方法上面添加缓存注解 @Cacheable



第一次查询，首先查询数据库，把数据库查询数据返回，并且返回数据放到缓存中

第二次查询，查询缓存，发现缓存有数据，直接返回

```
//查询所有banner
@Cacheable(value = "banner", key = "selectIndexList")
@Override
public List<CrmBanner> selectAllBanner() {
```

Redis是一个key-value的数据库，key一般是String类型，不过value的类型多种多样：

String	hello world	基本类型
Hash	{name: "Jack", age: 21}	
List	[A -> B -> C -> C]	
Set	{A, B, C}	
SortedSet	{A: 1, B: 2, C: 3}	特殊类型
GEO	{A: (120.3, 30.5)}	
BitMap	0110110101110101011	
HyperLog	0110110101110101011	

高级特性

String类型的三种格式：

- 字符串
- int
- float

Redis的key的格式：

- [项目名]:[业务名]:[类型]:[id]

## Hash类型

Hash类型，也叫散列，其value是一个无序字典，类似于Java中的HashMap结构。

String结构是将对象序列化为JSON字符串后存储，当需要修改对象某个字段时很不方便：

KEY	VALUE
heima:user:1	{name:"Jack", age:21}
heima:user:2	{name:"Rose", age:18}

Hash结构可以将对象中的每个字段独立存储，可以针对单个字段做CRUD：

KEY	VALUE	
	field	value
heima:user:1	name	Jack
	age	21
heima:user:2	name	Rose
	age	18

## List类型

Redis中的List类型与Java中的LinkedList类似，可以看做是一个双向链表结构。既可以支持正向检索和也可以支持反向检索。

特征也与LinkedList类似：

- 有序
- 元素可以重复
- 插入和删除快
- 查询速度一般

常用来存储一个有序数据，例如：朋友圈点赞列表，评论列表等。

## Set类型

Redis的Set结构与Java中的HashSet类似，可以看做是一个value为null的HashMap。因为也是一个hash表，因此具备与HashSet类似的特征：

- 无序
- 元素不可重复
- 查找快
- 支持交集、并集、差集等功能

## SortedSet类型

Redis的SortedSet是一个可排序的set集合，与Java中的TreeSet有些类似，但底层数据结构却差别很大。SortedSet中的每一个元素都带有一个score属性，可以基于score属性对元素排序，底层的实现是一个跳表（SkipList）加hash表。

SortedSet具备下列特性：

- 可排序
- 元素不重复
- 查询速度快

因为SortedSet的可排序特性，经常被用来实现排行榜这样的功能。

## 实战



## 缓存更新策略的最佳实践方案：

1. 低一致性需求：使用Redis自带的内存淘汰机制
2. 高一致性需求：主动更新，并以超时剔除作为兜底方案

### ◆ 读操作：

- 缓存命中则直接返回
- 缓存未命中则查询数据库，并写入缓存，设定超时时间

### ◆ 写操作：

- 先写数据库，然后再删除缓存
- 要确保数据库与缓存操作的原子性

## 有关缓存的问题

<https://blog.csdn.net/a898712940/article/details/116212825>

## 缓存穿透

### 缓存穿透

缓存穿透是指客户端请求的数据在缓存中和数据库中都不存在，这样缓存永远不会生效，这些请求都会打到数据库。

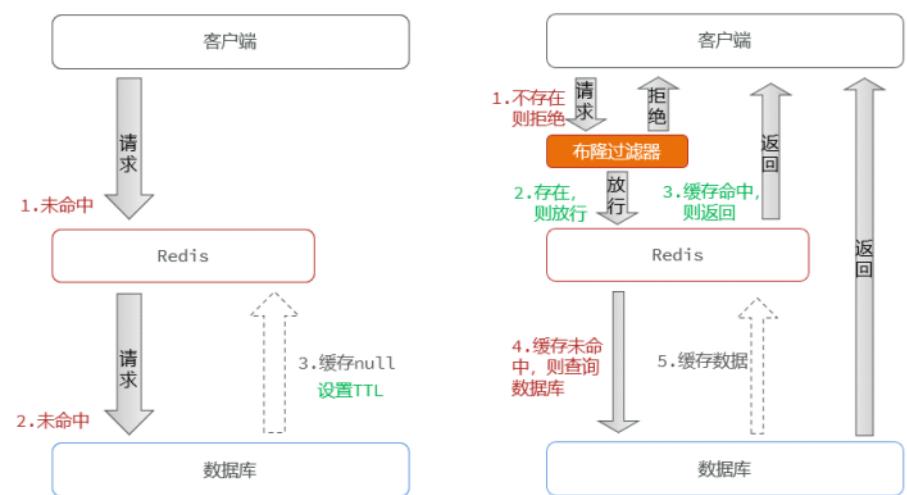
常见的解决方案有两种：

#### ● 缓存空对象

- ◆ 优点：实现简单，维护方便
- ◆ 缺点：
  - 额外的内存消耗
  - 可能造成短期的不一致

#### ● 布隆过滤器

- ◆ 优点：内存占用较少，没有多余key
- ◆ 缺点：
  - 实现复杂
  - 存在误判可能



## 缓存穿透产生的原因是什么？

- 用户请求的数据在缓存中和数据库中都不存在，不断发起这样的请求，给数据库带来巨大压力

## 缓存穿透的解决方案有哪些？

- 缓存null值
- 布隆过滤
- 增强id的复杂度，避免被猜测id规律
- 做好数据的基础格式校验
- 加强用户权限校验
- 做好热点参数的限流

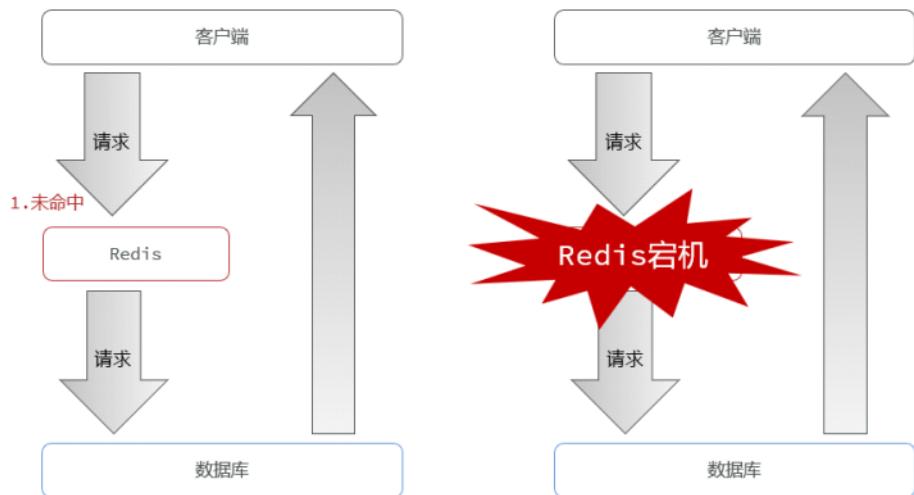
## 缓存雪崩

## 缓存雪崩

缓存雪崩是指在同一时段大量的缓存key同时失效或者Redis服务宕机，导致大量请求到达数据库，带来巨大压力。

### 解决方案：

- ◆ 给不同的Key的TTL添加随机值
- ◆ 利用Redis集群提高服务的可用性
- ◆ 给缓存业务添加降级限流策略
- ◆ 给业务添加多级缓存



## 缓存击穿

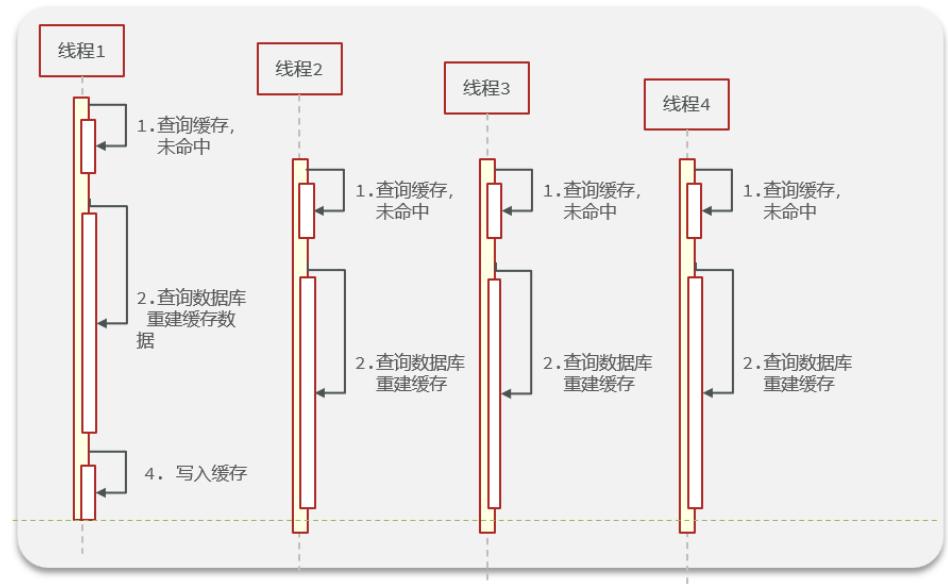
可以使用 Jmeter 进行并发测试

## 缓存击穿

缓存击穿问题也叫热点Key问题，就是一个被高并发访问并且缓存重建业务较复杂的key突然失效了，无数的请求访问会在瞬间给数据库带来巨大的冲击。

常见的解决方案有两种：

- ◆ 互斥锁
- ◆ 逻辑过期



## 缓存击穿

解决方案	优点	缺点
互斥锁	<ul style="list-style-type: none"><li>• 没有额外的内存消耗</li><li>• 保证一致性</li><li>• 实现简单</li></ul>	<ul style="list-style-type: none"><li>• 线程需要等待，性能受影响</li><li>• 可能有死锁风险</li></ul>
逻辑过期	<ul style="list-style-type: none"><li>• 线程无需等待，性能较好</li></ul>	<ul style="list-style-type: none"><li>• 不保证一致性</li><li>• 有额外内存消耗</li><li>• 实现复杂</li></ul>

## 总结

## 缓存工具封装

基于StringRedisTemplate封装一个缓存工具类，满足下列需求：

- ✓ 方法1：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置TTL过期时间
- ✓ 方法2：将任意Java对象序列化为json并存储在string类型的key中，并且可以设置逻辑过期时间，用于处理缓存击穿问题
- ✓ 方法3：根据指定的key查询缓存，并反序列化为指定类型，利用缓存空值的方式解决缓存穿透问题
- ✓ 方法4：根据指定的key查询缓存，并反序列化为指定类型，需要利用逻辑过期解决缓存击穿问题

## 工具类



### 使用办法

使用前要先将所有数据预加载到 redis 中

```
1. Shop shop = cacheClient .queryWithLogicalExpire(CACHE_SHOP_KEY, id, Shop.class, this::getById, 5L, TimeUnit.SECONDS);
2.
3. //Class<R> type, Function<ID, R> dbFallback, Long time , TimeUnit unit)
4. if(shop == null) {
5. return Result.fail("店铺不存在");
6. }
7. return Result.ok(shop);
```

# redis 秒杀技术

2022年12月24日 20:28

优惠券秒杀



- ◆ 全局唯一ID
- ◆ 实现优惠券秒杀下单
- ◆ 超卖问题
- ◆ 一人一单
- ◆ 分布式锁
- ◆ Redis优化秒杀
- ◆ Redis消息队列实现异步秒杀

## 全局唯一ID

每个店铺都可以发布优惠券：



当用户抢购时，就会生成订单并保存到tb\_voucher\_order这张表中，而订单表如果使用数据库自增ID就存在一些问题

：

- id的规律性太明显
- 受单表数据量的限制

## 全局唯一ID生成策略：

- UUID
- Redis自增
- snowflake算法
- 数据库自增

## Redis自增ID策略：

- 每天一个key，方便统计订单量
- ID构造是 时间戳 + 计数器

## 超卖问题

超卖问题是典型的多线程安全问题，针对这一问题的常见解决方案就是加锁：

### 悲观锁

认为线程安全问题一定会发生，因此在操作数据之前先获取锁，确保线程串行执行。

- 例如Synchronized、Lock都属于悲观锁

### 乐观锁

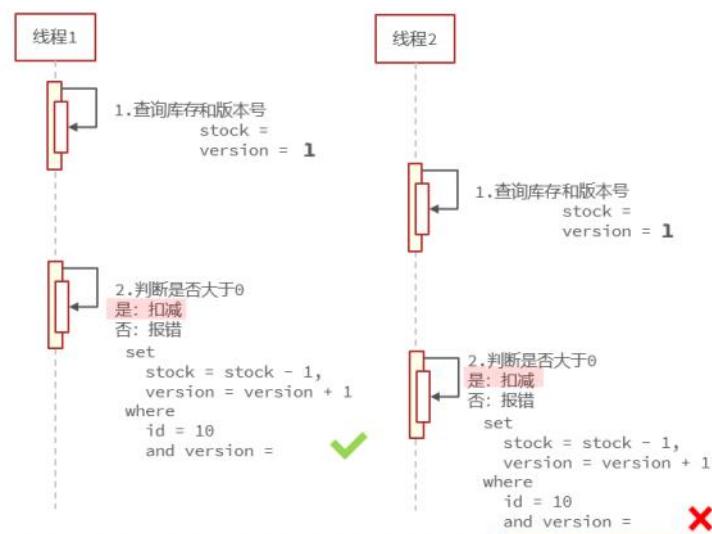
认为线程安全问题不一定发生，因此不加锁，只是在更新数据时去判断有没有其它线程对数据做了修改。

- ◆ 如果没有修改则认为是安全的，自己才更新数据。
- ◆ 如果已经被其它线程修改说明发生了安全问题，此时可以重试或异常。

乐观锁的关键是判断之前查询得到的数据是否有被修改过，常见的方法有两种：

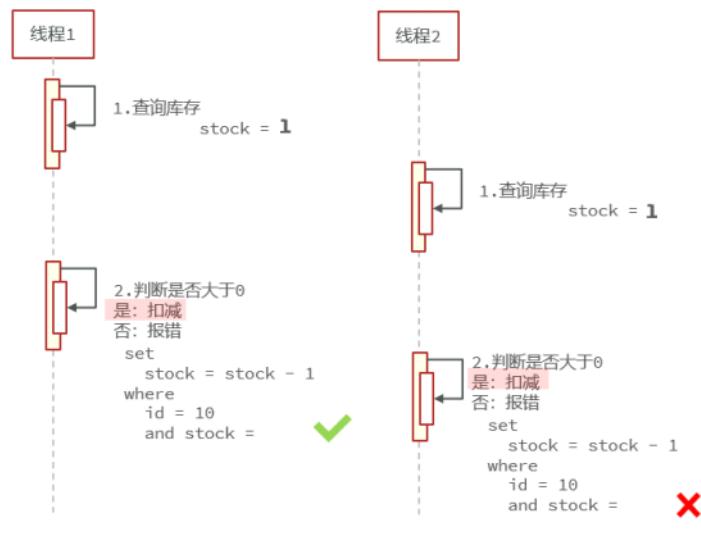
#### ◆ 版本号法

<b>id</b>	<b>stock</b>	<b>version</b>
10	1	1



## ◆ CAS法

id	stock
10	1



超卖这样的线程安全问题，解决方案有哪些？

1. 悲观锁：添加同步锁，让线程串行执行
  - 优点：简单粗暴
  - 缺点：性能一般
2. 乐观锁：不加锁，在更新时判断是否有其它线程在修改
  - 优点：性能好
  - 缺点：存在成功率低的问题

关于 事务处理之Spring代理失效

<https://blog.csdn.net/sayhitoloverOvO/article/details/120495506>

```
synchronized (userId.toString().intern()) {
 // 获取代理对象（事务）
 IVoucherOrderService proxy = (IVoucherOrderService) AopContext.currentProxy();
 return proxy.createVoucherOrder(voucherId);
}

@Transactional
public Result createVoucherOrder(Long voucherId) {
```

Redis 集群

原来的用户id锁没有用

▶ Running

- HmDianPingApplication :8081/
- HmDianPingApplication2 :8082/

```
proxy_next_upstream error timeout;
#proxy_pass http://127.0.0.1:8081;
proxy_pass http://backend;

}

upstream backend {
 server 127.0.0.1:8081 max_fails=5 fail_timeout=10s weight=1; I
 server 127.0.0.1:8082 max_fails=5 fail_timeout=10s weight=1;
}
```

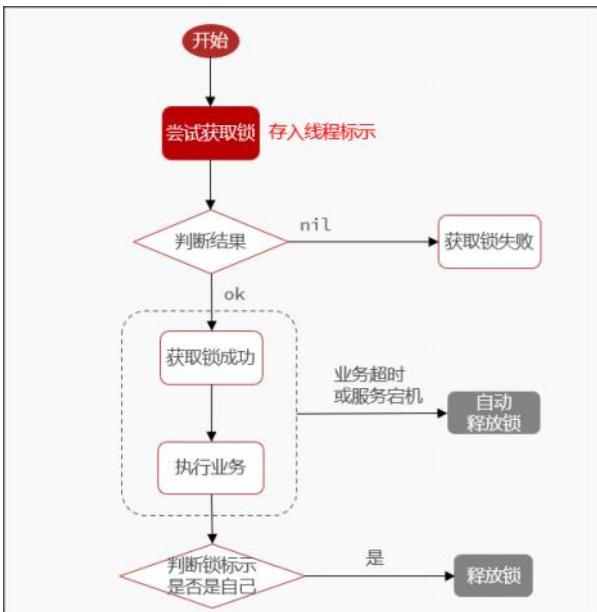
要用分布式锁

## 分布式锁的实现

分布式锁的核心是实现多进程之间互斥，而满足这一点的方式有很多，常见的有三种：

	MySQL	Redis	Zookeeper
互斥	利用mysql本身的互斥锁机制	利用setnx这样的互斥命令	利用节点的唯一性和有序性实现互斥
高可用	好	好	好
高性能	一般	好	一般
安全性	断开连接，自动释放锁	利用锁超时时间，到期释放	临时节点，断开连接自动释放

Redis 分布式锁就是多个集群调同一个 redis 服务



## 改进Redis的分布式锁

需求：修改之前的分布式锁实现，满足：

1. 在获取锁时存入线程标示（可以用UUID表示）
2. 在释放锁时先获取锁中的线程标示，判断是否与当前线程标示一致
  - ◆ 如果一致则释放锁
  - ◆ 如果不一致则不释放锁
  - ◆ 判断+操作 要原子性（lua脚本）

## 基于Redis的分布式锁

释放锁的业务流程是这样的：

1. 获取锁中的线程标示
2. 判断是否与指定的标示（当前线程标示）一致
3. 如果一致则释放锁（删除）
4. 如果不一致则什么都不做

如果用Lua脚本来表示则是这样的：

```

-- 这里的 KEYS[1] 就是锁的key, 这里的ARGV[1] 就是当前线程标示
-- 获取锁中的标示, 判断是否与当前线程标示一致
if (redis.call('GET', KEYS[1]) == ARGV[1]) then
 -- 一致, 则删除锁
 return redis.call('DEL', KEYS[1])
end
-- 不一致, 则直接返回
return 0

```

```
public void unlock() {
 //调用lua脚本
 stringRedisTemplate.execute(UNLOCK_SCRIPT,
 Collections.singletonList(KEY_PREFIX + name),
 ...args: ID_PREFIX + Thread.currentThread().getId());
}
```

## Redis 现成分布式锁 redisson

### Redisson入门

#### 1. 引入依赖:

```
<dependency>
 <groupId>org.redisson</groupId>
 <artifactId>redisson</artifactId>
 <version>3.13.6</version>
</dependency>
```

#### 2. 配置Redisson客户端:

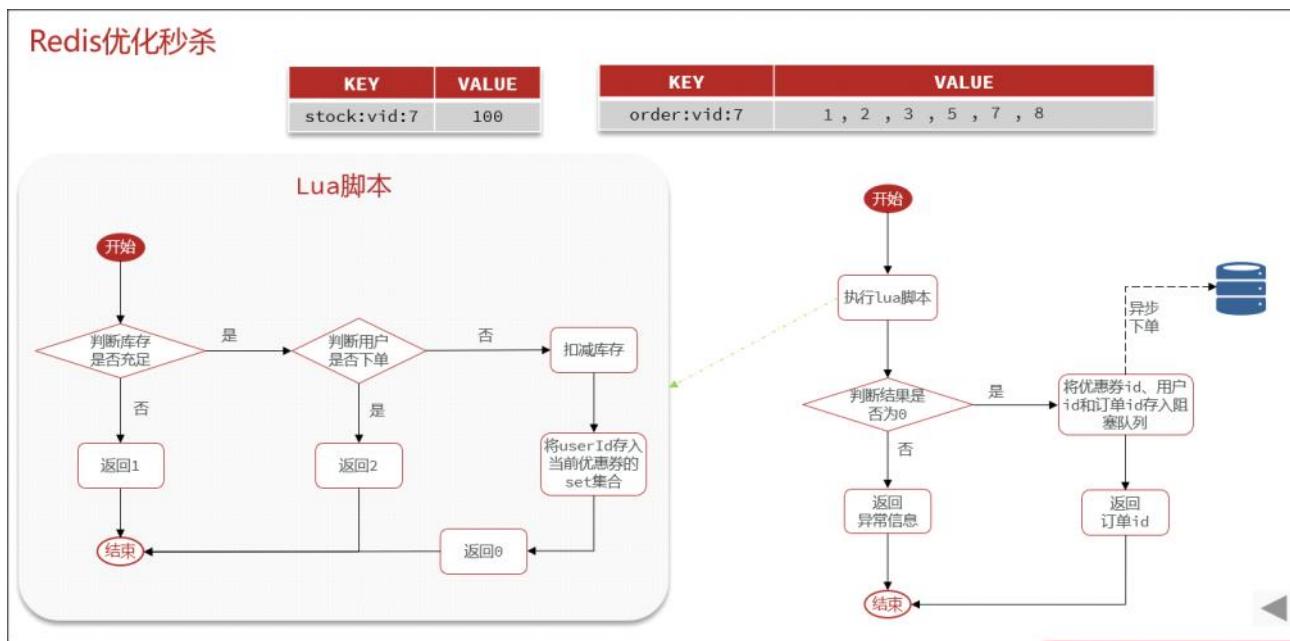
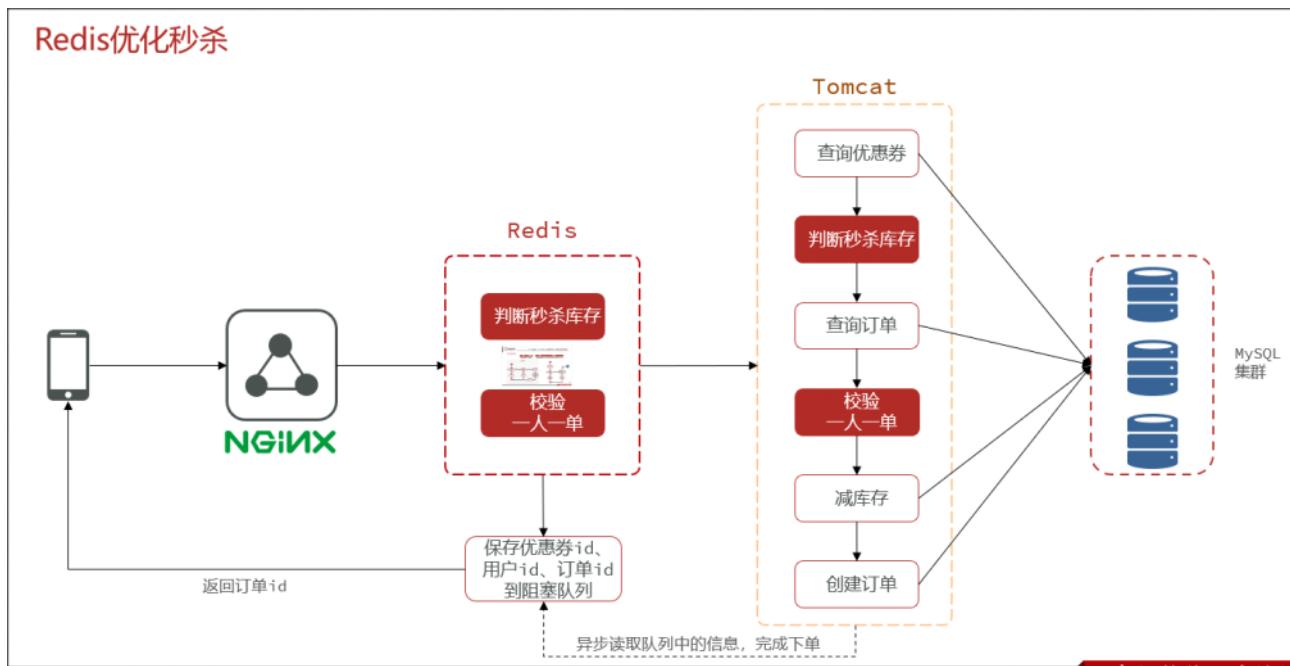
```
@Configuration
public class RedisConfig {
 @Bean
 public RedissonClient redissonClient() {
 // 配置类
 Config config = new Config();
 // 添加redis地址, 这里添加了单点的地址, 也可以使用config.useClusterServers()添加集群地址
 config.useSingleServer().setAddress("redis://192.168.150.101:6379").setPassword("123321");
 // 创建客户端
 return Redisson.create(config);
 }
}
```

#### 3. 使用Redisson的分布式锁

```
@Resource
private RedissonClient redissonClient;

@Test
void testRedisson() throws InterruptedException {
 // 获取锁(可重入), 指定锁的名称
 RLock lock = redissonClient.getLock("anyLock");
 // 尝试获取锁, 参数分别是: 获得锁的最大等待时间(期间会重试), 锁自动释放时间, 时间单位
 boolean isLock = lock.tryLock(1, 10, TimeUnit.SECONDS);
 // 判断释放成功
 if(isLock){
 try {
 System.out.println("执行业务");
 }finally {
 // 释放锁
 lock.unlock();
 }
 }
}
```

## Redis 消息队列 优化秒杀



需求:

- ① 新增秒杀优惠券的同时，将优惠券信息保存到Redis中
- ② 基于Lua脚本，判断秒杀库存、一人一单，决定用户是否抢购成功
- ③ 如果抢购成功，将优惠券id和用户id封装后存入阻塞队列
- ④ 开启线程任务，不断从阻塞队列中获取信息，实现异步下单功能

初始化（阻塞队列）（创建线程池和提交子线程）

```
//创建阻塞队列，队列中没有元素时会阻塞
private BlockingQueue<VoucherOrder> orderTasks = new ArrayBlockingQueue<>(capacity: 1024 * 1024); //队列长度不宜过长
//单线程池
private static final ExecutorService SECKILL_ORDER_EXECUTOR = Executors.newSingleThreadExecutor();

//实现类初始化的时候就执行线程池
@PostConstruct
private void init(){
 Log.info("开始执行线程池");
 SECKILL_ORDER_EXECUTOR.submit(new VoucherOrderHandler());
}
```

线程如下

```
//创建线程任务
private class VoucherOrderHandler implements Runnable{
 //将订单写入数据库
 @Override
 public void run() {
 while (true) {
 //获取队列中的头元素
 VoucherOrder voucherOrder = null;
 try {
 Log.info("开始处理订单");
 voucherOrder = orderTasks.take();
 handleVoucherOrder(voucherOrder);
 } catch (InterruptedException e) {
 Log.error("处理订单异常");
 }
 }
 }
}
```

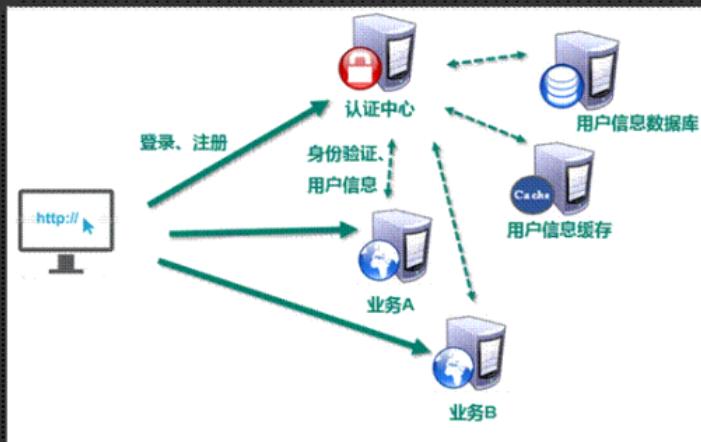
# 单点登录与JWT

2022年3月7日 17:16

一处登录，处处可用

## 1.2. SSO(single sign on)模式

分布式，SSO(single sign on)模式



优点：

用户身份信息独立管理，更好的分布式管理。

可以自己扩展安全策略

缺点：

认证服务器访问压力较大。

\* 单点登录三种常见方式：

第一种：session广播机制实现

session复制

第二种：使用cookie + redis实现

第三种：使用token实现

token是什么？

按照一定规则生成字符串，  
字符串可以包含用户信息

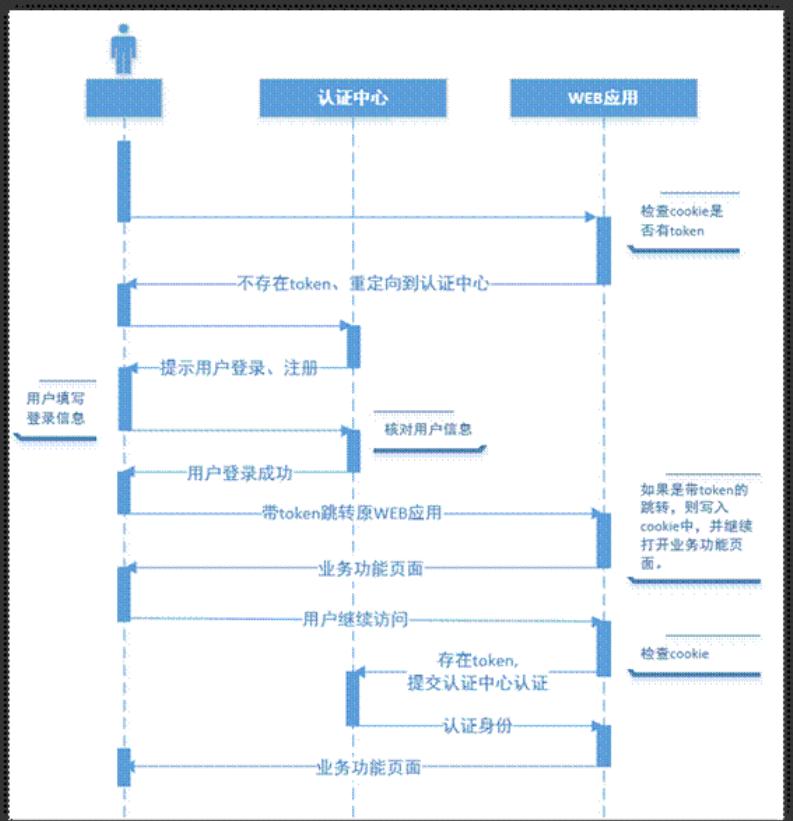
1、在项目中任何一个模块进行登录，登录之后，把数据放到两个地方  
(1) redis，在key：生成唯一随机值（ip、用户id等等），在value：用户数据  
(2) cookie：把redis里面生成key值放到cookie里面  
2、访问项目中其他模块，发送请求带着cookie进行发送，获取cookie值，拿着cookie做事情。  
(1) 把cookie获取值，到redis进行查询，根据key进行查询，如果查询数据就是登录

1、在项目某个模块进行登录，登录之后，按照规则生成字符串，把登录之后用户包含到生成字符串里面，把字符串返回  
(1) 可以把字符串通过cookie返回  
(2) 把字符串通过地址栏返回

2、再去访问项目其他模块，每次访问在地址栏带着生成字符串，在访问模块里面获取地址栏字符串，根据字符串获取用户信息。如何可以获取到，  
就是登录

### 1.3. Token模式

业务流程图{用户访问业务时，必须登录的流程}



#### 优点：

无状态： token无状态， session有状态的

基于标准化：你的API可以采用标准化的 JSON Web Token (JWT)

#### 缺点：

占用带宽

无法在服务器端销毁

注：基于微服务开发，选择token的形式相对较多，因此我使用token作为用户认证的标准

### JWT 规则

Encoded PASTE A TOKEN HERE



JWT生成字符串包含三部分

第一部分 jwt头信息

第二部分 有效载荷 包含主体信息（用户信息）

第三部分 签名哈希 防伪标志

[ ]

阿里云短信服务

# Jenkins 部署

2022年3月29日 22:44

## Java 的安装与环境变量

### ubuntu安装与卸载java

#### 安装java

- 查看java是否安装: `java -version`
- java版本:
  - 较新的版本是java11, 若要安装执行命令: `sudo apt install default-jre`
  - 我选择的是java8, 因为此版本是得到广泛支持的。安装命令: `sudo apt install openjdk-8-jdk`
- 安装好后, 执行 `java -version`, 如果输出java版本, 则说明安装成功
- 配置环境变量:
  - 通过上述方式安装的java, 目录是在: `/usr/lib/jvm/java-8-openjdk-amd64`
  - 所以环境变量配置时候添加下面内容即可:

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export JRE_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
```

<https://www.cnblogs.com/jaysonteng/p/13453244.html>