### 1.计划:

不懒，坚持，自律，不当场做决定15万，禁止想垃圾人 未cf来 历史的事

* + - 1. ing：，，，，，，，，，
      2. 最近： ，，，，，，，，，activiti，，，，复杂表，，，，，，，，，，，，
      3. 中期：代码展示js，油价，二维码，地图，，jwt接口，，限制某个IP一天内验证码发送次数,每日晚4点重置为0，台风，，汇率，，，，，，RabbitMQ
      4. 远期：，分布式，微信公众号开发，备份，高并发，，美食，，OAuth2获取接口，IM，大数据，，爬虫，硬件采集，

### 2.快捷键:

ctrl+alt+L整理代码，ctrl+f9编译整个项目，ctrl+f9+shift编译单个java文件，ctrl+tab idea页面切换，浏览器硬性重新加载：Ctrl+Shift+R，，alt+f8 idea debug查看变量，，，，，，，，，，，，，，，，，，，，

### 3.Java基础:

MTX项目搭建的配置：jdk1.8.0\_131，mysql5.5， hosts更改，Redis开启，Gitbook开启，，

由于之前电脑太卡，重装了系统，是大地系统ghost的win10版本；先制作了启动盘，然后把系统镜像放在U盘上，之后重启电脑，F12用u盘启动，选择好系统，即可安装；注意安装过程只与C盘有关，其他盘的数据不受影响，但之前安装的软件都会失效了，因为注册表没有添加到新的系统中；系统一直存在激活的问题，最后在淘宝买了注册码完事。

按步骤搭建zheng的开发环境，首先安装jdk1.8.0\_131，就是安装后配置一下环境变量；安装MySQL5.5；安装maven，配置环境变量，settings.xml里加入本地仓库地址与阿里云的镜像；安装git与TortoiseGit这两个是一起使用的，前者是命令行后者是客户端；安装idea作为开发工具；安装Redis，是一种数据库支持高并发注意把启动方式改为手动；安装Zookeeper，是用来支持分布式架构的，安装好修改zoo.cfg文件名，zkServer.cmd启动服务；安装ActiveMQ，activemq.bat启动；安装nginx；

电脑移动文件时会复制，而不是单纯的移动，编写：新建文本文档.reg 文件，内容是：

Windows Registry Editor Version 5.00

[HKEY\_CLASSES\_ROOT\\*]

"DefaultDropEffect"=dword:00000002

[HKEY\_CLASSES\_ROOT\AllFilesystemObjects]

"DefaultDropEffect"=dword:00000002

然后双击运行文件，就可以了，这个动作就是向注册表中修改信息的过程。

新建mtx工程，外壳是个空的Maven工程，上传GitHub，PRs Welcome的意思是欢迎其他人提交代码到我的项目，我来主导合并；搭好框架后从自动生成system代码开始做；项目有两个数据库，主库用来操作数据，从库用来查询数据；代码生成完毕配置springmvc准备开始搭建页面，包括spring；

Maven控制台输出中文乱码解决： Setting->maven->runner VMoptions:-Dfile.encoding=GB2312；BigDecimal在商业计算中用到，用来计算小数的，更加精确；Preconditions类似断言，减少if语句，错了直接抛出异常；

报错：Caused by: org.springframework.beans.BeanInstantiationException: Failed to instantiate [org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter]: Constructor threw exception; nested exception is java.lang.NoSuchMethodError: com.fasterxml.jackson.annotation.JsonFormat$Value.empty()Lcom/fasterxml/jackson/annotation/JsonFormat$Value;

最终发现是jar包冲突导致的，以后遇到类似问题应该早点察觉。

之后开始调试单元测试的内容；决定使用领域驱动，DO为与DB交互模型，VO是页面展示模型，DTO是保存修改模型，使用Dozer进行模型间的类型转换；lang3是Apache Commons 团队发布的工具包，要求jdk版本在1.5以上，lang是老版本的；break;结束整个循环，continue;结束此次循环；

登录时提交用户信息到服务器，服务器通过HTTP请求头返回给浏览器，浏览器保存cookie信息，之后再次请求服务器，请求头中会包含cookie信息，服务器依此判断；cookie是有时效的；

session存放在服务器上，发送到浏览器的只有sessionId，浏览器再请求时会将这个id带上，服务器根据id找到Session，session也是有时效的，可以在web.xml设置时间，JSESSIONID就是sessionId，浏览器打开新窗口会产生两个session；

由于项目调试过于麻烦，暂时取消rpc方式，并使用idea的JRebel插件热部署开发；首先hotswap 开启热更新，server项目热部署打钩，将jetty的热部署关闭；jetty:run-war先把项目打成war包再运行，启动时间太久；

由于父项目修改后不能热部署，因为子项目是以jar包的形式调用的，决定使用jrebel插件并关闭jetty的热更新，jrebel所有项目都要勾选；bootstrapValidator使用要把<div class="form-group"></div> 放在输入框外面包住，不然无法生效；在ajax的beforeSend方法上加一个load，用来防止重复提交数据；

**public static boolean** isOneEmpty(Object... os) {*//这样写非常灵活，可以写1个或多个参数*

@Data  
@ApiModel swagger相关注解  
@Accessors(chain = **true**) 链式编程开启注意会影响到构造方法   
**public class** SystemUserDto **implements** Serializable {

User u = new User().setUsername("用户名").setPassword("密码");

@Builder

User u = User.builder().username("用户名").password("密码").build();

**public class** SystemConstant {  
 **public interface** Number {*//内部接口总是静态的* **int *THOUSAND\_INT*** = 1000;

SystemConstant.Number.***THOUSAND\_INT 调用***

@JsonInclude(JsonInclude.Include.***NON\_NULL***)*//属性为NULL 转json不序列化***public class** Wrapper<T> **implements** Serializable {

防止空指针异常：尽量valueOf()替代toString()；不要List list = null;尽量使用List list = Collections.EMPTY\_LIST;

Map map =Collections.EMPTY\_MAP 但这个空集合是不可变的，空的集合不可以使用add put等方法 return null;尽量不要出现 注意空值主要出现在网络与DB的层次需要做判断

**this**.**resourceBundle** = ResourceBundle.*getBundle*(name);

这个类的作用就是读取资源属性文件（properties）

BooleanUtils.*toBoolean*(systemUserDto.getRememberMe()) true/false 字符串转boolean

序列化一个类就是要把 类信息 由内存保存到硬盘上以字符串形式存储，这个类要实现Serializable接口，

String result = RequestUtil.*getHtml*(**"https://graph.qq.com/oauth2.0/token?grant\_type=authorization\_code&client\_id="** + pageInfo.getQqAppId()+  
 **"&client\_secret="**+pageInfo.getQqClientSecret()+**"&redirect\_uri="**+URLEncoder.*encode*(pageInfo.getQqAuthPath())+**"&code="**+getPara(**"code"**)); 获取这个url的结果，注意遇到url里面的url要编码一下

重新启动程序，这个时候访问图片url后可以删除图片；但上传图片后如果不访问图片是可以删除的，但访问后

则显示文件已在java中打开，从而不能删除；原因可能是没有关闭流引起的，但具体原因没找到。

七牛云每个域名每日限总流量 10GB，每个测试域名自创建起 30 个自然日后系统会自动回收，仅供测试使用并且不支持 Https 访问

Resource interpreted as Document but transferred with MIME type image/png

由于html5的a标签的download有跨域的问题，即跨域不支持，这个是为了安全，一般js也有一些跨域的问题；

ThreadLocal，它是线程绑定的变量，A线程的ThreadLocal只能看到A线程的ThreadLocal，不能看到B线程的ThreadLocal；

Method countByExample = **mapper**.getClass().getDeclaredMethod(**"countByExample"**, example.getClass());  
Object result = countByExample.invoke(**mapper**, example); //先获取类的方法example.getClass()是传入参数类型，再执行这个方法example是传入的参数，result是结果

public class Box<T> {

private T t;

public void add(T t) {

this.t = t;

} 泛型类

}

Box<Integer> integerBox = new Box<Integer>();

integerBox.add(new Integer(10));

**public static** <T> T getBean(Class<T> clazz) { //<T> T 可以看做固定写法，表示返回该类型的 对象，这个就是泛型方法

SpringContextUtil.*getBean*(**mapper**.getClass()); //方法的调用

*// 打印的结果是：泛型参数String 是第一个泛型参数***class** AnonymousArrayListA **extends** HashMap<String,Integer> {  
  
}  
AnonymousArrayListA aa = **new** AnonymousArrayListA();  
Class typeA = (Class) ((ParameterizedType) aa.getClass().getGenericSuperclass()).getActualTypeArguments()[0];  
System.***out***.println(typeA);

***log***.debug(**"doAround>>>result={},耗时：{} ms"**, result, **endTime** - **startTime**);*//log动态写法*

*//任意数组类型转换***public** <V,E> List<V> convertList(List<E> list, Class<V> clazz){  
 List<V> voList=**new** ArrayList<>();  
 **if**(ToolUtil.*isEmpty*(list)){  
 **return** voList;  
 }  
 **for**(E e:list){  
 V v = convertModel(e,clazz);  
 voList.add(v);  
 }  
 **return** voList;  
}

*//抽象方法，为了在不同子类里实现自定义的 内容***protected abstract** <V,E> V convertAttribute(E e,V v);

Assert.*assertNotNull*(map);*//断言null则抛出异常 ，顺我者昌逆我者亡 原则*

**for** (Iterator iterator = sourceList.iterator(); iterator.hasNext();){*//迭代器* Object sourceObject = iterator.next();

**public abstract class** BaseCacheFactory **implements** ICache{  
 @Override  
 **public** <T> T get(String cacheName, Object key, ILoader iLoader) {  
 Object data = get(cacheName, key);*//抽象类里使用了 抽象方法，实际上方法会在子类的 实现方法中执行* **if** (data == **null**) {  
 data = iLoader.load();*//调用了其它接口的 抽象方法，可以在子类中重写实现* put(cacheName, key, data);  
 }  
 **return** (T) data;  
 }

CacheKit.*get*(**"1"**, **"2"**, **new** ILoader() {*//子类中重写* @Override  
 **public** Object load() {  
 **return null**;  
 }  
});

**private static** ICache *defaultCacheFactory* = **new** EhcacheFactory();*//多态 ICache是个接口*

*defaultCacheFactory*.put(cacheName, key, value); //调用接口的方法

@PostConstruct  
**private void** init() {*//构造方法执行后，用来初始化的* 也可以理解为在spring容器初始化的时候执行该方法**this**.**configMap** = **new** HashMap<String, String>();

泛型T 代表一种类型；?是通配符,泛指所有类型；? extends T 指T类型或T的子类型

对封装和子类可见（protected）,默认是相同package包下可见。

LinkedList 在插入和删除时更优于 ArrayList，而随机访问则比 ArrayList 逊色些。Set 接口是一种不包括重复元素的 Collection。涉及到多线程同步时采用 HashTable，没有涉及就采用 HashMap。vector 是线程同步的，所以它也是线程安全的，而 arraylist 是线程异步的，是不安全的，但效率高。在 Map 中插入、删除和定位元素，HashMap 是最好的选择;但如果您要按自然顺序或自定义顺序遍历键，那么 TreeMap 会更好。tree是有顺序的hash则没有。

### 4.Shiro框架:

Subject代表了当前用户;SecurityManager管理着所有Subject;Realm：域，Shiro从从Realm获取安全数据（如用户、角色、权限）可以把Realm看成DataSource;

Shiro支持粗粒度权限（如用户模块的所有权限）和细粒度权限（操作某个用户的权限，即实例级别的）

注意system:user 等于system:user:\*

/login=anon拦截路径 ?：匹配一个字符； \*：匹配零个或多个字符串； \*\*：匹配路径中的零个或多个路径

1. /bb/\*\*=filter1
2. /bb/aa=filter2
3. /\*\*=filter3

拦截顺序：如果请求的url是“/bb/aa”，因为按照声明顺序进行匹配，那么将使用filter1进行拦截。

anon表示可以不登录，authc表示需要认证才能使用更严格，user表示必须登录过rememberMe也算;

url会进入AuthenticationFilter，这个过滤器实现下面两个方法，

isAccessAllowed:即是否允许访问,false表示不允许,onAccessDenied:表示访问拒绝时是否自己处理，false表示已经处理；

我们通过继承HashedCredentialsMatcher，且使用Ehcache记录重试次数和超时时间，字典与配置的缓存也是保存在Ehcache里

登录成功把session信息保存到Redis，为了更快的信息交互

UpmsSessionDao是用来管理session的，当网站打开时就会创建一个session并保存到redis

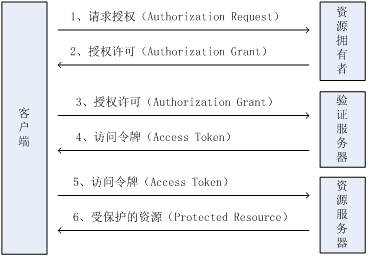
在用手机号或第三方登录时不会输入密码，则在校验完成后使用 免密登录

项目开启SSL，但是涉及证书问题，暂时就不用它了

单点登录是适用于多个子系统的，A系统登录会跳转到单点登录系统CAS，系统返回一个令牌和用户信息给A表示登录

当强制退出某个用户或同时只允许一个账号登录时，给相应session加上一个FORCE\_LOGOUT的状态即可通过过滤器判断登录与否

用RuntimeException这个可以抛出异常，并准确定位，缺点是不能catch处理这个异常



Oauth2.0 资源拥有者:新浪微博用户返回code；授权服务器：颁发授权令牌token；资源服务器：返回微博等信息；

Servlet中

request.getRequestDispatcher("new.jsp").forward(request, response);//转发到new.jsp

response.sendRedirect("new.jsp");//重定向到new.jsp

JSP中

<jsp:forward page="apage.jsp" />

<%response.sendRedirect("new.jsp");%><!--重定向到new.jsp-->

WebUtils.*toHttp*(response).sendRedirect(ssoServerUrl.append(**"/login"**).toString());

mv.setViewName(**"redirect:"**+backurl);

Forword只请求了一次由服务器转发,性能高；redirect浏览器请求了两次浏览器的地址栏URL会发生变化，可以防止f5刷新页面重复提交表单

### 5.Gitbook文档:

是用来做文档用的，先安装node.js，然后gitbook安装，最后gitbook的编辑器安装；项目放在C:\Users\Administrator.PC-201711130912\GitBook\Library\Import文件夹

gitbook serve运行项目，gitbook install安装插件，<http://admin.mtx.com:4000为项目默认的地址，注意book.json>是自己加入的为了添加相关插件用

"plugins": ["-search", 减号表示去除默认自带的插件

### 6.Jquery+Html文档:

<**script**>**var *BACK\_URL*** = **'${**param.backurl**}'**;</**script**>

这个是url后跟的？里的参数的值

line-height:140%; 行高，display:inline;展示在同一行，overflow:auto 页面元素过多出现滚动条，但大小不变

var d = $("#" + c).find('.required:visible') d=d.slice(0,2); 取一个集合中的前两个元素

**var** formData = **new FormData**(); //用ajax方式提交文件，注意两个false  
formData.append(**'file'**, *$*(**'#file'**)[0].**files**[0]);  
  
*$*.**ajax**({  
 **type**: **'post'**,  
 **url**: url,  
 **data**: formData,  
 **processData**: **false**,  
 **contentType**: **false**,  
 beforeSend: **function**() {  
 ***AttachEdit***.**index** = **layer**.load(1, {  
 });  
 },

<**c:choose**> 相当于if else 判断  
 <**c:when test="${not empty** exception**}"**>  
<**textarea cols=20** >**${**exception**}**</**textarea**>  
 </**c:when**>  
 <**c:otherwise**>  
<**textarea cols=20**>没有更详细的错误信息...</**textarea**>  
 </**c:otherwise**>  
</**c:choose**>

*$*(**"#codeImg"**).attr(**"src"**, 改变某个属性

*$*(**"#codeImg"**).bind(**"click"**, *changeCode*); 绑定某个方法

**location**.**href** =result.**message**; 当前页面到新的url

**window**.close();  
**window**.**opener**.**location**.**href** =result.**message**;

html 识别不了 ‘\n’，只要在结果所在的标签设置css样式：white-space: pre-line;

<**input onfocus="this**.blur();**" 禁止输入框聚焦**

**function** *removeField*(event) {*//通过event对象操作页面元素* **var** obj = *eventParseObject*(event); *//获取点击事件的源对象*  
 **if**(event.**target**.**nodeName** == **'SPAN'**){*//判断标签类型* obj.parent().parent().parent().remove();  
}**else** {  
 obj.parent().parent().remove();  
}  
}

**onClick="***removeField*(***event***)**"** >

垂直水平居中 vertical-align: middle;一行内可以被 <span> 划分成好几个区域，从而实现某种特定效果，<span> 本身没有任何属性；style="float:left;"> 可以使元素在同一行，不会自动换行的；overflow: hidden; 页面元素过多时隐藏不显示；cursor: pointer; 鼠标变手；!important放在样式后面表示最优先；background:url(../img/pho6.jpg) ; css里设置图片路径；margin 外边距 padding内边距；

<span style='white-space: nowrap;display: inline-block;width: 130px;overflow: hidden;text-overflow: ellipsis;line-height: 0.9;'> span控制文字不换行，且宽度固定

js页面的方法名不可为clear()，因为clear()是document的一个方法，所以不可用了；

<ul>

<li>Coffee</li>

<li>Tea</li>

<li>Milk</li>

</ul> 列表



<select>

<option value ="volvo">Volvo</option>

<option value ="saab">Saab</option>

</select>

<label for="male">Male</label>

<input type="radio" name="sex" id="male" /> 联系起来

<img src="/i/eg\_tulip.jpg" alt="上海鲜花港 - 郁金香" /> alt是图片的替代文本

<p>段落，<span>行内元素

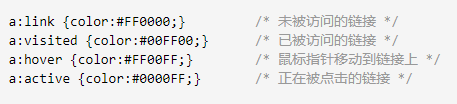
h1 {color:red; font-size:14px;}

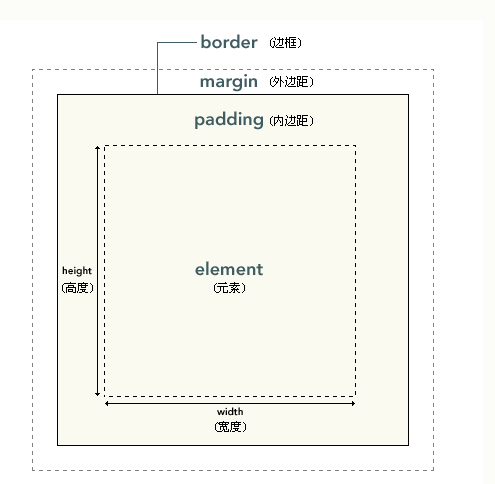
li strong {

font-style: italic;

font-weight: normal;

}





在 CSS 中，width 和 height 指的是内容区域的宽度和高度。增加内边距、边框和外边距不会影响内容区域的尺寸，但是会增加元素框的实际总尺寸。

white-space: normal;丢掉多余的空白符并忽略换行符，background-image: url(/i/eg\_bg\_04.gif);，line-height:90%行高，font-family: sans-serif;字体，font-weight:bold;字体加粗，font-size:60px;字体大小，li {list-style : url(example.gif) square inside}列表类型，vertical-align:bottom;垂直对齐方式，text-align:right;水平对齐方式，outline-color:#00ff00;轮廓颜色，padding-left: 20%;父元素宽度的20%，border-width: 20px;边框宽度，border-color: transparent;边框颜色透明，margin : 10px 0px 15px 5px;top right bottom left外边距，z-index:-1;放到后面，position:absolute;left:100px;top:150px;相对父元素定位以左上角为中心，position:fixed;相对浏览器定位，position:relative;于其正常位置进行定位，overflow:scroll;内容溢出元素框时发生的事情，overflow:inherit;规定应该从父元素继承 overflow 属性的值，float:right;向右浮动，margin-left:auto;均等地分配可用的外边距结果就是居中的元素，cursor:pointer;定义指针为手，display:block此元素将显示为块级元素，此元素前后会带有换行符，display:inline此元素会被显示为内联元素，元素前后没有换行符，display:inherit从父元素继承 display 属性的值，opacity:0.4;使图片透明，

把脚本置于 **<body>** 元素的底部，可改善显示速度，因为脚本编译会拖慢显示。单独js文件更快

document.getElementById("demo").innerHTML = "Hello JavaScript";

var carName; //值是 undefined 没有值的变量，其值是 undefined。typeof 也返回 undefined。

var length = 7; // 数字

var lastName = "Gates"; // 字符串

var cars = ["Porsche", "Volvo", "BMW"]; // 数组

var x = {firstName:"Bill", lastName:"Gates"}; // 对象

typeof "" // 返回 "string"

typeof 0 // 返回 "number"

typeof NaN; // 返回 "number"

Undefined 与 null 的值相等，但类型不相等：

typeof undefined // undefined

typeof null // object

var sln = txt.length; //字符串长度

var pos = str.indexOf("USA"); //未找到文本返回 -1

var res = str.slice(7,13); //提取部分字符串

var n = str.replace("Microsoft", "W3School"); //替换字符串内容

text3 = text1.concat(" ",text2); //连接字符串

var x = 100 / "Apple"; // x 将是 NaN（Not a Number）

isNaN(x); // 返回 true，因为 x 不是数

x.toFixed(2); // 返回 9.66 保留两位小数

var cars = new Array("Saab", "Volvo", "BMW"); //避免 new Array() 请使用 [] 取而代之

fruits.push("Lemon"); // 向 fruits 添加一个新元素 (Lemon)

fruits.splice(0, 1); // 删除 fruits 中的第一个元素

var myChildren = myGirls.concat(myBoys); // 合并数组

fruits.sort(); // 对 fruits 中的元素进行排序

points.sort(function(a, b){return a - b}); //可以自定义升序降序

cars.sort(function(a, b){return a.year - b.year}); //对象排序

numbers.forEach(myFunction); //遍历

function myFunction(value, index, array) {

txt = txt + value + "<br>";

}

switch (new Date().getDay()) {

case 6:

text = "今天是周六";

break;

case 0:

text = "今天是周日";

break;

default:

text = "期待周末~";

}

function person(firstName, lastName, age, eyeColor) {

this.firstName = firstName;

this.lastName = lastName;

this.age = age;

this.eyeColor = eyeColor;

this.changeName = function (name) {

this.lastName = name;

};

} //对象添加方法 myMother.changeName("Jobs"); 调用方法

function Person(first, last, age, eyecolor) {

this.firstName = first;

this.lastName = last;

this.age = age;

this.eyeColor = eyecolor;

}

Person.prototype.nationality = "English"; //prototype 属性允许您为对象构造器添加新属性或方法

var patt = /e/; patt.test("The best things in life are free!"); //字符串中有一个 "e"输出true

element.addEventListener("click", function(){ alert("Hello World!"); }); //添加监听方法

<title id="demo">DOM 教程</title>

var myTitle = document.getElementById("demo").innerHTML;

var myTitle = document.getElementById("demo").firstChild.nodeValue; //以上两行js等价，元素节点中不包含文本

var para = document.createElement("p");

var node = document.createTextNode("这是新文本。");

para.appendChild(node);

var element = document.getElementById("div1");

element.appendChild(para); //向div1中添加p标签

setTimeout(myFunction, 3000） //等待 3 秒执行方法

setInterval(myTimer, 1000); //每隔1秒重复执行方法

var myJSON = JSON.stringify(myObj); //对象转json文本

var obj = JSON.parse(text); //把 JSON 文本转换为 JavaScript 对象或数组，json的对象就是js的对象

转义符号\

JavaScript this 关键词指的是它所属的对象。let比var作用范围更局域一些。const是常量的意思。

$("#lastname") id；$(".intro") class；$("p")标签；$("p:hidden")所有隐藏的 <p> 元素；$("[href='#']")所有 href 属性的值等于 "#" 的元素；$("p").css("background-color","red");用于改变CSS 属性；

$("button").click(function(event){ //jQuery 事件函数

$("p").hide();

});

$("p").toggle(); //切换元素显示/隐藏

$("input").trigger("select"); //触发 input 元素的 select 事件：

$("div").animate({left:'250px'}); //<div> 元素移动到左边，直到 left 属性等于 250 像素为止

$("#test").text(); //获取纯文本

$("#test").html(); //获取html标记的文本

$("#w3s").attr("href") //获取某个属性

$("p:first").addClass("intro"); //向第一个 p 元素添加一个类

$("p").append(" <b>Hello world!</b>"); //在每个 p 元素结尾插入内容

$("p").empty(); //移除 p 元素的内容

$("p").remove(); //移除所有 <p> 元素

$("p").removeAttr("id"); //从任何 p 元素中移除 id 属性

$("p").replaceAll("<b>Hello world!</b>"); //用粗体文本替换每个段落

$(":text").val("Hello World"); //设置输入域的值

$("p").wrap("<div></div>"); //用div把p包裹起来

$("p").height(50); //设置 <p> 元素的高度

$("p").css({"background-color":"yellow","font-size":"200%"}); //设置多个 CSS 属性

$("#div1").innerWidth() //宽度包括内边距

$("li").each(function(){ //遍历

alert($(this).text())

});

$('li').has('ul').css('background-color', 'red'); //li中如果有ul就添加样式

$("div").children(".selected").css("color", "blue"); //类名为 "selected" 的所有 div 的子元素，并将其设置为蓝色

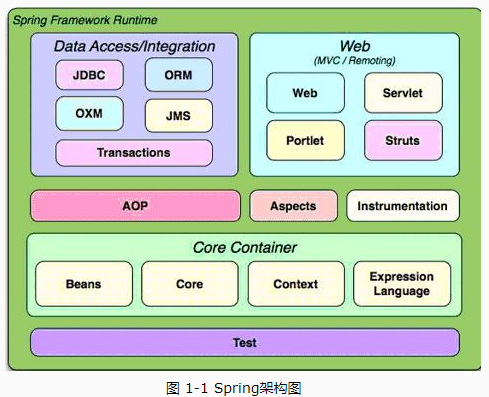
$("p").slice(0, 2).wrapInner(""); //选中所有段落，然后将所选内容缩减为只包含第一和第二个段落

$("h2").next(); //<h2> 的下一个同胞元素

$("form").serialize() //表单内容序列化成string

$("li").size() //输出被 jQuery 选择器匹配的元素的数量

### 7.Spring框架:



1. //1、读取配置文件实例化一个IoC容器
2. ApplicationContext context = **new** ClassPathXmlApplicationContext("helloworld.xml");
3. //2、从容器中获取Bean，注意此处完全“面向接口编程，而不是面向实现”
4. HelloApi helloApi = context.getBean("hello", HelloApi.**class**);
5. //3、执行业务逻辑
6. helloApi.sayHello();
7. <bean id="bean2" **class**="cn.javass.spring.chapter3.bean.BooleanTestBean">  相当于Class
8. <property name="success" value="on"/>   相当于Field
9. <property name="values">   相当于List类型
10. <list>
11. <value>1</value>
12. <value>2</value>
13. <value>3</value>
14. </list>
15. </property>
16. </bean>

Aop基于动态代理实现

<**aop:aspectj-autoproxy proxy-target-class="true"**/> 表示支持@Aspect注解 aop编程一定要注入spring容器

@Pointcut(**"@annotation(com.mtx.common.annotation.NotDisplaySql)"**)*//表示对什么样的类使用切面编程*

@Before(**"execution(\* \*..controller..\*.\*(..))"**)*//前置方法*

@After(**"execution(\* \*..controller..\*.\*(..))"**)*//不管是抛出异常或者正常退出都会执行*

@Around(**"execution(\* \*..controller..\*.\*(..))"**)*//环绕增强,相当于执行该方法，result是返回值*

@Autowired注入Spring Bean，则当前类必须也是Spring Bean才能调用它，不能用new xxx()来获得对象，这种方式获得的对象无法调用@Autowired注入的Bean

ApplicationContext是IOC的容器，ApplicationContext完全继承BeanFactory

Spring IoC容器目的就是管理Bean

lazy-init="true"延迟初始化Bean；depends-on="helloApi"明确的初始化及销毁顺序；

init-method="init"指定初始化方法；destroy-method="destroy"指定销毁方法；

工作单元内的操作是不可分割的，即要么所有操作都做，要么所有操作都不做，这就是事务，

在controller层不能操作数据，因为该层不支持事务；

* **原子性：**事务应该当作一个单独单元的操作，这意味着整个序列操作要么是成功，要么是失败的。
* **一致性：**这表示数据库的引用完整性的一致性，表中唯一的主键，转入转出金额一致等。
* **隔离性：**可能同时处理很多有相同的数据集的事务，每个事务应该与其他事务隔离，以防止数据损坏。
* **持久性：**一个事务一旦完成全部操作后，这个事务的结果必须是永久性的，不能因系统故障而从数据库中删除。

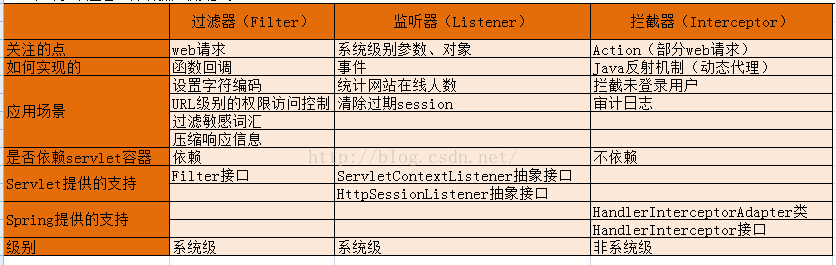
Spring使用ContextLoaderListener监听器来加载和关闭Spring Web容器，@Autowired：自动装配注入容器；

<context:component-scan>标签将自动开启“注解实现Bean依赖注入”支持，可开启@Component注解。

@Configuration注解的类本身是一个xml配置，@Bean是bean相当于@Component；这个注解也要放到扫描包去；

JSP最终还是会被编译为Servlet；DispatcherServlet是mvc的核心类；

@RequestParam绑定单个请求参数值；@PathVariable("id")绑定URI模板变量值；



拦截器(Interceptor)在一个Action执行的前后执行一段代码，类似AOP编程与，servlet无关；

监听器（Listener）类似观察者模式；

过滤器（Filter）基于servlet接口，调用目标资源url前后，让一段代码执行；

@Order作用是定义Spring IOC容器中Bean的执行顺序的优先级，而不是定义Bean的加载顺序；

@Cacheable(value=”accountCache”)，这个注释的意思是，当调用这个方法的时候，会从一个名叫 accountCache 的缓存中查询，如果没有，则执行实际的方法（即查询数据库），并将执行的结果存入缓存中，否则返回缓存中的对象。这里的缓存中的 key 就是参数 userName，value 就是 Account 对象。“accountCache”缓存是在 spring\*.xml 中定义的名称

<setting name="cacheEnabled" value="false"/><!-- 开启缓存 -->开启的是mybatis的二级缓存,测试开发中就暂时关闭了

<select id="selectArticleListPage" resultMap="resultUserArticleList" useCache="false"> 某个语句关闭缓存

Mybatis 的一级缓存是默认开启的，其存储作用域为 Session；采用 ehcache 来实现 Mybatis 的二级缓存。

mybatis返回自增长id id从model里get获取

<insert id="insertByName" useGeneratedKeys="true" keyProperty="village.id" parameterType="com.dgtec.gas.modular.meterManage.model.Village" >

lucene是专门用来做内容检索和文件检索的；导入包，三个配置文件；添加好索引后，到lukeall-3.5.0所在目录，shift打开cmd，输入java -jar luke-with-deps.jar可以查看索引,但现在双击bat文件即可；

IKAnalyzer是解词器；发现一直被锁上，是因为Lucene的多线程机制，但最终原因是解词器与lukeall的版本号太低导致的，以后也要注意这个版本的问题；

**@RestController 返回JSON 或 XML 形式数据 == @Controller +@ResponseBody**

IoC（Inverse of Control:控制反转）是一种**设计思想**，就是 **将原本在程序中手动创建对象的控制权，交由Spring框架来管理**

public class GiraffeService {

@PostConstruct

public void initPostConstruct(){

System.out.println("执行PostConstruct注解标注的方法");

}

@PreDestroy

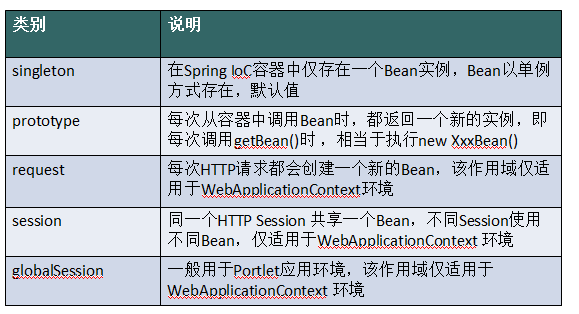
public void preDestroy(){

System.out.println("执行preDestroy注解标注的方法");

}

} //bean初始化与销毁前方法

在@Transactional注解中如果不配置rollbackFor属性,那么事物只会在遇到RuntimeException的时候才会回滚,加上rollbackFor=Exception.class,可以让事物在遇到非运行时异常时也回滚。



Bean的作用域：

@Service

@Scope("singleton")

public class ServiceImpl{

}

ZooKeeper 是一个开源的分布式协调服务，是一个典型的分布式数据一致性解决方案，RPC（Remote Procedure Call）—远程过程调用类似http接口的作用，Dubbo是一个rpc框架

quartz与spring整合，QuartzJob无法注入spring里的属性，在MyJobFactory类中进行spring的属性注入即可。

public class InitHelloWorld implements BeanPostProcessor

在bean初始化的前与后执行对应的方法

<bean id="beanTeamplate" abstract="true">

<bean id="helloIndia" class="com.tutorialspoint.HelloIndia" parent="beanTeamplate">

在bean的继承中，父 bean 自身不能被实例化，因为它是不完整的，而且它也被明确地标记为抽象的。

<bean id="exampleBean" class="examples.ExampleBean">

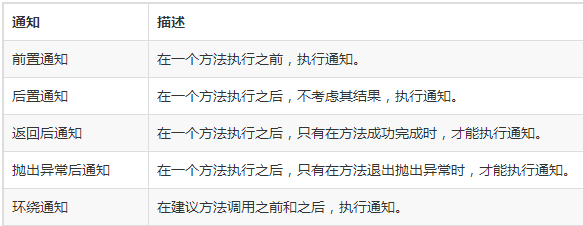
<constructor-arg type="int" value="2001"/>

<constructor-arg type="java.lang.String" value="Zara"/>

</bean> 初始化依赖注入

当 Spring遇到一个在 setter 方法中使用的 @Autowired 注释，它会在方法中视图执行 byType 自动注入，这是spring的注解。

Spring的AOP有5种通知类型：



### 8.多线程:

public class ThreadPoolDemo {

/\*\*

\* 定义静态内部线程类

\*/

public static class MyTask implements Runnable {

@Override

public void run() {

System.out.println(System.currentTimeMillis() + ":Thread name:"

+ Thread.currentThread().getName());

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

/\*\*

\* @param args

\*/

public static void main(String[] args) {

MyTask myTask = new MyTask();

ThreadFactory namedThreadFactory = new ThreadFactoryBuilder()

.setNameFormat("demo-pool-%d").build();

ExecutorService executorService = new ThreadPoolExecutor(5, 10,

0L, TimeUnit.MILLISECONDS,

new LinkedBlockingDeque<Runnable>(1024), namedThreadFactory, new ThreadPoolExecutor.AbortPolicy());

for (int i = 0; i < 10; i++) {

executorService.execute(myTask);

}

executorService.shutdown();

}

}

由于Timer类在实际使用中会存在多线程与抛出异常的问题，所以一般使用ScheduledThreadPoolExecutor作为定时器的替代,

java线程池ScheduledThreadPoolExecutor可以用来在给定延时后执行异步任务或者周期性执行任务，线程池用来处理任务的；

@Autowired  
ThreadPoolTaskExecutor **threadPoolTaskExecutor**;  
  
**public void** deleteFile(**final** String... filePaths) {  
 **this**.**threadPoolTaskExecutor**.execute(**new** DeleteFileThread(filePaths));  
}  
  
**private class** DeleteFileThread **implements** Runnable {  
 String[] **filePaths** ;  
 **private** DeleteFileThread(String[] filePaths) {  
 **super**();  
 **this**.**filePaths** =filePaths;  
 }  
 @Override  
 **public void** run() {  
 **for** (String filePath : **filePaths**) {  
 **if** (StringUtils.*isBlank*(filePath)) {  
 **continue**;  
 }  
 **uploadComponent**.delete(filePath);  
 }  
 }  
}

ThreadPoolTaskExecutor是spring的线程池用法相似

threadlocal是一个线程内部的存储类，相当于Map，可以在指定线程内存储数据，只有指定线程可以得到存储数据；

ThreadLocal和Synchronized都是为了解决多线程中相同变量的访问冲突问题，Synchronized是通过线程等待，牺牲时间来解决访问冲突；

cookie.setMaxAge(0)相当于删除cookie;

class RunnableDemo implements Runnable {

class ThreadDemo extends Thread { 创建线程两种方式，一般使用Runnable接口

BlockingQueue<String> bqueue = new ArrayBlockingQueue<String>(20);

for (int i = 0; i < 30; i++) {

//将指定元素添加到此队列中

bqueue.put("加入元素" + i);

System.out.println("向阻塞队列中添加了元素:" + i);

}

System.out.println("程序到此运行结束，即将退出----");

阻塞队列，类似数组，如果满了程序会进入等待状态。

### 9.数据库:

注意在因为版本修改DB时，要准备一个回滚的sql文件，防止不可挽回的错误发生

mysql导入数据是报错：2006 - MySQL server has gone away

找到你的mysql目录下的my.ini配置文件，加入以下代码

max\_allowed\_packet=1300M

wait\_timeout=288000

interactive\_timeout = 288000

最后管理员模式运行cmd

net stop mysql

net start mysql

update t\_ne set view\_code = '04\_06\_01' where ne\_id = '40289f6a5c18fc08015c19b0a1760001'

insert into t\_ne(view\_code,ne\_id) values ('40289f6a5c18fc08015c19b0a1760001','04\_06\_01')

select \* from t\_ne where ne\_id = '40289f6a5c18fc08015c19b0a1760001'

delete from t\_ne where view\_code = '40289f6a5c18fc08015c19b0a1760001'

select tkv.vector\_value,tn.display\_name,tkv2.vector\_value ,tkv.ne\_id

from t\_kpi\_value tkv

left join t\_ne tn on tkv.ne\_id = tn.ne\_id

left join t\_kpi\_value tkv2 on tkv.ne\_id =tkv2.ne\_id

where tkv.kpi\_code = 'SQL\_MEAN\_EVER\_TIME'

and tkv.vector\_value <> '0.00'

order by to\_number(tkv.vector\_value) desc

and length(view\_code) > 6

where rownum <=5

left join t\_service\_model\_item\_ne tsmin on tsmin.ne\_code = ( case when tne.top\_ne\_id is null then tne.ne\_id end )

select \*

from

(

select ''''||listagg(trik.kpi\_code,''',''') WITHIN GROUP(ORDER BY trik.top\_ne\_type\_code)||'''' as colum,trik.top\_ne\_type\_code as type\_code ,1 as IS\_SINGLE, 1 AS KPINETYPE

from t\_report\_inspection\_kpicfg trik

where trik.top\_ne\_type\_code = 'PF-TOP-HOST-LINUX-RHLE' and trik.leaf\_ne\_type\_code is null

group by trik.top\_ne\_type\_code

UNION

select ''''||listagg(trik.kpi\_code,''',''') WITHIN GROUP(ORDER BY trik.leaf\_ne\_type\_code)||'''' as colum,trik.leaf\_ne\_type\_code as type\_code ,0 as IS\_SINGLE, 3 AS KPINETYPE

from t\_report\_inspection\_kpicfg trik

where trik.top\_ne\_type\_code = 'PF-TOP-HOST-LINUX-RHLE' and trik.leaf\_ne\_type\_code is not null and trik.is\_single = 0

group by trik.leaf\_ne\_type\_code

) v

order by v.KPINETYPE

下面这两句效果一样

select listagg(tss.staff\_name,',') WITHIN GROUP(ORDER BY rownum) from t\_system\_staff tss

where tss.staff\_id in ('113174','113166')

select wm\_concat(tss.staff\_name) as ss from t\_system\_staff tss

where tss.staff\_id in (113174,113166)

select wm\_concat(tss.staff\_name) from t\_system\_staff tss where instr(tn.responsible,tss.staff\_id) >0 113174,113166 转 吉宗山,毕寒南

truncate table t\_subnet\_ip清空某个表的数据

select \* from t\_network\_segment t where t.network\_ip\_addr like '%10.0.0.0%' and

to\_number(substr(t.network\_ip\_addr,INSTR (t.network\_ip\_addr, '/', 1, 1) + 1, 1||2 )) =

(select max(to\_number(substr(t.network\_ip\_addr,INSTR (t.network\_ip\_addr, '/', 1, 1) + 1, 1||2 ))) s

from t\_network\_segment t

where t.network\_ip\_addr like '%10.0.0.0%' ) 查最大的那一行的数据

exp itime\_ytyh/itime\_ytyh@192.168.31.200:1521/orcl file=d:\orcl.dmp oracle导出dmp

imp itime\_ytyh/itime\_ytyh@192.168.154.128:1521/orcl file=C:\orcl.dmp full=y 导入

Mysql:

SELECT \* FROM Persons WHERE (FirstName='Thomas' OR FirstName='William') AND LastName='Carter'

UPDATE Person SET Address = 'Zhongshan 23', City = 'Nanjing' WHERE LastName = 'Wilson'

SELECT \* FROM Persons LIMIT 5

SELECT \* FROM Persons WHERE City NOT LIKE '%lon%'

SELECT \* FROM Persons WHERE City LIKE '[ALN]%' 选取居住的城市以 "A" 或 "L" 或 "N" 开头的人

SELECT \* FROM Persons WHERE LastName BETWEEN 'Adams' AND 'Carter' 包括

SELECT Persons.LastName,Orders.OrderNo INTO Persons\_Order\_Backup FROM Persons

INNER JOIN Orders ON Persons.Id\_P=Orders.Id\_P 把数据插入备份的新表Persons\_Order\_Backup

CREATE DATABASE my\_db 创建数据库

CREATE TABLE `orders` (

`id` int(11) unsigned NOT NULL AUTO\_INCREMENT COMMENT '主键',

`no` varchar(50) DEFAULT NULL,

`p\_id` int(11) DEFAULT NULL,

`time` datetime NOT NULL DEFAULT NOW(),

PRIMARY KEY (`id`)

)

DROP TABLE IF EXISTS `orders`;

ALTER TABLE Person DROP COLUMN Birthday

SELECT DATEDIFF('2008-12-30','2008-12-29') AS DiffDate

SELECT ProductName,UnitPrice\*(UnitsInStock+IFNULL(UnitsOnOrder,0)) FROM Products 如果为null则0代替

SELECT COUNT(DISTINCT column\_name) FROM table\_name

SELECT Customer,SUM(OrderPrice) FROM Orders GROUP BY Customer

SELECT ProductName, UnitPrice, FORMAT(Now(),'YYYY-MM-DD') as PerDate FROM Products

Oracle:

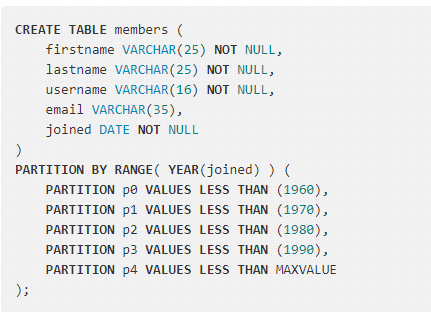
SELECT \* FROM Persons WHERE ROWNUM <= 5

CREATE SEQUENCE seq\_person MINVALUE 1 START WITH 1 INCREMENT BY 1 CACHE 10 oracle中的自增字段

INNER JOIN存在匹配时返回值

LEFT JOIN从左表 (table\_name1) 那里返回所有的行

RIGHT JOIN从右表 (table\_name2) 那里返回所有的行，也就是后面的表

分区的例子

mysql大表优化指的是单表数据过多的情况:限定数据的范围在一个月的范围内；读/写分离，主库负责写，从库负责读；垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表；水平分区类似行的拆分比如把数据不同年份拆分成不同的分区；尽量使用TINYINT、SMALLINT、MEDIUM\_INT作为整数类型而非INT；使用枚举或整数代替字符串类型；尽量使用TIMESTAMP而非DATETIME；应尽量避免在WHERE子句中对字段进行NULL值判断，否则将导致引擎放弃使用索引而进行全表扫描；不用SELECT \*；避免在WHERE子句中使用!=；对于连续数值，使用BETWEEN不用IN；使用程序的缓存；尽可能把所有列定义为 NOT NULL；避免使用子查询，可以把子查询优化为 join 操作；

索引是表的目录，主键id也是一个索引，索引会在查询中自动用到无需特殊查询语句

explain SELECT \* FROM table\_name WHERE column\_1='123' //查看查询语句使用索引的情况

ALTER TABLE 'table\_name' ADD INDEX index\_name(column\_name)； //普通索引

ALTER TABLE 'table\_name' ADD INDEX index\_name(column\_name1,column\_name2)；//组合索引把最常用作为检索或排序的列放在最左,该索引包含column\_name与column\_name1,column\_name2，而column\_name2无法使用索引

ALTER TABLE my\_table ADD FULLTEXT INDEX ft\_index(column\_name) //全文索引

SELECT \* FROM table\_name MATCH(ft\_index) AGAINST('查询字符串'); //全文索引不能使用LIKE %查询字符串%的模糊查询语法

alter TABLE users drop index name\_index ;

SHOW INDEX FROM tablename //查看索引

MyISAM适合SELECT密集型的表，而InnoDB适合INSERT和UPDATE密集型的表。

数据库会自动创建分区文件的，用户看到的只是一张普通的表，其实是对应的是多个分区，这个是对用户是屏蔽的、透明的，在使用上和使用一张表完全一样，不需要借助任何功能来实现。

Redis支持多种数据类型：

String类型的命令 set,get,decr,incr,mget ；Hash类型存储对象 hget,hset,hgetall；List类型 lpush,rpush,lpop,rpop,lrange；Set类型是不重复的list sadd,spop,smembers,sunion；Sorted Set类型是有序Set zadd,zrange,zrem,zcard；

### 10.工作流框架:

（1）先引入jar包，注入spring的bean在xml中；eclipse安装工作流的插件，把Activiti.Zip放到eclipse目录下的dropins文件夹即可；注意要查看流程图的具体属性需要打开properties视图；

（2）先完成流程定义，保证每个key只展示一个版本，画好流程图

（3）为支持组任务，要role，user与act\_的表实时同步，注意由于act\_的用户角色关系表是有外键的，故删除时先删关系表再删 用户表；

（4）开始流程实例，根据key获取是哪个订单页面，并自动生成流水编号作为名称：采购流程-101801170001；

（5）删除流程实例，不用先挂起，后删除，注意删除就是直接进入endEvent，去到历史表中；

（6）将流程变量设置在提交采购单时，报错Couldn't serialize value，是因为类没有实现

Serializable 接口；而且还不能是map类型，只能是object类型；

（7）Class<SerializeObject> beanClass=SerializeObject.class; 就是Class的对象；

（8）接下来是拾取任务，因为是组任务，谁拿到归谁；之后变成 归还任务与处理任务；

（9）在提交采购单时提交的变量是全局都可能用到的，而之后审核时提交的变量是与审核相关的变量；

（10）Activiti的关键字段

流程部署ID deploymentId

流程实例ID processInstanceId

业务ID businessKey 就是task表的ID，用于关联ACT与业务系统,启动流程时传入

任务id taskId 是当前该执行的小任务id

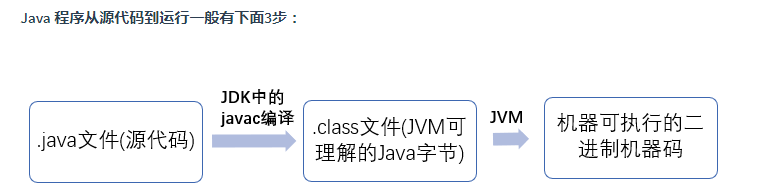
任务key taskDefinitionKey 小任务key

（11）startEvent→ userTask→ endEvent

排他网关 X，相当于if else；

并行网关 +，两个分支都会执行；

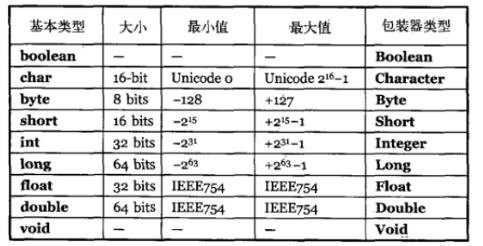
### 11.系统学习:



JDK是Java Development Kit，它是功能齐全的Java SDK。它拥有JRE所拥有的一切，还有编译器（javac）和工具（如javadoc和jdb）。它能够创建和编译程序。

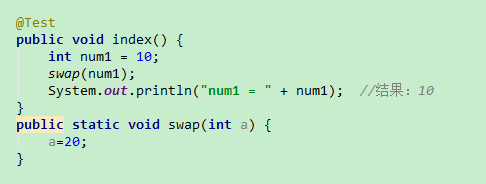
JRE 是 Java运行时环境。它是运行已编译 Java 程序所需的所有内容的集合，包括 Java虚拟机（JVM），Java类库，java命令和其他的一些基础构件。但是，它不能用于创建新程序。

* Java 不提供指针来直接访问内存，程序内存更加安全
* Java 的类是单继承的，C++ 支持多重继承；虽然 Java 的类不可以多继承，但是接口可以多继承。
* Java 有自动内存管理机制，不需要程序员手动释放无用内存

1个字节占8位

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）。

StringBuffer 对方法加了同步锁或者对调用的方法加了同步锁，所以是线程安全的。StringBuilder 并没有对方法进行加同步锁，所以是非线程安全的。



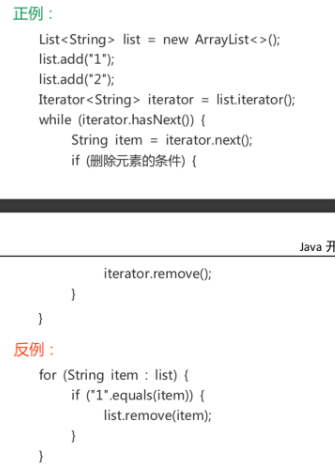
一个方法只可以改变一个 对象 中参数的状态，基本类型不可改变

当用final修饰一个类时，表明这个类不能被继承

float浮点数运算要用BigDecimal不能直接加减

总体来说使用Integer，只有局部变量使用int基本类型

List list = new ArrayList<>(Arrays.asList("a", "b", "c"))

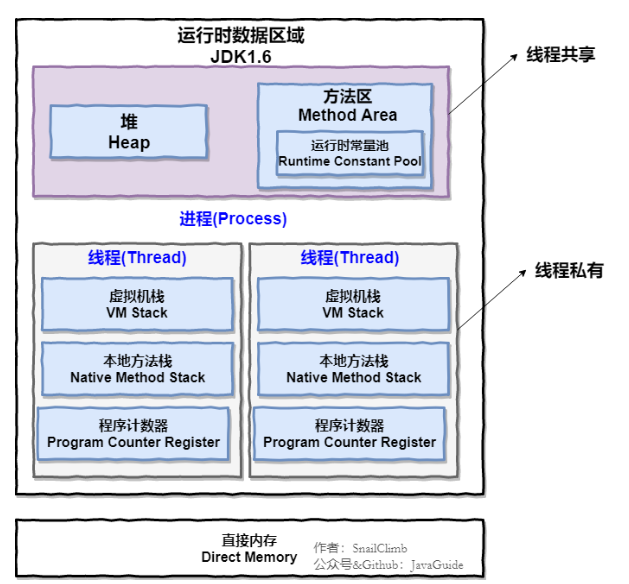
****

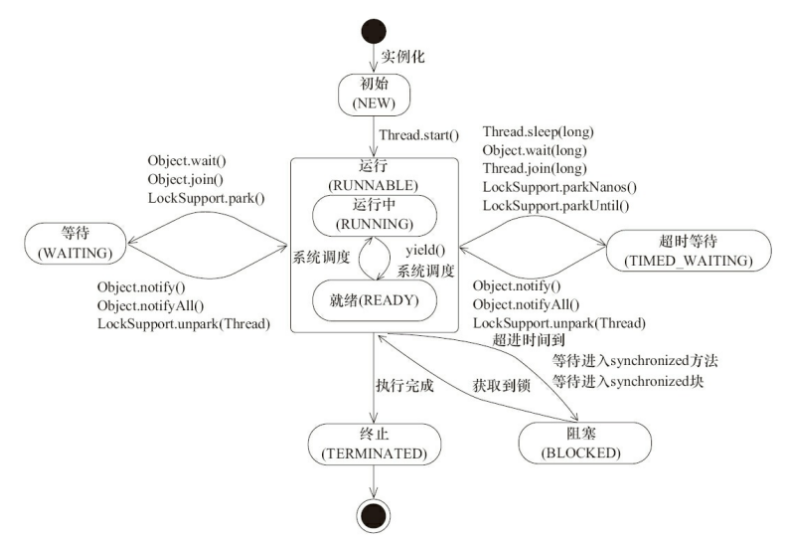
getParameter()返回的是String ，获取get/post 的参数

多线程下使用 HashMap 还是会存在其他问题比如数据丢失。并发环境下推荐使用 ConcurrentHashMap

主要根据集合的特点来选用，比如我们需要根据键值获取到元素值时就选用Map接口下的集合，需要排序时选择TreeMap,不需要排序时就选择HashMap,需要保证线程安全就选用ConcurrentHashMap.当我们只需要存放元素值时，就选择实现Collection接口的集合，需要保证元素唯一时选择实现Set接口的集合比如TreeSet或HashSet，不需要就选择实现List接口的比如ArrayList或LinkedList，然后再根据实现这些接口的集合的特点来选用。

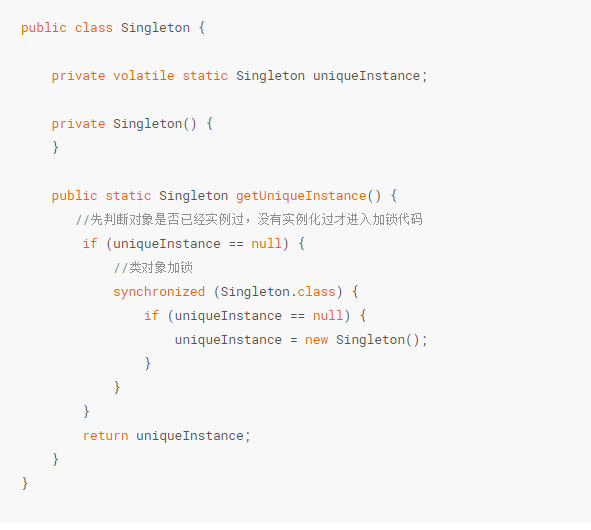
****

****

****

线程 A 持有资源 2，线程 B 持有资源 1，他们同时都想申请对方的资源，所以这两个线程就会互相等待而进入死锁状态。

**sleep 方法没有释放锁，而 wait 方法释放了锁** 。

****

**volatile性能肯定比synchronized关键字要好，但**第一选择应该是 synchronized 关键字。但是**volatile关键字只能用于变量而synchronized关键字可以修饰方法以及代码块**

* **ConcurrentHashMap:** 线程安全的HashMap
* **CopyOnWriteArrayList:** 线程安全的List，在读多写少的场合性能非常好，远远好于Vector.
* **ConcurrentLinkedQueue:** 高效的并发队列，使用链表实现。可以看做一个线程安全的 LinkedList，这是一个非阻塞队列。
* **BlockingQueue:** 这是一个接口，JDK内部通过链表、数组等方式实现了这个接口。表示阻塞队列，非常适合用于作为数据共享的通道。
* **ConcurrentSkipListMap:** 跳表的实现。这是一个Map，使用跳表的数据结构进行快速查找。

**多读场景使用乐观锁，不断的重试直到正确才完成操作；多写的场景下用悲观锁synchronized；**

Java 8使我们能够通过使用 default 关键字向接口添加非抽象方法实现。

Collections.sort(names, new Comparator<String>() {

@Override

public int compare(String a, String b) {

return b.compareTo(a);

}

});

Collections.sort(names, (String a, String b) -> {

return b.compareTo(a);

}); //java8，lambda表达式简化代码

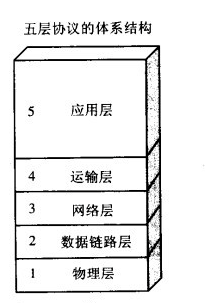
// 测试 Filter(过滤)

stringList

.stream()

.filter((s) -> s.startsWith("a"))

.forEach(System.out::println);//aaa2 aaa1 map类不支持这样的流操作 java8



运输层的TCP 三次握手和四次挥手,握手要求建立连接双方都是有去有回所以最少三次；

Shell脚本语言是利用计算机命令执行的小程序.sh结尾

**SQL 对大小写不敏感**

var patt1=new RegExp("e");

document.write(patt1.test("The best things in life are free")); //字符串中搜索字符 "e" ，输出true

var patt1=new RegExp("e");

document.write(patt1.exec("The best things in life are free")); //输出e，如果没有发现匹配，则返回 null

boolean isMatch = Pattern.matches(pattern, content);

Pattern r = Pattern.compile(pattern);

Matcher m = r.matcher(content);

if (m.find( )) {m.group(0);}

\* 零次或多次，+一次或多次，?零次或一次，{n}匹配确定的 n 次，{n,}至少匹配n 次，{n,m}最少匹配 n 次且最多匹配 m 次，[]范围内的任意字符，[^]匹配非范围内任意字符，|或，\d匹配一个数字字符，\w匹配字母、数字、下划线，^[0-9]\*$最前与最后字符分别是开头与结尾，.所有字符

**@Test**方法可以作为一个测试案例；**@Before**在 test 方法前运行；在**@After之后运行；@Ignore**忽略有关不需要执行的测试；

@Test(timeout=1000)最大时间测试

字符串的拼接，append()速度最快，concat()次之，+最慢。原因请看下面分解:

final 修饰数据表示常量，修饰方法类表示不可以被继承。

Class myObjectClass = MyObject.class; class对象

String className = myObjectClass.getName(); 类名

int modifiers = myObjectClass.getModifiers(); 类的修饰符private等

Package package = myObjectClass.getPackage(); 包信息

Class superclass = aClass.getSuperclass(); 父类

Class[] interfaces = aClass.getInterfaces(); 实现的接口

Constructor[] constructors = aClass.getConstructors(); 构造器

Method[] method = aClass.getMethods(); 方法

Field[] method = aClass.getFields(); 变量

Annotation[] annotations = aClass.getAnnotations(); 类注解

Annotation[] annotations = method.getDeclaredAnnotations();方法注解

Annotation[] annotations = field.getDeclaredAnnotations();变量注解

Class[] parameterTypes = constructor.getParameterTypes(); 获取构造器的参数

Field field = aClass.getField("someField"); 获取公有变量

Field field = aClass. getDeclaredField("someField"); field.setAccessible(true); 获取私有变量

String fieldName = field.getName(); 获取变量名称

Object fieldType = field.getType(); 获取变量类型

Object value = field.get(objectInstance); 获取变量值

field.set(objetInstance, value); 设置变量值

Method method = aClass.getMethod("doSomething"); 获取公有方法

Method method = aClass. getDeclaredMethod("doSomething"); method.setAccessible(true); 获取私有方法

Class[] parameterTypes = method.getParameterTypes(); 获取方法参数

Class returnType = method.getReturnType(); 获取返回值类型

Object returnValue = method.invoke(null, "parameter-value1"); 执行方法

Type genericFieldType = field.getGenericType(); Type[] fieldArgTypes = aType.getActualTypeArguments();获取泛型相关

int[] intArray = (int[]) Array.newInstance(int.class, 3); 创建对象

动态代理 利用Java反射机制你可以在运行期动态的创建接口的实现

public interface InvocationHandler{

Object invoke(Object proxy, Method method, Object[] args)

throws Throwable;

}

public class MyInvocationHandler implements InvocationHandler{

public Object invoke(Object proxy, Method method, Object[] args)

throws Throwable {

//do something "dynamic"

}

}

InvocationHandler handler = new MyInvocationHandler();

MyInterface proxy = (MyInterface) Proxy.newProxyInstance(

MyInterface.class.getClassLoader(),

new Class[] { MyInterface.class },

handler);

Java servlet 是运行在 Web 或应用服务器上的程序，作为在来自 Web 浏览器或其他 HTTP 客户机的请求和在 HTTP 服务器上的数据库或应用程序的中间层。

@Target({ElementType.***TYPE***})*//应用于类、接口（包括注解类型）、枚举*@Retention(RetentionPolicy.***RUNTIME***)*//注解的生命周期*@Documented *//使用了@Inherited注解的注解，所标记的类的子类也会拥有这个注解***public** @**interface** BaseService {  
}

Pattern p = Pattern.*compile*(regex);  
Matcher m = p.matcher(content);  
**return** m.matches();

### 12.设计模式:

单例模式，节省内存开销

public class Singleton {

private static Singleton singleton;

private Singleton(){}

public static synchronized Singleton getInstance(){

if(singleton==null){

singleton = new Singleton();

}

return singleton;

}

}

工厂方法模式，降耦合，为了生成复杂对象

interface IProduct {

public void productMethod();

}

class Product implements IProduct {

public void productMethod() {

System.out.println("产品");

}

}

interface IFactory {

public IProduct createProduct();

}

class Factory implements IFactory {

public IProduct createProduct() {

return new Product();

}

}

public class Client {

public static void main(String[] args) {

IFactory factory = new Factory();

IProduct prodect = factory.createProduct();

prodect.productMethod();

}

}

抽象工厂，多个复杂对象

interface IProduct1 {

public void show();

}

interface IProduct2 {

public void show();

}

class Product1 implements IProduct1 {

public void show() {

System.out.println("这是1型产品");

}

}

class Product2 implements IProduct2 {

public void show() {

System.out.println("这是2型产品");

}

}

interface IFactory {

public IProduct1 createProduct1();

public IProduct2 createProduct2();

}

class Factory implements IFactory{

public IProduct1 createProduct1() {

return new Product1();

}

public IProduct2 createProduct2() {

return new Product2();

}

}

public class Client {

public static void main(String[] args){

IFactory factory = new Factory();

factory.createProduct1().show();

factory.createProduct2().show();

}

}

建造者模式，更复杂的工厂

class Product {

private String name;

private String type;

public void showProduct(){

System.out.println("名称："+name);

System.out.println("型号："+type);

}

public void setName(String name) {

this.name = name;

}

public void setType(String type) {

this.type = type;

}

}

abstract class Builder {

public abstract void setPart(String arg1, String arg2);

public abstract Product getProduct();

}

class ConcreteBuilder extends Builder {

private Product product = new Product();

public Product getProduct() {

return product;

}

public void setPart(String arg1, String arg2) {

product.setName(arg1);

product.setType(arg2);

}

}

public class Director {

private Builder builder = new ConcreteBuilder();

public Product getAProduct(){

builder.setPart("宝马汽车","X7");

return builder.getProduct();

}

public Product getBProduct(){

builder.setPart("奥迪汽车","Q5");

return builder.getProduct();

}

}

public class Client {

public static void main(String[] args){

Director director = new Director();

Product product1 = director.getAProduct();

product1.showProduct();

Product product2 = director.getBProduct();

product2.showProduct();

}

}

原型模式，通过copy创建新对象，浅拷贝

class Prototype implements Cloneable {

public Prototype clone(){

Prototype prototype = null;

try{

prototype = (Prototype)super.clone();

}catch(CloneNotSupportedException e){

e.printStackTrace();

}

return prototype;

}

}

class ConcretePrototype extends Prototype{

public void show(){

System.out.println("原型模式实现类");

}

}

public class Client {

public static void main(String[] args){

ConcretePrototype cp = new ConcretePrototype();

for(int i=0; i< 10; i++){

ConcretePrototype clonecp = (ConcretePrototype)cp.clone();

clonecp.show();

}

}

}

模板方法模式，一个抽象类，一个继承类即可

中介者模式，将对象间的关系全部交给中介类去处理

观察者模式

abstract class Subject {

private Vector obs = new Vector();

public void addObserver(Observer obs){

this.obs.add(obs);

}

public void delObserver(Observer obs){

this.obs.remove(obs);

}

protected void notifyObserver(){

for(Observer o: obs){

o.update();

}

}

public abstract void doSomething();

}

class ConcreteSubject extends Subject {

public void doSomething(){

System.out.println("被观察者事件反生");

this.notifyObserver();

}

}

interface Observer {

public void update();

}

class ConcreteObserver1 implements Observer {

public void update() {

System.out.println("观察者1收到信息，并进行处理。");

}

}

class ConcreteObserver2 implements Observer {

public void update() {

System.out.println("观察者2收到信息，并进行处理。");

}

}

public class Client {

public static void main(String[] args){

Subject sub = new ConcreteSubject();

sub.addObserver(new ConcreteObserver1()); //添加观察者1

sub.addObserver(new ConcreteObserver2()); //添加观察者2

sub.doSomething();

}

}

访问者模式，通过访问类去操作方法

命令模式，把操作放到命令类中，用到时调用，就像命令行一样

责任链模式，简化if else代码，先判断，再依次交由对应处理器处理，也有些类似链式编程

策略模式，为不同策略执行不同的方法，常用在md5或64加密中

迭代器模式，把类改成迭代器，遍历统一执行某操作

解释器模式，不同类执行不同方法，例子是加减法

备忘录模式，类似事务，有回滚

代理模式，真正的业务功能还是由目标类来实现，代理类只是用于扩展、增强目标类的行为，类似AOP

**静态代理 就是在之前与之后加入了方法 ,动态代理 使用jdk自带方法 实现implements InvocationHandler接口 重写invoke方法**

Subject subject =(Subject)Proxy.*newProxyInstance*(RealSubject.**class**.getClassLoader(),realSubject.getClass().getInterfaces(),simpleHandle);

装饰模式，用于给对象添加新功能，又不改变原有类

享元模式，类似线程池

外观模式，就是把内部系统隔绝，只通过外观类访问

状态模式，根据数据不同状态，做出对应的操作

### 13.算法:

当数据处理量大，常规处理耗时多，需要用到算法以缩短时间与节省系统占用。

### 14.-: