达内 Java 笔记整理

包含内容:

Unix	**first-2**
Core Java，	**3-16**
OOAD 思想	**17**
Oracle	**18-26**
JDBC 笔记	**27-33**
Hibernate	**34-44**
HTML&javascript	**45-49**
Servlet	**50-55**
Jsp	**56-62**
Struts 1.2	**63-76**
Struts 2	**77-80**
Ajax	**81-94**
Spring，	**95-111**
SSH 和 Ajax 的整合	**112-116**
Ejb，	**117-144**
PL/SQL	**145-161**
java 和模式	**162-end**

Linux/Unix 笔记

inode ： 存储编号（地址）
ls -k：查看磁盘分区
ls -li：显示当前文件的 inode 号。
目录的大小跟文件的大小有关，跟目录里的文件（目录）数量无关。
一行多个命令的话，中间用分号分开。如：pwd;cal;date
last | grep pts/13 表示查看 pts/13 登陆服务器的记录。
find . -mtime -10 -print
-10：表示 10 天以内的，
+10：表示 10 天以前的，
.：表示当前路径
-mtime：最后一次修改时间。
-print:表示输出到显示器（可有可没有）。
-user 0:表示 UID 是 0。
size+400:表示大于 400*0.5K ，-400 表示小于 400*0.5K
-atime:表示最后一次访问时间。
grep:在文件里查找符合要求的那一行。通常用在管道（|）后面，表示对 | 前面的输出做查找。
如：cat /etc/passwd | gred liu | sort
sort:表示排序。
进程是作业，作业是进程。
前台就是终端，后台就是服务器。
当杀掉父进程，前台子进程会消失,，后台作业不依赖于任何终端，会继续运行
LINUX 常用命令（基础）
1. man 对你熟悉或不熟悉的命令提供帮助解释
eg:man ls 就可以查看 ls 相关的用法
注：按 q 键或者 ctrl+c 退出，在 linux 下可以使用 ctrl+c 终止当前程序运行。
2. ls 查看目录或者文件的属*，列举出任一目录下面的文件
eg: ls /usr/man ls -l
a.d 表示目录(directory)，如果是一个"-"表示是文件，如果是 l 则表示是一个连接文件(link)
b.表示文件或者目录许可权限.分别用可读(r)，可写(w)，可运行(x)。
3. cp 拷贝文件
eg: cp filename1 filename2 //把 filename1 拷贝成 filename2
cp 1.c netseek/2.c //将 1.c 拷到 netseek 目录下命名为 2.c
4. rm 删除文件和目录
eg: rm 1.c //将 1.c 这个文件删除
5. mv 移走目录或者改文件名
eg: mv filename1 filename2 //将 filename1 改名为 filename2
mv qib.tgz ../qib.tgz //移到上一级目录
6. cd 改变当前目录 pwd 查看当前所在目录完整路径
eg: pwd //查看当前所在目录路径
cd netseek //进入 netseek 这个目录
cd //退出当前目录
7. cat，more 命令
将某个文件的内容显示出来。两个命令所不同的是:cat 把文件内容一直打印出来，而 more 则分屏显示
eg; cat>1.c //就可以把代码粘帖到 1.c 文件里，按 ctrl+d 保存代码。
cat 1.c 或 more 1.c //都可以查看里面的内容。
gcc -o 1 1.c //将 1.c 编译成.exe 文件，我们可以用此命编译出代码。
8.chmod 命令 权限修改 用法：chmod 一位 8 进制数 filename。
eg: chmod u+x filenmame //只想给自己运行，别人只能读
//u 表示文件主人， g 表示文件文件所在组。 o 表示其他人 ;r 表可读，w 表可写，x 表可以运行
chmod g+x filename //同组的人来执行
9. clear，date 命令
clear:清屏，相当与 DOS 下的 cls;date:显示当前时间。
10. mount 加载一个硬件设备
用法:mount［参数］要加载的设备 载入点
eg: mount /dev/cdrom
cd /mnt/cdrom //进入光盘目录
11. su 在不退出登陆的情况下，切换到另外一个人的身份
用法: su -l 用户名(如果用户名缺省，则切换到 root 状态)
eg:su -l netseek (切换到 netseek 这个用户，将提示输入密码)
12.whoami，whereis，which，id
//whoami:确认自己身份
//whereis:查询命令所在目录以及帮助文档所在目录
//which:查询该命令所在目录(类似 whereis)
//id:打印出自己的 UID 以及 GID。(UID:用户身份唯一标识。GID:用户组身份唯一标识。每一个用户只能有一个唯一的 UID 和 GID)
eg: whoami //显示你自己登陆的用户名
whereis bin 显示 bin 所在的目录，将显示为：/usr/local/bin
which bin
13. grep，find
grep:文本内容搜索;find:文件或者目录名以及权限属主等匹配搜索
eg: grep success * /*查找当前目录下面所有文件里面含有 success 字符的文件
14. kill 可以杀死某个正在进行或者已经是 dest 状态的进程
eg; ps ax
15. passwd 可以设置口令
16. history 用户用过的命令
eg: history //可以显示用户过去使用的命令
17. !! 执行最近一次的命令
18. mkdir 命令
eg: mkdir netseek //创建 netseek 这个目录
19. tar 解压命令
eg: tar -zxvf nmap-3.45.tgz //将这个解压到 nmap-3.45 这个目录里
20. finger 可以让使用者查询一些其他使用者的资料
eg: finger //查看所用用户的使用资料
finger root //查看 root 的资料
ftp 上传下载 ftp 192.168.1.100
用户：xiangf Pwd xiangf
Put mput 上传多个 Get mget 下载多个
在 linux 下 Jdk 的安装
1. 先从网上下载 jdk(jdk-1_5_0_02-linux-i586.rpm)
进入安装目录
#cd /home
#cp jdk-1_5_0_02-linux-i586.rpm /usr/local
#cd /usr/local
给所有用户添加可执行的权限
#chmod +x jdk-1_5_0_02-linux-i586.rpm.bin
#./jdk-1_5_0_02-linux-i586.rpm.bin
此时会生成文件 jdk-1_5_0_02-linux-i586.rpm,同样给所有用户添加可执行的权限
#chmod +x jdk-1_5_0_02-linux-i586.rpm
安装程序
#rpm -ivh jdk-1_5_0_02-linux-i586.rpm
出现安装协议等，按接受即可
2. 设置环境变量。
#vi /etc/profile
在最后面加入
#set java environment
JAVA_HOME=/usr/java/jdk-1_5_0_02
CLASSPATH=.:$JAVA_HOME/lib.tools.jar
PATH=$JAVA_HOME/bin:$PATH
export JAVA_HOME CLASSPATH PATH
保存退出。
3.检查 JDK 是否安装成功。
#java -version
如果看到 JVM 版本及相关信息，即安装成功!
bash-profile 是配置文件，配置 java-home，path，classpath 等。空格。bash-profile 从新启动
Vim 。bash-profile 编辑
Javac 把.java 编译成。Class java 运行。Class 文件
java –d.按包的名字自动生成相对应的

批注 [U1]:

Core Java 笔记

人－－－>源文件－－－>编译器－－－>程序－－－>CPU

编译器：

1，编译执行:源文件－－－>可执行代码。如：C / C＋＋语言。
　　执行效率高。可移植性差。－－－－>开发成本的增加。

2，解释执行：源文件－－－>体系结构中立的中间代码(.class)－－－ >解释器 －－－>机器指令。 如 ： java 语言
　　执行效率低。 可移植性好。－－－－>　对硬件要求高。

JAVA 语言：

（源文件）－－－>（编译器 javac.exe）－－－>中间码－－－>（虚拟机 java.exe）－－－>机器指令－－－>CPU
　　（编译）　（解释）

.java ----> .class ---> 可执行文件

PATH：指定可执行程序（命令）在哪个目录。不同目录用（:）分开。－－－>SHELL

JAVA_HOME：指定 JDK 的安装目录。给其它应用程序看的。

CLASSPATH：指定（jar 文件）中间字节码文件在什么地方。由多个目录组成的集合。－－－>
　　让虚拟机可以找到中间字节码文件。就是可以找到.class 文件

服务器上 csh:.cshrc

bsh:.profile

客户端上。bash:.bash_profile

```
 1 # .bash_profile
 3 # Get the aliases and functions
 4 if [ -f ~/.bashrc ]; then
 5  . ~/.bashrc
 6 fi  //if 的结尾。
 8 # User specific environment and startup programs  //#代表注释。
 9 JAVA_HOME=/opt/jdk1.5.0_15        JDK 安装路径--- JDK = JRE {JVM(硬件)+编译器（ 软件）} +编译器工具+类库
10 PATH=$JAVA_HOME/bin:$PATH:$HOME/bin          / / 系统定义的＄PATH   启动命令
11 CLASSPATH=.:java/jar/servlet-api.jar:/java/jar/jsp-api.jar         //类路径
12
13 export PATH CLASSPATH JAVA_HOME    / / 使三个变量变成全局变量。。
```

Source .bash_profile：只能经过这条命令之后，才会让修改的变量生效。（csh）

. .bash_profile。 只能经过这条命令之后，才会让修改的变量生效。　（bash）

java -version:查看虚拟机的版本号。

2.，编写第一个 JAVA 程序。

1），以.java 结尾的文件。

2），所有内容，都要写在一个类中（类名要和文件名想同，包括大小写在内）

```
pbulic class HelloWorld{......}
```

3），main 函数是程序执行的入口，程序从 main 函数的第一条语句开始执行，执行到 main 函数结束为止。

```
public static void main(String argvs[]){
        }
```

4），输出语句：System.out.println(　　);

5），每条语句以（;）结尾

先指定到源程序存放的目录，然后进行以下步骤运行。

编译：javac 命令。如：javac -d . HelloWorld.java －－>生成 HelloWorld.class 类文件

启动虚拟机(java)，运行类文件。如：java com.work.core.HelloWorld

后面的.class 省略，无需写成 java com.work.core.HelloWorld.class

批注 [U2]:、一个“.java”源文件中是否可以包括多个类（不是内部类）？有什么限制？

可以。必须只有一个类名与文件名相同。

包（package）：

1，package com.kettas.corejava; / / 包声明。

2，必须保证类文件一定放在相应的目录结构中，HelloWorld.class 放在 com/kettas/corejava 目录中。

3，运行时，一定在顶层包的同级目录下运行 java 命令，

例如:com（shell 界面里运行）java com.kettas.corejava.HelloWorld

（如果顶级包所在的目录已经被配置到 CLASSPATH 中的话可以在任意目录中启动 java 命令）

1，类文件太多，不好管理。

2，解决重名。

javac -d . HelloWorld.java　　（不仅能够编译类文件，同时还能创建包结构）

运行命令 java xxxx 类的名字--- 启动虚拟机

（一）语言：适合于 internet

1，跨平台的　（属于解释执行的语言）

2，更容易开发出安全的程序：

1）垃圾回收器，（帮助程序员管理内存）

2）取消了指针，对数组越界进行检查

3）安全检查机制，（对生成的字节码检测,如：包结构检测）

Hash 算法，，只要加密的源是一样的。经过 Hash 运算之后，Hash 值都一样。

加密的源如果不一样了，经过 Hash 运算之后，值都不一样。

（二）变量：
如：　学生，姓名，性别，age
　　帐户，ID，password，balance，username
内存：没记忆，容量小，成本高，存储变量数据，最小逻辑单位：**byte（字节）=8bit（位）**
外存（硬盘）：有记忆，容量大，成本低，存储文件数据
1，变量声明：给变量起名，是给变量选择一种数据类型。如：int age;
不同的变量，
　　1）参与的运算是不同的，
　　2）存储的值是不同的，
　　3）需要的存储空间的大小也不同，

java 数据类型：
简单数据类型（原始数据类型）
　数字：　整数：byte(1 字节)－short(2 字节) －**int(在内存中占 4 个字节)**－long（8 个字节）
　小数：float(4 个字节，单精度浮点型) － double（8 个字节，双精度浮点型）
　字符：char(一个字符＝2 个字节):只能表示一个字。　如：char c='中'; c 存的是‘中‘这个字的编码。
布尔：boolean（true,false）,不能用 0 和非 0 表示。
String（字符串）复杂数据类型（类）
　String 类提供了数值不可改变的字符串 String s=new String(“abc”);创建了两个对象 1，在**字符串池**中创建一个对象（此对象是不能重复的）2，**new 出一个对象**。Java 运行环境有一个字符串池，由 String 类维护。执行语句 String s="abc"时，首先查看字符串池中是否存在字符串"abc"，如果存在则直接将"abc"赋给 s，如果不存在则先在字符串池中新建一个字符串"abc"，然后再将其赋给 s。执行语句 String s=new String("abc")时，不管字符串池中是否存在字符串"abc"，直接新建一个字符串"abc"（注意：新建的字符串"abc"不是在字符串池中），然后将其付给 s。
2，初始化（局部变量而言，必须初始化才能使用）　如：age=10;
3，通过变量的名字对变量进行操作，如：age=other;
　　赋值时，左＝右
　　1）数据性质是否一样。
　　2）所占用的内存空间上考虑　（左＞右）不用类型转换。（左＜右）要类型强制转换，
　　如：int age=1;
　　long l=10;
　　age=(int)l;
符号位：0 代表正数，1 代表负数。
BigDecimal 比 double 更精确，一般用于处理高精度运算。
&和&&的区别。
&是位运算符，表示按位与运算，&&是逻辑运算符，表示逻辑与（and）
Java 中的标识符的要求：
1，只能是字母，数字，__，$。　2，数字不能作为标识符的开头。
3，大小写敏感。　　4，没有长度限制。如：ArrayOutOfBoudleException
5，标识符不能是关键字。

一般来说，类的名字以大写字母开头。
方法和变量的名字以小写字母开头。
标识符的名字应当具有含义，如　　age　，name

表达式：1，由运算符和变量组成。2，都会有返回值。
简单表达式：a/b;　　复合表达式："a/b="+(a/b);

作业，打印润年，
1，能被 4 整除，但不能被 100 整除
2，能被 400 整除。

自增（++），自减（－－）运算符。
前＋＋：++a;先对 a+1，再返回 a+1 之后的值。
后＋＋：a++;先返回 a 的值，然后对 a 进行+1。
前＋＋，后＋＋共同的特点：a 本身的值都会+1;
区别在于表达式的返回值是不一样；用返回值来参与计算。

？　：　　－－＞（boolean express）？A　：　B;如：char ch=(5>2)?'Y':'N';
？前面的布尔表达式如果为真就执行 A，否则执行 B。
　　（‘：‘左右两侧的返回类型要一致，且与 ch 的返回类型也一样）

批注 [U3]: 25、short s1 = 1; s1 = s1 + 1;有什么错? short s1 = 1; s1 += 1;有什么错?
short s1 = 1; s1 = s1 + 1;（s1+1 运算结果是int 型，需要强制转换类型）
short s1 = 1; s1 += 1;（可以正确编译）

批注 [U4]: 4、String 和 StringBuffer 的区别
JAVA 平台提供了两个类：String 和StringBuffer，它们可以储存和操作字符串，即包含多个字符的字符数据。这个 String 类提供了数值不可改变的字符串。而这个 StringBuffer 类提供的字符串进行修改。当你知道字符数据要改变的时候你就可以使用StringBuffer 。典型地，你可以使用StringBuffers 来动态构造字符数据。

java 打包（压缩包　.zip　.rar　.tar　.gz　.jar）。

```
root/
  yesq/
  |-com/
    |-work/
      |-core/
        |-VarTest.class
        |-RunNian.class
          .  .  .  .  .  .
```

压缩命令： jar -cvf abc.jar dir

解压命令：jar -xvf abc.jar

abc.jar 为生成压缩包的名字

dir 为被打包的文件目录下的东西

c：是做压缩。v：为压缩过程的信息。f：指定压缩文件的名字。x：是做解压

打包一般从顶层包开始打。如：[java@localhost yesq]$ **jar -cvf first.jar com**

//当前目录（yesq）下的 com 文件夹开始打

1，获得别人提供的 jar 文件

2，将 jar 文件的名字以及他的绝对路径配置到 CLASSPATH 中

3，使用 import 语句，将其他包的内容导入到本文件中，如：引入包 com.kettas.common 包中的 SystemIn 类

import com.kettas.common.SystemIn;

－－－－－－　java 中的流程控制　－－－－－

1，顺序

2，条件（代码是否执行，依赖于一个特定的条件）

```
  if(boolean express){
     XXXXXX;   //如果布尔表达式为真就执行 XXXXXX。
     XXXXXX;
 }else{
     XXXXXX;   //如果布尔表达式为假就执行 XXXXXX。
     XXXXXX;
 }
```

3，循环（相同的代码，被重复的执行多次）

a,初始化循环条件。b，确定循环条件。c，确定循环变量在循环体中的变化。

```
(1)       a;
while(boolean express){
          XXXXXXXXX;    //如果布尔表达式为真就执行 XXXXXX
          XXXXXXXXX;
                c;
     }
 (2)   for(a;b;c){

}
 (3) do{
          XXXXXXXXX;      //循环体至少执行一次；
          XXXXXXXXX;
          }while(boolean express);

while(XXX){
     XXXXXX;
     XXXXXX;
     bread;           //或者 continue；
     XXXXXX;
}
```

break ： 用在循环或分支里，用于退出整个循环或分支

continue ：用在循环里，用于结束本次循环。接着开始下一次循环

4，分支

```
     switch(var){
          case 1 : XXXXXX;
                break;
          case 2 : xxxxxx;
                break;
                .....
                default : xxxxxxx;    //   default 后面可以不用 break；
          }
```

函数（方法）：是由一些相关代码组成的集合，目的是为了解决一个特定的问题，或者实现某个特定的功能。

函数（方法）一定有一个自己的名字，通过函数的名字执行函数中的代码。

2，在 java 中如何定义函数：

a，声明：public static void printMenu(int a,double b){.....}

b,编写函数体：

```
如：  public static void printMenu(int a,double b){
          XXXXXXXXXX;
          XXXXXXXXXX;
     }
```

函数（方法）里面不允许再有其它的函数（方法），函数（方法）的位置必须是并列的。

3，调用函数：

通过函数名，如 pirntMenu(a,b);

```
public static void main(String argv[]){
     XXXXXXXXXXX;
     printMenu(a,b);
     XXXXXXXX;
}
```

调用者－原数据－＞函数

return 作用：1，结束本方法。2，返回结果。

一个函数返回的值只能有一个。

批注 [U5]: var：是一个整数表达式。因此传递给 switch 和 case 语句的参数应该是 int、 short、 char 或者 byte。long,string 都不能作用于 swtich。

4，值传递。传递的是实参的值，
被调函数使用的数都是**实参的拷贝**。
是否改变实参，要看被调函数的设计。

数组：一维数组－－>：三种声明方式
一，（1）首先要声明。如：**int[] array**;或 int array[];
（2）申请内存空间如：array=new int [2];
array 代表数组 array[]的首地址（引用：引用就是 C＋＋里的指针。）；当参数传递时，只要用数组的首地址就可以。
1，数组中的元素类型要一致。
2，数组长度不能缺省，不能改变，但可以删了重建。
3，内存空间连续，有索引下标值（从 0 开始，最大 length-1）
优缺点：查询快，增删慢（链表，查询慢，增删快）

只要有 new 在，就要为变量分配内存。 array.length //表示数组 array 的长度。
array 存的是 array 数组首变量的地址。
二，数组的显示初始化：int[] array={1,2,3,4,5,6};
三，int[] array=new int[]{1,2,3,4,5,6,7};

数组的拷贝：
```
public static int[] expand(int[] a){
      int[] b=new int[a.length*2];
/* for(int i=0;i<a.length;i++){
            b[i]=a[i];
      }
这段代码等于下面那条代码
*/
      // 从 a 数组下标为 0 的位置开始拷贝，到 b 数组下标为 0 的位置开始粘贴。
      // a.length 为粘贴的个数。（一般为 a 数组的长度）。
      System.arraycopy(a,0,b,0,a.length);
      return b;  //返回扩展后的数组 b。b 是代表数组 b 的首地址。
}
```

二维数组（数组的数组）－－>：三种声明方式

二维数组的第二个 [] 在声明时可以留空，如：int[][] array=new int[4][];
//留空说明该二维数组的列是以行的数组下标为数组名，再成为新的数组。

一，声明：int[][] array=new int[3][4]; //声明一个三行四列的整型二维数组。
二，int[][] array={{1,2,3},{2,3,4},{2,5,7},{2,3,6}};
三，int[][] array=new int[][]{{1,2,3},{2,3,4},{2,5,7},{2,3,6}};
1，找对象（客观存在的事物）；
2，利用对象的功能。就是让对象干点什么；
3，对象的属性可以是其他对象。
4，对象还可以通过调用其他对象的函数产生联系。
5，简单的对象构成复杂的系统。

有什么：属性--------》 实例变量
做什么：方法--------》函数
对象的属性可以是其他对象
对象还可以通过调用其他对象的函数产生联系
简单的对象构成复杂的系统

思想：解决问题的过程，

总纲（优势）
1，符合人的思维过程。
2，编写出来的代码比较合理。如：可扩展性，可维护性，可读性等等。（存在就是合理），（贴近生活）
3，弱耦合性，
4，可扩展性
5，可复用性。 不要重复制造轮子。
6，更尽所能，更施其职。

＝＝＝＝＝＝＝＝**类：**＝＝＝＝＝＝＝＝＝＝＝＝＝＝
面向过程：是代码的容器。
面向对象：对象所共有的功能和属性进行抽像，成为了类。客观事物在人脑中的主观反映。在程序里类是创建对象的模板。

java 中的对象：对现实对象的抽象（获得有用的，舍弃没用的）。
存储空间中的一块数据区域，用于存数据。如：人（nama sex age）
属性：实例变量（定义在类以内，方法之外）
1.默认的初值
2.实例变量的使用范围至少是类以内

批注 [U6]: 垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？
对于GC 来说，当程序员创建对象时，GC 就开始监控这个对象的地址、大小以及使用情况。通常，GC 采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。当GC 确定一些对象为"不可达"时，GC 就有责任回收这些内存空间。可以。程序员可以手动执行 System.gc()，通知 GC 运行，但是 Java 语言规范并不保证 GC 一定会执行。

3.实例变量调用必须有对象，实例变量和局部变量重名，局部优先。

例：

```
public class Test{
      public static void main(String[] args){

      //新建对象 stu；stu 存的是新建对象 stu 的地址。stu 的专业术语：引用 / 对象变量 / 引用变量 / 实例变量
            Student stu=new Student();
            stu.name="tom";        //给对象 stu 的 name 属性赋值。
      }}
class Student{              //用类声明的数据（str）为引用类型，如：Student str;

      //实例变量：定义在方法之外，类以内的，没有 static 修饰的变量
      //实例变量会有一个默认的初始值。初始值与变量定义时的类型有关。
      //实例变量（成员变量）－－－＞属性。可通过新建对象的引用来访问类的实例变量。如，stu.name;
      String name;
      int age;
      String sex;
}
```

实例变量和局部变量的区别：

1，位置：**局部变量定义在方法里面。实例变量定义在类以内方法之外**。

2，使用的范围：局部变量只能在定义他的方法里面使用，直接调用变量名就可以了。

实例变量至少可以在定义他的整个类内使用，使用时必须用对象去调用。只有跟对象一起实例变量才有意义。

3，局部变量使用之前必须初始化。**实例变量不需要赋初值，系统会给默认的初值。**

4，局部变量在同一方法里不能重名。局部变量和实例变量可以重名，在方法里采用就近原则。

＝＝方法：＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝＝

包括：

方法:

做什么：方法的定义

修饰符　返回类型　方法名（参数列表） 异常

怎么做：方法的实现 {******}

修饰符（如：public） 返回类型（如：int） 方法名/函数名 （参数表－－形参）

如：

```
      public void eat(String fish){                //eat(),吃的功能。
            //怎么做.
      }
```

使用实例方法时也需要用对象去调用。如：stu.eat("fish");

方法重载（overloading）：编译时多态。

在一个类的内部，方法名相同形参数不同的方法，对返回类型不要求，这种现象称之为重载；

编译器会自动选择使用的方法。体现了一个编译时多态。

好处：对使用者屏蔽因为参数不同造成方法间的差异。

找方法时如果没有合适的，采取自动向上扩展原则。

调用时形参之间的区别一定要明确。

1. 形参的个数不同
2. 形参的类型不同
3. 形参的顺序不同
4. 形参的名字相同

方法覆盖（override）：运行时多态。

1，发生在父类和子类之间

2，方法名相同，参数相同，返回类型相同

3，子类方法的访问权限不能更严格，只能等于或更加宽松。

构造方法：

1，没有返回类型，方法名必须和类同名。

2，构造方法不能手动调用，它只用在创建对象在时候，只出现在 new 之后。

只要构造方法被调用运行，就一定有对象产生。

3，在一个对象的生命周期里，构造方法只能被调用一次。

4，类写好后一定有构造方法，

如果程序没有显示的提供构造方法，JVM 会提供一个默认的构造方法，public classname(){}

如果程序显示提供，系统不再提供默认的

5，同一个类的多个构造方法一定重载。

6，创建对象的同时构造方法的代码块里还可以写需要运行的代码，还可以给属性（实例变量）赋值，

引用类型的属性赋值就是用 new 创建对象，然后调用构造方法。如：Student stu=new Student();

用 new 创建对象时 JVM 做的三件事：

如：Student stu=new Student();

1，申请空间;（把值放到空间里，再把空间的地址给引用变量。）－－－－创建父类对象

2，初始化实例变量：没显示声明值的话就用默认值。

3，执行构造方法，

因为实例变量在创建对象的过程中被初始化，所以使用实例变量前必须创建对象（要用对象去调用），否则实例变量根本不存在

批注 [U7]: 方法的重写 Overriding 和重载 Overloading 是Java 多态性的不同表现。重写Overriding 是父类与子类之间多态性的一种表现，重载Overloading 是一个类中多态性的一种表现.我们说该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被“屏蔽”了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载 (Overloading)。Overloaded 的方法是可以改变返回值的类型。

＝＝＝＝＝**关键字：this**＝＝＝＝＝＝＝
1，在普通方法里，指代当前对象引用（哪个对象调用该方法，this 就指向该对象）
2，this 不能出现在静态方法里。
3，在构造方法里，this 用来指代本类的其他构造方法。在本类构造方法之间互相调用。如：this(形参);
　　使用时放在构造方法里的第一行。
4，不能写成死循环（不能递归调用）
＝＝＝＝＝**关键字：super**＝＝＝＝＝＝＝
（和 this 的用法类似）
1,调用父类的构造方法，在子类调用父类的构造方法必须出现在第一句，构造方法第一句可能出现的三种情况（调用父类的构造方法看子类构造方法的第一句）①super（）；②super（参数）注意传递的参数一定是实体，有值的。③this（），先在本类的构造方法间调用，再看调用那个构造方法的第一句是什么
2，super 访问父类的变量和方法，及 super 代表父类的对象，super.name;super.setName();
＝＝＝＝＝**参数传递**＝＝＝＝＝＝＝＝
1，参数相当于局部变量
2，参数传递相当于赋值语句
3，基本类型数据传递的是本身的值，引用类型数据传递的是引用 xx(Animal animal)（地址，对象变量本身的值）

面向对象的三大特性：

一，封装（Encapsulation）：一个系统以合理的粒度出现。
定义：封装就是将客户端不应看到的信息包裹起来。使内部执行对外部来看不一种不透明的、是一个黑箱，客户端不需要内部资源就能达到他的目的。(封装是把**过程和数据包围起来**，对数据的访问只能通过已定义的界面。面向对象计算始于这个基本概
念，即现实世界可以被描绘成一系列完全自治、封装的对象，这些对象通过一个受保护的接口访问其他对象。)
1．事物的内部实现细节隐藏起来
2．对外提供一致的公共的**接口**－－间接访问隐藏数据　　3．可维护性

访问控制修饰符：public（公开的），protected（受保护的，1，本包内可见；2，其他包的子类可见）
　　default（默认，本包内可见），private（私有的，类内部可见）

访问控制修饰符	（可以范围）	（可修饰）下面的类（指外部类）
private	本类	方法，属性
default	本包	类，方法，属性
protected	本包+子类	方法，属性
public	到处可见	类，方法，属性

1，属性：隐藏所有属性，用 private。隐藏后属性只能在类以内访问 。程序可以根据需要提供 get 和 set
2，方法（函数）：该公开的公开，该私有的就私有。(视情况而定)
3，公开方法的功能，隐藏方法的实现细节。

二，继承（inheritance）：抽象出不变性。
从一般到特殊的关系，可以设计成继承
　特点：共性写在父类里面，特性写在子类
所有类的总父类是 Object (Object 是类的祖先)
父类里写的是共性，子类里写的是特性。
父类中用 private 修饰的属性和方法不能被子类继承；
但是父类里的属性子类都有，如果子类有特殊属性的话，就要在子类里定义
且在创建该子类对象的时候初始化属性（包括父类所有属性和该子类所有属性）；

什么样的功能和属性可以继承到子类?
对于父类里的属性和方法，子类有权访问的，我们称为可以继承；

用 new 创建子类对象，JVM 执行的过程：Dog d=new Dog(); 为 d 申请空间。 class Dog extends Animal{}
（1）申请空间；(把值放到空间里，再把空间的地址给引用变量。)
（2）看本类构造方法的第一句
（3）默认的创建父类对象：
　　执行顺序：子类（2）－－－＞　父类（2－－＞3－－＞4－－＞5）－－－＞　子类（4－－＞5），
　　新建一个对象空间只申请一次（该空间存放所有父类和子类)。)
（4）初始化本类的实例变量（该类的所有属性）；
（5）执行本类的构造方法，(构造方法只会在创建一个对象的时候被执行一次)

(创建是先执行父类的构造方法，在执行子类的构造方法，访问时先访问子类自己的特殊属性和方法再访问父类的属性和方法)
用 super 调用父类被子类覆盖的普通方法和遮盖的属性，
指代的是在创建子类对象过程中，由 JVM 自动创建的那个父类，如：super.方法名（） / 属性
用 super 调用父类的构造方法；必须出现在子类构造方法的第一句。如：super(形参);
1，在子类的构造方法里如果没有指明调用哪一个父类的构造方法（就是子类中没有 super（形参）语句;），
　　JVM 会默认调用父类的无参构造方法，跟本类构造方法的形参没有关系。
2，显示的写 super，JVM 会根据参数调用相应的父类构造方法。
3，有 this(形参)，在本类的构造方法之间调用，看被调用的那个构造方法的第一行。

三，**多态（polymorphism）**：多态只有方法多态，没有属性多态。
用父类类型屏蔽子类之间的差异

所有的子类对象都可以当父类对象来用，一个父类型的引用可能指向的是一个子类对象，
如：把狗（对象）看作动物（类型）。Animal a=new Dog();　　编译看前面，运行看后面。
　　（编译时类型）　（运行时类型）
1，运行时对象不会改变（对象是客观存在的），如：狗这个对象的属性和方法是不会改变的。
2，对一个引用，只能调用编译时类型里的已知方法。
　　如：编译器知道动物里已有的属性和方法，但不知道狗的属性和方法。
3，运行时会根据运行时类型自动寻找覆盖过的方法运行。

引用　instanceof 类名　　/ / 结果为 boolean 值，
引用所指向的对象和类名类型是否一致（对象是否属于类名类型）
类型的转换：转换编译时类型
　Sub　su= (Sub) s;
　子类型的引用向父类型转换时，不需要强转
　父类型的引用向子类型转换时，需要强转
　　Animal a=new Cat();
　　Dog d=(Dog)a;　　/ /　类型转换异常
　　引用　instanceof　类名 -----> boolean
　　引用所指向的对象和类名所代表的类型是否一致
　　a instanceof Animal ----->　true　　a instanceof Cat---------->　true　　a instanceof Dog---------->false
　　Employee e=new Manager();
　　e instanceof Employee ------>true
　　e instanceof Manager------> true

属性没有多态，属性只看编译时类型

编写程序的顺序：
class 类名 {
　　private 属性（有什么）
　　无参的构造方法（public 类名（）{}）
　　有参的构造方法（作用：给属性赋值）
　　set 和 get（设置和获得属性）
　　业务方法（做什么）
}

一，**修饰符：static**
　　static 变量：如：static int index=2;
　　类的所有对象共同拥有的一个属性；可以用类名直接访问，又称为类变量，
　　类的所有对象共同拥有的一个变量；类第一次被加载时会初始化静态变量
　　（也就是会先执行 static 修饰的变量）；
　　跟类创建了多少对象无关；任何对象对静态变量做的修改，其他对象看到的是修改后的值。
　　可以用作计数器，记录类创建对象的个数，　**static 变量在类加载的时候只会被初始化一次**，
　　static 方法：如：public static void teach(){}
　　可以用类名直接去调用，不需要对象所以不能直接访问（在没有对象的情况下）实例变量，
　　在静态方法里不能出现 this 和 super，类的所有对象共同拥有的一个方法；跟类创建了多少对象无关。
　　在继承里，父类的静态方法只能被子类的静态方法覆盖，且覆盖以后没有多态
　　（访问的是父类的静态方法）；
　　static 初始化块：如：class Teacher(){
　　　　static int index=4;
　　　　static{　　//static 初始化块
　　　　.....
　　　　}
　　　　}
　　静态初始华块：用 static 修饰类里面的一个独立的代码块，**类第一次被 JVM 加载的时候执行，只被执行一次**。
　　类加载：JVM 在第一次使用一个类时，会到 classpath 所指定的路径去找这个类所对应的字节码文件，
　　并读进 JVM 保存起来，这个过程称之为类加载，**一个线程一个 jvm**。

二，**final　（最后的，最终的）**final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承
　　final 类：如：final class Animal{}
　　表示该类不能被继承，意味着不能改变里面的代码；
　　对虚拟机的正常运行有重要作用的类通常用 final 修饰，如：String,System,Math ...等类
　　final 方法：如：public final void sleep(){}　　**该方法不能被覆盖（修改），但能被子类访问。**

　　final 变量：如：final (static) int index=4;
　　该变量是常量能被继承（访问）；
　　final 修饰的变量就是常量，通常和 static 一起连用，来声明常量；
　　final 修饰引用类型数据，指的是引用（地址）不能变，但引用里的数据不受限制。
　　final 修饰的变量，只要在初始化的过程中就可以赋值。

批注 [U8]:（java 最主要的特性，实现的原理是因为有 jvm 虚拟机，在编译时不用考虑真实的对象，运行时才考虑，象封装继承在 c++中都可以模拟实现，而多态无法实现）

批注 [U9]: 静态变量和实例变量的区别？
static i = 10; //常量
class A a; a.i =10;//可变

实例变量：声明的同时或构造方法里赋值；
静态变量：声明的同时或在静态代码块里赋值；

三，abstract

abstract 类：如：abstract class Animal{}
抽象类，不能创建对象(如一些父类)，但是可以声明一个抽象类型的引用
（可以声明父类类型子类对象，编译时利用多态调用抽象方法）。
含有抽象方法的类一定是抽象类，但抽象类并不一定要有抽象方法；
抽象类一般是用来被继承的；子类继承自抽象类，就要实现里面的抽象方法，
如果不想让子类也是抽象类的话，必须实现父类里面所有的抽象方法。
抽象类有构造方法，有父类，也遵循单继承的规律。

abstract 方法：如：public abstract void sleep();
抽象方法，只有方法名的定义，没有实现体（只定义了能做什么，没定义怎么做），不能被调用，
用于被子类的方法覆盖或重新实现。只能放在抽象类中。
好处：允许方法的定义和实现分开。
public protected default private static final abstract
可以： public static
private static
public final
public static final
不可以：abstract final void eat();
private abstract void eat();
static abstract void eat();
abstract 不能和 final,private,static 连用。

四，interface：是抽象类的变体，。在接口中，所有方法都是抽象的。如：interface M{
int num=3;
void eat();
}
理解为接口是一个特殊的抽象类，所以接口不能创建对象，且接口没有构造方法，
但可以声明一个接口类型的引用（m 是接口类型实现类对象，如：M m=new N();）
接口存在的意义是被子类实现，如果不想让子类也抽象，
就要实现接口里面所有的抽象方法，实现过程中注意访问权限；

用 implements 关键字实现接口，如：class N implements M{
public void eat(){...}
}
接口里面的常量默认都是 public static final 的；
接口里面的方法默认都是 public abstract 的。

接口本身支持多继承，继承了父接口里功能的定义，如，interface A extends B,C,D{} //A，B，C，D 都是接口；
类可以同时继承一个父类和实现接口（或多个接口）。
如：class AA extends BB implements CC,DD,EE{}//AA，BB 是类，CC，DD，EE 是接口；
作用：1，用接口去实现多继承，接口是对类的共性进行再次抽象，抽象出类的次要类型。
如：蜘蛛侠，拥有人和蜘蛛的属性，但主要类型是人，次要类型（接口）是蜘蛛，
因为接口是次要类型，所以在类关系里不占一个节点，不会破坏类层次关系的树状结构，
2，标准（保证弱耦合）：**一个接口就是一个标准**（里面的属性不能改变，只定义了功能，
但没有被实现），接口将标准的制定者，标准的实现者以及标准的使用者分离开，
降低实现者和使用者的耦合。接口是 java 里一种重要的降低耦合的工具；
接口可以屏蔽不同实现类的差异，
当底层的实现类更换后，不会对上层的使用者产生影响，体现在参数和返回值。

写程序时，应该先写实现者再写使用者，如：Bank.java 是实现者，View.java 是使用者，
但是有了接口之后，就可以用接口回调的功能；
接口回调：先定义接口，然后写使用者和实现者的顺序随便（一般是先写使用者，
后写实现者）；利用参数把实现者传给使用者（即：实现者是使用者的属性），
使用者利用接口调用实现者相应的功能。
**接口和抽象类的区别 1 一个类可以 implements 多个接口，而只能 extends 一个抽象类
2，一个抽象类可以实现部分的方法，而接口都是抽象的方法和属性

Object 是 Java 里所有类的直接或间接父类，Object 类里面的所有功能是所有 java 类共有的
1，JVM 调用垃圾回收器回收不用的内存（没有引用指向的对象）前运行 finalize()，给 JVM 用的方法。
程序显示的通知 JVM 回收没用的内存（但不一定马上就回收）：System.gc();或 Runtime.getRuntime().gc();

2，toString()返回对象的字符串表现形式，打印对象时，虚拟机会自动调用 toString 获取对象的字符串表现格式，
如：System.out.println(str.toString());＝＝System.out.println(str);
如果本类不提供（覆盖）toString()，那么使用的是 Object 类里的相应方法，打印的就是地址。
如：public String toString(){
return ".....";
}

3，基本类型时“==“判断变量本身的值是否相等；引用类型时，判断的是地址是否相等。
equals 判断的是对象内容是否相等。对于自己创建的类，应该覆盖 Object 类的 equals()方法;
否则使用的是 Object 类里的 equals()方法，比的是地址。

覆盖方法如下：

```
/*****************************************************
public boolean equals(Object o){
  if(o==null)  return false;
  if(o==this)  return true;
  if(!(o.getClass()==this.getClass())) return false;
  final Student s=(Student)o;
  return this.name.equals(s.name) && this.age==s.age ;  //比较原则;
 }
*****************************************************/
```

覆盖 euqals()方法时遵循的原则：
自反性：a.quals(a);　　//true
对称性：a.equals(b);<==> b.equals(a);//true
传递性：a.equals(b);//true　　b.equals(c); //true
　　－－－>则：a.equals(c);　//为 true

封装类（Wrapper class）：

OverLoading 时，基本类型时采用向上匹配原则，
如果没有基本类型的话就向包装类转换，如果还没有就让这个基本类型在包装类里也采用向上匹配原则：

基本类型－转换到－－>包装类
boolean－－－－－>Boolean
int－－－－－－－－－>Integer　　//Integer 是引用类型，
int－－－－－－－－－>Ddouble　　//合法，　　但 Integer－－－－－－>Double　　非法
double－－－－－－>Double
　.....　－－－－－－－－>　.....
任何类型－－－－－>Object
基本数据类型 int 可以向 double 自动扩展，但是包装类型之间不能自动的相互转换，
基本类型数据－－－>包装类型
int i=3;
Integer it=new Integer(i);　　//手动转换：基本类型向包装类型转换。
int <－－－－> Integer <－－－－> String
转换时 String 类型必须为全数字字符串。如："2515"　　不能为："abc265","aec"...等
String str=”123”; int it=Integer,parseInt(str);把字符串转换成数字。String str2=it+“”; 把数字转化成字符串

＝＝内部类＝＝－－－－－－－－－－－－
定义在其他代码块（类体或者方法体）里的类称为内部类；
编译后每一个内部类都会有自己的独立的字节码文件，
文件名：Outer$Inner.class－－>**内部类也可以有父类和实现接口。也可以有抽象方法**。

根本位置和修饰符的不同分为四种：
1，member inner class　　**成员内部类，**当实例方法或变量一样理解。
　1）定义的位置：类以内，方法之外，没有静态修饰符（static）。
　2）本身能定义的属性和方法：只能定义非静态的属性和方法。
　3）能直接访问的什么：能访问外部类的所有静态和非静态的属性或方法。
　4）怎么创建对象：在外部类内的方法内：Outer.Inner inner=new Outer().new Inner();
　　在外部类外的类的方法内：Outer.Inner inner=new Outer().new Inner();或
　　在 Outer 类里提供一个 getInner()方法,返回内部类的对象，这样在外部类外的类的方法内也可以用该成员内部类。

2，static inner class 静态内部类（嵌套内部类），当静态方法或变量一样理解。
　static 只能修饰内部类，不能修饰外部类。
　1）定义的位置：类以内，方法之外，有静态修饰符（static）。一般写在外部类的属性下面。
　2）本身能定义的属性和方法：可以定义静态和非静态的属性或方法。
　3）能直接访问的什么：只能访问外部类的静态属性和方法。
　4）怎么创建对象：在外部类内的方法里：　Outer.Inner inner=new Outer.Inner();
　　在外部类外的类方法里：　Outer.Inner inner=new Outer.Inner();
3，local inner class　　**局部内部类**　　当局部变量一样理解。
　1）定义的位置：方法里面的类，前面不能用 public 或 static 修饰。
　2）本身能定义的属性和方法：只能定义非静态的属性和方法。
　3）能直接访问的什么：能访问方法内用 final 修饰的局部变量（不能与该类内的变量名相同）。
　　能访问外部类的所有静态和非静态的属性或方法。
　4）怎么创建对象：只能在方法内创建对象，如：Inner inner=new Inner(); 对象的作用范围只在方法内。
4，annonymous inner class　　**匿名内部类**　　如：Teacher tc=new Teacher(){
　1）没有名字的类，没有构造方法。是一个特殊的局部内部类，　　public void teach(){...}
　　可以实现一个接口，　或者一个类，　　}
　　生命周期里只能产生一个对象(tc)，也就是说只能被一个对象（tc）调用，
　2）除了没有名字外，看匿名内部类所在的位置，他的定义和访问将和成员内部类、静态内部类、局部内部类一样。
　　一般像局部内部类的定义和访问比较多。

批注 [U10]: Java 提供两种不同的类型：引用类型和原始类型（或内置类型）。Int 是 java 的原始数据类型，Integer 是 java 为 int 提供的封装类。Java 为每个原始类型提供了封装类

3）当试图创建接口或者抽象类对象的时候，用匿名内部类。
表示类体的{...}紧跟在抽象实例（接口）之后，表示实现该抽象实例（接口）。
调用匿名内部类的方法只能用写类时创建的那个对象（tc）。
作用：1，不破坏访问权限的情况下，内部类可以使用外部类的私有成员变量和方法。
2，将接口公开，将实现类（实现公开的接口）作成内部类隐藏起来，强制要求使用者使用接口，强制降低偶合度。
3，Java 通过接口和内部类两种机制来实现多继承。在类内部可以建立本类的实例，然后调用本类内的其他方法。

Exception(异常)：运行时的概念。

1，Throwable：运行时可能碰到的任何问题的总称；
1）Error：指非常严重的错误，系统不要求程序员处理，也处理不了。如：硬件坏了.....等。
2）Exception：从代码的角度是程序员可以处理的问题；
UncheckedException(RuntimeException 的子类) （未检查异常）如果是 RuntimeException（或子类）就是为检查异常，其他就是已检查异常
程序员小心谨慎完全可以避免的异常，系统不要求程序员处理（可以不管,运行会提示错误），
如：3/0　　数组下标越界。
CheckedExcepiton　（已检查异常）
系统要求必须处理异常。

2，异常处理：**异常是相对于方法来说的**。
1）声明抛出异常（消极的处理）
throws（抛弃）：写在方法名的定义上，后面跟要抛弃的异常类型。
如：public void m1() throws Exception{.}
异常产生时，责任可能并不在当前方法，向外抛弃（把异常抛弃，留给调用者处理）可以让异常找到一个最佳的位置处理
抛弃过程中可以对异常类型进行扩展，但是不能缩小。
throw（抛出）：一般出现在方法实现里，用来抛出异常对象（或者是产生异常），
如：throw new FileNotFoundException();
当代码出现异常时，代码不会向下执行，JVM 会将异常封装成相应的异常类的对象，
然后向外抛出。之后这个方法里剩下的代码就不会再执行了。
对于一个方法的返回值：
1）正常运行时，要求方法必须返回定义的类型的值。
2）如果运行不正常（出现异常），方法返回的是异常对象
方法覆盖：名相同，参数相同，返回类型相同，访问权限不能更小，子类抛弃的异常不能比父类更多。
2）try....catch（积极的处理）：
一个 try 语句后可以跟多个 catch 语句；catch 时异常子类放上面，异常父类放下面。
如果没有父子关系，先后无所谓；

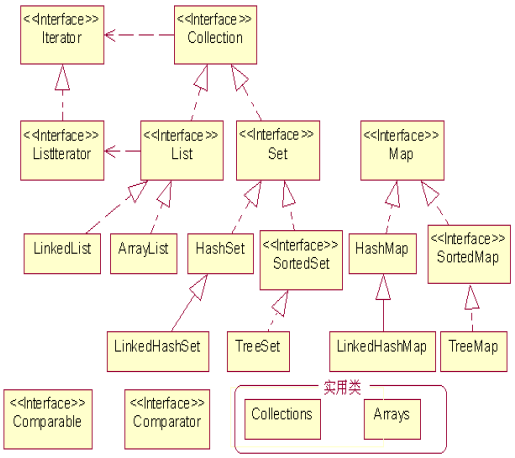
```
－－－方法－－－( ){
    try{
        //可能会出现异常的代码
        xxxxxxxxxx;  (1)
        xxxxxxxxxx;  (2)
    }catch(Exception1 e1){
        //当 try 代码块出现异常时,执行 catch 代码块。
        xxxxxxxxx;   (3)
    }catch(Exception2 e2){
        xxxxxxxxx;   (4)
    }finally{
        //不管有没有异常出现都要执行的代码。
        xxxxxxxxx;   (5)
    }
        xxxxxxxxx;   (6)
}
```

1）如果（1），（2）没产生异常，（2）执行后直接执行（5），然后执行（6）。
2）如果（1）产生异常，（2）不会被执行，直接跑出 try{..},匹配 catch，和 catch 里定义的类型一致，
执行 catch 完了后，直接跳到（5）执行，最后再执行（6）,如果异常类型都不一致，将导至语法问题。
3）自定义异常类型（业务异常）：

```
如：class MyException extends Exception{
        public MyException(String str);
        super(str);
    }
```

Exception 异常总结：
1、如果程序用了 System.exit(0);则不会执行 finally 里的程序
2、在程序 return 前执行 finally 里的程序
3、Java 中异常分为两类：
1) checked Exception
处理方式一、继续抛出，消极做法，直到抛出到 JVM
处理方式二、用 try..catch
2) unchecked Exception (runtime exception)
throws ArithmeticException,IOException 应该是出现这两种异常就向上抛吧。
什么情况下一个方法抛出几个异常？一般来说应该是会抛出几种异常，然后在一级调用这个方法时处理一下。
如果没有特殊需要的话要把可能出现的异常都截获并处理。

```
try{
  method();
}catch(exception1 e1){
 do something;
```



}catch(exception2 e2){
 do something;
}……
e1 的范围小于 e2.

课外考题：12、final，finally，finalize 的区别。
final 用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，类不可继承。
finally 是异常处理语句结构的一部分，表示总是执行。
finalize 是Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。

Java 高级部分==========
集合，常用的接口，Collection，List Set，SortedSet，Map，SortedMap
用来管理／容纳多个对象的对象（或称为容器）；
面向对象可重用性的体现，对数组作了功能封装。
Collection 是一个接口：是以对象为单位来管理元素的。
　　基本操作：**add　remove　size**，
　　遍历：迭代遍历
有两个子接口：List 接口和 Set 接口
1，**List 接口**：元素是有顺序（下标），可以重复。有点像数组。可以用迭代器（Iterator）和数组遍历集合。
　　List 接口里本身有自己的方法，还有从父接口 Collection 里继承的方法。
　　remove(int)删除某一位置的对象、add(int,Object)往某一位置插入对象 、get(int)查询某一位置上的元素。
　　遍历：迭代遍历、for 遍历
　　Collections.sort(list)；引入 list 集合然后对 list 集合里的对象（如：Student 类）进行排序时，
　　只要在让对象（类）实现 Comparable 接口，再覆盖接口里面的方法（compareTo()方法），
　　在 compareTo()方法里写出进行什么样的方式排序，
```
　　Public int compareTo(Object o){
　　　Work w=(Work)o;
　　 If(this.salary!=w.salary){return (int)w.salary-(int)this.salary;}
　　 Else If(this.age!=w.age){return w.age-this.age;}else  retuen this.name.compareTo(w.name);sss
　　}
```
　　然后在主函数里使用 Collections.sort(list)；　，就对 list 里的对象进行排序了，
　　而不用管 Collections.sort()方法是怎么实现的，既不用管 Collections.sort()方法的实现体。
　　排序规则：对象实现 Comparable 接口，覆盖 compareTo()方法，Collections.sort(List);

1）**ArrayList** 集合类实现 List 接口，轻量级，底层是以数组实现的，查询快，增删慢，线程不安全，用 get(int)方法多时，
Vector 集合也实现 List 接口，底层也是以数组实现的。但这个类已经放弃了。重量级，查询慢，增删快，线程安全。（Vector 和 HashTable 是线程同步的（synchronized），所以线程安全。性能上，ArrayList 和 HashMap 分别比 Vector 和 Hashtable 要好。）
2）LinkedList 集合类实现 List 接口，他底层是以链表实现的，查询慢，增删快，
　　用 remove(int)、add(int,Object)方法多时，
　　自己实现栈（stack），并对栈里进行增删操作。
```
　　class MyStack{
　　　private LinkedList list=new LinkedList();
　　　public void push(Object o){
　　　　list.addFirst(o);
　　　}
　　　public void pop(Object o){
　　　　list.removeFirst();}}
```
2，**Set 接口：无顺序，不可以重复**（内容不重复，而非地址不重复）。**只能用迭代器（Iterator）遍历集合。**
　　Set 接口里本身无方法，方法都是从父接口 Collection 里继承的，
　　遍历：迭代遍历
　　实现类：保证元素内容不重复。
1）HashSet 集合类实现 Set 接口，**底层是以数组实现的**。HashSet 集合里不允许有重复对象
　　每向 HashSet 集合类里添加一个对象时，先使用 HashSet 集合类里 add(o)方法，
　　再调用对象 o 的 hashCode()方法算出哈稀码，保证相同对象返回相同的哈希码。
　　如果哈稀码一样，再调用对象 o 里的 equals()方法对对象的属性进行比较是否相等，
集合也可以构造集合，如：List<Object> list=new ArrayList<Object>(); Set<Object> set=new HashSet<Object>(list);原来在 list 集合里的对象是可以重复的，但被 set 集合构造之后，重复的对象将被去掉
按这三种顺序来理解接口：　　1）接口特点；2）接口常见操作（方法）；3）接口实现类的特点
Map:对应一个键对象和一个值对象，可以根据 key 找 value，
　　（Key－－value）不可重复且无序－－可重复
　　排序（SortedMap 是 Map 的子接口）：TreeMap 类实现 SortedMap 接口；对集合进行排序。
　　基本操作：put()　get()
　　遍历：键遍历　ketSet()返回键的集合(Set)（说明 map 的键是用 set 实现的，不能重复）
　　　　值遍历 values()　返回值的集合。
　　HashMap 类实现 Map 接口：　　查询快，线程不安全，允许键和值对象为 null
　　Hashtable 类实现 Map 接口：查询慢，线程安全
　　TreeMap 类实现 Map 接口：SortedMap 的实现类，自动对键进行排序。

批注 [U11]: 说出 ArrayList,Vector, LinkedList 的存储性能和特性
ArrayList 和Vector 都是使用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插
入元素，它们都允许直接按序号索引元素，但是插入元素要涉及数组元素移动等内存操作，所以索引
数据快而插入数据慢，Vector 由于使用了synchronized 方法（线程安全），通常性能上较ArrayList
差，而LinkedList 使用双向链表实现存储，按序号索引数据需要进行前向或后向遍历，但是插入数
据时只需要记录本项的前后项即可，所以插入速度较快。

```
Iterator 实例
Static void printValue(Map map){
  Collection col=map.values();
Iterator it=col.iterator();
  While(it.hasNext()){
        Object obj=it.next();…;
}}
```

```
Jdk1.5 新特性加了 forEach 循环方便数组和集合的遍历（是在泛型的基础上提出的）
Static void printValue(Map<Integer Product> map){
  Set<Integer> s=map.ketSet();
For(Integer i:s){xxxx}
}
```

注意：如果需要定位，就得用“普通”的 for，在列表遍历期间无法删除集合对象。

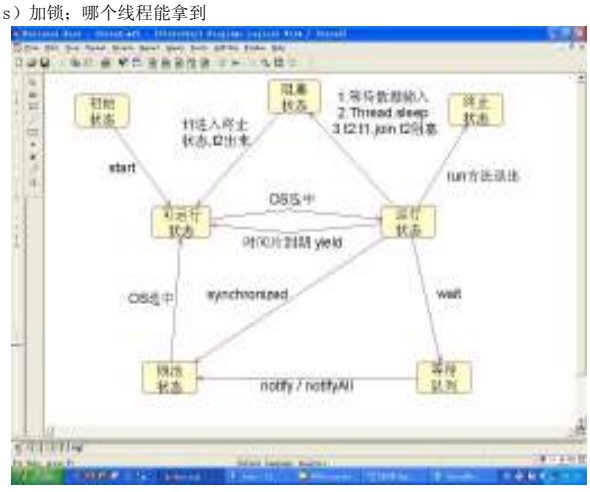
课外考题、Collection 和 Collections 的区别。
Collection 是集合类的上级接口，继承与他的接口主要有 Set 和 List.
Collections 是针对集合类的一个帮助类，他提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作

＝＝＝＝＝＝＝**图形界面（GUI）**＝＝＝＝＝＝＝==========

1，选择容器：图形界面容器，容器也是一个对象（相当一个图形界面），用来装组件的。
　　JFrame：窗口，相当于一个容器；如一些按钮，最大化，最小化等一些。默认的布局管理器是 BorderLayout
　　JPanel：面版，透明的。默认布局是 FlowLayout， 一般放在窗口里
　　javax.swing.*;存在于该包中。
2，设置布局管理器
　　FlowLayout()：流式布局。组件会随着容器的大小而改变位置，
　　BorderLayout()：东西南北中分布组件
　　GridLayout()：网格布局。一个格只能放一个组件，
　　CardLayout()：卡片布局。如按上一步和下一步。界面是一个卡片式的布局。
　　GridBagLayout()：复杂的网格布局，一个格可以放多个组件。
　　java.awt.*;存在于该包中。
　　setLayout()：用于设置什么样的布局。
3，添加组件：一个组件就是一个对象，
　　JTextField ：单行文本框
　　JTextArea ：多行文本区
　　JComboBox：下拉列表
　　JScrollPane：左右拉或上下拉按钮
4，设置事件监听
AWT 事件模型，事件三要素：事件对象，事件源，事件监听器
1，事件对象：事件也是一个对象。事件对象继承：java.util.EventObjct 类
2，事件源：发生事件的对象，也是报告事件的对象。（点击 b1 按钮，那么 b1 按钮就是事件源）
3，事件监听器：处理事件的对象。 监听器接口必须继承 java.util.EventListener 接口。
　　事件源预先就某种事件注册监听器，当事件发生时，事件源就给所有注册的监听器发送一个事件对象，
　　所有监听器存在一个数组集合里（如 ArrayList 集合），由监听器做出相应处理。
　　事件源可以同时是多种事件的事件源，就某种事件单独注册监听器。
　　事件源就同一种事件，可以注册多个监听器。
　　一个监听器可以同时注册在多个事件源当中。
　　事件对象中会封装事件源对象。
　　事件监听接口中的每一个方法，都应以对应的事件对象作为参数类型。
　　所谓的事件源给监听器发送事件对象，其实就是事件源以事件对象为参数，调用监听器的方法。
　　getSource()方法：是事件对象（EventObject）里的一个方法，用事件对象 e 调用（如：e.getSource()；）
　　　　getSource()方法返回的是事件对象里的事件源

＝＝＝＝＝＝＝**多线程**＝＝＝＝＝＝＝==========

线程：进程中并发的一个顺序执行流程。
并发原理：CPU 分配时间片，多线程交替运行。宏观并行，微观串行。
Tread t=new Thread();表示新建一个线程对象，并不表示线程。
　　当调用 t.start();才起动线程,当得到 CPU 时，就会执行线程 t 的方法体。
线程三要素：CPU、Date、Code
多线程间堆空间共享，栈空间独立。堆存的是地址，栈存的是变量（如：局部变量）
创建线程两种方式：继承 **Thread 类或实现 Runnable** 接口。
Thread 对象代表一个线程。
多线程共同访问的同一个对象（临界资源），如果破坏了不可分割的操作（原子操作），就会造成数据不一致的情况。
在 java 中，任何对象都有一个互斥锁标记，用来分配给线程。
synchronized(o){..同步代码块. .}
　　　　对 o（o 是临界资源）加锁的同步代码块，只有拿到 o 的锁标记的线程，
　　　　才能进入对 o 加锁的同步代码块，退出同步代码块时，会自动释放 o 的锁标记。
synchronized 的同步方法，如：public synchronized void fn(){} 对访问该方法的当前对象（this）加锁；哪个线程能拿到该对象（临界资源）的锁，哪个线程就能调用该对象（临界资源）的同步方法。
一个线程，可以同时拥有多个对象的锁标记



在 java 中，任何对象都有一个锁池，用来存放等待该对象锁标记的线程，
线程阻塞在对象锁池中时，不会释放其所拥有的其它对象的锁标记。
在 java 中，任何对象都有一个等待队列，用来存放线程，
线程 t1 对（让）o 调用 wait 方法,必须放在对 o 加锁的同步代码块中!
1.t1 会释放其所拥有的所有锁标记;
2.t1 会进入 o 的等待队列
t2 对（让）o 调用 notify/notifyAll 方法,也必须放在对 o 加锁的同步代码块中!
会从 o 的等待队列中释放一个/全部线程，对 t2 毫无影响，t2 继续执行。
当一个现成对象调用 yield()方法时会马上交出执行权，回到可运行状态，等待 OS 的再次调用
线程的生命周期
下面为线程中的 7 中非常重要的状态：（有的书上也只有认为前五种状态：而将“锁池”和“等待池”都看成
是“阻塞”状态的特殊情况：这种认识也是正确的，但是将“锁池”和“等待池”单独分离出来有利于对程序的理解）
1,初始状态，线程创建，线程对象调用 start()方法。
2，可运行状态，也就是等待 Cpu 资源，等待运行的状态。
3，运行状态，获得了 cpu 资源，正在运行状态。
4，阻塞状态，也就是让出 cpu 资源，进入一种等待状态，而且不是可运行状态，有三种情况会进入阻塞状态。
1)如等待输入（输入设备进行处理，而 CPU 不处理），则放入阻塞，直到输入完毕，阻塞结束后会进入可运行状态。
2)线程休眠，线程对象调用 sleep()方法，阻塞结束后会进入可运行状态。
3)线程对象 2 调用线程对象 1 的 join()方法，那么线程对象 2 进入阻塞状态，直到线程对象 1 中止。
5，中止状态，也就是执行结束。
6，锁池状态
7，等待队列

课外问题：71、简述 synchronized 和 java.util.concurrent.locks.Lock 的异同 ？
主要相同点：Lock 能完成 synchronized 所实现的所有功能
主要不同点：Lock 有比 synchronized 更精确的线程语义和更好的性能。synchronized 会自动释放锁，
而 Lock 一定要求程序员手工释放，并且必须在 finally 从句中释放。
＝＝＝＝＝＝＝I/O＝＝＝＝＝＝＝==========
File：代表了磁盘上的文件或者目录
I/O：jvm 和外部数据源的数据交换。File,db—in-→jvm---out-→file,db
流一共有三种分类：
方向分：输入流和输出流；
单位分：字节流和字符流；
字节流：
InputStream/OutputStream 字节流的父接口
（1）FileInputStream/FileOutputStream 文件字节流 ((可以向下转换))
DataInputStream/DataOutputStream 读写 8 种基本类型和以 UTF-8 读写 String
BufferedInputStream/BufferedOutputStream 带缓冲的输入/出流
PrintStream 融合 Data 和 Buffered， System.out 所属的类
Piped 管道 用于线程间交换数据
RandomAccessFile 随机访问文件
字符流:处理字符编码问题
Reader/Writer 字符流的父接口
FileReader/FileWriter 文件字符流，和 FileInputStream/FileOutputStream 文件流，
((可以向下转换)) 与上面的（1）是相等，
只不过一个是字节流，下面是字符流，所以两个无法相传
InputStreamReader/OutputStreamWriter 桥转换 将字节流转成字符流 在桥转换的过程中,可以制定编解码方式
BufferedReader/PrintWriter 有缓冲
字符流转换为字节流时，指定编解码方式是在桥转换时指定的。
功能分：节点流和过滤流；
节点流：用于传输数据。
过滤流：帮助节点流更好的传输数据。
piped（管道节点流）：用于两个线程间传输数据。一个线程的输出，是另一个线程的输入。
对象序列化：
把对象放在流上传输 ObjectInputStream/ObjectOutputStream
只有实现了 Serializable 接口的对象才能序列化
用 transient 修饰的属性,为临时属性,不参与序列化，只能修饰对象的属性。

==========**====网络=**==========
1 网络通信的本质是进程间通信。
2 Tcp 协议和 UDP 协议
TCP：开销大，用于可靠性要求高的场合。
TCP 的过程相当于打电话的过程
UDP：用在对实时性要求比较高的场合。
UDP 的过程相当于写信的过程。
注意：socket 是套接字，ip 和 port（端口号 0~65535 个端口，一个端口只能有一个进程）
3， TCP 通信机制，tcp 是面向连接的，实现多线程的方法有三个
① 为每个客户分配一个工作线程。
② 创建一个线程池，由其中的工作线程来为客户服务。

③ 利用 JDK 的 Java 类库中现成的线程池，由它的工作线程来为客户服务。

下面以为每个客户分配一个工作线程来实现多线程

在服务端

```
import java.net.*;
import java.io.*;
public class TCPServer2 {
   public static void main(String[] args) throws
   Exception{
      ServerSocket ss=new ServerSocket(9000);//端口号
      while(true){
            Socket s=ss.accept();//连接监听客户端
            System.out.println(s.getInetAddress());
            Thread t=new ServerThread(s);//实现多线程
   连接
            t.start();
      }
   }
}
class ServerThread extends Thread{//分配线程
   Socket s;
   public ServerThread(Socket s){
      this.s=s;
   }
public void run(){
   try {
      OutputStream os=s.getOutputStream();//在网络中
获取输出流
      PrintWriter out=new PrintWriter(os);
      for(int i=1;i<=30;i++){
      out.println("Hello "+i);//通过网络发消息给客户
   端
      out.flush();
      Thread.sleep(1000);
            }
      } catch (Exception e) {
            e.printStackTrace();
      }
      finally{
            try {
                  s.close();//注意关闭线程而不关闭流
            } catch (IOException e) {
                  e.printStackTrace();
            }
      }
   }
}
```

在客户端

```
import java.net.*;
import java.io.*;
public class TCPClient {
   public static void main(String[] args) throws
   Exception{
Socket s=new Socket("192.168.0.10",9000);//连接服务端
         的 ip 和端口
      InputStream is=s.getInputStream();//获得输入流，
读取服务端发来的信息
      InputStreamReader ir=new InputStreamReader(is);
      BufferedReader in=new BufferedReader(ir);
      for(int i=1;i<=30;i++){
            System.out.println(in.readLine());
      }
      s.close();
   }
}
```

4， Udp 面向无连接，不可靠，效率高，资源占用少 ，先从客户端发起通信，让服务端知道其地址等信息，不同的协议其端口号的含义也不一样

例子：

在客户端

```
import java.net.*;
public class UDPClient {
  public static void main(String[] args) throws
Exception{
DatagramSocket ds=new DatagramSocket();注册邮箱
String text1="I am here!";
byte[] bs1=text1.getBytes();//转换成字节用来传输
DatagramPacket letter1=new DatagramPacket(
bs1,0,bs1.length,InetAddress.getLocalHost(),9000);
//写好信（地址和内容）
      ds.send(letter1);//发信
DatagramPacket letter2=new DatagramPacket(
                  new byte[100],0,100);
//准备空信用来接受服务端的信
      ds.receive(letter2);//收信
      byte[] data=letter2.getData();//得到信的内容
      int offset=letter2.getOffset();
      int length=letter2.getLength();
   String str=new String(data,offset,length);//转换成
   字符串读取
      System.out.println(str);
      ds.close();
      }
}
```

在服务端

```
import java.net.*;
public class UDPServer {
public static void main(String[] args) throws
Exception{
DatagramSocket ds=new DatagramSocket(9000);//服务端口
      while (true) {
      byte[] bs1 = new byte[100];
      DatagramPacket letter1 = new DatagramPacket(bs1,
0, bs1.length);//制作空信封用来收信
      ds.receive(letter1);
      InetAddress address = letter1.getAddress();
      int port = letter1.getPort();
      String str = "北京 2008 年举行奥运会";
      byte[] bs2 = str.getBytes();
DatagramPacket letter2 = new DatagramPacket(bs2, 0,
bs2.length,address, port);//获得客户端的 ip 和 port 然
后将信息发给客户端
      ds.send(letter2);
            }

      }
```

DatagramSocket，相当是邮箱有 send（发消息）receive（收消息）

DatagramPacket，相当是信，里面有 ip 端口，信息内容

==========**jdk1.5 新特性**=====

1， 可变参数 相当一个数组 m（String …s）一个方法只能有一个可变参数，且只是最后一个参数

2， Foreach 循环（for（object o：list））方便遍历数组和集合

3， 枚举

```
enum Course{
     UNIX ("Luxw") {
          public void study(){}
     },
     COREJAVA ("Huxz"){
          public void study(){}
     };
  Public void study ();
}
```

使用：

```
class test { Course [] cs=Course.values(); for(Course c:cs ){System.out.println(s.ordinal+s+s.getName())}}
```

注意的问题：1 枚举值之间用逗号隔开，最后一个用分号 2 枚举中可以有抽象，但必须在枚举值中实现；

4， 泛型 泛型解决集合中不安全性，泛型强制集合中都是一个类型

```
List<Integer>  l=new ArrayList<Integer>();Map<Integer,String> m=new HashMap<Integer,String>();
泛型方法：public static <T> void copy(List<T> l,T[] os){ for(T o:os){l.add(o);}}
                              <T extends Integer> <? Extends Integer>
```

5,反射 类对象，类加载时把类的信息保存在 jvm 中就会产生一个相应的对象（记录类信息的对象），只要类对象存在则类信息就在

获得类对象应用的三种方式：

```
①Class c1=ArrayList.class; ②Object l=new ArrayList();Class c2=l.getClass();
③String className="java.util.ArrayList"; Class c3=Class.forName(className);
Class[] cs=c1.getInterfaces();//获得接口
Class c=Animal.class;
Field[] fs=c.getDeclaredFields();//获得属性的数组
Method[] ms=c.getDeclaredMethods();//获得本类所有公私有的方法，getMethods（）获得父类所有公开的方法
Constructor[] cons=c.getDeclaredConstructors();//获得构造方法的数组
Object o2=c.newInstance();//创建类的对象
Method m1=c.getDeclaredMethod("study");//找到方法名是 study 的方法
m1.setAccessible(true);//设置为可以调用
m1.invoke(o2);//调用 m1 方法
Method m2=c.getDeclaredMethod("study", String.class);
m2.setAccessible(true);
m2.invoke(o2,"CoreJava");//调用 m2 的方法，方法的参数是 CoreJava
```

6,注释,===注释是一种类型（总共有四种类型：类，接口，枚举，注释）

定义注释

```
如：@Override;
标记：@注释名
单值：@注释名（属性名＝属性值）
多值：@注释名（属性 1＝值 1，属性 2＝值 2，.....）

import java.lang.annotation.*;
@Retention(RetentionPolicy.RUNTIME)
@Target()
public @interface MyType {
  String authorName();
  String lastModified();
  String bugFixes() default "ok";
  }
```

在类中应用注释：

```
import java.lang.annotation.*;
@MyType(authorName="hadeslee",lastModified="20061207
")
    public class Test1 {
    /** Creates a new instance of Test1 */
    public Test1() {
    }
    @Deprecated

@MyType(authorName="hadeslee",lastModified="20061207
",bugFixes="what")
    public void doSth(){
    }
```

这里我定义了一个我自己的注释类,声明方式和声明接口差不多,只不过在 **interface 前面多了一个@**符号.
注释类也可以用注释类注释,如此下去.
@Retention(RetentionPolicy.RUNTIME) 这句表示它的保存范围是到 RUNTIME,也就是运行时,这样在类运行的时候,我们也可以取到有关它的信息.
@Target() 这句表示它的适用对象,它可以用在哪里地方,我这里定义的是它可以用在类的定义和方法的定义上
然后我们看我们是怎么为我们写的类加上注释的

OOAD 思想

1 继承关系：指的是一个类（称为子类、子接口）继承另外的一个类（称为父类、父接口）的功能，并可以增加它自己的新功能的能力，继承是类与类或者接口与接口之间最常见的关系

2 关联关系

① 关联关系是对于两个相对独立的对象，当一个对象的实例与另一个对象的一些特定实例存在固定的对应关系时，这两个对象之间为关联，关系是使用实例变量实现的，比如公司和职员

② 聚合关系是关联关系的一种，他体现的是整体与部分、拥有的关系，此时整体与部分之间是可分离的，他们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享；比如计算机与 CPU、公司与员工的关系等；表现在代码层面，和关联关系是一致的，只能从语义级别来区分，它也是通过实例变量实现的，在 java 语法上是看不出来的，只能考察类之间的逻辑关系

③ 组合也是关联关系的一种特例，他体现的是一种 contains-a 的关系，这种关系比聚合更强，也称为强聚合；他同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束；比如你和你的大脑；表现在代码层面，和关联关系是一致的，只能从语义级别来区分

④ 依赖关系是类与类之间的连接，依赖总是单项的（人和车，人和房子），依赖关系在 java 中体现在局部变量，方法的参数，以及静态方法的使用。在 A 类中的方法中使用 B 类的引用

代码表示

```
class A{
  B b;//聚合
  C c=new C();//组合
  Public A（B b）{this.b=b;}
 Public void ect(D d){ d.m(); //依赖}
 }
 class D{void m(){}}
```

在 UML 图形表示

批注 [U12]: UML 方面标准建模语言UML。用例图,静态图(包括类图、对象图和包图),行为图,交互图(顺序图,合作图),实现图。

ORACLE

第一天：==========

1， **关系数据库的三要素：** 1 实体----》表（存放数据）
2 属性---》列
3 关系---》外键

2， sqlplus 常用命令

help index
可以看到所有的命令，
不会的命令用：
help <命令>;例如：help list
exit 退出 SQL*PLUS
desc 表名 显示表的结构
show user 显示当前连接用户
show error 显示错误
show all 显示所有 68 个系统变量值
edit 打开默认编辑器，Windows 系统中默认是 notepad.exe，把缓冲区中最后一条 SQL 语句调入 afiedt.buf 文件中进行编辑
edit 文件名 把当前目录中指定的.sql 文件调入编辑器进行编辑
clear screen 清空当前屏幕显示

SQLPLUS 到 ORACLE 的运行图(见笔记本)

select columnname（列名）
1) select *
2) select id,last_name,dept_id
3) select id,last_name "Name",salary*12 anual
4) select last_name||' '||first_name
//如果 comm 为 null 时，nvl(comm,0)等于 0；否则等于 comm；
5) select last_name,salary*(1+nvl(comm,0)/100)
6) select distinct last_name,first_name

from tablename
预定义: 作用的时间长 所有的叫这个列名的列都生效
sql: 当次生效 当前 sql 的当前表

order by:
select id,last_name,dept_id
from s_emp
1)order by dept_id;
order by title;
2)order by last_name,dept_id;
先按名字升序排,名字相同再按 dept_id 升序排 ，中间用“ , ”隔开；
3)order by last_name desc;
order by dept_id,last_name desc;
dept_id 升序,last_name 降序
4)null
在 oracle 里是最大，也就是升序的时候放在最后面；
5)order by 2; <=======> order by last_name
select * 里面的第二列
6)select id,last_name name
from s_emp
order by NAME;

==========

select from where order by

select id,last_name,dept_id,start_date
from s_emp
1)where dept_id=41;
where last_name='Smith'; ---->字符串区分大小写，SQL 命令不区分大小写；
where start_date='08-mar-90';
2)where salary>1000;
3)where dept_id between 41 and 45;
包括边界值
where dept_id between 45 and 41; //小的在前,如果反了,查不出记录
//dept_id 的值只能是 41，43，45 三个数中的一个,且三个数都会被计算一次。
4)where dept_id in(41,43,45);
5)where commission_pct=null; 查不出记录
where commission_pct is null;
6)where last_name like 'S%';
unix: * ?
oracle: % _ (下划线)
查出 last_name 是以'S_'
where last_name like 'S_%'; ----> error
where last_name like 'S_%' escape '\'; //escape '\'表示“\”为换码字符；
7) 非
!= <>
not between and
not in
not like
is not null
8)and > or
where dept_id=41 and id>5;
where dept_id=41 or id>5;
----->where salary>1000
and dept_id=41
or dept_id=42;
----->where salary>1000
and (dept_id=41
or dept_id=42);
----->where dept_id=41
or dept_id=42
and salary>1000;

```
select s_customer.name,s_emp.last_name
from
s_customer,s_emp
where s_emp.id(+)=s_customer.sales_rep_id;
//(+)让员工表补个空值。为的是让所有的客户都有对应的员工
//，因为有些客户没有对应的销售代表，所以设为空
```

第二天：

Single Row Function:

varchar2:

1) nvl('hello','world')-----> 'hello'
   nvl(null,'world')--------> 'world'
2) select id,last_name,dept_id
   from s_emp
   where lower(last_name)='smith';
3) select id,concat(last_name,first_name),dept_id
   from s_emp;
   concat(last_name,first_name)--->result1;
   concat(result1,title);
   --> concat(concat(last_name,first_name),title);
   System.out.println(stu.toString());
   select substr('String',-4,3)
   from dual;
   dual: dummy table ,为了维护 select 的完整性
   select id,last_name,dept_id
   from s_emp
   where last_name like 'S%';
   where substr(last_name,1,1)='S';
   number: round trunc

date:日期类型

1) select sysdate from dual;

标准的日期格式:

年: yy 08
yyyy 2008
rr 08
rrrr 2008
year two thousand and eight

月: mm 06
mon JUN
month june

日: dd 19
ddth 19th
ddsp ninteen
ddspth ninteenth

星期: d 4
dy thu
day thursday

小时: hh24 22
hh 10
am pm

分钟: mi 秒: ss

```
select id,last_name,dept_id,
   to_char(start_date,'yyyy-month-dd,hh24:mi:ss') "sdate"
from s_emp
where dept_id=41;

update s_emp
set
start_date=to_date('19-jun-08,11:24:56','dd-mon-yy,hh24:mi:ss')
where id=100;

update s_emp
set start_date=to_date('19-jun-90','dd-mon-yy')
where last_name='S_abc';

select id,last_name,to_char(salary,'$99,999.00')
from s_emp;

select id,last_name,
   to_char(start_date,'fmdd-mm-yyyy,fmhh:mi:ss') "sdate"
from s_emp;
```

fm:在不引起歧义的情况下去掉前置的零和空格
类似于开关变量,第一次写打开功能,再写关闭

Join:**数据来源于多张表**,叫多表联合查询,多个表之间要做连接

1.等值连接(内连接) 条件: fk-----pk

```
select s_emp.last_name,s_emp.dept_id,
       s_dept.id,s_dept.name
from s_emp,s_dept
where s_emp.dept_id=s_dept.id;


oracle:select e.last_name,e.dept_id,d.id,d.name
       from s_emp e,s_dept d
       where e.dept_id=d.id;

标准 sql:select e.last_name,e.dept_id,d.id,d.name
       from s_emp e inner join s_dept d
       on e.dept_id=d.id
       where e.last_name='Smith';

       select e.last_name,e.dept_id,d.id,d.name
       from s_emp e join s_dept d
       on e.dept_id=d.id
       where e.last_name='Smith';
```

打印出员工的名字,所在部门的名字,以及所在地区的名字

```
select e.last_name,d.name,r.name
from s_emp e,s_dept d,s_region r
where e.dept_id=d.id
  and d.region_id=r.id;

select e.last_name,d.name,r.name
from s_emp e join s_dept d
  on e.dept_id=d.id
  join s_region r
  on d.region_id=r.id;
```

2.外连接(左外,右外,全外)

打印所有的客户名字,以及所对应的销售代表的名字

1)左外：如果左边对应的右边的值为空时，右边补 null

```
oracle:select  c.name,c.sales_rep_id,e.id,e.last_name
                from s_customer c,s_emp e
                where c.sales_rep_id=e.id(+);
标准 sql:select c.name,c.sales_rep_id,e.id,e.last_name
         from s_customer c left outer join s_emp e
         on c.sales_rep_id=e.id;
```

2)右外 ：如果右边对应的左边的值为空时，左边补 null

```
select  c.name,c.sales_rep_id,e.id,e.last_name
from s_customer c,s_emp e
where  e.id(+)=c.sales_rep_id;

select c.name,c.sales_rep_id,e.id,e.last_name
  from s_emp e right join s_customer c
  on c.sales_rep_id=e.id;
```

3)全外 ：都不补空值（null）

```
select c.name,c.sales_rep_id,e.id,e.last_name
 from s_customer c full outer join s_emp e
 on c.sales_rep_id=e.id;
```

3.自连接: 本表的 fk 指向本表的 pk

可以给表起不同的别名，再做连接。

打印出员工的姓名和他的经理的名字

```
select e.last_name,m.last_name
from s_emp e,s_emp m
where e.manager_id=m.id;
```

打印出所有员工的姓名和他的经理的名字

```
select e.last_name,m.last_name
from s_emp e,s_emp m
where e.manager_id=m.id(+);
```

4.非等值连接(t1 和 t2 没有 pk 和 fk,还想连接 t1,t2)

table1: id value

1	A
2	B
3	C
4	D

table2: id value

1	A
2	B
3	C
4	D

请写出一句 sql 语句,打印输出

AB AC AD BC BD CD

```
select    t1.value,t2.value
from    table1 t1,table2 t2
where t1.id<t2.id;
```

请写出一段 sql 语句,打印输出

AB AC AD BA BC BD CA CB CD DA DB DC

```
select    t1.value,t2.value
from    table1 t1,table2 t2
where t1.id!=t2.id;
```

5.笛卡儿连接(不给连接条件)

请写出一段 sql 语句,打印输出

AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD

```
select t1.value,t2.value
from table1 t1,table2 t2;
```

SubQuery:

1.请打印出和 Smith 在同一部门的员工的姓名,start_date,dept_id

1) select dept_id

```
from s_emp
where last_name='Smith'; -----> result1
select last_name,dept_id,start_date
from s_emp
where dept_id=result1;
```

2) select last_name,dept_id,start_date

```
from s_emp
where dept_id=(
    select dept_id
    from s_emp
    where last_name='Smith');
```

2.主查询和子查询靠值联系

1) 值的类型要相同

```
select last_name,dept_id,start_date
from s_emp
where dept_id=(
    select last_name
    from s_emp
    where last_name='Smith');

select last_name,dept_id,start_date
from s_emp
where dept_id=(
    select salary
    from s_emp
    where last_name='Smith');

select last_name,dept_id,start_date
    from s_emp
    where dept_id=(
        select id
        from s_region
        where id=1);
```

2)值的个数要一致

```
    select last_name,dept_id,start_date
        from s_emp
        where dept_id=(
                select dept_id,last_name
                    from s_emp
                where last_name='Smith');

    select last_name,dept_id,start_date
        from s_emp
        where dept_id=(
            select dept_id
            from s_emp
            where last_name='Smith'
              or id=3);    ---->error
        --->改进
        select last_name,dept_id,start_date
        from s_emp
        where dept_id in (select dept_id
                                from s_emp
                                where last_name='Smith'
                        or id=3);
```

sql----->dbsv

1)编译

a.权限检测

b.语法检测

c.sql 翻译成内部指令

2)运行内部指令

3)哪些地方可以用子查询

需要值的地方就可以用子查询

1)select	不可以
from	可以（需要表的时候在这里写子查询）
where	可以（需要值的时候在这里写子查询）
order by	可以(按 select 里列的下标值排序 ，如：order by 2)
group by	不可以
having	可以（跟 where 用法一样）

```
select *
from (select e.last_name,e.dept_id,d.id,d.name
        from s_emp e,s_dept d
        where e.dept_id=d.id) e;
```

jobs:

1.SQL-1.pdf (4)

2. 请打印出公司入职最早的五个员工的姓名（last_name）,dept_id,start_date

```
select *
from (
    select last_name,dept_id,start_date
    from s_emp
    order by start_date
) e
where rownum    between 1 and 5;
```

伪列:rownum(代表显示在界面上的行号，数据库本身不提供，查询时会默认的提供，只有在 select 中显示的写 rownum,才会显示行号；)

rowid(数据库里本身就有的，代表在数据库里存的物理地址)

```
select * from s_dept;
desc s_dept;
```

分页:子查询和 rownum

打印表里面第五到第十条记录:

```
select last_name dept_id,start_date
from(
select e.last_name,e.dept_id,e.start_date,rownum rownumid
        from(
                select last_name,dept_id,start_date
                from s_emp
                order by start_date
        ) e
        where rownum between 1 and 10
) a
where a.rownumid between 6 and 10;
```

第三天：==========

Group function（组函数）:

1) select avg(salary),max(salary)
   from s_emp;
   select count(*) from s_emp;
   select count(id) from s_emp;
   select count(commission_pct) from s_emp;
   统计 commission_pct 列不为空的值的个数;
   number: max min sum avg count
   date: max min count
   char: max min count

2) select dept_id,avg(salary),count(*)
   from s_emp
   group by dept_id;
   select from where group by order by
   select dept_id,avg(salary)
   from s_emp
   where dept_id in (41,50)
   group by dept_id;

3) 当 select 里面出现组函数和单独的列并存时,要求必须写 group by 子句,并且单独的列必须出现在 group by 里面

```
select dept_id,avg(salary)
from s_emp
group by dept_id;
select salary,avg(salary)
from s_emp
group by salary;
```

如果 group by 里出现多列,那么按照列的联合分组,只有多列的值完全相同才会分到一组

```
select last_name,first_name,avg(salary)
from s_emp
group by last_name,first_name;
```

请打印出工资低于公司平均工资的人的姓名,dept_id,start_date

```
select last_name,dept_id,start_date
from s_emp
where salary<(select avg(salary) from s_emp );
```

4) 只要写 group by,那么 select 里单独列必须出现在 group by 里

```
select dept_id,last_name,start_date
from s_emp
group by dept_id,last_name,start_date;
```

5)where 执行时数据处于独立个体状态,里面不能对组函数进行判断,只能对单独记录判断

having 一定出现 group by 后面,此时数据处于组的状态

所以 having 只能对组函数或者组的共性进行条件判断,

请打印出部门平均工资大于 1500 的这些部门的 id 以及平均工资

```
select dept_id,avg(salary)
from s_emp
where avg(salary)>1500
group by dept_id;          ------>error
```

==========

```
select dept_id,avg(salary)
from s_emp
group by dept_id
having avg(salary)>1500;

select dept_id,avg(salary)
from s_emp
group by dept_id  -------> group
having salary>1000;     ----->整个组  error

select dept_id,avg(salary)
from s_emp
group by dept_id
having dept_id>40;

select dept_id,avg(salary)
from s_emp
where dept_id>40
group by dept_id ;
```

6) order by 列名,2,别名;

原来对 order by 里的列没有任何限制,只要表里有就可以

select from where group by having order by

在 order by 之前如果出现了 group by,那么 order by 里面就只能按照组函数或者是组的共性排序

```
eg: select dept_id,avg(salary)
    from s_emp
    where dept_id > 40
    group by dept_id
    having avg(salary)>1000
    order by last_name; ------>error
    order by max(last_name); ---->correct
    order by dept_id;       --------->dept_id
```

作业: 请打印出工资低于本部门平均工资的员工的姓名,dept_id,start_date

```
select dept_id,avg(salary) asalary
    from s_emp
    group by dept_id;     ---->a1
select e.last_name,e.dept_id,e.start_date
    from s_emp e,a1
    where e.dept_id=a1.dept_id
      and e.salary<a1.asalay;
--->select e.last_name,e.dept_id,e.start_date
    from s_emp e,(
        select dept_id,avg(salary) asalary
                from s_emp
                group by dept_id
            ) a1
    where e.dept_id=a1.dept_id
      and e.salary<a1.asalary;        //数据库表和表连接的时候，比较的是同一元组；
```

Define Variable
1. select last_name,dept_id,start_date
from s_emp
where dept_id=&did;
where last_name=&name;
where last_name='&name';
2.预定义
1) define varname=value
define did=41
简单,定义出来的变量都是 char 类型
不能写进 sql 文件
查询一个数是否已经预定义用 define varname
取消用 undefine varname
2) accept varname type prompt ' ' hide //单独一句话，为的是永久性的给 varname 一个值
accept did number prompt 'please input dept_id value:' hide
accept did prompt 'please input dept_id value:'----> char

比 define 复杂， 可以指定变量的类型,可以写进 .sql 文件，
在 bash 下：1）进入 csh；
2）setenv EDITOR VI ；
3）sqlplus 进入 SQL 的用户名和密码；
4）写一个要保存进文件的命令；
5）save sqlfile；
6）ed 进入 vi 编辑命令，在最上面加上（accept varname type prompt ' ' hide）语句；
7）ZZ（保存并退出 vi）；
8）start sqlfile.sql 就可以运行刚才保存的命令了。

下面是用于实验预定义的
/*select e.last_name
from s_emp e
where e.dept_id='&deptid';
accept deptid number prompt 'please input dept_id value:' hide
*/
3) define varname 查看状态
undefine varname 取消预定义
3.哪些地方可以用变量定义:
select
from
where 全部可以
group by
having
order by

select &col1,&col2,&col3
from &tab1
where &con1;

==========

第四天：
Create table:
1.db structure
1)table---> store data 2)view----> select 3)sequence 4)index
2.create table luxw1
(id number(7),
name varchar2(15),
registdate date
);
create table luxw2
(id number(7),
film blob
);

1)列一级约束
create table yesq5
(id number(7) constraint yesq5_id_pk primary key,
name varchar2(15) constraint yesq5_name_nn not null,
age number(3) constraint yesq5_age_ck_age check(age>18 and age<60),
sex varchar2(2) default 'M' constraint yesq5_sex_ck_sex check(sex in('F','M')),
birthday date default sysdate,
phone varchar2(15) constraint yesq5_phone_uk unique,
personid char(18)
constraint yesq5_personid_nn not null
constraint yesq5_personid_uk unique,
class_id number(7) constraint yesq5_class_id_fk references s_dept(id) on delete cascade
);
2)表一级约束
create table yesq5
(id number(7),
name varchar2(15) constraint yesq5_name_nn not null,
age number(3),
sex varchar2(2) default 'M',
birthday date default sysdate,
phone varchar2(15),
personid char(18) constraint yesq5_personid_nn not null,
class_id number(7),
constraint yesq5_id_pk primary key(id),
constraint yesq5_age_ck check(age>18 and age<60),
constraint yesq5_sex_ck check(sex in('M','F')),
constraint yesq5_phone_uk unique(phone),
constraint yesq5_personid_uk unique(personid),
constraint yesq5_class_id_pk foreign key(class_id) references s_dept(id) on delete cascade
);
五种约束: 无约束的地方不用写 constraint。
1) not null
2) unique
3) primary key
4) foreign key references （on delete cascade）
5) check
两种位置定义约束的 区别:
1) 约束前面是逗号就是表一级,
约束前面是空格就是列一级;
2) not null 只能定义在列一级
3) 联合键只能定义在表一级
eg:联合唯一： unique(last_name,first_name)
联合主健： primary key(product_id,order_id)
on delete cascade:该句在子表最后声明时，说明他的数据随外键的消失而消失
删除父表记录
1)写了 on delete cascade
sql: delete from fathertable where id=1;
2)没写 on delete cascade
sql: delete from sontables where fk=1;
delete from fathertable where id=1;

3 用子查询建表
1) 表的列名,列数,类型靠子查询定
create table luxw04
as
select id,last_name,dept_id,start_date
from s_emp
where dept_id in(41,42);

create table asalary
as
select dept_id,avg(salary)
from s_emp
group by dept_id;

2) 主 sql 规定了列名,列的个数,不能定义列的类型
在建表过程中只保留了非空（not null）约束,其他约束丢失

```
create table asalary
(did,avgsal)
as
  select dept_id,avg(salary)
    from s_emp
    group  by dept_id;
```

3) 可以制定约束

```
create table luxw06
(id primary key,
 last_name,
 dept_id references s_dept(id),
 start_date default sysdate
)
as
  select id,last_name,dept_id,start_date
  from s_emp
  where dept_id in(41,42);
```

约束和数据的关系:
谁先进入表,谁说了算

Data dictionary(表的集合):

1) 表:存放描述数据库状态的数据
db server 在建库时建立
由 db server 维护,在线动态更新
user 只能查询

2)user:由用户创建的 ，属于这个用户的。
all: 用户有权使用
dictionary: 保存了所有数据字典的名字,和描述信息

3)查出 yesq5 表 personid 列上的唯一性约束的名字
用到下面这两个表：
user_constraints
user_cons_columns

```
select
cons.constraint_name,cons.constraint_type,cols.column_name
  from user_constraints cons,user_cons_columns cols
  where cons.constraint_name=cols.constraint_name
    and cons.table_name='YESQ5'
    and cols.column_name='PERSONID';
```

DML:

insert:

* 1) 全表插入,值的顺序,类型,个数必须和表里的列一致

```
insert into luxw8
values(1,'zhangsan','m','12-sep-98');
insert into luxw8
values(2,'lisi','f',null);
insert into luxw8
values(3,'wangwu','f', to_date('23-06-08','dd-mm-rr'));

insert into luxw8
values(4,'lisi','f',&da);
```

* 2)选择插入,列的个数,类型,顺序必须和指定的相等
选择时非空的列必须出现

```
insert into luxw8(id,name,sex)
    values(5,'huxz','m');

    insert into luxw8(name,id)
    values('wangwu',6);

    insert into luxw8(name,sex)
    values('liucy','m');        ----->error 无 PK
3)insert into luxw8(id,name)
    values('7','liucy')
    values('8','luxw');       ---->error       不能同时添加两个数据
    insert into luxw8(id,name)
    values(&idd,&na);

  子查询提供值,值的类型和个数要匹配
    insert into luxw8(id,name)
    select id,last_name
    from s_emp
    where dept_id=45;
update:
  1) update  luxw8
     set name='sun'
     where id=1;
  2)update  luxw8
     set name='fengwang',sex='m',registdate=sysdate
     where id=2;
delete:
  delete from luxw8
  where id between 2 and 5;
```

Transaction（事务）:必须具有原子性

ACID：

A：原子性，加钱和减钱都要成功或都失败，保持原子性；
C：一致性，数据一致性，如：A 向 B 转帐 5000 块，那 B 就应该收到 5000 块，不能多也不能少；
I：隔离性，两个 SQL 语句的操作只操作自己的，不能影响到另一个语句，也就是相当于每个 SQL 都要有自己的回滚段；
D:持久性，数据的改变用 commit 或用 rollback，让数据持久化；

1) 一组不可再分的 sql 命令组成的集合， 如果分了数据将不统一，如：银行转帐
事务的大小跟业务需求有关

2) 事务结束时结果
success: commit 回滚段内的数据写到 FILE，锁释放，其它用户也可以看到结果
fail: rollback 清空回滚段内容，没写到 FILE，锁释放，其它用户看不到修改的值，

3)

```
insert update  delete update  delete commit;
create--->auto commit;
insert
update
delete
create --->auto commit;
insert
update
rollback;
insert update delete  exit
```

==========

第五天：

DDL:

alter:改表结构

1)增加一列,添加到表的最后一列

```
alter table luxw8
add(age number(3) default 18 check(age<40));
```

2)删除一列(在 oracle8i)

```
alter table luxw8
drop (age,registdate);

alter table luxw8
drop column age;
```

3)更改列名(在 oracle9i)

```
alter table luxw8
rename column oldname to newname;
```

4)更改列(类型,size,添加约束,添加默认值)

```
alter table luxw8
modify(name varchar2(20) default user unique);
```

如果表里没有数据,随便改

如果表里有数据,改的时候必须符合表里的数据

```
alter table luxw8
modify(sex not null);
```

非空约束只能用 modify 子句加

5)添加约束

联合键约束加的时候只能用 add 子句

```
alter table luxw8
add unique(personid);
alter table luxw8
add constraint luxw8_personid_uk unique(personid);
```

当 A 表的 PK 是 B 表的 FK 且 B 表的 PK 是 A 表的 FK 时，

只能先让一个表（A 表）的 FK 为空，再用 add 以表结构向该表（A 表）添加 FK，如下：

```
create table product
( id number(7) primary key,
  name varchar2(15) not null,
  price number,
  b_id number(7)
);
create table bid
( id number(7) primary key,
  userid number(7) references users(id),
  bprice number,
  product_id number(7) references product(id)
);
alter table product
add foreign key(b_id) references bid(id);
```

6)删除约束

```
alter table luxw8
drop constraint luxw8_personid_nn ;
alter table luxw8
drop primary key cascade;
```

7)失效约束

```
alter table luxw8
disable constraint luxw8_personid_uk;

alter table luxw8
disable constraint luxw8_id_pk cascade;
```

级联失效

8)生效约束

```
alter table luxw8
enable constraint luxw8_personid_uk;
alter table luxw8
enable constraint luxw8_id_pk;
```

生效时无法级联,只能手动一个一个做

```
rename oldtablename to newtablename        改表名
truncate table tablename;   截取表
drop table tablename cascade constraint;
```

删除父表,并且级联删除子表的外键约束

Sequence:

数据库提供的产生一系列唯一值的对象,用来生成主键值;

它是数据库里独立的对象,不依赖任何表

1)create sequence seq1

```
increment by 1     //每次递增 1
start with 100              //从 100 开始计数
maxvalue 99999          //最大值为 99999
nocycle             //无循环
nocache;       //没有缓冲

create sequence sql_seq
    increment by 1
    nocycle;
```

2) nextval:

如果有 cache 取出 cache 下一个可用的值

如果没有 cache,取出序列下一个可用的值

```
insert into    sql_users(id,name,passwd,phone,email)
 values(       sql_seq.nextval,'wangwu','123456abc',
12321,'kdfdfk@gmail.com');

select seq2.nextval
from dual;
```

currval:当前的连接最后一次使用序列的值

```
select seq1.currval
from dual;
```

last_number(column):序列下一个可用的值

```
select last_number
from user_sequences
where sequence_name='SEQ2';
```

3)改序列

```
alter sequence seq1
increment by     maxval     minval     cycle      cache
```

序列的起始值不可能改变

4)删除序列

```
drop sequence seq1;
```

5) insert   insert insert rollback;

只能朝前走

==========

view:

起了名字的查询语句

1) create view emp_v1

```
as    select id,last_name,dept_id
      from s_emp;

select * from emp_v1;
```

简化 select 语句的复杂度

限制数据的显示

节省空间

2) 复杂视图:里面包含了函数,连接,组函数
不能应用 DML(增删改),只能查询
简单视图:

```
create view emp_v3
(dept_id,asalary)
as
    select dept_id,avg(salary)
    from s_emp
    group by dept_id;

update emp_v3
set asalary=1111
where dept_id=41;     ---->error
```

3)create view emp_v4
as
select id,last_name,dept_id
from s_emp
where dept_id=41
with check option constraint emp_v4_ck; // 表 示 该 视 图（emp_v4）里的数据无法删除或改变，但可以添加数据

创建数据库用户：

建立表空间

```
create tablespace si datafile '/u/oradata/en73/si.dat' size 70M online
    default storage ( pctincrease 2 );
```

建立用户

```
create user zhangwf identified by pwd default tablespace si temporary tablespace si;
```

给用户授权

```
grant dba, connect, resource, create table to si;
```

建立表在用户 SI 上

```
create table si.tbname(col1 int,col2 varchar2(10));
```

修改密码：alter user zhangwf identified by newpwd

```
create user username identified by password
default tablespace users
temprory tablespace temp;

alter user username identified by newpassword;

grant resource,connect to username;
grant create view to cksd0804;

revoke create view from cksd0804;
```

至少掌握数据库以下四点：
1)crud（增删改查）
2)create table（建表）
3)transaction（事务）
4)create sequence //sequence:序列

JDBC 跟下面有很大的联系：
1 interface
2 try catch finally
3 thread
4 static{}
5 socket (url)
6 encapsulation inheritance poly... javabean
7 collection map

考点：行列转换， 重复数据的过滤，索引的建立

JDBC 笔记

第一天：

OracleDriver{
public void connectOracle(){
.....
} }
DB2Driver{
public void connectDB2(int i){}
}
telnet 192.168.0.200
Unix,Linux: echo $ORACLE_SID 在 Unix 和 Linux 下查看连接的哪个 ORACLE
windows: net start 在 windows 下查看连接的哪个 ORACLE
username
passwd
DriverManager.getConnection();
class OracleDriver implements Driver{ //OracleDriver 类的大体实现
static{
OracleDriver d=new OracleDriver();
DriverManager.addDriver(d);
}
}
class DriverManager{ //DriverManager 类的大体实现
private static List<Driver> dri=new ArrayList<Driver>();
public static void addDriver(Driver d){
.....
dri.add(d);
}
public static Connection getConnection(){
for(Driver d:dri){
if(d!=null) return d.getconnect();
}
} }
boolean execute(sql) ---> all
boolean 有没有 ResultSet 返回
true:有 false:没有
rs=stm.getResultSet();
ResultSet executeQuery(sql){}-----> select
int executeUpdate(sql){} -----> insert update delete
运行程序可能出现的下面几种异常：
1 Class not found
-------> ojdbc14.jar
2 No suitable Driver
----->url
3 Socket Exception
ping 200
4 SQLException ----> sql
table: jdbc_users
sequence: jdbc_users_seq
driver=oracle.jdbc.driver.OracleDriver
url=jdbc:oracle:thin:@192.168.0.200:1521:oradb10g
username=cksd0804
password=cksd0804
BufferedReader
1）String str=br.readLine();
2）String[] s=str.split("=");
3）if(s[0].equals("driver"){}
以上三步相当于以下两步：
Properties info=new Properties();
info.load(Inputstream); //表示（1）加载输入流，（2）读取文本内容，（3）解析文本文件，//（4）再存入 Properties 集合
.properties ------> Map<String,String>
------>Properties extends Hashtable<String,String>
JdbcUtil.class.getResourceAsStream("/com/kettas/jdbc/cfg/config.properties"); //为虚拟路径，第一个/代表 classpath
/com/kettas/jdbc/cfg/config.properties
/<====>classpath
/home/java/jdbc/..... //全路径
Student s=new Student();
第一次加载类时，JVM 加载类的顺序：
1. search file ---->Student.class 通过 classpath 寻找类文件
path---->command
classpath--->.class
2. read Student.class 读取类文件
3. Class c----> 获得 Student 类详细信息（类对象）
Student s2=new Student();
.class----> jdbc.jar--->user--->classpath
1 分散代码集中管理（封装）
2 变化的内容写进文件（配置文件）
配置文件一般格试：config.xml 和 config.properties
3 .properties （是一种特殊的文本文件）
4 Properties (是 Hashtable 的子类)
5 ExceptionInInitializerError
6 Class.getResourceAsStream(虚拟路径)

1. jdbc 六个步骤

1)注册 Driver；
Class.forName("oracle.jdbc.driver.OracleDriver");
2)获得连接
String url="jdbc:oracle:thin:@192.168.0.200:1521:oradb10g";
Connection conn=DriverManager.getConnection(url,"用户","密码");
3)创建 Statement，stm=conn.createStatement();
4)执行 sql， stm.executeUpdate(sql);
5)select--->处理结果集
ResultSet rs=stm.executeQuery(sql);
while(rs.next()){
System.out.println(rs.getInt(1)+"------"+rs.getString(2));
}
6)释放资源(rs,stm,conn)
if(rs!=null) try{ rs.close();} catch(Exception ee){}
if(stm!=null) try{ stm.close();} catch(Exception ee){}
if(conn!=null) try{ conn.close();} catch(Exception ee){}

2. 注册 Driver 的三种方式

1)Class.forName("oracle.jdbc.driver.OracleDriver");
2)Driver d=new oracle.jdbc.driver.OracleDriver();
DriverManager.registDriver(d);
3)程序里没有指定
java-Djdbc.drivers=oracle.jdbc.driver.OracleDriver classname

3. ResultSet 遍历

1) next()---->boolean
2) get***(int) get***(columnname)
eg: getString("name");
开始时指针指向第一行的上一行,最后指针指向最后一行的下一行

4. 三种 execute 方法的区别

1)stm.execute(sql) all boolean(ResultSet)(返回布尔型，判断是否有结果集)
2)stm.executeQuery(String selectsql) --->ResultSet（返回结果集，sql 是查询语句）
3)stm.executeUpdate(sql) --->int(db row) （返回 int，判断改变的行数，一般执行，update，delete，insert）
---->delete update insert

Statement 和 PreparedStatement
Statement 是逐条发送语句（可以执行多条语句），PreparedStatement 是先存储 sql 再一起发送（在 sql 需要设值的时候，效率要高，但只能执行一条语句）例子：
String sql=
"insert into jdbc_users(id,name) values(users_seq.nextval,?)";

```
pstm=conn.prepareStatement(sql);
pstm.setString(1, names[i]);//1 代表是提几个问号，后面是设值
pstm.executeUpdate();一起提交
```

5.JdbcUtil 类

1)分散代码集中管理(封装)

2)经常变化的写进文件(配置文件)

config.xml　　config.properties

3)Properties extends Hashtable<String,String>

专门处理 properties 文件,提供 load(InputStream is)

4)在程序加载时读一次文件,所以读文件的代码写在静态代码块,里面只能抛一种类型的错误 ExceptionInInitializerError

5)利用 jvm 的类加载器读字节码文件的功能,可以获得输入流,

InputStream is=JdbcUtil.class

.getResourceAsStream(虚拟路径);

绝对路径:/是真是的目录里面的根

虚拟路径:/是 classpath

例子：

```
private static Properties info=new Properties();
        static{          try {
InputStream is=JdbcUtil.class.getResourceAsStream(
"/com/kettas/jdbc/cfg/config.properties");
        info.load(is);
         is.close();
                } catch (Exception e) {
                        throw    new ExceptionInInitializerError(e);
                }
        }
```

线程 ThreadLocal　每一个线程对象创建好以后，JVM 会为其分配一块内存空间用来存放当前线程对象独占的数据，（一个线程对象和另一个独占的数据（对象）绑定（如：(tl , conn)代表某一线程的独占数据 ））空间以 map 形式存放独占数据，相当于 Map 集合里的健对象和值对象

每个线程的独占数据不共享，即：健对象（tl）和值对象（connection）不共享：Map<ThreadLocal,Object>

注意：ThreadLocal 无法直接获取 Map 对象，操作 Map 只能**通过 ThreadLocal 的 set 和 get 方法**

第二天：==========

Statement 和　PreparedStatement

1)建立

```
stm=conn.createStatement();
pstm=conn.prepareStatement(sql);
```

2)执行

```
stm.execute(sql);     包含数据的完整的 sql 命令
pstm.execute();       数据
```

3)stm 可以执行多条 sql

pstm 只能执行一条 sql

4)使用 pstm 的环境

a.执行同构 sql,用 pstm 效率高

执行异构 sql,用 stm 效率高

b.构成 sql 语句时需要外部的变量值,用 pstm

dao:

```
public void update(Account a){
    sql=update account set username=?,passwd=?,
          personid=?,balance=?;
    pstm=conn.....
    pstm.setString(1,a.getName());
}
```

==========

Transaction: 一组不可再分的 sql 命令,根具体业务有关

```
private static final ThreadLocal<Connection> tl
                        =new ThreadLocal<Connection>();
public static Connection    getConnection() throws Exception{
  Connection conn=tl.get();//用 get 获取当前线程的连接对象
      if(conn==null){
      Class.forName(info.getProperty("driver"));
      conn=DriverManager.getConnection(info.getProperty("url
"),
      info.getProperty("username"),info.getProperty("password
"));
      tl.set(conn);//用 set 把连接对象放在当前线程中
      }       return conn;   }
```

线程 ThreadLocal 的大致实现过程

```
public class ThreadLocal<T>{
    public T get(){
        Map m=Thread.currentThread().getMap();
        return m.get(this);
                    }
    public void set(T t){
        Map m=Thread.currentThread().getMap();
        m.put(this,t);
            }
     }

    ThreadLocal<Integer>   ta=new   ThreadLocal<Integer>()
//例子
    ta.set(new Integer(3));
```

7. Driver 方展的四个阶段

1)Jdbc-odbc （桥连接）　　2)db client（客户端 api）

3)base net pure java （网络服务器）　4)pure java native（纯客户端）

7. 写程序,向 jdbc_users 表插入五条记录

```
String[] names={"mary","tom","anna","jerry","george"};
```

==========

A:原子性　：事务要么一起成功，要么一起失败，如：银行转帐

C:一致性　：如：存钱时，存 5000，那么在数据库里的钱就要有 5000，不能多也不能少

I:隔离性　：连接 db 的两个线程操作 db 时，要操作个自的元组，不能同时操作同一个元组

D:持久性--->commit,rollback ，就是让更改后的数据要么写加 db 要么不写进 db；

手动划分事务的边界，一般写在业务层（biz）的代码块里。

划分事物边界的时候一般在要划分的地方加 commit 和 rollback；

```
conn.setAutoCommit(false);
    新事物开始
conn.commit();
    旧事务的结束,新事务的开始
conn.rollback();
    旧事务的结束,新事务的开始
```

没有手动划分,采用的是默认事务提交策略：

一条 sql 一个事务

Connection 是线程不安全的,不能在线程间共享连接（connection）用 ThreadLocal 实现安全的连接

threada　save　withdraw　transfer

threadb save withdraw
1.common: 5 个 Connection
2.singleton: 1 个 Connection
3.一个线程内部共享连接: 2 个 Connection
t1.start();
t2.start();
class ThreadA extends Thread{
private Connection conn1=null;
public void run(){
conn1=JdbcUtil.getConnection();
save(10000,conn1);
withdraw();
}
public Connection getConn(){return this.conn1;}
}
class Bank{
public void save(double balance){
conn2=Thread.currentThread().getConn();
sql="update account set balance=?";
pstm=conn.prepareStatement(sql);
//设置上面问号(？)，如 pstm.setDouble(int parameterIndex, double x);
pstm.set***;
pstm.executeUpdate();
}
public void withdraw(double balance){
conn2=Thread.currentThread().getConn();
sql="update account set balance=?";
pstm=conn.prepareStatement(sql);
pstm.set***;
pstm.executeUpdate();
}
}

save ----> update insert
withdraw----> update insert
transfer ---> update insert
update insert
update() insert()
save---->update() insert();

dao: 1)对业务层屏蔽底层数据库设计的细节 ，只有功能被业务层调用
2)o-r mapping：就是把对像自动的保存到数据库的表中

第三天:==========
1 PreparedStatement
Statement
stm=conn.createStatement();
stm.execute(sql);
String sql="insert into table value(?,?,?);
pstm=conn.prepareStatement(sql);
pstm.set***(int,value);
pstm.execute();
pstm 同构 sql,sql 语句需要外界变量值
stm 异构 sql
2 Transaction
sqls,跟具体业务有关
ACID
biz:
try{
conn.setAutoCommit(false);
conn.commit();
}catch(Exception e){
conn.rollback();
}
3 ThreadLocal
1)将 Connection 做成 Thread 的属性
通过参数的传递,保证一个线程一个连接
2)Thread t1-----> map(线程的独占的数据)
Map: 主键 ThreadLocal,无法直接获得 Map 对象
操作 map 只能通过 ThreadLocal 的 set get 操作
eg:让 Thread 对象独占一个 Integer 对象
Integer id=new Integer(3);
ThreadLocal<Integer> tl=new ThreadLocal<Integer>();
tl.set(id);
4 Layer
1) app: showinterface
业务判断
==========
save
transfer
showResultSet
---
view ---> showinterface
biz----> 业务处理
1)业务判断
2)业务处理
==========
account.balance+=money;
3) Connection conn
stm.execute(sql);
commit();
4)return result --->view
save(): update insert
withdraw(): update insert
transfer(): update insert
update insert
update 4 insert 4
view---->showinterface
biz----> 业务处理
1)业务判断 如：查看密码是否正确，或余额是否不足。
2)业务处理如：密码不正确或正确的处理，余额足时的操作
account.balance+=money;
3) 封装对象 如：用户注册
4)用对象调用 dao 里的方法存贮数据
5）提交事物
dao:----->data access object
1)完成对某一张的增删改查
update(): update
insert(): insert
delete(): delete
query(): select
2)接收从业务层传来的数据(对象),将其存进 db
从 db 里面查数据(零散),封装对象,返回给业务层
object----relation mapping
save(): update() insert()
withdraw: update() insert()
transfer: update() insert();
update:1 insert: 1

1 javabean--->Student
class Student{
private Integer id;
private String name;
private String sex;
private String address;
private int age;
private String studentId;
public Student(String name,String sex,String address,int age,String studentId){
.....
} }
2 create table jdbc_student
(id number(7) primary key,

```
        name varchar2(15) not null,
        sex varchar2(3) check(sex in('f','m')),
        age number(3),
        address varchar2(20),
        studentid varchar2(15) not null unique
      );
  3 class StudentDao{
        public void insertStudents(Set<Student> stus) throws
Exception{}
        public void insert(Student s) throws Exception{
            Connection conn=null;
            PreparedStatement pstm=null;
            try{
               conn=JdbcUtil.getConnection();
               String sql="insert into jdbc_student"
                  +" values(jdbc_student_seq.nextval,?,?,?,?...)";
               pstm=conn.prepareStatement(sql);
               pstm.set***();
               pstm.execute();
               sql="select jdbc_student_seq.currval from dual";
               s.setId(?);
               }finally{
                JdbcUtil.release(...);
            } }
        public void update(Student s) throws Exception{
             String sql="update jdbc_student set
name=?,age=?,address=?,sex=?,studentId=? where id=?";
        }
        public void delete(Student s) throws Exception{
            delete from jdbc_student where studentId=?
        }
        public Student queryById(Integer id) throws
Exception{}
        public Student queryByStudentId(String studentId)
throws Exception{}
        public Set<Student> queryByName(String name) throws
Exception{
      String sql="select * from jdbc_student where name=?";
               pstm=conn.prepareStatement(sql);
               pstm.setString(1,name);
               rs=pstm.executeQuery();
               Set<Student> stu=new HashSet<Student>();
               while(rs.next()){
                   Student s=new Student();
                   s.setId();
                   s.setName();
                   .....
                   stu.add(s);
               }
               return stu;
          }
      }
  4 class StudentBiz{
              public void registStudent(String name,String
      sex,int age,String address,String studentId) throws
      Exception{
            1. 判断      如：判断密码是否正确
            2. 封装对象        如：新建对象
               3. dao.insert(s);
            4. 事务处理        conn.commit          或
conn.rollback;
        }
        public void dropStudent(String studentId)...
        public void queryStudent(String studentId)...
        public void updateStudent(String name,String
address,String oldstudentId,String newStudentId)....
    }
    com.kettas.student
      biz
        |--- StudentBiz
              impl
                |---StudentBizImpl
      entity
      dao
      |--- StudentDao
            impl
              |---StudentDaoImpl
      config
      util
      excp
      view
      test
      sql

  Repeatable read
     select *
     from s_emp for update;
  Serializable

  备份数据库提高效率策略:
   1) 手动控制事务的大小,节约了 db 服务器的内存
      减少了回滚带来损失
   2) 事务结束时写日志,减少了查找失败事务的时间
   3) 删除备份数据库表里的约束,减少数据进入时验证的时
间
      降低数据库服务器维护索引的资源
   4) 降低数据库的隔离级别,设置到 read uncommited
   5) 插入时选择 PreparedStatement,执行同构 sql,减少编译
      sql 命令的时间
   6)插入时使用批处理，减少网络传输时间。
```

第四天：==========

```
Statement stm=conn.createStatement();

stm2=conn.createStatement(int value1,int value2);
   value1:是否可滚动
       ResultSet.TYPE_SCROLL_INSENCITIVE
   value2:是否可更新
       ResultSet.CONCUR_UPDABLE
 rs=stm2.executeQuery();
pstm=conn.prepareStatement(sql,int,int);
 数据库表数据的分页
 1.db--sql
    子查询和 Rownum
 2.jdbc---->可滚动的结果集
    String sql="select * from s_emp order by start_date";
    rs=stm.execute(sql);
  ======================================
 int num=1;
   rs.absolute(5);
   while(rs.next()){
       if(num>5) return;
       .........
   }
      可用 addBatch();方法添加批处理
   int[] a=stm.executeBatch(); //执行批处理
      批处理里放的是增删改语句，可以减少网络传输时间；
   stm 批处理里缓存的是 sql 命令，如：stm.addBatch(sql);
   pstm 批处理里缓存的是数据，如：pstm.addBatch();
   Object----->db
处理大数据的类型
1，Blob------》二进制的大对象（电影，图片，压缩包）
2，clob-------》字符的大对象（文本小说等）
以 Blob 为例
第一步：用 empty_blob 占一个空位置
```

```
        String sql="insert into largetable(id,name,largefile)"
                    +" values(3,'file3',empty_blob())";
                stm.executeUpdate(sql);
第二步：查出位置
            sql="select largefile from largetable where id=3";
                rs=stm.executeQuery(sql);
第三步：读文件然后插入数据库
    if(rs.next()){
        Blob blo=rs.getBlob(1);   //获得结果集的列值
                    //强转成 Blob 对象这样才有输出流
         oracle.sql.BLOB bo=(oracle.sql.BLOB)blo;
        OutputStream os=bo.getBinaryOutputStream();
      InputStream is=new FileInputStream(filename);//获得文件
                                                的输入流
        byte[] b=new byte[1024];   每次读文件的字节数
        int len=0,num=0;
        while(true){
        len=is.read(b);   每次读的自己长度一个数组
          if(len<=0) break;
              os.write(b,0,len);  向数据库写文件
              if(++num%10==0){
                System.out.println("10 k    ok");
              }}
        is.close();
        os.close();
        }        conn.commit();

   （类）class(entity)-------table
   （对象）object(persist object)--------row
   （属性）field------column(oid-----pk)
  relation---------> pk    fk
    class Student{
        private Integer oid;
        String name;
        int age;
        String sex;
        public Student(String name,int age,String sex){
                }
    }
    Student s1=new Student("zhangsan",18,"male");
    Student s2=new Student("lisi",19,"male");
    Student s3=new Student("zhangsan",20,"male");
```

各种关系表的建立

one-to-one(Car ---- License)，一对一：互相保留对方的引用，既互相都以对方为属性。

```
 1.java
    class Car{
        //oid
        private Integer cid;      //与数据库对应的 cid 不用写进
构造方法

        // 业务属性
        private String manufactory;
        private String productType;

        //关系属性：其它对象做为该对象的属性，关系属性不
用写进构造方法里
        private License license;
        // get 和   set
        // 有参和无参的构造方法
        public Car(String manufactory, String productType,
License license) {
                super();
                this.manufactory = manufactory;
                this.productType = productType;
        }
          public void addLicense(License license){
             this.license=license;
             license.addCar(this);
          }
          public void removeLicense(){
              this.license.removeCar();
              this.license=null;
          }
    }
    class License{
        private Integer lid;
        private String serialNum;
        private String LicenseType;
        private Car car;
        // get 和   set
        // 有参和无参的构造方法
        public void addCar(Car car){
            this.car=car;
        }
        public void removeCar(){
            this.car=null;
        }
    }
    //一对一的数据库建立
 2. db
    create table jdbc_car
    ( cid number(7) primary key,
      manufactory varchar2(15) not null,
      producttype varchar2(15) not null,
    );
    create table jdbc_license
    ( lid number(7) primary key,
      serialnum varchar2(15) not null,
      licensetype varchar2(15) not null,
      car_id number(7) references jdbc_car(cid) unique
    );
3. dao( CarDao--->jdbc_car , LicenseDao--->jdbc_license)
       //当一个程序需要用到两个数据库时，就要建立两个数
据访问层（dao）
      class CarDao{
        public void insertCar(Car c) throws Exception{
            1.insert into cartable
            2.c.setCid(?);
            3.insert into license
            4.l.setLid(?);
        }
        public void deleteCar1(Car c) throws Exception{
            1.delete from license
            2.delete from car
            3.c.setOid(null);
            4.license.setLid(null);
            5.c.removeLicense(l);
        }
        public void deleteCar(Car c) throws Exception{
            1.update license   set   car_id=null
            2.delete from car
            3.c.setOid(null);
            4.license.setLid(null);
            5.c.removeLicense(l);
        }
        public void updateCar(Car c) throws Exception{
              1. update car table;
              2. update license table set car_id=null;
        }
        public Car queryByCid(Integer cid) throws Exception{
          db-->1.select car table
                2.select license table
          内存--> 3. car.addLicense(license);
          4.return car;
                    }
    }
        main(){
```

```
            Car c=queryByCarid(222);
            c.addLicense(license);
            dao.update(c);
                    }
```

1. 维护数据库
2. 维护内存中的 oid （insert delete）
3. 维护关系（主要对象）

one-to-one :

弱耦合: fk+uk fk null

强关联: pk+fk fk not null

one-to-many(Company----Employee)

1. java

```
class Company{
    private Integer cid;
    private String cname;
    private String address;
    private Set<Employee> employees=new
HashSet<Employee>();
}
class Employee{
    private Integer eid;
    private String ename;
    private Date birthday;    //   java.sql.Date
    private int age;
    private Company company;
}
//一对多的数据库建立
```

2. db

```
create table jdbc_company
( cid number(7) primary key,
  cname varchar2(15) not null,
  address varchar2(15)
 );
create table jdbc_employee
( eid number(7) primary key,
  ename varchar2(15) not null,
  birthday date ,
  age number(3),
  company_id number(7) references jdbc_company(cid)
);
```

理性思维，一步一步的往下做就 OK 了：

3 dao

```
insert(Object o){
  1. insert object
  2. object oid
  3. relation
}
delete(Object o){
  1. relation
  2. delete object
  3. object oid
}
update(Object o){
  1. update object field
  2. update relation( option )
}
Object query(value){
  1. query object
  2. query relation
}
```

---

many-to-many (Course <----> Student)

1. java

```
class Student{
    private Integer sid;
    private String sname;
    private int age;
    private Set<Course> courses=new HashSet<Course>();
}
class Course{
    private Integer cid;
    private String cname;
    private int hours;
    private Set<Student> students=new HashSet<Student>();
}
```

2. db //多对多的数据库建立

```
create table jdbc_student
( sid number(7) primary key,
  sname varchar2(15) not null,
  age number(3)
);
create table relationtable
( studentid number(7) references jdbc_student(sid),
  courseid  number(7) references jdbc_course(cid),
  primary key(studentid,courseid)
);
create table jdbc_course
( cid number(7) primary key,
  cname varchar2(15) not null,
  hours number(5)
);
```

3. dao

StudentDao CourseDao

---

inheritance ---> pojo

1. java

```
class Account{
    private Integer id;
    private String name;
    private String passwd;
    private String cardId;
    private double balance;
    private String personId;
}
class SavingAccount extends Account{
    private String cardType;
    public SavingAccount(String name,String passwd,
            String cardId,double balance,
            String personId,String cardType){
        super(name,passwd,cardId,balance,personId);
        this.cardType=cardType;
    }
}
class CreditAccount extends Account{
    private double overdraft;
}
```

2. db ---> one table //一个超级大表

```
create table inheritance_account
( id number(7) primary key,
  name varchar2(15) not null,
  passwd varchar2(15) not null,
  cardid varchar2(15) not null unique,
  balance number(10,2),
  personid char(18) not null unique,
  cardtype varchar2(10),
  overdraft number(5),
  atype varchar2(3)
);
```

3. dao

```
class AccountDao{
  public void insert(Account a){
      sql= insert into inheritance_account(.....);
      pstm.setInt(1,....);
      pstm.setString(2,a.getName());
      pstm.setString(3,a.getPasswd());
      pstm.setString(4,a.getCardId());
      pstm.setDouble(5,a.getBalance());
```

```
            pstm.setString(6,a.getPersonId());
            if(a instanceof SavingAccount){
                SavingAccount sa=(SavingAccount)a;
                pstm.setString(7,sa.getCardType());
                pstm.setDouble(8,null);
                pstm.setString(9,"SA");
            }else{
               .....
            }
            a.setId(...);
        }
        public void delete(Account a){
            sql= delete from table where id=?
            pstm.setInt(1,a.getId());
            pstm.executeUpdate();
        }
        public void update(Account a){
            sql= update table set ..... where id=?
            pstm.setString(1,a.getName());
            ..............
            pstm.setString(5,a.getPersonId());
            if(a instanceof SavingAccount){
                SavingAccount sa=(SavingAccount)a;
                pstm.setString(6,sa.getCardType());
                pstm.setDouble(7,null);
            }else{
               ......
            }
            pstm.setInt(8,a.getId());
        }
        public Account queryByCardId(String cardId){
           sql=select
id,name,passwd,cardId,balance,personId,cardtype,overdraft,atype
                from table where cardid=?
            pstm.setString(1,cardId);
            rs=pstm.executeQuery();
            if(rs.next()){
                String type=rs.getString(7);
                if(type.equals("SA"){
                   SavingAccount sa=new SavingAccount();
                   sa.set****;
                }else{
                   ca.set****
                }
            }
        }
        public Account queryById(Integer id){
        }
    }
```

---

jobs:

1. license dao ----> self
2. employee dao ----> self -->update(){self relation}
3. companydao----> queryByCname()
4. cardao------> delete2(),queryByCid2()
5. 改写 bam
   1) value object
   2) dao ---> delete(Account a)
   3) data---->db
   4) biz,view 不改

1.基本六个步骤
2. Statement 和 PreparedStatement
3. JdbcUtil
   1) ThreadLocal 2) 配置文件
   3) 虚拟路径(Class.getResourcAsStream())

   1) 分散代码集中管理
   2) 变化的数据写进文件
4. Transaction
   1)API 2) ACID 3)并发问题 4)隔离级别
5. Jdbc2.0
   1) Batch
6. ormapping:
   1) dao 步骤
   2) 各种关系表的建立
7. 1)数据备份时提高效率的方法
   2)数据分页
8.Driver 的四个发展阶段

Hibernate

第一&二天：

ORM，即 Object-Relationl Mapping，它的作用是在关系型数据库和对象之间作一个映射，这样，我们在具体的操作数据库的时候，就不需要再去和复杂的 SQL 语句打交道，只要像平时操作对象一样操作它就可以了（把关系数据库的字段在内存中映射成对象的属性）

1 hibernate---->ormapping
1) CRUD 2)HQL 3)mapping file 4)性能

2 基于 hibernate 的应用步骤:
1)javabean(pojo)
2) relation db -----> create table
3)mapping file(object----table)

filename.hbm.xml ----> 跟 javabean 放在一个路径

```
<hibernate-mapping package="packagename">
  <class name="classname" table="tablename">
<id name="oidname" column="pkname" unsaved-value="null">
      <generator class="生成器 classname">
        <param name="type"> value </param>
      </generator>
    </id>
<propertyname="javaproperty"
column="dbcolumnname"></property>
  </class>
  <class>.....</class>
</hibernate-mapping>
```

4) configure file ----> classpath(src)
hibernate.cfg.xml

```
<hibernate-configuration>
  <session-factory>
    <property>
    <mapping resource="filepath">
```

5) application (base hibernate)
a. start hibernate
Configuration cfg=new Configuration().configure(...);
B . create sessionfactory
SessionFactory sf=cfg.buildSessionFactory();
c. open Session
Session sess=sf.openSession();
d. begin Transaction
Transaction ta=sess.beginTransaction();
e. persist object
sess.save(Object o); oid=null ; insert
sess.delete(Object o); oid!=null delete
sess.update(Object o); oid!=null update
Object o=sess.get(classname.class,id);
f. ta.commit()/ta.rollback()
g. release resource
(sess , sf) ----- sess
sess: 线程不安全
sf: 线程安全, 重量级的组件

3 hilo
1) create table hilotable
(value number(7));
insert into hilotable
values(1);
2) <generator class="hilo">
<param name="table">hilotable</param>
<param name="column">value</param>
</generato>

name= java name
class= java type
type= hibernate type (lower)
table , column= db name (insensitive)

----------one-to-one----------
1. javabean (双向（bi）,单向 （ui）)
2. create table (fk+uk , pk+fk)
3. mapping file
4. change configure file
5. java app
property-ref="car"
1)双向的一对一
2) 没有写 property-ref, 没有写 cascade
sess.update(car);
a. update jdbc_car set ***** where cid=?
b. update jdbc_license set car_id=? where lid=?

没有写 property-ref, 有写 cascade
sess.update(car)
a. update jdbc_car set **** where cid=?
b. update jdbc_license set car_id=? where lid=?
c. update jdbc_license set sn=?,lt=?,car_id=? where lid=?

有写 property-ref, 有写 cascade
a.update jdbc_car set **** where cid=?
b.update jdbc_license set **** where lid=?

```
class People{
  Integer id;
  String name;
  int age;
  Set<Hand> hands=new HashSet<Hand>();
}
class Hand{}
```

1. one-to-one fk+uk(单向)
2. one-to-one pk+fk(单向)
3. one-to-many (单向)
4. many-to-many (javabean table mappingfile)

第三天：==========

1. 标签: 类驱动表，真实反应 Java 里的类，即：以类为主
hibernate-mapping---->package
class ----> name table
id ----> name column unsaved-value
generator----> class
param ----> name
property----> name column unique not-null type
one-to-one----> name property-ref cascade constraint
many-to-one---> name column unique cascade
set----> name table cascade inverse
key----> column
one-to-many---> class

2 id 生成器类的别名:
sequence , hilo , sequencehilo , foreign
assigned , increment , identity(mysql) , native
3 assosiation:

==========

1) one-to-one

a. fk+uk

```
<class name="Car"....>
  <one-to-one name="license" property-ref="car"
cascade="save-update"/>
</class>
<class name="License"....>
    .....
    <many-to-one name="car" column="car_id"
unique="true" cascade="save-update"/>
</class>
```

b. pk+uk

```
<class name="Car"....>
    <id> ....</id>
    .....
    <one-to-one name="license" property-ref="car"
        cascade="all"/>
</class>
<class name="License"....>
    <id name="lid".....>
        <generator class="foreign">
            <param name="property"> car</param>
        </generator>
    </id>
    ...
    <one-to-one name="car" constrained="true"
        cascade="save-update"/>
</class>
```

2) many-to-one

```
<class name="Company"....>
    .....
    <set name="employees" table="jdbc_employee">
        <key colum="company_id"/>
        <one-to-many class="Employee"/>
    </set>
</class>
</class name="Employee"...>
    .....
    <many-to-one name="company" column="company_id"
        cascade="save-update"/>
</class>
```

当写双向级联关系时,确定只有一方维护关系,(一般多的一方维护关系)

1 one-to-one 用 property-ref

2 one-to-many many-to-many **用 inverse="ture"**

一的一方要 设 <set **(inverse="false")** casecade="save-update" />, 多的一方要设<many-to-one **inverse=true** (casecade="none") />

两边都用 true 或都用 false 都是不对的，关系最好让一方维护

---

many-to-many

1. javabean

```
class Student{
        private Integer sid;
        private String sname;
        private int age;
        private String sex;
        private Set<Course> courses=new HashSet<Course>();
    }
class Course{
    private Integer cid;
    private String cname;
        private int hours;
    private Set<Student> students=new HashSet<Student>();
}
```

2 db table

```
create table hibernate_student
();
create table hibernate_relation
();
create table hibernate_course
();
```

3 mapping file

```
<class name="Student" table="hibernate_student">
<id name="sid" column="sid"  unsaved-value="null">
</id>
<property name="…"/>
<set name="courses" table="hibernate_relation"
        cascade="save-update"
inverse="true">
    <key column="student_id"/>
    <many-to-many class="Course" column="course_id">
    </set>
    </class>
    <class name="Course" table="hibernate_course">
    <id name="cid" column="cid"  unsaved-value="null">
    </id>
    <property name="…"/>
    <set name="students" table="hibernate_relation"
                cascade="save-update">
    <key column="course_id"/>
    <many-to-many class="Student" column="student_id" />
    </class>
```

1.student and course 没有 inverse=true 就会多一次维护关系

student and course 都有 cascade

update(student)

1)update student table

2)update relation table

3)update course table

4)update relation table 就会多一次维护关系

2.student and course 有 inverse=true 就不会维护关系

student and course 都有 cascade

insert(student)

1)insert student table

2)insert course table

3.student 有 inverse=true 一方维护关系才正确

course 没有 inverse=true

student and course 都有 cascade

update(student)

1)update student table

2)update course table

3)update relation table

```
sess.beginTransaction();
sess.save(Object);
    1. insert student table
    2. stm.addBatch(sql1);
    3. insert course table
    4. stm.addBatch(sql2);
    5. insert relation table
    6. stm.addBatch(sql3);
    7. stm.executeBatch();
ta.commit;
catch(Exception e){ta.rollbacka();}
```

==========

获得线程安全的 session

1. 修改配置文件

```
<property
name="current_session_context_class">thread</property>
```

2.在 java 程序里获取 session 时

```
Session sess=sf.getCurrentSession();
```

session 特点:

1) 线程安全

2) sess 在事务结束时会自动关闭,一个事务用一个 sess,当数据量过多时，

可以用 sess.flush()；将缓存的 sql 命令发往数据库执行,执行结果保存在回滚段.

---

persist:

Session:一级 Cache (缓存 session 操作过的所有对象)

SessionFactory: 二级 Cache ，用安全的 session 时，尽量使 sf 关闭时，线程也要关闭；

(缓存生成的 sql,以及 hibernate 管理过的部分对象

Hibernate 的对象状态，即生命周期

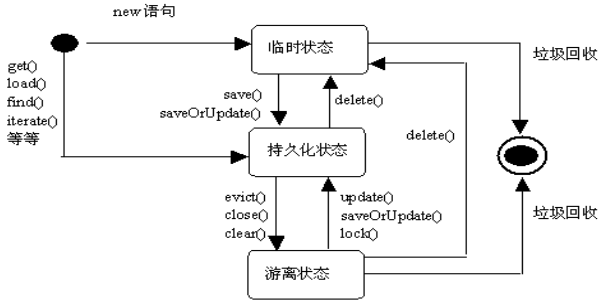


图4：对象的状态转换图

1.Transient(临时,暂时)

Student s1=new Student("zhangsan",18);

1)在数据库里没有对应的记录

2)一般情况下,没有数据库唯一标识符

特殊情况 1)assingned 临时对象，即新建（new）对象

2)sess.delete(o) 对象已被删除

3)不在 Session 里面

2.persist(持久化状态)

sess.save(s1); saveOrUpdate()

Student s2=sess.get(Student.class,123);

1)在数据库里有对应的记录

2)有数据库唯一标识符

3)在 Session 里面

3.detached(游离,脱管状态)

sess.close(); sess.clear();sess.evict()

1)在数据库里有对应的记录

2)有数据库唯一标识符

3)不在 Session 里面

Session 保留备份（两个集合）

```
class SessionImpl implements Session{
    private LinkedList  entities=new LinkedList();  //实体集合，用于存放临时对象
    private LinkedList  copy=new LinkedList();  //深度拷贝集合，跟集合的数据一样，可以减少访问 DB
    public Serializable save(Object o){
        1.Serializable pk=.....
        2.db insert
        3.entities.add(o);
        4.copy.add(o.deepcopy);
        5.return pk;
    }

public void delete(Object o){
        1.db delete
        2.entities.remove(o);
        3.copy.remove(o.deepcopy);
    }
    public void close(){
        entities.removeAll();
        copy.removeAll();
    }
    public void evict(Object o){
        entities.remove(o);
        copy.remove(o.deepcopy);
    }
}
```

sess 保留备份的作用:

优势:

1) 查询同一个对象,不用再访问数据库,

直接返回 sess 里的数据

2) 脏的检测，当检测到实体集合与深度拷贝集合里的数据不同时，会把改变的对象同步到数据库里。

```
Student s=new Student("zhangsan",18);
sess.save(s);
s.setName("lisi");
sess.close(); (sess.update(s))  -----> db s name="lisi"，在sess 关闭前会自动更新脏的数据到 db 里。

<class name="Student" table="" dynamic-update="true">
s.setName("wangwu");
sess.update(s);
1)sess=HibernateUtil.getSession();
  ta=sess.beginTransaction();
  Student s4=new Student("zhangsan",18);
  sess.save(s4);
  s4.setName("lisi");
  ----> 在 sess 没有关闭前,脏的检测起作用
  ta.commit();
  db---------> name="lisi"
2)sess=HibernateUtil.getSession();
  ta=sess.beginTransaction();
  Student s4=new Student("zhangsan",18);
  sess.save(s4);
  ta.commit();   ----> sess 关闭
  s4.setName("lisi");  ----->s4 此时已处在游离状态
  db---------> name="zhangsan"
3)sess=HibernateUtil.getSession();
  ta=sess.beginTransaction();
  Student s4=new Student("zhangsan",18);  --->临时
  sess.save(s4);                  -----> 持久
  ta.commit();                    ---->s4 游离
  ta2=HibernateUtil.beginTransaction(); --->新的 sess
  s4.setName("lisi");     ----->s4 游离
  ta2.commit;               ---> 关闭新的 sess
  db---------->name="zhangsan"
4)sess=HibernateUtil.getSession();
  ta=sess.beginTransaction();
  Student s4=new Student("zhangsan",18); --->transient
  sess.save(s4);                          --->persist
  sess.evict(s4);                         --->detached
  s4.setName("lisi");
  ta.commit();
--->sess.close
  db---------> name="zhangsan"
  缺点:不适合做大量数据的插入或者修改
  Student s=new Student("zhangsan",18); -->transient
 sess.save(s);                          -->persist
 sess.evict(s);                         -->detached
 sess.delete(s);
 detached--->persist--->transient
```

jobs:

1. m2m---> javabean table mappingfile dao biz test
2. 如何获得线程安全的 session
3. 三种状态,以及它们之间的转换
4. sess 脏的检测
5. 测试 get 和 load

第四天：==========

1 many-to-many

1) java

class Student

Set<Course> courses

class Course

Set<Student> students

2)db

hibernate_student relationtable hibernate_course

sid studentid cid cid

3)

```
<class name="Student" table="hibernate_student">
  <id name="sid..>
  <property>
  <set name="courses" table="relationtable" inverse="true"
cascade="save-update" >
    <key column="studentid" >
    <many-to-many class="Course" column="cid">
</set>
</class>
<class name="Course" table="hibernate_course" >
  <id.....>
  <property>
  <set name="students" table="relationtable"
                        cascade="save-update">
    <key column="cid">
    <many-to-many .....>
  </set>
</class>
```

---

2. hibernate---->ormapping (object-->db)

hibernate--->dao

connection---> not safe ---> ThreadLocal<Connection>

Session---->not safe---> ThreadLocal(Session)

1) ThreadLocal<Session>

2) current_session_context_class thread

Session sess=sf.getCurrentSession();

线程安全,一个事务一个 sess,事务结束会自动关闭 sess

ta=sess.beginTransaction();

Student stu=new Student();

sess.save(stu);

ta.commit();

dao: save() update() delete() get()

saveOrUpdate()

3.Persist

1)Transient:

id=null (assigned , sess.delete())---->saveOrUpdate()

not in sess

not in db----not row

Persistance

in db ---- row

id!=null

in sess

Detached

in db----row

id!=null

not in sess

2) 一级 cache(Session)----> 集合

session 有两个集合(entities,deepcopy)

a.缓存操作过的对象，加快查询速度

sess.get(Student.class,14);

get--->一级 cache---->db

b.脏的检测(关闭时)

在内存中改变对象属性:

显示调用 update():detached

隐含调用 update():Persistance（已经持久化的对象，如果被修改，就会隐含调用）

3)状态图

4 Student s=new Student();

持久化 s;

1) sess.save(s) sess.saveOrUpdate(s)

2) 利用持久化对象的扩散

```
sess=sf.getCurrentSession();
Course c=sess.get(Course.class,18);
c.getStudents().add(s);
sess.update(c);
ta.commit();
```

A--->B--->C--->D

save(A);

cascade:

a. none(default)

b. save-update

c. all (save-udpate-delete)

d. delete

e. delete-orphan

```
com1-------> delete
emp1  emp2  emp3 ----> delete-orphan
delete(com1); ---> emp1 emp2 emp3
delete(emp1);  --->emp1
```

5 查找 Student s

1) query by id (get() or load())

```
Student s=sess.get(Student.class,10);
Student s=sess.load(Student.class,10);

class SessionImpl implements Session{
    public Student get(Class c,Serializable id){
        1. sql= " select * from student where id=? ";
        2. Student o=data
        3. return o;
            }
    public Object load(Class c,Serializable id){
        Student s=c.newInstance();
        s.setId(id);
        return s;
          }
  }
```

get 和 load

a. get 时先查询 Session 的 Cache,如果 cache 里面没有,再生成 sql 语句查询数据库,默认查询所有的值

load 先在内存中生成一个只有 oid 的对象, 且返回没有查询 db,需要其他属性时再去查 db

b. get 回来的对象,所有的属性均有值

load 回来的对象只有 oid 有值,其他属性为 null

c. get---->Student

load----> SubStudent(cglib.jar)

2) 曲线查询:

```
Student s;
Course c=sess.get(Course.class,167);
for(Student s: c.getStudents()){
    System.out.println(s.getSname());
}
```

3)Query(interface)-----> Hibernate 查询语言：hql(Hibernate query language)

```
String hql="from Student s where s.sid>150";
Query q=sess.createQuery(hql);
List l=q.list();
```

batch insert/update:9999

1) sess=sf.getCurrentSession();

ta=sess.beginTransaction();

```
String hql="from Student s";
List stus=sess.createQuery(hql).list();
sess.setFlushMode(FlushMode.COMMIT);
for(int i=1;i<=stus.size();i++){
    if(i%50==0){
        sess.flush();     // batch  里缓存多少 sql 命令
          将缓存的 sql 命令发往数据库执行,执行结果保存 在回滚段
        sess.clear();
          清空一级 cache
    }
    stus.get(i).setName("hello"+i);
    sess.update(stus.get(i));
}
ta.commit();          // 回滚段的内容写回数据文件
```

1) 存放临时对象 1000 块
2) Session 里面留有 1000 个备份对象
3) hibernate 自动使用 batch,缓存 1000 条 sql

默认情况下一个事务一个 batch,flush()以后,把一个事务划分成若干个 batch

```
2) sess=sf.getCurrentSession();
   ta=sess.beginTransaction();
   String hql="update Student s set s.sname=?";
   Query q=sess.createQuery(hql);
   q.setString(1,"haha");
   q.executeUpdate();
   ta.commit();
```

总结: Session(entities deepcopy)
1. 三种状态
2. 持久化一个对象
   save saveOrUpdate
   persistance（持久保存） object ---->脏的检测
3. 查询对象:
   1) get load ----> 加快速度
   2) relation object //关系对象
   3) hql

4. batch update/insert
   1) query----> change----> update
      batch_size flush() clear()
   2) hql

```
-----------------------------------------------------
interfact Transaction{
    public Transaction beginTransaction();
    public void commit();
    public void rollback();
}
class JdbcTansaction implement Transaction{
  public Transaction beginTransaction(){
     .....
    conn.setAutoCommit(false);
  }
    public void commit(){
        conn.commit();
    }
    public void rollback(){
        conn.rollback();
    }
}
```

事务的并发控制

一、多个事务运行时的并发问题

并发问题归纳为以下几类：

1 **第一类丢失更新**：撤销一个事务时，把其他事务已经提交的更新数据覆盖。

2 赃读：一个事务读到另一个事务未提交的更新数据。

3 虚读：一个事务读到另一个事务提交的新插入的数据。

4 不可重复读：一个事务读到另一个事务已经提交的更新数据。事务 A 对统一数据重复读两次却得到不同的结果，有可能在 A 读取数据的时候事务 B 对统一数据做了修改

5 第二类丢失更新：这是不可重复读中的特例，一个事务付给另一个事务已提交的更新事务。

二数据库系统提供了四种事务隔离级别供用户选择：

A.Serializable（串行化）：一个事务在执行过程中完全看不到其他事务对数据库所做的更新。

B.Repeatable Read（可重复读）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，但是不能看到其他其他事务对已有记录的更新。

C.Read Commited（读已提交数据）：一个事务在执行过程中可以看到其他事务已经提交的新插入的记录，而且能看到其他事务已经提交的对已有记录的更新。

D.Read Uncommitted（读未提交数据）：一个事务在执行过程中可以拷打其他事务没有提交的新插入的记录，而且能看到其他事务没有提交的对已有记录的更新。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大。对于多数应用程序，可以有优先考虑把数据库系统的隔离级别设为 Read Commited，它能够避免脏读，而且具有较好的并发性能。尽管它会导致不可重复读、虚读和第二类丢失更新这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制

在 hibernate.cfg.xml 中设置隔离级别：

```
<session-factory>
  <!-- 设置隔离层次，并发,缺省时 Read Committed: 2 -->
    <property name="connection.isolation">2</property>
<!-- 配置事务实现类 -->
  <property name="transaction.factory_class">
org.hibernate.transaction.JDBCTransactionFactory </property>
    </session-factory>
```

悲观锁: 在查询时加

1. Student s=sess.get(Student.class,id**,LockMode.UPGRADE**);

五种模式

LockMode.NONE: 查询时先在 cache（缓存）里找,如果没有，再到 db 里加载 无锁机制。

LockMode.READ: 不管 cache 有没有,都查询数据库，Hibernate 在读取记录的时候会自动获取。

LockMode.UPGRADE: 不管 cache 有没有,都查询数据库,并且 对查询的数据加锁,如果锁被其他事务拿走, 当前事务会一直等到加上锁为止. 利用数据库的 for update 子句加锁。

LockMode.UPGRADE_NOWAIT: 不管 cache 有没有,都查询数据库,并且对查询的数据加锁, 如果锁被其他 事务拿走,当前事务会立刻返回. ，hiberna Oracle 的特定实现，利用 Oracle 的 for update nowait 子句实现加锁

LockMode.WRITE:。在做 insert,update，delete 会自动使用模式.内部使用

2． 以下情况用悲观锁：查询数据时，就给数据加锁
1)数据资源被多个事务并发访问的机会很大
2)修改数据所需时间非常短

乐观锁，大多是基于数据版本 （Version）记录机制实现

1 在 javabean 中加上 version 的属性提供 set 和 get 方法
2，在数据库表上加上 vershion 列
3 在映射文件的<id></id>后加上<version>
<version name=”version” column=”version”/>

实现原理：读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提 交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据 版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。

第五天：==========

Inheritance:

1. javabean
   Account ---> SavingAccount CreditAccount
2. db
   1)one table
   2)one class one table
   3)subclass---->table
3. **mapping file**
   1) one table

```
<class  name="Account" table="inheritance_table"
                discriminator-value="AC" >
        <id>
        <discriminator column="atype" type="string" length="3"/> 在数据库加一列 atype 用来分开各种不同的父类和子类
        <property>
        <subclass name="SavingAccount" discriminator-value="SA">
          <property name="">          子类的特别属性 discriminator-value="SA"默认值是 SA
        </subclass>
        <subclass name="CreditAccount" discriminator-value="CA">
          <property name="">
        </subclass>
    </class>
```

2) one class one table 每个类都有一个表 子类的主键同时是父类的外键 即父表的主键也子表主键一样

```
<joined-subclass name="SavingAccount" table="hibernate_savingaccount">
        <key column="id"></key>
        <property name="cardType"></property>
    </joined-subclass>
```

3) one subclass one table 每个子类一张表

```
<class name="Account" abstract="true">      抽象父类不能 new 对象，父类没有对应的表
    <id name="id" column="id" unsaved-value="null">
      <generator class="sequence">
        <param name="sequence">account_seq</param>
      </generator>
    </id>
    <property name="name"></property>     父类的属性
    <union-subclass name="SavingAccount" table="hibernate_savingaccount">
      <property name="cardType"></property>
    </union-subclass>
    <union-subclass name="CreditAccount" table="hibernate_creditaccount">
      <property name="overdraft"></property>
    </union-subclass>

</class>
```

4. change config file
5. dao

```
public void insertAccount(Account a){
    sess.save(a);
}
```

6. biz
7. view
8. test

```
class Person{
   private Integer id;     ------> oid
   private String name;    ------>业务属性
   private int age;
   private Head head;      ---->业务属性
   private Set<Hand> hands=new HashSet<Hand>();   ---->业务属性
   private Computer computer; ---->关系属性
   private Set<Book> books=new HashSet<Book>();---->关系属性
}
class Hand{
   private String finger;
}
class Book{
   private Integer id;
   private String bname;
```

```
}
class Head{
    private String eye;
}
class Computer{
   private Integer id;
   private String Screen;
   private Person person;
}
```

以下标签和属性要结合着映射文件看和理解 hibernate

base（基础）:

hibernate-mapping：表示映射文件的根标签 --> package：表示要映射的类所在的包。

class：该标签下包含映射的类和 DB 里的表--> name：要做映射的类　table：对应 DB 里的表 ；
optimistic-lock：经常与 version 连用，作用是一个数据被多个事务访问时，
如果其中一个事务修改了 DB 数据，那么其它事务将不能得到提交数据，只能重新访问 DB；
1 none 无乐观锁 2 version 通过版本机制实现乐观锁
3 dirty 通过检查发生变动过的属性实现乐观锁 4 all 通过检查所有属性实现乐
dynimic-update：默认值为"false",当值为"true"时表示更新被修改的列，而不全表更新；
abstract：表示要映射的类是抽象类，不能 new 对象，只能用他的实现类，否则 hibernate 会报错；
lazy：表示延时加载，即用到 DB 里的数据时才去 DB 里加载，默认为"true"。

id：---> name：表示在类里的对应的属性名； column：表示在 DB 表里对应的列名；
unsaved-value：未存进 DB 前或当 id 为临时对象时，默认为 null;
generator:表示生成器标签 ---> class：是用哪种生成器，在 Oracle 里是系列（sequence）。

param：参数标签 ---> name：指代用什么配置或初始化生成器的值。

property：该标签用于属性与表里的映射----> name：对应类里的属性名；
column：对应 DB 表里的列名； unique：表示 DB 表里的列的值是否唯一；
length：表示 DB 里那一列的数据存放长度； type：DB 里那一列的数据存放类型；
not null：DB 里那一列的数据是否可以为空。
version：该标签经常与乐观锁连用，表示其它事务是否修改 DB 里的值--->
name：在类里对应的属性名； column：DB 里对应的列名。

assosiation（关联）:在类里是一个类的属性是别一个类的类型，在 DB 里是多表关联。
one-to-one--> name：表示在类里对应的属性名；constrained：表示该属性必须有值；
property-ref：表示只维护该表里的属性列，而不维护以该表的 ID 为外健的其它表；
cascade：级联操作，即级联关系属性，当值为"save-update"时表示更新 A 表时，也更新与之对应的 B 表，（cascade 在 A 表）。
many-to-one-->name：表示在类里对应的属性名； column：表示在 DB 表里对应的列名； lazy:表示当用到该类的对象（A）的属性（name）时，
会自动去 DB 里加载这个属性对应的行，并用这行生成对象保存在该类的对象（A）的属性里。
cascade unique
one-to-many--> class：在类里对应哪个类；
set：该标签表示类里的属性是 set 集合----> name：表示 set 集合在该类里的名称；
table：表示 set 集合里的数据在 DB 里对应哪个表；
cascade：表示更新 cascade 所在的表时，级联更新 table 对应的表（或 name 属性所在的表）；
inverse：表示只维护该表里的属性列，而不维护以该表的 ID 为外健的其它表；

key：该标签表示外健 ----> column：表示表里的外健名；

many-to-many----> class：在类里对应的类；column：在表里对应的外健。

inheritance（继承）:
subclass：用于多个子类做成一张表时 ---> name：表示在类里对应的子类名；
discriminator-value：在 DB 表里对应的列的值；
discriminator：该标签用于描述 discriminator-value ---> column:表示该值在哪一列 ；
type：在 DB 里是什么样的数据类型； length：在 DB 里以什么样的长度存储的。

joined-subclass：一个类一个表--->name：表示在类里对应的子类名； table：在数据库里对应哪个表；
union-subclass：一个子类一个表---->name table

valuetype（值类型）：没有 oid 的 Java 对象（某个对象的业务属性，离开该对象无法存在）
component：该标签表示该类(comA)是另一个类(A)的某一部分（属性）--->name：表示 comA 类在 A 类里的名字；
parent：该标签表示属于哪个类---> name：在 comA 类里的属性名；
set-->.....
element；值元素（如在 map 标签里表示值对象）---> type：在对象在 DB 里存的数据类型； column：在 DB 里对应哪一列；

list ---> name table

list-index：表示 list 集合的下标（在数据库中自动生成）---> column:对应 DB 里 table 值的哪一列
idbag:hibernate 里特有的一个集合标签，可重复无序。----> name table
collection-id：表示给一重复数据的区别---> column：对应 DB 表里的哪一列；
composit-element：该标签表示里面的数据是某一集合的组成元素---> class：表示该集合里的元素是属于哪一个类的；
map----> name：表示 map 集合在该类里的名称； table：表示 map 集合里存的数据在 DB 里对应哪个表；
map-key：map 集合里的健对象---> column：表示健对象对应 table 表里的一列； type：表示健对象的类型；
query

第六天：==========

1.optimistic-lock

1) old: <class optimistic-lock="all/dirty/version"
dynimic-update="true">
2) new: a. javabean ---->version(int)
b. table----->version(number)
c. mapping file---> <version> (紧跟在<id>)

2. 1)javabean ---> extends
2)db ----> **a. one table**
(pk, 父类共性 ，各个子类特性 ，类型)
b. one class one table
父表(pk 共性)
子表(pk-fk 特性)
c. one subclass one table
表(pk 共性 特性)
表 (pk 共性 特性)
3) mapping file
a. one table
<class>
<discriminator column="colname" type="string">
<subclass name="subname" discrinator-value="SA">
</subclass>
</class>
b. one class one table
<joined-subclass name="subname" table="subtablename">
<key columm="">
<property>
</joined-subclass>
c. one subclass one table
1) <class> <class>
2) <class name="fatherclassname">
共性
<union-subclass name="" table="">
特性
</union-subclass>
</class>

3. valuetype 值对象 (没有 oid 的 java 对象)

1) one value ---> component （两个类一张表）
db: pk 简单类型 v1property1 v1property2
mapping: <component name="javavaluetype name">
<parent>
<property>
</component>

2) one collection （两个类两张表）
set: java---> Set<String>
db-----> table(id(fk) value pk(id,value))
mapping--->
<set name="setname" table="">
<key column="id">//外键
<element column="value" type="string">
list: java---> List<String>
db-----> table(id(fk) index value pk(id,index))
mapping--->
<list name="listname" table="">
<key column="id">
<list-index column="index"> 自动生成下标
<element column="value" type="string">
idbag: java---> List<String>
db-----> table(id(fk) key(pk) value)
mapping--->
<idbag name="listname" table="">
<collection-id column="key">
<generator>
<key column="id">
<element column="value" type="string">
3) one map
java---> Map<String,String>
db-----> table(id(fk) key value pk(id,key))
mapping--->
<map name="setname" table="">
<key column="id">
<map-key column="key">
<element column="value" type="string">

1.query object(Employee): Company--Employee
1) Company c=sess.get(Company.class,18);
Set<Employee> employees=c.getEmployees();
2) query oid;
Employee e=(Employee)sess.get(Employee.class,18);
Employee e=(Employee)sess.load(Employee.class,18);
3)hql
String hql="from Employee e where e.name='zhangsan'";
Query q=sess.createQuery(hql);
List<Employee> es=q.list();
mapping file:
<query name="querybyname">
<![CDATA[from Employee e where e.name='zhangsan']]>
</query>
java: Query q=sess.getNamedQuery("queryname");
4) Criteria
Criteria ct=sess.createCriteria(Employee.class);
Criterion ce1=Expression.eq("name","zhangsan");
Criterion ce2=Expression.like("address","W%");
ct.add(ce1);
ct.add(ce2);
List<Employee> es=ct.list();
5) 本地 sql
str="select e.NAME as {emp.name},e.AGE as {emp.age}
from EMPLOYEE e where e.NAME='zhangsan'";
SQLQuery sq=sess.createSQLQuery(str);
sq.addEntity("emp", Employee.class);
str=" select {e.*} from EMPLOYEE e where e.NAME='zhangsan'";
sq.addEntity("e",Employee.class);

2. 支持多态查询

public Account queryById(Integer id){
String hql="from Account a where a.id=?"
Account a=query.list().get(0);
}

3.排序(list()和 iterate()):

hql1="from Employee e order by e.eid";
Query q1=sess.createQuery(hql1);
List<Employee> es1=q1.list();
hql2="from Employee e where e.age>30";
Query q2=sess.createQuery(hql2);
List<Employee> es2=q1.list();
// select * from Employee where e.age>30; --->sql

Iterator<Employee> et=q.iterate(); --->效率高
// select e.eid from employee e where e.age>30--->sql

4. 数据分页:

1)sql--->rownum

2)jdbc---> ResultSet

3)hibernate--->

```
hql="from Employee e order by e.start_date";
Query q=sess.createQuery(hql);
q.setFirstResult(5);//从结果集 q 中的第五行开始
q.setMaxResult(5);  //得到五行
List<Employee> es=q.list();
list.size===5;
q.setFirstResult(5);
q.setMaxResult(1);
Employee e=q.uniqueResult();
```

根据 pk 查询

5.根据条件查询(where)

```
hql=" from Employee e "+"where..."
1)where e.name='zhangsan'
  where lower(e.name)='zhangsan'
2)where e.age<30
3)where e.age between 30 and 60
4)where e.age in(30,40,50)
5)where e.name like 'S%'
6)where e.salary>1000 and e.dept_id=41
  where e.salary>1000 or e.dept_id=41
7)where e.address is null
8)!= , not between and , not in   ,not like ,is not null
```

6. 参数绑定:

```
1) hql=" from Employee e where e.name=?";
   Query q=sess.createQuery(hql);
   q.setString(1,"zhangsan");
2) hql=" from Employee e where e.name=:name";
   Query q=sess.createQuery(hql);
   q.setString("name","zhangsan");
   Company c=(Company)sess.get(Company.class,18);
   hql="from Employee e where e.company=:com";
   Query q=sess.createQuery(hql);
   q.setEntity("com",c);
   public List<Student> query(String name,Integer age){
    StringBuffer sb=new StringBuffer("from Student s ");
    if(name!=null) sb.append(" where s.name=:name");
    if(age!=null && name!=null)
        sb.append(" and s.age=:age");
    if(age!=null && name==null)
        sb.append(" where s.age=:age");
    Query q=sess.createQuery(sb.toString());
    if(name!=null) q.setString("name",name);
    if(age!=null)   q.setInt("age",age);
    return q.list();
  }
  public List<Student> query2(String name,int age,String
address,String sex){
    Student s=new Student();
    s.setName(name);
    s.setAge(age);
    s.setAddress(address);
    s.setSex(sex);
    Criteria c=sess.createCriteria(Student.class);
    Example e=Example.create(s);
    c.add(e);
    return c.list();
  }
```

7. 连接:

```
sql: inner join ;   outer join ;   no equal join ; self join

hql: inner join --- inner join fetch(迫切内连接)
     left outer join -- left outer join fetch(迫切左外连接)
     right outer join
```

隐式连接

```
1)left outer join:( left join )
   hql="from Company c left join c.employees e";
   Query q=sess.createQuery(hql);
   List<Object[]> list=q.list();
  /* Company 对象关系属性没有被初始化,所有每条记录
       被封装成两个独立的对象
   */
   sql= select c.*,e.*
        from   jdbc_company c left join jdbc_employee e
        on c.cid=e.company_id;
   result:
     cid   cname   address   eid   ename   age
      1    kettas    wdk      1     mary    18
      1    kettas    wdk      2     tom     19
      1    kettas    wdk      3     ann     20
      2     IBM         sd
  ------------------------------------------------
  hql="select c from Company c left join c.employees e";
  Query q=sess.createQuery(hql);
  List<Company> list=q.list();
  sql= select c.*
       from   jdbc_company c left join jdbc_employee e
       on c.cid=e.company_id;
```

2)left join fetch:

```
  hql="from Company c left join fetch c.employees e";
  Query q=sess.createQuery(hql);
  List<Company> list=q.list();
  Set<Company> set=new HashSet<Company>(list);

  /* list 里的每一个 Company 对象,所有的属性都有值,
     包括关系属性也被初始化,并且里面存有查回来的
     Employee 对象
  */
  sql= select c.*,e.*
       from   jdbc_company c left join jdbc_employee e
       on c.cid=e.company_id;
  result:
    cid   cname   address   eid   ename   age
     1    kettas    wdk      1     mary    18
     1    kettas    wdk      2     tom     19
     1    kettas    wdk      3     ann     20
     2     IBM         sd
```

3)隐式连接: 如：写 hql 语句时，hibernate 会自动的使用 sql 对数据库进行查找。

```
   hql=" select e from Employee e
         where e.company.cname='kettas'";
   sql= select e.*
        from jdbc_employees e,jdbc_company c
        where e.company_id=c.cid   and c.cname='kettas'
```

8. 组函数

```
sql: avg()   sum()    max()   min()   count()
hql: avg()   sum()    max()   min()   count()

需求:打印出地区名字,以及该地区公司的个数
hql=" select c.address,count(*)
        from Company c
        group by c.address
        having count(*)>1 ";
hql=" select e.dept_id,avg(salary)
        from Employee e
        group by e.dept_id";
```

9 子查询:

```
1)相关: 请查询员工数量大于一的公司
  hql=select c   from Company c
      where 1<( select count(*) from c.employees e );
```

2)无关: 查询工资高于平均工资的员工对象

```
hql=select e1 from Employee e1
where e1.salary>( select avg(e2.salary) from Employee e2);
```

3)all(查询出的所有记录)

```
some/any/in(任意一条记录)

打印出公司所有员工年龄大于 30 的公司
hql="select c from Company c
        where 30< all(select e.age from c.employees e)";
打印出公司有员工年龄大于 30 的公司
hql="select c from Company c
        where 30< any(select e.age from c.employees e)";
```

10.投影查询:

```
hql="select c.cname,c.address,e.ename,e.age
      from Company c left join c.employees e";
Query q=sess.createQuery(hql);
List<Object[]> list=q.list();
--->Object[] o1={"kettas","wdk","tom",18};
result:
    cname    address    ename   age
    kettas     wdk        tom     18
    kettas     wdk        mary    19
    IBM          sd
-------------------------------------------------
packate hibernate.test;
class UserData{
   private String cname;
   private String address;
   private String ename;
   private int age;
   public UserData(String cname,String address,String
ename,int age){
      this.cname=cname;
      this.address=address;
      this.ename=ename;
      this.age=age;
   }
}
hql=select new hibernate.test.UserData(c.cname,c.address
        ,e.ename,e.age)
      from Company c left join c.employees e;
Query q=sess.createQuery(hql);
List<UserData> list=q.list();
```

UserData 在 session 中没有数据备份，是把零散的数据从数据库中拿回来自己现封装的对象。

11.集合过滤:

```
Company c=sess.get(Company.class,18);
c.employees 没有初始化,默认 lazy=true
取回 c 的所有员工
Set<Employee> es=c.getEmployees();
sql="select * from jdbc_employee where
company_id=c.getId()
```

取回的是 c 的年龄大于 30 的员工

```
1)hql="from Employee e where e.company=:com and
e.age>30";
   Query q=sess.createQuery(hql);
   q.setEntity("com",c);
   List<Employee> e=q.list();
2)Query q=sess.createFilter(c.getEmployees(),"where
this.age>30");
   List<Employee> es=q.list();
```

12. batch update

```
1)sess
  a. List<Employee> list=sess.createQuery("from
Employee").list();
  b. for(int i=0;i<list.size();i++){
        if(i%100==0){
           sess.flush();
           sess.clear();
                                }
        list.get(i).setName("hello"+i);
           }
     ta.commit();

2) hql="update Employee e set e.ename=:name";
   Query q=sess.createQuery(hql);
   q.setString("name","hello");
   q.executeUpdate();

3) hql="delete Employee e where e.eid>20";
```

13. SQL 语句是对数据库的表操作，HQL 语句是对类的操作。

以下是要掌握的内容：

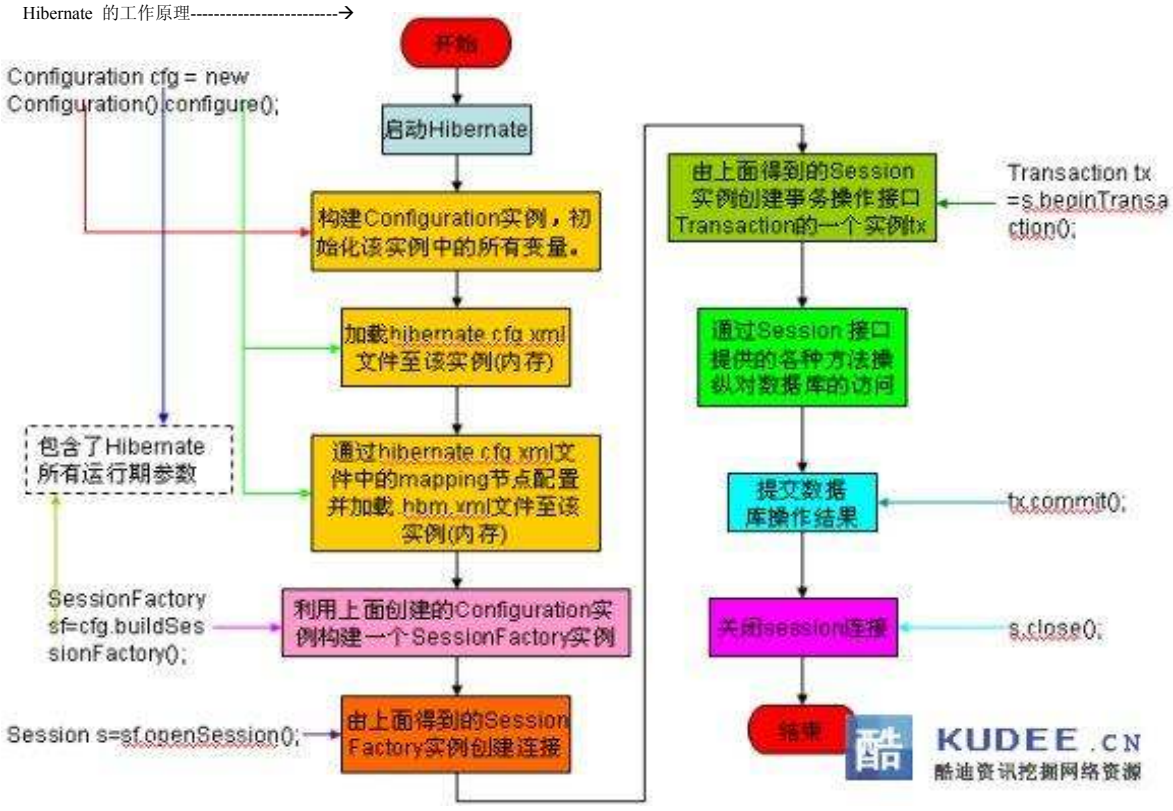
```
oracle: create table , create sequence
        select update delete insert
jdbc:  基本步骤
       jdbcUtil
       dao 的概念
       Transaction
       association(one-to-one)

hibernate: mapping file
           Session
  1)标签  2)三种状态  3)Session(集合)
  4)association  5)Inheriatance(one table)
  6)hql
```

hibernate.cfg.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
      "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
      "http://hibernate.sourceforge.net/hibernate-configuration-
3.0.dtd">
<hibernate-configuration>
<!--=====>      配置文件      <===== -->
    <session-factory>
        <!--<<  配置事务实现类 ，下面可以不写， 默认情况下为 JDBC>>  -->
        <property name="transaction.factory_class">
                org.hibernate.transaction.JDBCTransactionFactory
        </property>

        <!--<<   配置 JDBC 里 Batch 的大小  >>  -->
        <property name="jdbc.batch_size">50</property>
        <property name="cache.use_second_level_cache">false</property>

        <!--<<   获得线程安全的 Session  >>  -->
        <property name="current_session_context_class">thread</property>

        <!--<<   运行时，是否显示 SQL 语句  >>  -->
        <property name="show_sql">true</property>
```

```
        <property name="format_sql">true</property>

        <!--<<    配置数据库方言    >> -->
        <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

        <!--<<    配置数据库连接    >> -->
        <property name="connection.driver_class">oracle.jdbc.driver.OracleDriver</property>
        <property name="connection.username">cksd0804</property>
        <property name="connection.password">cksd0804</property>
        <property name="connection.url">jdbc:oracle:thin:@192.168.0.200:1521:ORADB10G</property>

        <!--<<    c3po   配置连接池    >> -->
        <property name="c3p0.max_size">2</property>
        <property name="c3p0.min_size">2</property>
        <property name="c3p0.timeout">5000</property>
        <property name="c3p0.max_statements">100</property>
        <property name="c3p0.idle_test_period">3000</property>
        <property name="c3p0.acquire_increment">2</property>
        <property name="c3p0.validate">false</property>

        <!-- <<    指定（hibernate 管理的）映射文件    >> -->
        <!--<mapping resource="day1/Users.hbm.xml"></mapping>-->
<!-- <<    当用 tomcat 服务器配置的连接池时用下面代替上面的连接    >>-->
<!--<property name="hibernate.connection.datasource">java:comp/env/oracle/ds</property> -->

<!--该路径是相对于 hibernate.cfg.xml 的路径 （相对路径）-->
        <!--<mapping resource="orm/o2o/fkuk/bi/Car.hbm.xml"></mapping> -->
        <!--<mapping resource="orm/o2o/fkuk/bi/License.hbm.xml"></mapping> -->
        <!--<mapping resource="orm/o2o/fkuk/ui/CarLicense.hbm.xml"></mapping>-->
        <!--<mapping resource="orm/o2m/bi/CompanyEmployee.hbm.xml"></mapping>-->
        <mapping resource="orm/m2m/bi/StudentCourse.hbm.xml"></mapping>
    </session-factory>
</hibernate-configuration>
```

Hibernate 的工作原理---------->



HTML&JavaScript

Html：==========

Head 相关
Meta 描述属性
Style 样式
Script 脚本
字体相关
h1 ... h6 字体的大小 一次变小
p 段落标签 换行且有段间距
pre 预备役标签，保留你的输出
font --> size | color 字体变化，设置大小和颜色
b | strong 加粗
center 居中
em 斜体，着重显示，不同的显示器，着重显示不一样
br 换行符
hr 分割线
图片/超链接
图片 img src（名字） | width height（宽高） | alt （图片找不到的时候显示的东西） | align（控制写在图片之后文字的位置 bootom 下方 middle 中间） | border （控制图片边框）
超级连接 a href（连接目标的 url） | name（锚点名） | target（4 个备选 _new 第一次连接开一个新窗口，第 2，3 次在同一个窗口里刷新 _self 自己窗口里打开 _blank 连接打开一个新窗口，点几次开几个窗口 _top（我在最顶端，把 frameset 框架替换了，要代替真个框架显示出来））href=“#”表示本页面
表格
table (tbody) border | bordercolor | bgcolor | cellsapcing | width height | align
|- tr
|- td colspan | rowspan 和并列| 合并行
表单
form --> method | action
|- input --> type ： text | password | checkbox | radio | button | submit | reset | hidden
|- select
|- option
|- textarea --> cols rows
name=value
框架
frameset ---> 不允许出现 body ---> rows cols
|- frame ---> name | src
C/S B/S
web 1.0 ---> 发布式体系
web 2.0 ---> 强调用户参与 ---> blog
==========
internet ---> 铺路
web ---> 车
http --- 底层传输层协议 来自于 ARPA
---> 无状态协议
---> 一次性连接
---> 80 端口
HTML ---> 标识性语言 --- 被浏览器解释
Active X
W3C 制定的

```
<body xxx="xxx">
    xxxx
</body>
```

```
<html>
    <head>  ---->
    </head>
    <body>
    </body>
</html>
```

字体相关标签

```
<h1 --- h6>
<p>                 <pre>
<font size color>
<center>
<br><hr>
<b><strong>
```

图片\超链接

```
<img>
 |- src
 |- width height
 |- alt
 |- align

<a>
 |- target --->  _new _self _blank _top 如果不加_表示一个 frameset 的 name
 |- href
 |- name
```

表格

```
<table> --> border | bordercolor | width / heigth | align | cellspacing
 |- <tr>
        |- <td>   --->  colspan | rowspan
```

表单

```
<form>
    |- method
        |- get
            参数以 URL 方式进行提交
        参数用来提交给服务器看,需要服务器端主动拿取
            缺点: 不安全
                URL 长度有限制
                编码问题
            |- post
                参数包含在请求体中提交
                服务器端直接从请求中获得数据

    |- action

    file:///C:/Documents%20and%20Settings/user/  桌  面 /html/form.html
            ? --> 连接符 表示后面提交参数
            userName=kettas
            & --> 参数连接符用于多个参数分隔
            password=123456
            &radio1=hibernate
            &heshang=JinChuangGuang
            &heshang=SunShuai&sel=tianjin
    <input>
</form>
```

框架<frameset>

```
<html>
<frameset rows="15%, 70%, 15%">
    <frame name="title" src="title.html">
    <frameset cols="15%,*">
        <frame name="leftBar" src="leftBar.html">
```

```
        <frame name="main" src="main.html">
    </frameset>
    <frame name="footer" src="footer.html">
</frameset>
</html>
```

Input→type:text 文本|password|checkbox 多选框||radio 单选框|botton|submit|reset
Select 下拉列表 option 标签
Textarea

==========

1) Css （Cascading Style Sheets）什么用

优点: 1 避免传统 html 标签过于繁琐的步骤
2 比传统 html 标签更灵活
3 可重用性

```
<head>
 <style type="text/css">
    h1.rb, h2.rb{
            color : red;
            background-color : black ;
            }

    h1.gy{
            color :green ;
        background-color : yellow;
    }

    #p1{
            color : red;
    }
    .mylayout{ //一般用点号“。”后面是 css 的名字
        color:red;
       }
            </style>
    </head>
    <body>
        <h1 class="rb">Hello Css World!</h1>
        <h1 class="gy">Have a nice day!</h1>
        <h1 class="rb">Good luck & Have fun!</h1>

        <h2 class="rb">this is h2 line</h2>
        <h3 class="rb">this is h3 line</h3>
        <p id="p1">this is p line</p>
        <textarea class=”mylayout”/> 用 class 调用 css
2) selector{
        property : value;
     }
```

3) 外部引入 css 使用的时候和内部一样

1) <style type="text/css"></style>
2) <link rel="stylesheet" type="text/css" href="…/home.css"/>
3) nested

内部标签使用 <h1 **style**="color:blue">this is 3 line</h1>

4，常用的 css 相关标签

文本相关
text-align : left | center | right
text-indent : 10px ;
color : 制定文字颜色

字体相关
font-family : 字体：
“times new Roman" , times , serif ....
font-weight : normal , bold , lighter , bolder
font-size : xxxx
% , smaller , larger .font-size:10pt}

```
.s1 {font-size:16px}
.s2 {font-size:16pt}
.s3 {font-size:80%}
.s4 {font-size:larger}
.s5 {font-size:xx-large}
```

letter-spacing : 字符间隔
word-spacing : 单词间隔

```
 .p1 {text-indent: 8mm}
 .d1 {width:300px}
.p2 {text-indent:50%}
```

颜色背景
background-color :
background-image:url(images/xxxx.jpg)---背景图片
background-repeat : 决定显示方式。
repeat : 重复显示
repeat-x : 左上方开始横向重复显示。
repeat-y : 纵向重复显示
*no-repeat : 只显示一次 。

关于 div span

边框相关
border : style width color
style : 边框的种类.
none , hidden , dotted , dashed, solid , double, groove , ridge, inset , outset
border-style :
border-top-style :
border-left-style :
border-right-style:
border-bottom-style :

border-width:
border-top-width:
border-left-width:
border-right-width:
border-bottom-width:
border-color :
border-top-color:
border-left-color:
border-right-color:
border-bottom-color:

高度，宽度 ：
width : height :

补白:变相的缩进

padding:
padding-top
padding-right
padding-left
padding-bottom
margin:
margin-top
margin-right
.....

JavaScript_1:==========

动态网页技术
---> 服务端技术
---> 客户端技术
javascript 基于解释性的语言
动态网页：
服务器端动态
客户端动态
减少服务器压力
功能受浏览器控制 需要看浏览器支持什么
========= 词法特性 ==========
采用 unicode 字符集,但是仅限于注释和字符串变量值, 变量和函数的标识符不能使用
Unicode 字符集。
基本特征 ：
变量没有数据类型 。
JAVA:
int a = 123;
String b = "123";
Javascript :
var a = 123;
var b = "123";
基本程序控制和 java 一样。
==========

将 javascript 代码引入到 Html 中
1，代码直接嵌入
```
<script language="javascript">
    ......
    ......
</script>
```

2, 引入外部文件
```
<script type="text/javascript" src="js/functions.js"></script>
<link rel="stylesheet" type="text/css" href="*.css">
```

javascript 简单交互手段
```
alert( "" );
document.write( "" ) ;
```
只有 function 里才算局部变量
If、for 里都不算
for 循环
第一种
```
for (i = 0; i <= 5; i++)
{
document.write(i)
document.write("<br>")
}
```
第二种使用 for...in 循环语句
for...in 循环中的循环计数器是一个字符串，而不是数字。它包含了当前属性的名称或者表示当前数组元素的下标。
```
<script type="text/javascript">
// 创建一个对象 myObject 以及三个属性 sitename, siteurl, sitecontent。
var myObject = new Object();
myObject.sitename = "布啦布啦";
myObject.siteurl = "blabla.cn";
myObject.sitecontent = "网页教程代码图库的中文站点";
//遍历对象的所有属性
for (prop in myObject)
{
document.write("属性 '" + prop + "' 为 " + myObject[prop]);
document.write("<br>");
}
```
prop 是数组元素的下标和 java 种的 foreach 有不同的含义

数据类型和值 ：
弱数据类型 设计的比较简单 随着功能愈加强大 已经成为了一个缺陷
在程序中，变量的值可以是 ：
三种基本数据类型 ：
数字: 123, 123.22
文本字符串: "zhongguo" ,"中国", '123'
boolean 类型: true | false
非 0 和 0 非 null| null

除基本数据类型以外， javascript 还支持复合类型 ：
Object(对象) , Array(数组)
boolean ：
boolean 的其他表示方法 ：
1 ， 0 和 非 0 值 。
2， 空 和非空 。
特殊数据类型：null 和 undefine （未定义的）.
javascript 是 弱数据类型的语言，其变量没有数据类型。所有变量声明时都使用 var 类型 。 而且统一变量可分别存储不同类型的值

```
var a = 123;
a = "123";
var a = 1;
var b = "2";
var c = a + b ;      "12"
```

创建并使用对象 。
1，
```
var obj = new Object();
obj.name = "zhangsan" ;
obj.age =123 ;
obj.email = "liucy@cernet.com" ;
```

属性的两种访问方式：
alert(obj.name) ;
alert(obj["name"]) ;
本质：多个属性的集合
缺点：不严谨
```
var obj = { name : "zhangsan" , age : 24 , email : "liucy@cernet.com" } ;
alert( obj.gender ) ;
```

创建并使用数组 。
1,
```
var arr = new Array();
var[0] = 1 ;
var[1] = 3 ;
```
2,
```
var arr = [ 1,2,3,4,5,6 ] ;
```
3,
```
var arr = [ 1,,,,6] ;
```
4,
```
var arr = [ 1, 4.4, "sd" , true]
```
类似一个集合
不需要指定数组长度
使用变量 ：
变量需要先声明，后使用 。
未付值的变量初始值是 undefine .
重复声明 ：
使用 var 重复声明是合法的. 如果重复声明中有初始值的话, 则相当于付值
语句 , 没有初始值的话,变量保留以前的值 .
遗漏声明 ：
如果使用了一个未声明的变量 ， javascript 会对这个变量作隐式声明。
但是所有隐式声明的变量，都会成为全局变量，即使声明是发生在函数体
之内的 。
函数声明和使用 ：

function name(a , b , c) {}
支持内联函数 ：
function a(){
function b();
b();
}
内联函数只能在作用域内使用。
变量作用域 ：
在 javascript 中不存在块作用域 ， 声明在块中的变量，在块的外面一样可以使用
if(){
var a = 10 ;
}
alert(a) ; //合法 。
作为数据的函数 ：
function a(x, y){ .... }
var b = a ;//用变量存函数；
b(1 , 2) ;//相当于 a(1,2);
思考：
var student = new Object();
student.name = "zhangsan";

通过构造函数创建函数 。
var a = new Function("a" , "b" , "return a + b ");
a , b , 新建对象的参数名称 ， 如果有多个可以依次填入 ：
new Function("a" , "b" , "c" , ... " return a + b + ... + n ; ") ；
调用 ： a(10 , 20) ;
通过函数直接量：
var a = function (x , y){ return x + y ; }
参数数量验证：arguments.length
变量作用域：
不存在块作用域 注意 这里所说的块 并不是函数块
window.parent

JavaScript_2:==========

javascript 中的常见事件 ：
一般性事件 ：
onclick 单击事件
ondblclick 双击事件
onmouseomove 鼠标移动
onmouseover 鼠标移入
onmouseout 鼠标移出
onmousedown 鼠标键按下
onmouseup 鼠标键松开
适用 几乎全部的可显示元素（标签） 。
页面相关事件 ：
onload : 页面加载时触发。即把页面所有的东西都下载完时触发 <body>
onscroll : 页面滚动时触发。
<body>
onstop : 按下 stop 按钮时触发。
<body>
onresize : 调整大小时触发 。
<body>
onmove : 窗口移动时触发。
<body>
表单相关事件 ：
onblur : 当前元素失去焦点时触发。 <input>
onchange : 当前元素失去焦点，并且值发生变化时触发。<input>
onfocus : 当前元素获得焦点时触发，指光标移到当前处，即获得焦点 。 <input>
onsubmit : 表单被提交时触发 <form onsubmit=”return tes()”>

==========

DOM : 是 W3C 提供的一组规范 ， 可以在独立于平台的前提下修改文档的内容和结构。
DOM 将文档的内容封装到对象及对象的属性和关系中 。通过操作 DOM 对象及对象的属性，达到修改文档内容及结构的目的 。
DOM 里有各种方法，用于修改文档内容和结构；
可以将 DOM 理解为文档内容的树状表示法 。

```
<table>
    <tbody>
        <tr><td>zhangsan</td><td>20</td></tr>
        <tr><td>lisi</td><td>21</td></tr>
    </tbody>
</table>
```

用于遍历 XML 文档的 DOM 方法：
document.getElementById("") XMLElement
document.getElementsByTagName("name") **array**
用于处理 XML 文档的 DOM 属性 ：
childNodes Array //返回的是一个数组
firstChild XMLElement //第一个子标签
lastChild XMLElement //最后一个子标签
nextSibling XMLElement //同级的下一个标签
previousSibling XMLElement //同级的上一个标签
parentNode XMLElement //直接父标签
通过 "." 访问 element 属性 。
document 对象为 DOM 的内置对象，代表 XML 文档的根
在 HTML 文件中可以理解为 body 标签 。
document.createElement("div") ;
document.createTextNode("text") ; 创建文本

```
var txtA = document.createTextNode("hello");  //创建文本内容
var colA = document.createElement("td");  //创建标记
colA.appendChild(txtA);  //添加子标记
```

element.getAttribute(Name) ;
element.setAttribute("name" , value) ;
element.appendChild()
element.insertBefore(newNode , targetNode) ;
element.removeAttribute(node)
element.removeChild(node) ;
element.replaceChild(newNode , oldNode) ; // 用 newNode 替换 oldNode
element.hasChildnodes()
浏览器差异。
1) table 和 tbody
2) 设置属性 ff element.setAttribute("name" , "value") ;
ie element.name = value
3
设置 css ff element.setAttribute("style","color:blue") ;
ie element.style.cssText = "color:blue" ;
ff element.setAttribute("class","xxx") ;
ie element.className ;

1 变量没有数据类型 ---> 值有
基本数据类型 ：
数字 | 字符串 | boolean
|-
0,非 0 | null 和非 null
复合类型:
数组(没有固定长度,可以装任何类型的值)
对象(只有属性没有方法)
2 变量可以重复声明
3 变量如果没有经过声明,系统会自动对其做隐式声明(作为全局变量)
4 function 函数名(a, b){}
5 通过构造方法创建函数: var fun = new Function("");
通过函数直接量创建函数: var fun = function(){}
6 作为变量函数

```
var fun;
function testFun(a, b){
    return a+b;
}
fun = testFun;
fun(1,2);
```

7 事件句柄：onclick……

8 DOM ---> getElementById
getElementsByTagName

9 各种表单验证==========

1） 两次输入密码是否相同

```
<FORM METHOD=POST ACTION="">
<input type="password" id="input1">
<input type="password" id="input2">
<input type="button" value="test" onclick="check()">
</FORM>
<script>

function check(){
with(document.all){
if(input1.value!=input2.value) {
alert("false")
input1.value = "";
input2.value = "";
}
else document.forms[0].submit();
}}
</script>
```

2）6. 验证油箱格式

```
<SCRIPT LANGUAGE=javascript RUNAT=Server>
function isEmail(strEmail) {
if(strEmail.search(/^w+((-w+)|(.w+))*@[A-Za-z0-9]+((.|-
)[A-Za-z0-9]+)*.[A-Za-z0-9]+$/) != -1)
return true;
else alert("oh");
}
</SCRIPT>
<input type=text onblur=isEmail(this.value)>
```

3）表单项不能为空

```
<script language="javascript">
function CheckForm(){
if (document.form.name.value.length == 0) {
alert("请输入您姓名!");
document.form.name.focus();
return false;
}
return true;
}
</script>
```

4) 比较两个表单项的值是否相同(密码比较)

```
<script language="javascript">
function CheckForm()
if (document.form.PWD.value !=
document.form.PWD_Again.value) {
alert("您两次输入的密码不一样！请重新输入.");
document.ADDUser.PWD.focus();
return false;
}
return true;
}
</script>
```

servlet

Servlet 是位于 Web 服务器内部的服务器端的 Java 应用程序，与传统的从命令行启动的 Java 应用程序不同，Servlet 由 Web 服务器进行加载，该 Web 服务器必须包含支持 Servlet 的 Java 虚拟机。
* 它们不是独立的应用程序，没有 main()方法。* 它们不是由用户或程序员调用，而是由另外一个应用程序(容器)调用。
* 它们都有一个生存周期，包含 init()和 destroy()方法。

要使用 Servlet，必须导入包 servlet-api.jar。**这里使用的服务器是 Tomcat**，其主目录结构为 ：

```
|- bin : 用于存放可执行命令( catalina.sh )
|- conf : 用于存放 tomcat 需要的配置文件.
|- lib : 存放 tomcat 在运行时要用到的类文件(jsp-api.jar、servlet-api.jar、...).
|- webapps : 最重要的一个目录，其下放置各种 Servlet 文件、网页文件(JSP HTML ...)、
            配置文件以及资源文件.此目录的结构 ：
    |- 应用目录(比如是一个学生管理网站，就在 webapps 文件夹下建一个 StudentManage 目录)
        |- WEB-INF 目录
            |- classes 目录, 放置所有的 Servlet 文件或其他类文件
            |- lib 目录, 放置本应用所要用到的类文件(jar 包)
            |- web.xml 配置文件
        |- 资源文件(比如图片), 网页文件(JSP HTML ...)

|- logs : 日志文件 .
|- work : 内部临时文件.
|- temp : 临时文件.
```

安装 tomcat 环境变量的配置，和配 jdk 差不多 servlet-api.jar；jsp-api.jar

1 Servlet 接口、GenericServlet 类、HttpServlet 类 ：

Servlet 是最顶层的接口，其提供的方法有 ：

```
init(ServletConfig config) : void    // 初始化
getServletConfig() : ServletConfig   // 取得该 Servlet 配置信息
getServletInfo() : String            // 取得相关信息
service(ServletRequest req, ServletResponse res) : void    //核心方法
destroy() : void    // Servlet 生命周期结束时候执行的方法
```

显然我们最关心的是 service 方法，其他的几个方法在实现的时候是千篇一律、无关痛痒的。故提供了 GenericServlet 类，此类实现了 Servlet 接口，我们在使用 Servlet 的时候，只需继承这个类然后覆盖其中的 service 方法(抛出 ServletException、IOException 异常)即可。

由于 Servlet 基本上是在 http 协议下使用的，故提供了 HttpServlet 这个类，此类继承自 GenericServlet 类，我们在使用 Servlet 时，只需继承 HttpServlet 类然后覆盖以下方法 ：

```
service( HttpServletRequest request ,
         HttpServletResponse response )
         throws ServletException , IOException : void
```

注意：HttpServletRequest 和 HttpServletResponse 分别是从 ServletRequest 和 ServletResponse 继承

此外，**HttpServlet 还提供了 doPost 和 doGet 方法**，参数和返回值与 service 方法一样。只是 service 方法可以针对客户端的任何请求类型(GET 和 POST)，而 doPost 和 doGet 方法分别只能对应客户端的 POST 方式请求和 GET 方式的请求。

HttpServlet-----→extends GenericServlet ----→ implements Servlet

2 使用 GenericServlet 实例 ：

```
package com.kettas.servlet;
import javax.servlet.* ;
import java.io.* ;
public class GenDateServlet extends GenericServlet{
    @Override
    public void service( ServletRequest request , ServletResponse response )
        throws ServletException ,IOException
    {       response.setContentType( "text/html" ) ; // 设置响应内容类型
            PrintWriter out = response.getWriter();// 获得文本写入流
            // 给客户端回应的 html 文本
            out.println( "<html>" ) ;
            out.println( "<body>" ) ;
            out.println( "<h1>Hello Servlet ！ </h1>" );
            out.println( "</body>" ) ;
            out.println( "</html>" ) ;
            out.flush();// 刷新写入
    }}
```

配置文件 web.xml 如下 ：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    version="2.5">
```

```
        // Servlet 文件路径
        <servlet>
            <servlet-name>query</servlet-name>
            <servlet-class>com.kettas.servlet.GenDateServlet</servlet-class>
        </servlet>
        // 指定映射，说明在浏览器中输入".../query"则对应当前 Servlet
        <servlet-mapping>
            <servlet-name>query</servlet-name>
            <url-pattern>/query</url-pattern>
        </servlet-mapping>
    </web-app>
```

3 使用 HttpServlet 简单实例 ：

```
package com.kettas.servlet ;
import javax.servlet.* ;
import javax.servlet.http.* ;
import java.io.* ;
public class LoginServlet extends HttpServlet{
    @Override
    public void service( HttpServletRequest request , HttpServletResponse response )
        throws ServletException , IOException
    {   response.setContentType("text/html; charset=GB2312"); // 注意设置编码的方式
        request.setCharacterEncoding("GB2312");
        PrintWriter out = response.getWriter();
        // 取得客户端提交的数据
        String name = request.getParameter( "userName" ) ;
        String pwd = request.getParameter( "password" ) ;
        out.println("<html>");
        out.println("<body>");
        out.println("<h1>");
        out.println("Name : " + name + "      " + "Password : " + pwd);
        out.println("</h1>");
        out.println("</body>");
        out.println("</html>");
        out.flush();
    }}
```

配置文件 web.xml 片段 ：

```
        <servlet>
            <servlet-name>login</servlet-name>
            <servlet-class>com.kettas.servlet.LoginServlet</servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>login</servlet-name>
            <url-pattern>/login</url-pattern>
        </servlet-mapping>
```

4 请求转发 ：

实现不同 servlet 之间的数据传递，这样便可实现业务逻辑和显示逻辑的分离

实例 ：

(1) 第一个 servlet，负责业务

```
package com.kettas.servlet ;
import javax.servlet.* ;
import javax.servlet.http.*;
import java.io.*;
import java.util.* ;
public class ForwardA extends HttpServlet{
    @Override
    public void service( HttpServletRequest request , HttpServletResponse response )
        throws ServletException , IOException
    {   System.out.println( "=== This is forward A ===" ) ;
        // 将业务部分的数据存储在 request 对象中，传递给下一个 servlet 使用
        Date d = new Date();
        request.setAttribute( "date" , d ) ;
        /* 注意转发的过程
           * 首先获得一个知道下一棒地址的"接力棒"对象，然后将这个"接力棒"传给下一个
           * servlet，这样便将请求转发了。
           */
        RequestDispatcher disp = request.getRequestDispatcher( "/forwardB" ) ;
        disp.forward( request , response ) ;
    }
}
```

注意 ：1 这种请求转发的方式是共用一个连接的，不管你中途经过了多少个 servlet，正因如此，
这些 servlet 才能共享 request 中存储的数据。
2 只有最后一个 servlet，才能在客户端浏览器中显示。

(2) 第二个 servlet，负责显示

```
package com.kettas.servlet ;
import javax.servlet.* ;
import javax.servlet.http.*;
import java.io.*;
import java.util.* ;
public class ForwardB extends HttpServlet{
    @Override
    public void service( HttpServletRequest request , HttpServletResponse response )
        throws ServletException , IOException
    {       response.setContentType( "text/html" );
            PrintWriter out = response.getWriter();
            out.println( "<h2>This is forwared B</h2>" ) ;
            // 通过 getAttribute 方法，从 request 中取得数据
            // 由于此方法返回的是 Object 对象，故要强转
            Date d = (Date)request.getAttribute( "date" ) ;
            out.println( "<h2>" + d + "</h2>" ) ;
            System.out.println( "=== This is forward B ===" ) ;
            out.flush();
    }
}
```

(3) web.xml 片段 ：

```
<servlet>
    <servlet-name>a</servlet-name>
    <servlet-class>com.kettas.servlet.ForwardA</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>a</servlet-name>
    <url-pattern>/forwardA</url-pattern>
</servlet-mapping>
<servlet>
    <servlet-name>b</servlet-name>
    <servlet-class>com.kettas.servlet.ForwardB</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>b</servlet-name>
    <url-pattern>/forwardB</url-pattern>
</servlet-mapping>
```

页面跳转的两种方式 1）request.getRequestDispatcher(“/forwardA”).forward(**request , response**); 这种跳转是在服务器内部是 servlet 之间跳转，显示的总是最后一个 servlet A-→B--→----→D

2)response.sendRedirect(“mayservlet/query”) 它其实是在客户端的 url 发生改变,相当一次新的请求，故不能传递数据，但能在不同的应用中跳转

5 关于 Cookie, 在客户端浏览器保存用户状态的一种机制

servlet 中的 Cookie 含有三个属性: name, value, maxAge

maxAge = 60 表示：此 cookie 在客户端存在 1 分钟

两个特殊值：

maxAge = -1 表示 ：此 Cookie 生命周期由保存它的浏览器决定 ，(浏览器开则生，关则死)，默认的

maxAge = 0 表示 ：删去以前的相应 cookie 存储

Cookie 应用实例 ：

```
package com.kettas.servlet ;
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
public class CookieServlet extends HttpServlet{
    @Override
    public void service( HttpServletRequest request , HttpServletResponse response )
        throws ServletException , IOException
    {// 创建一个新的 Cookie 对象, 构造参数分别为 Cookie 的 name 和 value 属性
        Cookie c = new Cookie( "test" , "1234567890" );
        // 将 Cookie 对象加入 response 中，这样才能被带入客户端
        response.addCookie( c ) ;
        // 从请求中获取客户端 Cookie 数组
        Cookie[] cookies = request.getCookies();
        response.setContentType( "text/html" );
        PrintWriter out = response.getWriter();
        out.println("<html>");
```

```
                out.println( "<body>" ) ;
                out.println( "<h1>Cookie List</h1><hr/><p></p>" ) ;
                if( cookies != null ){
                        for( Cookie cookie : cookies ) {
                                out.println( "<h2>" + cookie.getName() + "=" + cookie.getValue() + "</h2>" ) ;
                        }
                }else{
                          out.println( "<h2>No cookie</h2>" ) ;
                }
                out.println( "</body>" ) ;
                out.println("</html>");
                out.flush();
            }
        }
```

6 关于 HttpSession, 在服务器端保存用户状态的一种机制

(1) 获取 HttpSession 对象的方法 :

```
// 参数为 true，表示若存在对应的 HttpSession 对象，则返回。若不存在，则创建一个新的。
// 若参数为 false，表示若存在对应的 HttpSession 对象，则返回。若不存在，则返回 null。
  HttpSession session = request.getSession(true);
```

(2) 对 HttpSession 对象, 进行存取数据的操作

```
// 两个参数，分别为命名属性和对应的数据
session.setAttribute("name", data);
// 一个参数，命名属性，注意返回的为 Object 对象，要强转
session.getAttribute("name");
```

(3) 比较 Session 和 request :

request :

创建 : 当用户请求到达服务器的时候

销毁 : 当本次请求的应答回到客户端的时候.

客户端的一次请求应答之间

session :

创建 : 当用户第一次调用 request.getSession(true)

销毁 : 超时 (两级超时限制)

1) 内存 ---> 文件 .

2) 从文件系统销毁 .

session 的原理 :

给每个浏览器一个 cookie，这个 cookie 的 name 属性为"jsessionid"，value 属性为这个 session 对应的 ID 值。

(4) 当浏览器拒绝 cookie 时可以用 URL 把 session 的 id 提交给服务器

如 : http://localhost:8989/servletapp/forwardB;jsessionid=37D50D093CCD4A37CC1118785E38F438

```
"url;jessionid="+ session.getId()
response.encodeURL("url") ： 对 url 进行编码
```

7 **ServletConfig 对象和 ServletContext 对象**

(1)ServletConfig : 用来保存一个 Servlet 的配置信息的(比如 : name, class, url ...)

这些配置信息没什么大用处，我们还可以在 ServletConfig 中保存自己在 web.xml 文件中定义的数据

此时的 web.xml 文件片段如下 :

```
<servlet>
        <!-- 自己定义的，要保存在 ServletConfig 对象中的数据 -->
    <init-param>
        <param-name>jdbc.driver</param-name>
        <param-value>oracle.jdbc.driver.OracleDriver</param-value>
    </init-param>
    <init-param>
        <param-name>jdbc.user</param-name>
        <param-value>yinkui</param-value>
    </init-param>
    ...
    <servlet-name>query</servlet-name>
    <servlet-class>com.kettas.servlet.Query</servlet-class>
</servlet>
```

在 Servlet 中取得这些数据 :

```
// getServletConfig 方法继承自父类 GenericServlet
ServletConfig sc = this.getServletConfig();
// 显然，getInitParameter 方法返回的只能是字符串类型数据
String driver = sc.getInitParameter("jdbc.driver");
String user = sc.getInitParameter("jdbc.user");
```

注意: 1 ServletConfig 对象只能从 web.xml 文件中获取自定义数据(字符串数据)，不存在 setAttribute 方法去存入自定义数据。

2 在 Servlet 中，若要覆盖父类的 init(ServletConfig config)方法，必须这么做 :

```
public void init( ServletConfig config ){
```

```
                // 覆盖之前调用父类的这个方法, 否则 ServletConfig 对象会丢失
                // 此时 this.getServletConfig()返回的是 null, 那样我们就不能使用它了
                super.init( config ) ;
            ...     }
```

(2)ServletContext : 用来保存数据的全局唯一对象，一个应用中只有一个 ServletContext 对象

1 : 通过 web.xml 文件，在 ServletContext 对象中存入数据

此时的 web.xml 文件片段如下所示 :

```
<!-- 在此处写入我们要存入 ServletContext 对象中的数据 -->
<context-param>
    <param-name>jdbc.driver</param-name>
    <param-value>oracle.jdbc.driver.OracleDriver</param-value>
</context-param>
<context-param>
    <param-name>jdbc.url</param-name>
    <param-value>jdbc:oracle:thin:@192.168.0.201:1521:kettas</param-value>
</context-param>
...
<servlet>
    <servlet-name>...</servlet-name>
    <servlet-class>...</servlet-class>
</servlet>
```

取得其中的数据 : String driver = servletContext.getInitParameter("jdbc.driver");

2 : 通过 setAttribute 方法，在 ServletContext 对象中存入数据

```
servletContext.setAttribute("name", data); // 两个参数分别为命名属性以及对应的数据
// 取得 ServletContext 对象中的数据, 参数为命名属性
// 返回的是 Object 对象, 故要强转
servletContext.getAttribute("name");
```

3 : 取得 ServletContext 对象的三种方法(this 指代当前 Servlet)

```
(1) ServletContext sc = this.getServletContext();
(2) ServletContext sc = this.getServletConfig().getServletContext();
(3) ServletContext sc = request.getSession(true).getServletContext();
ServletContext 对象的一个重要方法 :
InputStream is = sc.getResourceAsStream( "fileName" ) ;
    fileName : 使用的是虚拟目录, 不依赖于实际路径/books/ajax.pdf
    最左边一个"/" : web 应用的根目录
  // 获得实际路径          String path = ctx.getRealPath( "/books/ajax.pdf" )
```

8 监听事件和过滤器

监听包括三种情况，分别是 HttpRequest、Session、ServletContext 监听。

常用的是 implements servletContextListener（全局变量）两个方法 public void contextInitialized(ServletContextEvent arg0)

arg0.getServletContext()

Session 监听事件所示 :

```
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
import com.kettas.upp02.util.Constant;
public class SessionListener implements HttpSessionListener {
    public void sessionCreated(HttpSessionEvent ent) {
        HttpSession session = ent.getSession();
        synchronized (this) {
            ServletContext ctx = session.getServletContext();
            Integer counter = (Integer) ctx.getAttribute("sessionCount");
            ctx.setAttribute("sessionCount", counter.intValue() + 1);
            System.out.println(Constant.LOGO + "SessionCount:"
                    + (counter.intValue() + 1));
        }}
    public void sessionDestroyed(HttpSessionEvent ent) {
        HttpSession session = ent.getSession();
        synchronized (this) {
            ServletContext ctx = session.getServletContext();
            Integer counter = (Integer) ctx.getAttribute("sessionCount");
            ctx.setAttribute("sessionCount", counter.intValue() - 1);
            System.out.println(Constant.LOGO + "SessionCount:"
                    + (counter.intValue() - 1));
        }
    }}
```

在 web.xml 文件中配置如下 :

```
<listener>
    <listener-class>shop. SessionListener </listener-class>
</listener>
```

其他两个监听事件的实现同上并无二致。

过滤器　　// 实现 Filter 接口

```
import java.io.IOException;
import javax.servlet.*;
public class EncodingFilter implements Filter{
//销毁时执行，没必要覆盖
        public void destroy() {}
        //发送请求时执行
        public void doFilter(ServletRequest request, ServletResponse response,
                FilterChain chain) throws IOException, ServletException {
            //设置发送请求和接收请求时的编码方式，统一才能达到过滤作用
            request.setCharacterEncoding("UTF-8");
            response.setCharacterEncoding("UTF-8");
            try {
                chain.doFilter(request, response); 请求转发
            } catch (RuntimeException e) {
                e.printStackTrace();
            }}
        //加载时执行，也没必要执行
        public void init(FilterConfig arg0) throws ServletException {}
    }
```

web.xml 文件中：

```
  //配置当发生什么要的请求时，让那个过滤流执行操作
    <filter>
        <filter-name>encodingFilter</filter-name>
        <filter-class>filter.EncodingFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>encodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

9，解决乱码问题　1）response.setContentType(“text/html;charset=gbk2312”)

2)requset.setCharacterEnconding(“gbk”)　------是 post 的时候

3)在 server.xml 中加 URLEncoding=“gbk”------是 get 发送数据的时候

10,servlet 的生命周期

1） Servlet 在容器中运行，其实例的创建及销毁等都不是由程序员决定的，而是由容器进行控制的。

创建 Servlet 实例有两个时机：1 客户端第一次请求某个 Servlet 时，系统创建该 Servlet 的实例：大部分的 Servlet 都是这种 Servlet。

2，Web 应用启动时立即创建 Servlet 实例，即 load-on-startup　　<load-on-startup>1</load-on-startup>

Servlet 的生命周期通过 **javax.servlet.Servlet** 接口中的 **init()、service()和 destroy()方法**来表示。

每个 Servlet 的运行都遵循如下生命周期。

（1）加载和实例化：找到 servlet 类的位置通过类加载器加载 Servlet 类，成功加载后，容器通过 Java 的反射 API 来创建 Servlet 实例，调用的是 Servlet 的默认构造方法（即不带参数的构造方法），

（2）初始化： 容器将调用 Servlet 的 init()方法初始化这个对象。初始化的目的是为了让 Servlet 对象在处理客户端请求前完成一些初始化的工作，如建立数据库的连接，获取配置信息等。对于每一个 Servlet 实例，init()方法只被调用一次

（3）请求处理：Servlet 容器调用 Servlet 的 service()方法对请求进行处理。要注意的是，在 service()方法调用之前，init()方法必须成功执行

（4）服务终止：容器就会调用实例的 destroy()方法，以便让该实例可以释放它所使用的资源

考点 2）从始至终只有一个对象，多线程通过线程池访问同一个 servlet

Servlet 采用多线程来处理多个请求同时访问，Servelet 容器维护了一个线程池来服务请求。

线程池实际上是等待执行代码的一组线程叫做工作者线程(WorkerThread)，Servlet 容器使用一个调度线程来管理工作者线程(DispatcherThread)。

当容器收到一个访问 Servlet 的请求，调度者线程从线程池中选出一个工作者线程，将请求传递给该线程，然后由该线程来执行 Servlet 的 service 方法。

当这个线程正在执行的时候，容器收到另外一个请求，调度者线程将从池中选出另外一个工作者线程来服务新的请求，容器并不关系这个请求是否访问的是同一个 Servlet 还是另外一个 Servlet。

当容器同时收到对同一 Servlet 的多个请求，那这个 Servlet 的 service 方法将在多线程中并发的执行。

3)、如何现实 servlet 的单线程模式

<%@ page isThreadSafe=”false”%>

JSP

1 JSP 语法:

jsp 中嵌入 java 代码的方式 :

1) 表达式标签 <%= 1 + 1 %>

a) 计算表达式的返回值 .

b) 能将返回值在网页上显示出来 .

不能出现 ";"

<%= 1+1%>则在网页上显示 2

2) 声明标签 : <%! %>

用来声明变量和函数, 在声明标签中声明的变量和函数, 可以在本页面的其他的 java 代码中使用.

声明的位置是首是尾皆无妨. 建议尽量少声明变量, 因为 jsp 最终要被解释为 servlet, 声明的变量在 servlet 中就体现为实例变量, 在多个线程访问这个 servlet 的时候, 由于实例变量被多个线程共享使用(实例变量线程不安全, 局部变量线程安全), 有可能出现问题, 并且不好解决.

3) 普通脚本 : <% %> 普通脚本是不能嵌套的

```
<%
  for( int i = 0 ; i < 10 ; i++ ){
        <%= 1+ 1 %>
  }
%>
```

2 指令元素 : 用来说明一个 jsp 文件自身的一些特点.

以便服务器(tomcat)作出正确的处理 .

页面指令 :

<%@page contentType="text/html;charset=utf-8" %>

<%@page import="java.util.*" %>

标签库指令

<%@taglib %>

<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

包含指令 :静态包含

用来包含其他 jsp 的源代码 (静态包含).

所谓静态包含, 就是先将引入的 jsp 页包含入本页面中, 然后解释为同一个 servlet

<%@include file="xxx"%> 静态包含包含的是源代码，考虑变量冲突，并且 xxx 后面不能传参数

3 动作元素

<jsp:include page="/a.jsp" />

动态包含: 包含的是对方的输出结果, 分别解释为不同的 servlet.

动态包含实例 :

(1) header.jsp :

```
        <%
        String text = request.getParameter( "text" ) ;
    %>
        <center>
        <h1>
        <font color="blue">
          <%
            if( text != null ){
          %>
              <h1><%=text%></h1>
           <%
          }else{
            %>
              <h1>Welcome to kettas</h1>
            <%
        }
              %>
        </font>
        </h1>
    </center>
```

(2) body.jsp :

```
        <%@page contentType="text/html" %>
            <html>
            <body>
<!-- 相当于 <jsp:include page="/header.jsp?name=This is param"/> -->
        <jsp:include page="/header.jsp">
            <jsp:param name="text" value="This is param"/>
            </jsp:include>
            <h1>This is body</h1>
            <%
                for(int i = 0 ; i< 3 ; i++ ){
            %>
            <h1><%= new java.util.Date() %></h2>
            <%
                }
            %>
            <%
                for( int i =0 ; i < 3 ; i++ ){
        out.println( "<h2>" + new java.util.Date() + "</h2>" ) ;
                }
            %>
            </body>
        </html>
    <jsp:forward page="/b.jsp" />
```

页面转向 : 相当于 servlet 中的"接力棒"转向, 是在同一个连接中的页面转向.

<% response.sendRedirect("/a.jsp"); %>

页面转向 : 连接已经换了一个.

4 jsp 中的 隐含 9 对象 . (可以直接拿来使用)

request ----> HttpServletRequest .
response ---> HttpServletResponse .
session ----> HttpSession .
application -> ServletContext .
|-> web.xml
|-> setAttribute, getAttribute.
|-> 全局唯一 .
以下四个用的很少, 知道有这个东西即可
out ---------> response.getWriter();<% out.println()%>
config -------> ServletConfig <在 xml 中也可以配置 servlet，可以配置初始化参数>
exception ---> Exception
page ------> Object
相当重要的隐含对象, 重点说明之
pageContext --> javax.serlvet.jsp.PageContext
关于 pageContext :
1, 本身也是一个能存储命名属性的作用域 .
setAttribute("name", data)
getAttribute("name")
pageContext 作用域和声明周期 .
声明周期只局限在本页面 .
在同一页面的不同标签之间传递数据 .（本页面共享数据）
同时保证数据不流传到其他页面上 .
2, 可以管理其他作用域中的命名属性 .
pageContext.getAttribute("name");
pageContext.getAttribute("name",int scope);
scope 值为：
PAGE_SCOPE
REQUEST_SCOPE
SESSION_SCOPE
APPLICATION_SCOPE
为了选择作用域
pageContext.setAttribute("name" , value);
pageContext.setAttribute("name" , value , int scope);
pageContext.findAttribute("name") ;
按照从小到大的顺序依次查找作用域中的命名属性 .
pageCOntext --> request ---> session --> application
pageCOntext.findAttribute("a") ;
3, 获得其他所有的隐含对象 .
pageContext.getRequest() ---> request
pageCOntext.getSession()
pageCOntext.getConfig()
pageCOntext.getOut()
注意 : 隐含对象在表达式标签和普通脚本中都可以使用
<%= request.getParameter("name") %>
<% sesison.getAttribute() %>
但是在声明脚本中不能用, 比如 :
<%!
void fn(){
session.getAtrreibute();
}
%>

5 JSP 的几个页面指令 :

页面指令 : 向服务器说明页面自身的特征,以便服务器
1 <%@page contentType="text/xml;charset=utf-8" %> 客户端--->>服务端的编码
2 <%@page import="" %> 引入名字空间
3 <%@page pageEncoding="GBK/GB2312/utf-8"%>(网页的静态内容乱码想到 pageEncoding,
jsp 文件文本编辑器所采用的编码)--->服务器内部使用的用来指定 jsp 源文件所采用的编码方式.
4 <%@page isELIgnored="false" %> EL 一种表达式语言，默认值不同的服务器不一样，最好明确写出
${ 1+1 } (false：显示结果。true：显示 ${ 1+1 }源代码)
5 <%@page errorPage="/error.jsp"%>
指定错误页面 .
jsp ---> 业务代码 .
6 <%@page isErrorPage="true|false"%>当是 TRUE 时就会有 exception 的隐含对象
<%@page isErrorPage="true" errorPage="/other.jsp"%> 不能这样转
A(源页面) ------------------------> B(错误页面)
errorPage="B" isErrorPage="true"
7 <%@page session="true"%>--默认值, 表示本页面是否需要 session 对象.
在 servlet 中只有直接调用才能得到，而 jsp 中默认是有的

<%@page language="java"%>默认的语言
<%@page extends="XXX" %>服务器自己决定
<%@page buffer=""%> 服务器自己决定调节

6 JSP(Servlet)中从连接池获取连接

1) 建立连接 .
2) 执行 SQL .
3) 处理结果 .
4) 释放资源 .

Connection pool 连接池
DataSource
LDAP (Light directory access protocal)轻量级目录访问协议
JNDI (java naming director interface) Java 命名目录接口

使用连接池 :

1) 配置连接池

改配置文件 conf/context.xml

```
<Resource driverClassName="oracle.jdbc.driver.OracleDriver"
  注：jar 包应该放在外边的 lib 包中，因为它是给 tomcat 用的
        url="jdbc:oracle:thin:@127.0.0.1:1521:XE"
        username="kettas"
        password="kettas"
        maxActive="2"  最大连接数
        type="javax.sql.DataSource"  连接类型
        auth="Container"  一般不改
        name="oracle/ds"
        />
```

2) 使用 DataSource

用法：%@page import="javax.naming.*"%>

```
<%
  Context ctx = new InitialContext();
  DataSource ds = (DataSource)ctx.lookup("java:comp/env/oracle/ds");tomcat 特点：必须加 java:comp/env/*
  Connection conn = ds.getConnection();
  out.println( "<h1>get connection!</h1>" );
  conn.close();
%>
```

7 处理 javabean 的 JSP 标签

(1) 关于 javabean 要求:

1 具有无参的构造函数 .
2 针对每一个成员变量,因改提供相应 get/set
3 implments Serializable（实现才能对象序列化）

(2) <jsp:useBean />

使用一个保存在某个作用域(pagecontext , request ,session, application)中的 javabean .

```
<jsp:useBean id="user" class="beijing.User" scope="session"/>
  实际上执行的代码为 : User u = session.getAttribute("user");
  id 的含义 : 1, 为要使用的 javabean 取个名字.
        2, javabean 在 session 作用域中的命名属性.
  class : java bean 的类型.
  scope : 指定一个作用域(page,request,session,application), 若不指定, 则由小到大.
```

(3) <jsp:setProperty /> 五种设置属性的方法

```
<jsp:setProperty name="" property="" value=""/>
  name: 被操作的 javabean 的名字. 即 useBean 中的 id
  property: 属性名称 id , name , phone
  value: 值
<jsp:setProperty name="" property="" param=""/>
  param:表示该属性的值来自于客户端提交的参数. param 用来指定这个参数的名字.
<jsp:setProperty name="" property=""/>
  客户端提交的参数名称和成员变量的名称正好一致. 省略 param
<jsp:setProperty name="" property="*"/>
  用一个标签为所有的属性赋值, 属性的值来自于客户端提交的参数,
  参数名字和属性名字 一一对应.
```

(4) <jsp:getProperty />

使用 javabean 中的属性, 可用于页面显示 .

```
<jsp:getProperty name="" property=""/>
  name: 被操作的 javabean 的名字. 即 useBean 中的 id
  property: 属性名称 id , name , phone
```

使用片段如下 :

```
<html>
  <body>
    <h2><jsp:getProperty name="user" property="name" /></h2>
    <h2><jsp:getProperty name="user" property="id"/></h2>
```

```
                    <h2><jsp:getProperty name="user" property="phone"/></h2>
                    <h2><jsp:getProperty name="user" property="password"/></h2>
                </body>
            </html>
```

8 使用自定义的标签

(1) 构思, 比如写一个对指定名字说 hello 的标签, 应该是<前缀:hello user="zhagnsna"/>

(2) 写类

要实现的基础接口 : javax.serlvet.jsp.tagext.SimpleTag

其中含有五个方法 :

doTag ---> 实现标签的功能 .

setJspContext ---> pageContext .

setParent ---> 父标签的实现类

setJspBody ---> JspFragement

getParent

要实现五个方法, 显得很繁琐, javax.servlet.jsp.tagext.SimpleTagSupport 类实现了这个基础接口, 我们在写类时只需要继承这个类, 然后覆盖 doTag 方法即可.

类如下 :

```
package com.kettas.tag ;
    import javax.servlet.http.*;
    import javax.servlet.jsp.tagext.* ;
    import javax.servlet.jsp.* ;
    import java.io.* ;
    public class HelloHandler extends SimpleTagSupport{
            private String user ;
            public void setUser( String user ){
                this.user = user ;
            }
            public String getUser(){
                return user ;
            }
            @Override
            public void doTag() throws JspException , IOException{
                PageContext ctx = (PageContext)this.getJspContext();
                ctx.getOut().println( "hello " + user    );
                ctx.getOut().flush();
            }
    }
```

(3) 写 tld 配置文件(位置为 : /WEB-INF/***.tld)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>2.0</jspversion>
    <shortname>mytag</shortname>
    <!-- 指定命名空间 引入标签时使用 -->
    <uri>liucy@cernet.com</uri>
    <tag>
        <!-- 标签名 -->
        <name>hello</name>
        <!-- 标签对应的类 -->
        <tag-class>com.kettas.tag.HelloHandler</tag-class>
        <!-- 标签体之中是否含有内容 -->
        <body-content>empty</body-content>
        <!-- 标签属性 -->
        <attribute>
            <name>user</name>
            <type>java.lang.String</type>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

(4) 在 jsp 文件中使用

```
<%@page contentType="text/html"%>
<%-- 注意引入自定义标签的方式, prefix 属性指定标签前缀, 前缀解决不同标签库标签重名 --%>
<%@taglib uri="liucy@cernet.com" prefix="liucy"%>
<html>
    <body>
```

```
                    <h2>
                        <liucy:hello user='<%= request.getParameter( "name" )%>'/>
                    </h2>
                </body>
            </html>
```

自定义一个循环标签 ：

类 ：==========

```
package com.kettas.tag ;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.*;
public class LoopHandler extends SimpleTagSupport{
  private int begin ;
  private int end ;
public void setBegin( int begin ){
        this.begin = begin ;
        }
        public int getBegin( ){
                return begin ;
                }
        public void setEnd(int end ){
                this.end = end ;
        }
        public int getEnd( ){
        return end ;
                }
        @Override
        public void doTag()throws JspException ,IOException{
           // JspFragment 对象可以获取标签体内部的内容
           JspFragment f = this.getJspBody();
           PageContext ctx = (PageContext)this.getJspContext();
          for( int i = begin ; i < end ; i++){
                f.invoke( ctx.getOut() );
                        }
                }
        }
```

配置文件如下 :==========

```
<?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
        "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd" >
                <taglib>
                        <tlibversion>1.0</tlibversion>
                        <jspversion>2.0</jspversion>
                        <shortname>mytag</shortname>
                        <uri>liucy@cernet.com</uri>
                        <tag>
                        <name>hello</name>

        <tag-class>com.kettas.tag.HelloHandler</tag-class>
                        <body-content>empty</body-content>
                        <attribute>
                        <name>user</name>
                        <type>java.lang.String</type>
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
                        </attribute>
                        </tag>
                        <tag>
                        <name>loop</name>

                        <tag-class>com.kettas.tag.LoopHandler</tag-class>
                        <!-- 标签体内含有内容, 并且是非脚本的内容 -->
                        <body-content>scriptless</body-content>
                                <attribute>
                                <name>begin</name>
                                <type>int</type>
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
                                </attribute>
                                <attribute>
                                <name>end</name>
                        <type>int</type>
                        <required>true</required>
                        <rtexprvalue>true</rtexprvalue>
                        </attribute>
                        </tag>
                </taglib>
```

使用如下 :==========

```
<%@taglib uri="liucy@cernet.com" prefix="liucy"%>
<liucy:loop begin="0" end="3">
        <h2>Hello !</h2>
</liucy:loop>
如此便可连续在网页上输出三个"Hello !"
```

9 EL 表达式(${ })

(1) 完成一些简单运算.

数学运算

+ - * % / ${ a + b }

布尔运算

>	gt	(great than)
<	lt	(less than)
>=	ge	(great equal)
<=	le	(less equal)
!=	ne	(not equal)
==	eq	(equal)

${ a > b } ${ a gt b }

逻辑运算

&& || !

and or not

非空运算 :

a == null

${ not empty a }

|-> a 不存在返回 true

|-> a 存在 返回 false

(2) 通过 EL 表达式,快捷的访问作用域中的命名属性

<%= session.getAttribute("name")%>

用 EL 表达式 : ${ name }

(3) 快速访问 javabean 的属性.

<jsp:getProperty name="user" property="name"/>

用 EL 表达式 : ${ user.name }

(4) 常用隐含对象 .

${ param }

${ param.age }

${ param.name }

相当于 : <%= request.getParameter("name") %>

用来访问客户端提交的参数.

${ cookie.age }

实际要执行的代码 :

```
Cookie[] c = request.getCookies();
for( Cookie a : c ){
    if(a.getName() == "age"){
        a.getValue();
        ...
    }
}
```

10 为 JSP 写的一套核心标签, 有了这套标签, 根本不需要自定义标签了

(1) 准备

需要 **standard.jar, jstl.ja**r 两个 jar 包, 放入 Tomcat 6.0/lib 目录中(或者是/WEB-INF/lib)

(2)core<%@tagliburi="http://java.sun.com/jsp/jstl/core"

prefix="c"%>

forEach 循环

①一般用法 相当普通的 for 循环

```
<c:forEach begin =”1” end=”10” varstatas="st">
    ${ st.count }</c:forEach>
```

② 迭代循环 集合

```
<c:forEach var="u" items="${ users }">
        ${ u .name}
</c:forEach>
```

若是 map 集合${ user.key}得到值的集合

set

```
(1)<c:set var="a" value="1234"/>
(2)<c:set var="a">
 xxxxx
</c:set>
```

把标签体内所有的输出都当作 var 的值，不直接显示在网页上

使用需要调用${a}

remove

```
<c:remove var="a"/>
    ${a}
```

url

```
<c:url var="date" value="/jspapp/date.jsp">
 <c:param name="age" value="23"/>
 <c:param name="name" value="lisi"/>
 <c:param name="passwd" value="ffff"/>
</c:url>

<a href="${date}">test</a>
/jspapp/xxx.jsp?age=23
<a href="/jspapp/xxx.jsp?age=23&name=lsis&passwd=123">
  test
</a>
```

同 if swith

```
<c:if test=”s{4>2}”>xxxxx</c:if>

<c:choose>
  <c:when test="a">
      ccccccc
  </c:when>
  <c:when test="b"></c:when>
  ....
</c:choose>
<c:choose>
  <c:when test="">
      cddddddd
  </c:when>
  <c:when test="">
   sssss
  </c:when>
  <c"when test="">
   xxxxxx
  </c:when>
  <c:otherwise>

  </c:otherwise>
</c:choose>
```

第五天：==========

MVC :

M : model (业务逻辑与业务数据)： javabean

V : view (显示逻辑)

将数据按照用户的 要求显示出来 .

对同一份数据而言,可以 以 多种形式

体现 (类表, 屏图,柱图 等等)

用 JSP 实现。

C : controller (控制器 , 负责程序的流程控制)

接收用户请求 , 根据业务逻辑的执行情况 返回相应的结果 .

用 Servlet 来实现。

==========

好处 :

1) 各司其职, 解耦合 .

代码可重用。

==========

前端控制器(Servlet)的工作指责 :

1) 能够接受所有的用户请求 .

<url-pattern>*</url-pattern>

2) 能跟据请求的不同 调用 不同的处理(javabean) .

a, 请求的不同 ---> url ---> servletpath

http://loxxx:8080/app/login

http://loxxx:8080/app/query

request.getServletPath() --> path

login=A f1 f2 f3

==========

query=B

delete=C

b, 通过一个配置文件,向 servlet 说明 被调用组件 和 serlvetpath 之间的对应关系 .

C, 要求所有的被调用的组件 必须按照某个接口的规发来进行开发 .这样才能由 servlet 正确的调用 .

Action 的主要作用是获得参数，让业务（Biz）处理，然后把处理的结果返回给 Action 再返回给 Servlet。

```
public interface Action{
 public String  execute( request , response )
}
LoginAction
   execute( Request , response ){
   String name = request.getParameter( .. ) ;
   String pwd = request.getParameter( "" );
        UserBiz biz = new UserBiz();
        biz.login( name , pwd );
        return "ok.jsp|error.jsp" ;
 }
UserBiz
   login( userName , password )
```

Servlet ---> Action ---> Biz

1) biz 实现业务逻辑

2) 写 Action

a, 准备参数

b, 调用 biz

c, 根据 biz 的运行结果,返回对应 URL

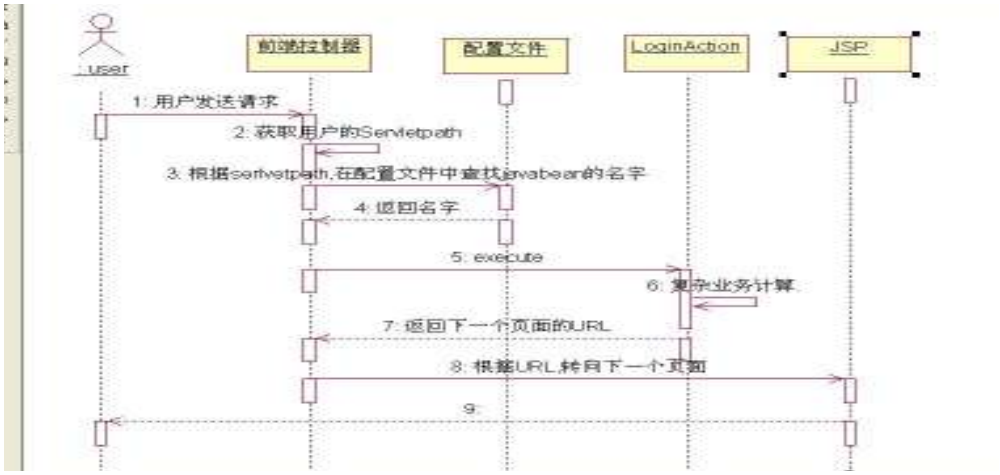
3)在配置文件中, 指定servletpath 与 Action之间的对应关系 .

4) 编写下一个 JSP

```
==========================================
ProductBiz
   |-> Collection<Product> getAllProducts();
   return Collection;

   QuerProductAction
   |-> execute()
         |-> biz.getAllProducts(); --> disp.jsp
                                        return disp.jsp;
```

MVC 的大至执行顺序：

1,用户发送请求＞2 获取用户的 servletpath＞3 根据 servletpath 在配置文件中查找 javabean 的名字，＞4 返回名字＞5，execute＞6，复杂业务计算＞7，返回下一个页面的 URL＞8 根据 URL 转向下一个页面＞9 把页面返回给用户。



Struts 1.2

第一天：

MVC : M : 业务逻辑,业务数据 (可以重复利用) java Bean (VO BO) EJB　　其实 struts1 没有实现业务层，也无法实现
V : 显示逻辑 ,同一份数据 ,对应多种显示方法. JSP 代码实现。
C：流程控制器 , Servlet 代码实现。javabean Servlet javabean 其实他有两个部分组成：
　　1，系统核心控制器 由 Struts 1 框架提供，就是 **ActionServlet**
　　2，业务逻辑控制器 由 Struts 1 框架提供就是用户自己写的 **Action** 实例
在 struts-config.xml 配置文件中的一些属性： >
Struts : 基于 mvc 的软件框架 . Struts :
1 使用 struts 需要完成的准备工作 :
1) 前置控制器 (ActionServlet) .
2) 定义了配置文件的格式 . **/WEB-INF/struts-config.xml**
3) 提供了一个父类, Action , 要求所有的 javabean 都来继承这个父类 , 覆盖其中的方法 . 使用 struts1 需要完成的准备工作 :
version 2.0 version 1.2.x 下面是 1.2 的版本
1) 获取 struts 的 jar 包 --> 拷贝到 WEB-INF/lib
2) 对 struts 提供的前端控制器(ActionServlet) 在 自己的 web.xml 中 进行配置
3) 确保 struts 用到的配置文件被放置到指定的位置 . struts1.2.7/webapps/struts-blank.war 解
　　压之后,从中拷贝 struts-config.xml 到我们 自己的应用里 .
4) 将 struts 提供的 jar 包配置到 classpath 中. struts.jar : 编译时依赖

2 常用的业务流程
0) 写 jsp , 这个 jSP 中有用户需要的连接或表单,
1) javabean , 通过它里面的方法,完成业务逻辑 .
2) 写 Action , 调用 javabean ,并根据 javabean 的结果,返回不同的页面 .
3) 在配置文件中对 Action 进行配置 .
4) 写 jsp login.do http://localhost:8989/strutsapp/ok.jsp Resource file
1) 编写资源文件. a, properties
2) 在 struts-config.xml 中对资源文件进行配置 . 通知 struts, 要使用的资源文件的名称 .
3) 在 jsp 中使用资源文件的内容 . 编程时使用 key 显示时看到的是 value
　　errors.header //errors 会在所有错误的开头自动加上该 key 所代表的值（value）， errors.footer //errors 会在所有错误的结尾自动加上该 key 所代表的值（value）， ActionForm : 用来保存客户端提交的数据 .
　　name password ----> Object |->name |->password
1) 写一个类(ActionForm) , 用于保存客户端提交的参数.
2) 编辑配置文件, 告知 struts,什么样的请求 使用哪个 ActionForm 对象 .
3) 在 Action 的 execute 方法中, 从 ActionForm 对象中获取数据 .通过 ActionForm 做验证 .
　　1) 在配置文件中说明,哪些请求是需要做验证的 .
　　2) 在 ActionForm 里编写验证代码 . @Override validate() ActionMessage ---> Resource
　　3) 在错误页面上显示错误信息 .
　　0) 编写 biz 对象的方法,用于处理用户请求 . |-> 测试功能
　　1) 编写用户发送请、求的页面
　　2) 根据表单的参数 编写 ActionForm
　　3) 编写 Action 对象.execute 方法 |-> 调用 biz 对象的方法. |->
　　根据 biz 方法的返回结果,返回不同的 ActionForward
　　4) 对 ActionForm , Action 进行配置 .
　　5) 编写与 ActionForward 对应的 jsp 页面 .

第二天：具体实现

```
<form-beans>
    <form-bean name="" type=""/>
</form-beans>
<action-mappings>
    <action path="" type="" name="" validate="true" input="">
        <forward name="" path="" redirect="true"/>
    </action>
</action-mappings>
<html:errors />
```

3，常用配置文件 最原始的 Action 接口，验证登陆的实例
(1) web.xml 文件 :

```
<?xml version="1.0" encoding="utf-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">

  <!—在 web.Xml 配置前端控制器 -->
    <servlet>
        <init-param>
```

```
                <param-name>config</param-name>
                <param-value>/WEB-INF/struts-config.xml</param-value>
            </init-param>
            <init-param>
                <param-name>debug</param-name>
                <param-value>2</param-value>
            </init-param>
            <init-param>
                <param-name>detail</param-name>
                <param-value>2</param-value>
            </init-param>
            <servlet-name>actionServlet</servlet-name>
            <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
            <load-on-startup>3</load-on-startup>
        </servlet>

        <!-- 对任意请求，都采用 Struts 提供的那个 ActionServlet 进行转发 -->
        <servlet-mapping>
            <servlet-name>actionServlet</servlet-name>
            <url-pattern>*.do</url-pattern>
        </servlet-mapping>

    </web-app>
```

(2) struts-config.xml 文件 ：

```
    ActionServlet<?xml version="1.0" encoding="utf-8" ?>
    <!DOCTYPE struts-config PUBLIC
            "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
            "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
    <struts-config>
    <form-beans>
            <!-- 将表单提交的数据封装为一个对象 name 属性对应 Action 中的 name  自己手动写的 actionform-->
             <form-bean name="loginForm" type="com.kettas.struts.LoginForm"/>
             <!—动态 actionForm
             <form-bean name="regForm" type="org.apache.struts.action.DynaActionForm">
                    <form-property name="userName" type="String"/>
                    <form-property name="age" type="java.lang.Integer"/>
            </form-bean>
        </form-beans>

        <!-- 指定映射 指定请求路径对应指定 Action
               name 属性指定与此 Action 对应的表单对象
               validate 指定是否采用表单验证 input 指定若是表单验证出错 要去的页面 -->
        <action-mappings>
            <action path="/login" type="com.kettas.struts.LoginAction" name="loginForm"
                validate="true" input="/login.jsp">
                <forward name="ok" path="/ok.jsp" redirect="true"/>
                <forward name="error" path="/error.jsp"/>
            </action>
        </action-mappings>

        <!-- 指定资源文件 注意 ： 以 WEB-INF/classes 为根目录 -->
        <message-resources parameter="res.AppResources"/>
    </struts-config>
```

(3) 表单对象 自己手动写 actionForm

```
    package com.kettas.struts ;
    import org.apache.struts.action.* ;
    import javax.servlet.http.* ;
    // 要继承 ActionForm 类
    public class LoginForm extends ActionForm{
        // 成员变量的名字要和客户端参数一一对应.
        private String userName ;
        private String password ;
        // 添加 get、set 方法
        。。。。。。。。

        /**************************************************
         * 用于对表单的数据进行验证, 该方法在调用 execute 之前
```

```
     * 被前端控制器调用,根据他的返回结果,来判断验证是否
     * 成功
     **********************************************/
    @Override
    public ActionErrors validate( ActionMapping mapping ,
                                  HttpServletRequest request )
    {// errors 用于存放错误的集合
        ActionErrors errors = new ActionErrors();
        if( userName == null || userName.trim().length() == 0 ){
            // ActionMessage 对象用来装载错误信息 初始化参数为资源文件中的键
            ActionMessage m = new ActionMessage( "errors.username.required" );
            errors.add( "error" , m );
        }
        if( password == null || password.trim().length()<3 ){
            ActionMessage m = new ActionMessage( "errors.password.required" ) ;
            errors.add( "error" , m ) ;
        }
        // 若返回的对象 size 为 0 则说明验证通过 否则进入 Action 中 input 属性指定的页面中
        return errors ;
    }
}

// 在 input 指定的页面中要显示 ActionErrors 中的错误信息 则要用到标签 : <html:errors/>
   errors.header : 错误信息的头
   errors.footer : 错误信息的主体
   引入标签库 : <%@taglib uri="http://struts.apache.org/tags-html" prefix="html"%>

// <bean:message key="login.password"/> : 用于显示资源文件中指定键值对应的内容
   由于规范的 jsp 页面中除了标签是不能含有静态文本内容的，故这个标签会被大量使用
   引入标签库 : <%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
```

(4) Action : 的实现

```
package com.kettas.struts ;
import org.apache.struts.action.* ;
import javax.servlet.http.* ;
import com.kettas.biz.*;

public class LoginAction extends Action{
    @Override
    public ActionForward execute( ActionMapping mapping , ActionForm form ,
                                  HttpServletRequest request ,  HttpServletResponse response )

    {    // 传统的获取表单数据方式
         // String name = request.getParameter("userName" ) ;
         // String pwd = request.getParameter("password" ) ;
         // 得到表单对象 从中获取表单数据
         LoginForm lf = (LoginForm)form ;
         // 调用 biz 对象,返回下一个页面 .
         try{
             UserService biz = new UserService();
             biz.login( lf.getUserName() , lf.getPassword() ) ;

             ActionForward af = mapping.findForward("ok");
             return af ;
         }catch( Exception e ){
             request.setAttribute( "errorMessage" , e.getMessage() ) ;
             return mapping.findForward( "error" );
         }
    }
}
```

(5) Biz 的实现 :

```
package com.kettas.biz ;
public class UserService{
    public void login( String name , String pwd ){
        if( ! name.equals( "liucy" ) ){
            throw new RuntimeException( "user " + name + " not found" ) ;
        }
        if( ! pwd.equals( "12345" ) ){
            throw new RuntimeException( "Invalid password" ) ;
```

```
                }
            }
        }
```

4，资源文件 .国际化

1, 默认资源文件 ： 当没有专门为每个用户准备资源文件
时 ,使用默认资源文件 .

```
<message-resources parameter="res.AppResources"/>
```

2,其他资源文件在命名上有一个规定 .
默认资源文件名_国家缩写 .
AppResources_zh
AppResources_ja

内容上是键值对 login.username=USER NAME: 前面是名称后面是显示的值
在 jsp 上的使用：<bean:define id="add"> **<bean:message key=" login.username "/>**</bean:define>

3, 所有的资源文件要以 utf-8 的编码来编写 .如果不是的话要进去以下转换。
a,用中文编写一个临时的资源文件 a.properties (中国字 GBK)

	临时文件中的编码	临时文件的名字	新文件的文件名
b, native2ascii -encoding	GBK	a.properties	AppResources_zh.properties

.

5，struts 中的异常处理 :==========

---> Action ---> Biz ---> Dao
Action 中对异常进行捕获 .

```
try{
        biz.fn();
        return mapping.findforward( "ok" ) ;
}catch( Exception e){
        return mapping.findforward( "error" );
}
```

系统级异常:
* 由于没个具体的技术操作导致的异常 ,
* 系统级异常通常是定义好的类型(SQLException , IOException)
* 一般发生在 DAO 层 .
* 在处里上要粒度粗一些 RuntimeException

应用级异常
* 由于用户违反了特定的业务逻辑导致的异常
* 应用级一般是用户自定义的异常 .
* 一般发生在 biz 层 .
* 粒度细一些 .

最好做一个异常的体系结构

实例 ：

(1) struts-config.xml 文件 ：

```
<?xml version="1.0" encoding="UTF-8" ?>
    <!DOCTYPE struts-config PUBLIC
            "-//Apache Software Foundation//DTD Struts Configuration 1.2//EN"
            "http://jakarta.apache.org/struts/dtds/struts-config_1_2.dtd">
    <struts-config>
        <form-beans>
            <form-bean name="loginForm" type="com.kettas.struts.LoginForm"/>
            动态 actionForm
            <form-bean name="regForm" type="org.apache.struts.action.DynaActionForm">
                <form-property name="userName" type="String"/>
                <form-property name="age" type="java.lang.Integer"/>
</form-bean>
        </form-beans>
        <!-- 全局变量 可用在一些 struts 标签中 如<html:link /> -->
        <global-forwards>
            <forward name="login" path="/login.jsp"/>
        </global-forwards>
        <action-mappings>
            <action path="/login" type="com.kettas.struts.LoginAction" name="loginForm"
                validate="true" input="/login.jsp">
                <!-- 将 Action 中抛出的异常捕获 并跳转到对应的错误页面
                    在错误页面中 还是用标签<html:errors />来显示错误
                    可见 这里 key 标签中指定的资源文件内容已经被封装到
```

```
                    ActionErrors 对象中去了 -->
                <exception type="com.kettas.struts.expt.SystemException"
                    path="/error.jsp" key="errors.system"/>
                <exception type="com.kettas.struts.expt.UserNotFoundException"
                    path="/error.jsp" key="errors.user"/>
                <exception type="com.kettas.struts.expt.PasswordException"
                    path="/error.jsp" key="errors.pwd"/>

                <!-- redirect="true"表示在一个新的连接中打开指定页面
                    默认在同一个连接中打开指定页面 -->
                <forward name="ok" path="/ok.jsp" redirect="true"/>
                <forward name="error" path="/error.jsp"/>
            </action>
        </action-mappings>
        //引入资源文件
        <message-resources parameter="res.AppResources"/>
    </struts-config>
```

(2) 定义根系统异常 作为其他一切系统异常的父类 这一步是有意义的

```
package com.kettas.struts.expt ;
// 继承至 RuntimeException 类而不是 Exception 如此 无需 try、catch 捕获 可自动抛出异常
public class SystemException extends RuntimeException{
    // 注意写这三个构造函数 后面都这么写
    public SystemException( ){
        super();
    }
    public SystemException( String msg ){
        super( msg ) ;
    }
    public SystemException( Throwable t ){
        super( t );
    }}
// 定义根业务异常 作为其他一切业务异常的父类
package com.kettas.struts.expt ;
public class BizException extends RuntimeException{
    ..
}
// 密码错误异常 继承至根业务异常
package com.kettas.struts.expt ;
public class PasswordException extends BizException{
    ...
}
// 用户未发现异常 继承至根业务异常
package com.kettas.struts.expt ;
public class UserNotFoundException extends BizException{
    ...
}
```

(3) dao 的实现 :

```
import ...
// 由于是采用 tomcat 的连接池获取连接 故这里引入这两个名字空间
import javax.sql.* ;
import javax.naming.* ;

public class UserDao{
    public User queryByLoginName( String loginName ){
        Connection conn = null ;
        PreparedStatement pstm = null ;
        ResultSet rs = null ;
        String sql = "select id , loginname , password from liucy_users where loginname=?" ;
        User u = null ;
        try{
            // 从连接池中获取连接的三个步骤
            Context ctx = new InitialContext();
            DataSource ds = (DataSource)ctx.lookup( "java:comp/env/oracle/ds" ) ;
            conn = ds.getConnection();

            pstm = conn.prepareStatement( sql ) ;
            pstm.setString( 1 , loginName ) ;
            rs = pstm.executeQuery();
```

```
                    if( rs.next() ){
                        u = new User();
                        u.setId( rs.getInt( 1 ) ) ;
                        u.setLoginName( rs.getString(2 ) ) ;
                        u.setPassword( rs.getString( 3 ) ) ;
                    }
                    return u ;
                }catch( Exception e){
                     e.printStackTrace();
                     // 抛出前面定义的系统异常 抛给 biz 再从 biz 抛出 抛给 Action 然后被系统捕获
                     throw new SystemException( e );
                }finally{
                    if( rs != null ) try{ rs.close(); }catch( Exception e){}
                    if( pstm != null ) try{ pstm.close();}catch( Exception e ){}
                    // 由于是连接池中的连接 故不会真的关闭 只是被连接池回收
                    if( conn != null ) try{ conn.close();}catch( Exception e ){}
                }}}
```

(4) biz 的实现 ：

```
public class UserService{
    public void login( String name , String pwd ){
        UserDao dao = new UserDao();
        User u = dao.queryByLoginName(name);
        if( u == null ){
            // 抛出用户不存在业务异常
            throw new UserNotFoundException( "user " + name + " not found" ) ;
        }
        if( ! u.getPassword().equals( pwd ) ){
            // 抛出密码错误业务异常
            throw new PasswordException( "Invalid password" );
        }}}
```

(5) Action 的实现 ：

```
public class LoginAction extends Action{
    @Override
    public ActionForward execute( ActionMapping mapping , ActionForm form ,
                                  HttpServletRequest request ,  HttpServletResponse response )
{   LoginForm lf = (LoginForm)form ;
        /*
        try{
        */
            // 这里会自动抛出 biz 和 dao 中的异常 抛给 struts 处理
            UserService biz = new UserService();
            biz.login( lf.getUserName() , lf.getPassword() ) ;
            ActionForward af = mapping.findForward("ok");
            return af ;
        /*
        // 若自己处理异常 则比较麻烦，可以在 struts 配置文件配置
        }catch( SystemException e ){
            request.setAttribute( "errorMessage" , "System is busy" ) ;
            return mapping.findForward( "error" );
        }catch( UserNotFoundException  e){
            request.setAttribute( "errorMessage" , e.getMessage() ) ;
            return mapping.findForward( "error" ) ;
        }catch( PasswordException e ){
            request.setAttribute( "errorMessage" , e.getMessage() ) ;
            return mapping.findForward( "error" ) ;
        }catch( Exception e ){
            request.setAttribute( "errorMessage" , "you catch an error" );
            return mapping.findForward( "error" ) ;
        }
        */
    }}
```

4 ActionMessages 对象 ：

```
public class MessageAction extends Action{
    @Override
    public ActionForward execute( ActionMapping mapping , ActionForm form ,
                                  HttpServletRequest request , HttpServletResponse response )
    {ActionMessages messages = new ActionMessages();
    // 用资源文件内容初始化
    ActionMessage m1 = new ActionMessage( "msg1" );
```

```
                ActionMessage m2 = new ActionMessage( "msg2" ) ;
                messages.add( "message" , m1 ) ;
                messages.add( "message" , m2 ) ;
                request.setAttribute( "msgs" , messages );
                return mapping.findForward( "ok" );
            }
        }
        // 在 jsp 中显示 ActionMessages 内容
        <html>
            <body>
                This message jsp
                <!-- name 指定命名属性 -->
                <html:messages id="m" name="msgs">
                    <h1>${m}</h1>
                </html:messages>
            </body>
        </html>
<bean:message />
<html:errors />

html: 处理页面的显示问题
bean: 定义变量和 javabean   <c:set />
logic: <c:if > <c:forEach>
        <forward name="ok" path="/path.jsp"/>

html:
  <html:link /> <===> <a href="">

  <html:link forward="xxxx" page="xxx">Test</html:link>
      page : 指向一个明确的连接地址 .
      forward : 全局的 forward 的 逻辑名字 .
    * 可以自动加应用名称
    * 可以使用逻辑名字 .
    * 在不支持 cookie 的情况下, 自动增加 jsessionid
  <html:link forward="ok"></html:link>

 <html:form action="/login.do" method="">
    <input type="text" name="age"/>
    <html:text property="age"/>

    <input type="password" name=""/>
    <html:password />

    <select name="">
      <option>aaa</option>
    </slelect>

    <html:select>
      <html:option></html:option>
    </html:select>

    <input type="hidden" name="" value=""/>
    <html:hidden />
</html:form>

<html:messages id="" name="">
</html:messages>
name : 被迭代的消息结合在作用域中的名字 .
id   : 当前整备迭代的消息对象 .
* 被该标签迭代的集合类型是 ActionMessages 类型 .
  集合中的每个元素都是 一个 ActionMessage , 他
  能够封装资源文件中的数据 .
<html:errors />

定义一个变量：id 表示变量名，value：表示变量值
<bean:define id="a" value="1234"/>
<bean:define id="">
      XXXXXXXX
</bean:define>
定义命名属性：
<bean:define id="a" name="data"/>
Action ---> a.jsp
request.setAttribute( "a" , "hello")
${ a }
```

第三天：标签的使用==========

5 struts 标签 :

```
<html:link forward="login"page="/webdir/login.jsp">
        Test</html:link>
      page : 指向一个明确的连接地址 .
      forward : 全局的 forward 的逻辑名字, 在配置文件
中指定的 :
    <global-forwards>
        <forward name="login" path="/login.jsp"/>
      </global-forwards>
    * 可以自动加应用名称
    * 可以使用逻辑名字, 在配置文件中说明实际路径
    * 在不支持 cookie 的情况下, 自动增加 jsessionid

     在 html 中表单标签用 struts 标签代替 :
     <!-- action 中无需写应用名 会自动添加上 -->
      <html:form action="/login.do" method="">
        <-- <input type="text" name="age"/> -->
            <html:text property="age" value=""/>
      <!-- <input type="password" name="pwd"/> -->
            <html:password property="pwd"/>
 <!--<select name="">
    <option>aaa</option>
 </slelect> -->
     <html:select>
        <html:option></html:option>
     </html:select>

  <!-- <input type="hidden" name="" value=""/> -->
            <html:hidden />
     </html:form>

     html :
          <html:link  forward|page="/" >
          <html:form action="" method="">
           <html:text property="" value=""/>
           <html:password  property=""/>
           <html:select property=""/>
       <html:option/>
       <html:radio  property=""  />
       <html:checkbo property=""/>
     </html:form>
     <html:errors/>
     <html:messages id="m" name="">
              ${m}
     </html:messages>

     bean :
     <!-- 在当前页面范围内 定义一个变量 -->
      <bean:define id="" value=""/>
       <bean:define id="">
                  ...
       </bean:define>
      <!-- 将某一个命名属性对应的值赋予变量 -->
      <bean:define id="a" name="student"/>
 <bean:define id="a" name="student" property="name"/>
     相当于 : String a = student.getName();
<!-- 将客户端提交的指定表单数据赋予变量 -->
      <bean:parameter id="a" name="age"/>
      <!-- 将客户端提交的指定 cookie 的值赋予变量 -->
       <bean:cookie id="c" name="JSESSIONID"/>
       <!-- 显示资源文件内容 -->
       <bean:message key=""/>

     logic :
     <logic:present parameter|cookie|name="age">
                    xxxxxxx ;
                    xxxxxxx ;
                    xxxxxxx ;
                </logic:present>
                <logic:notPresent >
                    Xxxxx
                </logic:notPresent>

 <logic:greaterThan  parameter="age" value="23">

 </logic:greaterThan>
 <logic:greateThan cookie="age" value="23">
                    cxddddd
 </logic:greateThan >
<logic:greateThan
name="student"property="age"value="23">
 </logic:greateThan>

<logic:equal>
  <logic:notEqual>
 <logic:greaterThan>
 <logic:lessThan>
 <logic:greaterEqual>
 <logic:lessEqual>
 <!-- 循环标签 -->
 <logic:iterate id="" collection="">
 </logic:iterate>
```

==========

request ----> ActionForm ---> Action

简化对 ActionForm 的开发 .直接在配置文件配置，不用写单独的类

DynaActionForm (动态 ActionForm)

1) 在配置文件中配 .

```
  <form-beans>
       <form-bean name="" type=""/>
       <form-bean name="regForm" type="org.apache.struts.action.DynaActionForm">
          <form-property name="userName" type="String"/>
          <form-property name="age" type="java.lang.Integer"/>
       </form-bean>
  </form-beans>
```

2) Action 中使用.

```
    execute( ActionForm form ){
           DynaActionForm af = ( DynaActionForm )form ;
               af.get( "age" ) ;
               af.get( "userName" ) ;
```

```
        }
==========================================
login , register , query
```

7 关于 MappingDispatchAction, DispatchAction, LookupDispatchAction

(1) MappingDispatchAction : 容许一个 action 中包含多个方法，分别处理与之对应的请求

实例 :

配置文件 :

```
<form-bean name="dynaLoginForm" type="org.apache.struts.action.DynaActionForm">
     <form-property name="userName" type="java.lang.String"/>
     <form-property name="password" type="java.lang.String"/>
</form-bean>

<form-bean name="dynaRegForm" type="org.apache.struts.action.DynaActionForm">
     <form-property name="userName" type="java.lang.String"/>
     <form-property name="password" type="java.lang.String"/>
     <form-property name="email" type="java.lang.String" />
</form-bean>
...
...
<action parameter="login" path="/login" type="com.kettas.struts.
     UserSelfServiceAction" name="dynaLoginForm">
     <exception type="com.kettas.struts.expt.SystemException" path="/error.jsp"
          key="errors.system"/>
     <exception type="com.kettas.struts.expt.UserNotFoundException"
          path="/error.jsp" key="errors.user"/>
     <exception type="com.kettas.struts.expt.PasswordException" path="/error.jsp"
          key="errors.pwd"/>
     <forward name="ok" path="/ok.jsp" redirect="true"/>
</action>

<action parameter="register" path="/reg" type="com.kettas.struts.
     UserSelfServiceAction" name="dynaRegForm">
     <forward name="ok" path="/regok.jsp"/>
</action>
...
```

Action :

```
public class UserSelfServiceAction extends MappingDispatchAction{
     // 用于处理用户登陆请求 .
     public ActionForward login( ActionMapping mapping , ActionForm form ,
                              HttpServletRequest request , HttpServletResponse response )
    {
          DynaActionForm daf = (DynaActionForm)form ;
          String userName = (String)daf.get( "userName" );
          String password = (String)daf.get( "password" ) ;

          UserService biz = new UserService();
          biz.login( userName , password );
          return mapping.findForward( "ok" ) ;
     }
     // 负责处理用户的注册请求
     public ActionForward register( ActionMapping mapping , ActionForm form ,
                              HttpServletRequest request ,   HttpServletResponse response )
    {
           DynaActionForm daf = ( DynaActionForm ) form ;
           System.out.println( daf.get( "userName" ) ) ;
           System.out.println( daf.get( "password" ) ) ;
           System.out.println( daf.get( "email" ) ) ;
           return mapping.findForward( "ok" ) ;
    }}
```

(2) DispatchAction :

1 将相关的操作合并在同一个 Action 中处理，这些相关的操作，必须使用同一个 ActionForm

2 客户端提交的参数值决定使用 Action 中的哪个方法

3 不能覆盖 execute 方法，为每一个请求单独写一个方法

4 采用 DispatchAction 的方式简化了配置文件

实例 :

配置文件 :

```
<!-- parameter 指定客户端请求中决定 action 方法的那个参数 -->
<action parameter="method" path="/all" type="com.kettas.struts.
      DispatchActionSample" name="allForm">
      <exception type="com.kettas.struts.expt.SystemException" path="/error.jsp"
            key="errors.system"/>
      <forward name="ok" path="/ok.jsp" redirect="true"/>
      <forward name="regOk" path="/regok.jsp"/>
</action>

Action :
      public class DispatchActionSample extends DispatchAction{
            public ActionForward login( ActionMapping mapping , ActionForm form ,
                                    HttpServletRequest request , HttpServletResponse response )
            {
                  DynaActionForm daf = (DynaActionForm)form ;
                  String userName = (String)daf.get( "userName" ) ;
                  String password = (String)daf.get( "password" ) ;
                  UserService service = new UserService();
                  service.login( userName , password ) ;
                  return mapping.findForward( "ok" ) ;
            }
            public ActionForward register( ActionMapping mapping , ActionForm form ,
                                    HttpServletRequest request ,   HttpServletResponse response )
            {
                  DynaActionForm daf = (DynaActionForm)form ;
                  System.out.println( daf.get( "userName" ) );
                  System.out.println( daf.get( "password" ) ) ;
                  System.out.println( daf.get( "email" ) ) ;
                  return mapping.findForward( "regOk" ) ;
            }}
      客户端 :
            <a href="/webdir/all.do?method=login">登录</a>
            <a href="/webdir/all.do?method=register">注册</a>
            Method 有不同的参数，对应处理的 action 方法
(3) LookupDispatchAction :
      (1) 不能覆盖父类的 execute 方法，为每一个按钮单独写一个方法
      (2) 覆盖父类 getKeyMethodMap 方法，在这个方法中指明按钮的值与方法名称的对应关系
      (3) 按钮的值，不能直接写死在程序中，要来自于资源文件
      (4) 在配置文件中对 Action 进行配置
          <action parameter="指明按钮的名称" path="" type="" name="" ></action>
      (5) 适合一个表单多种请求方式的时候

      实例 :
            配置文件 :
                  <action parameter="btn" path="/math" type="com.kettas.struts.MathAction"
                        name="allForm">
                        <forward name="ok" path="/submit.jsp"/>
                  </action>

            Action :
                  public class MathAction extends LookupDispatchAction{
                        // 覆盖父类 getKeyMethodMap 方法，在这个方法中指明按钮的值与方法名称的对应关系
                        @Override
                        public Map getKeyMethodMap(){
                              // 按钮的值应来自于资源文件 在 map 中保存是资源文件中的 key,values 是对应的方法
                              Map m = new HashMap();
                              m.put( "btn.add" , "addOperate" );
                              m.put( "btn.subtract" , "subOperate" );
                              return m ;
                        }
                        // 加法运算
                        public ActionForward addOperate( ActionMapping mapping,  ActionForm form ,
                                    HttpServletRequest request , HttpServletResponse response )
                        {
                              DynaActionForm daf = (DynaActionForm)form ;
                              Integer a = (Integer)daf.get( "a" ) ;
                              Integer b = (Integer)daf.get( "b" ) ;
                              int ret = a.intValue() + b.intValue();
                              request.setAttribute( "ret" , ret ) ;
```

```
                return mapping.findForward( "ok" ) ;
            }

            // 减法运算
            public ActionForward subOperate( ActionMapping mapping, ActionForm form ,
                                HttpServletRequest request , HttpServletResponse response )
            {
                DynaActionForm daf = (DynaActionForm)form ;
                Integer a = (Integer)daf.get( "a" ) ;
                Integer b = (Integer)daf.get( "b" ) ;
                int ret = a.intValue() - b.intValue();
                request.setAttribute( "ret" , ret ) ;
                return mapping.findForward( "ok" ) ;
            }
        }
```

客户端 :

```
<%@page contentType="text/html"%>
<%@page isELIgnored="false"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<html>
    <body>
        <!-- 为变量赋值 获得资源文件的值 -->
        <bean:define id="add">
            <bean:message key="btn.add"/>
        </bean:define>
        <bean:define id="sub">
            <bean:message key="btn.subtract"/>
        </bean:define>

        <form action="/strutsapp/math.do" method="GET">
            Number A :<input type="text" name="a"/><br/>
            Number B :<input type="text" name="b"/><br/>
            <!-- 按钮的 name 属性与 action 配置中的 parameter 属性是对应的 -->
            <input type="submit" name="btn" value="${add}"/>
            <input type="submit" name="btn" value="${sub}"/>

        </form>

        <!-- 显示计算结果 -->
        <c:if test="${!empty ret}">
            <h2>ret = ${ret}</h2>
        </c:if>
    </body>
</html>
```

```
================================ ================================================
n m ( m-1 ) * n ---> Begin
    (m-1)*n + n ---> end
   select id
     from
          ( select id , rownum rn from product )
     rn> ? and rn < ?
==================================================
```

ResultSet
1) 改 bug.
2) 增加上一页 , 下一页 功能 .
3) 总页数 通过 ServletContextListener
保存到 ServvletContext 对象中 .

第五天:

Validation --> 表便验证 .

Overide
validate()
1) dynaActionForm
2)
3)

validation :
1,varliator.jar | jakarta-ora.jar .

|-> 在这些类中实现了通用的验证方法.

2, validator-rules.xml .
|-> 相当于类的使用说明. 写给 struts 看的.
not null --->class ---> function

3, validation.xml :
|-> 相当于程序员编写的需求说明书 .
loginForm
|--> userName
|--> not null .

Validation 框架的使用方式 :

1) 程序员负责编写需求说明书 . 说明
哪些表单 , 哪些数据,需要做什么验证 .

2) struts 通过阅读 validator-rules.xml,
调用相应方法,实现验证逻辑 .

==========

使用 validation 框架的准备工作 :

1) 保证 validation.jar , jakarta-ora.jar
保存在应用的/WEB-INF/lib

2) 保证两个配置文件 validator-rules.xml
validation.xml 要保存到/WEB-INF 中 .

3) 将 validation 以插件的方式嵌入到 struts 中.
修改 struts-config.xml ,插件的配置标签
可以从 validator-rules.xml 的注释中拷贝.

4) 确保需要验证的请求 **validate="true"**

```
<action path="login" type="xxxAction"
        name="xxxForm" validate="true"
        input="xxx.jsp"/>
```

5) 如果要通过 Validation 作验证的话 .

ActionForm
- DynaValidatorActionForm 在 validation.xml 配置文件中的 form 标签中的属性为 path
- DynaValidatorForm 在 validation.xml 配置文件中的 form 标签中的属性为 name

DynaActionForm
- org.apache.struts.validator.DynaValidatorForm
- DynaValidatorActionForm

6) 将错误提示信息从 validator-rules.xml 中拷贝到自己的资源文件中 .

编辑需求说明书(validation.xml) 提出自己的验证需求

formset

|-> form
|-> field
|-> form

tiles .

```
<script>
    function fn(){
        xxxxxxxxx ;
        xxxxxxxxxx;
        return false ;
    }
</script>
<form action="" method="" onsubmit="return xxxx();">
</form>
```

1) 将 validation 写的 javascript 代码嵌入到自己的网页中.

```
<html:javascript formName=""/>
```

2) 获取 javascript 验证函数的名字 .

allForm ---> valdiateAllForm
abc -------> validateAbc

login.do
regster.do
AllForm
MappingDisaptcherAction

```
<action path="/login" type="" name="all" validate="true" >
<action path="/reg" type="" name="all" validate="true">
```

9 struts 中的模板

(1) **在 struts-config.xml** 文件中指定使用模板 : 通过插件件的方式

```
<action-mappings>  ... </action-mappings>
<plug-in className="org.apache.struts.tiles.TilesPlugin" >
    <!-- 指定模板配置文件的路径 -->
    <set-property property="definitions-config" value="/WEB-INF/tiles-defs.xml" />
```

```
        <!-- 使模板识别生效 -->
        <set-property property="moduleAware" value="true" />
    </plug-in>
```

(2) tiles-defs.xml 文件 :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE tiles-definitions PUBLIC
       "-//Apache Software Foundation//DTD Tiles Configuration 1.1//EN"
       "http://jakarta.apache.org/struts/dtds/tiles-config_1_1.dtd">
<tiles-definitions>
  <!-- template.jsp 是个模板页面 里面只有一个页面布局 -->
  <definition name="base" path="/tiles/template.jsp">
    <!-- 向模板页面的指定位置插入其他页面 -->
      <put name="header" value="/tiles/header.jsp"/>
      <put name="menu" value="/tiles/menu.jsp"/>
  </definition>
  <!-- 继承至上面的添加了部分内容的模板 再添加新的内容 -->
  <definition name="register" extends="base">
          <put name="body" value="/tiles/regBody.jsp"/>
  </definition>

  <!-- 继承至上面的添加了部分内容的模板 再添加新的内容 -->
  <definition name="login" extends="base">
          <put name="body" value="/tiles/loginBody.jsp"/>
  </definition>
</tiles-definitions>
```

(3) template.jsp 页面 :

```
<%@page contentType="text/html;charset=utf-8"%>
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>
<html>
     <body>
          <table width="100%" height="100%">
               <tr height="10%" bgcolor="gray">
                    <td colspan="2" align="center">
                         <!-- 指定此地可以插入新的页面 -->
                         <tiles:insert attribute="header"/>
                    </td>
               </tr>
               <tr height="90%">
                    <td width="15%" bgcolor="blue">
                         <!-- 指定此地可以插入新的页面 -->
                         <tiles:insert attribute="menu"/>
                    </td>
                    <td width="85%" align="center">
                         <!-- 指定此地可以插入新的页面 -->
                         <tiles:insert attribute="body"/>
                    </td>
               </tr>
          </table>
     </body>
</html>
```

(4) 使用的时候 页面格式如下 register.jsp :

```
<%@taglib uri="http://struts.apache.org/tags-tiles" prefix="tiles"%>

<!-- definition 属性指定 tiles-defs.xml 文件中装配完成的页面名称 -->
<tiles:insert definition="register"/>

<!-- 若在 tiles-defs.xml 文件中没有声明 则要这么写 :
<tiles:insert page="/tiles/template.jsp">
     <tiles:put name="header" value="/tiles/header.jsp"/>
     <tiles:put name="menu" value="/tiles/menu.jsp"/>
     <tiles:put name="body" value="/tiles/regBody.jsp"/>
</tiles:insert>
-->
```

一个客户在浏览器的地址栏输入了如下:
URL: http://www.tarena.com/webdev/account/deposit?accno=1212&amt=1000
调用 EG 的方法 F 可以获得初始参数 interestRate 的值。

在 accountServlet 中调用 HttpServletRequest 的 getRequestURI 返回 H ，
调用 getQueryString 返回 B ，调用 getContextPath 返回 A ，
调用 getServletPath 返回 C ，调用 getPathInfo 返回 D 。

A. /webdev
B. accno=1212&amt=1000
C. /account
D. /deposit
E. Servletconfig
F. getInitParameter
G. HttpServlet
H. /webdev/account/deposit

Struts 2

1， Struts2 的体系和 struts1 的体系差别很大，因为 struts2 使用 WebWork 的设计核心，而不是原来 struts1 的设计核心。
Struts2 大量使用拦截器来处理用户请求，从而允许用户的业务逻辑控制器与 Servlet API 分离。

2， struts2 框架的大致处理流程如下：

①浏览器发送请求，例如请求 /mypage.action 或 /report/myreport.pdf 等

②核心控制器 FilterDispatcher 根据请求决定调用合适的 Action

③WebWork 的拦截器链自动对请求应用通用功能，例如 workflow，validation 或文件下载和上传

④回调 Action 的 execute 方法(其实可以是任意方法)，该方法 execute 方法先获得用户的请求参数，然后执行某种数据操作，调用业务逻辑组组件来处理用户请求。

⑤Action 的 execute 方法处理结果信息将被输出发哦浏览器中，可以是 html 页面，pdf 还有 **jsp，Velocity，FreeMarker** 等技术模板。

3，在 struts2 中的 web.xml 配置 加拦截器（拦截器是整个 struts 的核心技术）

```
<?xml version="1.0" encoding="GBK"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
 <!-- 定义 Struts2 的 FilterDispathcer 的 Filter -->
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class>
    </filter>
 <!-- FilterDispatcher 用来初始化 struts2 并且处理所有的 WEB 请求。 -->
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

3， struts.xml 的配置

```
<struts>
 <!-- include 节点是 struts2 中组件化的方式 可以将每个功能模块独立到一个 xml 配置文件中 然后用 include 节点引用 -->
    <include file="struts-default.xml"></include>
    <!-- package 提供了将多个 Action 组织为一个模块的方式
        package 的名字必须是唯一的 package 可以扩展 当一个 package 扩展自
        另一个 package 时该 package 会在本身配置的基础上加入扩展的 package
        的配置 父 package 必须在子 package 前配置
        name：package 名称 extends:继承的父 package 名称
        abstract:设置 package 的属性为抽象的 抽象的 package 不能定义 action 值 true:false
        namespace:定义 package 命名空间 该命名空间影响到 url 的地址，例如此命名空间为/test 那么访问是的地址为
          http://localhost:8080/struts2/test/XX.action    -->
<package name="com.kay.struts2" extends="struts-default" namespace="/test">
    <interceptors>
        <!-- 定义拦截器 name:拦截器名称  class:拦截器类路径  -->
        <interceptor name="timer" class="com.kay.timer"></interceptor>
        <interceptor name="logger" class="com.kay.logger"></interceptor>
        <!-- 定义拦截器栈 -->
        <interceptor-stack name="mystack">
            <interceptor-ref name="timer"></interceptor-ref>
            <interceptor-ref name="logger"></interceptor-ref>
        </interceptor-stack>
    </interceptors>
    <!-- 定义默认的拦截器 每个 Action 都会自动引用 如果 Action 中引用了其它的拦截器 默认的拦截器将无效 -->
    <default-interceptor-ref name="mystack"></default-interceptor-ref>
    <!-- 全局 results 配置 -->
    <global-results><result name="input">/error.jsp</result> </global-results>
    <!-- Action 配置 一个 Action 可以被多次映射(只要 action 配置中的 name 不同)
        name：action 名称 class: 对应的类的路径  method: 调用 Action 中的方法名 -->
    <action name="hello" class="com.kay.struts2.Action.LoginAction">
        <!-- 引用拦截器 name:拦截器名称或拦截器栈名称  -->
        <interceptor-ref name="timer"></interceptor-ref>
        <!-- 节点配置 name : result 名称 和 Action 中返回的值相同
            type : result 类型 不写则选用 superpackage 的 type struts-default.xml 中的默认为 dispatcher
         -->
     <result name="success" type="dispatcher">/talk.jsp</result>
```

批注 [U13]: Struts1 是*.do 来处理

批注 [U14]: 多个拦截器组合成一个拦截器栈，同时拥有几个拦截器的功能

批注 [U15]: Struts2 的 name 就是处理 url 的前半部分，/hello.action，而 struts1 的 name 是该 action 关联 ActionForm，path 才是处理的 url

```
        <result name="error" type="dispatcher">/error.jsp</result>
        <result name="input" type="dispatcher">/login.jsp</result>
         <!-- 配置 Action 返回 cancel 时重定向到 Welcome 的 Action-->
        <result name="cancel" type="redirect-action">Welcome</result>
         <!—异常处理  result 表示出现异常时返回的 name 为 success 的结果处理—》
        <exception-mapping exception=" java.lang.Exception"  result=" success" />
        <!-- 参数设置 name：对应 Action 中的 get/set 方法 -->
        <param name="url">http://www.sina.com</param>
       </action>
    </package>
         <!—引用国际化文件的 base 名-->
        <constant name=" struts2.custom.i18n.resources" value=" messageResource"
</struts>
```

4，struts2 的 action 编写

例 1 HelloWorld.jsp

```
<% @ page contentType = " text/html; charset=UTF-8 " %>
<% @ taglib prefix = " s " uri = " /struts-tags " %>
< html >
< head >
    < title > Hello World! </ title >
</ head >
< body >
    < h2 >< s:property value ="message" /></ h2 >
</ body >
</ html >
```

例 1 classes/tutorial/HelloWorld.java

```
 import com.opensymphony.xwork2.ActionSupport;
 public class HelloWorld extends ActionSupport  {
     private String message;
     public String getMessage()  {
         return message;
    }
   public void tring setMessage(String message)  {
         this.message=message;
    }
    @Override
     public String execute()  {
        message = " Hello World, Now is " + DateFormat.getInstance().format( new Date());
         return success;
    }
}
```

例 1 classes/struts.xml 中 HelloWorld Action 的配置

```
< package name ="ActionDemo" extends ="struts-default" >
    < action name ="HelloWorld" class ="tutorial.HelloWorld" >
        < result > /HelloWorld.jsp </ result >
    </ action >
</ package >
```

4，　拦截器的编写

编写权限控制器

```
public class AuthorityInterceptor extends AbstractInterceptor {
    private static final long serialVersionUID = 1358600090729208361L;
    //拦截 Action 处理的拦截方法
    public String intercept(ActionInvocation invocation) throws Exception {
        ActionContext ctx=invocation.getInvocationContext();    // 取得请求相关的 ActionContext 实例
        Map session=ctx.getSession();
        String user=(String)session.get("user"); //取出名为 user 的 session 属性
        //如果没有登陆，或者登陆所有的用户名不是 aumy，都返回重新登陆
        if(user!=null && user.equals("aumy")){
            return invocation.invoke();
        }
        //没有登陆，将服务器提示设置成一个 HttpServletRequest 属性
        ctx.put("tip","您还没有登录，请登陆系统");
        return Action.LOGIN;
    }
}
```

Struts2.xml 的配置文件

批注 [U16]: Struts2 只用引入一个标签就可以

批注 [U17]: 在 struts2 中通过标签直接访问访问 action 中的属性值。当要访问集合，数组等时就要用表达式语言来访问这些对象，OGNL 是一种数据访问语言

批注 [U18]: 没有 struts1 中的 actionForm，struts2 只用设置属性的 get 和 set 方法就可以得到值

批注 [U19]: 方法名可以任意，只是 execute 是默认的

批注 [U20]: 可以继承 Intercept 接口还要实现 init()和 destroy（）方法，但是最好使用抽象方法 AbstractInterceptor

批注 [U21]: 该方法就像 action 的 execute()方法一样，放回一个逻辑视图的字符串。

```
<struts>
    <include file="struts-default.xml"/>

    <!—受权限控制的 Action 请求配置-->
    <package name="authority" extends="struts-default">
        <interceptors>
            <!--定义一个名为 authority 的拦截器—>
            <interceptor  class="com.aumy.struts.example.intercepter.AuthorityInterceptor"  name="authority"/>
            <!--定义一个包含权限检查的拦截器栈—>
            <interceptor-stack name="mydefault">
                <interceptor-ref name="defaultStack"/>   <!--配置内建默认拦截器-->
                <interceptor-ref name="authority"/>  <!--配置自定义的拦截器-->
            </interceptor-stack>
        </interceptors>

        <default-interceptor-ref name="mydefault" />
        <!--定义全局 Result-->
        <global-results>
            <result name="login">/login.jsp</result>
        </global-results>

        <action name="show" class="com.aumy.struts.example.LoginAction"
            method="show">
            <result name="success">/show.jsp</result>
        </action>
        <action name="add" class="com.aumy.struts.example.LoginAction"  method="add">
            <result name="success">/add.jsp</result>
        </action>
    </package>
</struts>
```

批注 [U22]: 定义之后整个包都会有权限过滤

还要一种方法拦截器，可以对某个 action 的方法进行拦截 编码类似 action 拦截器

```
  public class MyInterceptor3 extends MethodFilterInterceptor {
    @Override
    protected String doIntercept(ActionInvocation invocation) throws Exception {
        System.out.println("use MethodFilterInterceptor");
        String result = invocation.invoke();
        return result;
    }
}
```

只是在配置的时候和其他 interceptor 有些区别：

```
<interceptor name="myInterceptor3" class="com.langhua.interceptor.MyInterceptor3">
                <param name="excludeMethods">execute，login</param>  <!-- execute，login 两个方法不需要拦截
               <!—addmessage 方法需要拦截 可以指名多个方法，只需要用逗号隔开
                <param name="includeMethods">addmessage</param>
</interceptor>
```

6，拦截结果的监听器，用来精确定义在 execute 方法执行之后，在处理物力资源转向之前的动作，这个监听器是通过手动注册到拦截器内部的

```
public class MyListener implements PreResultListener {
    public void beforeResult(ActionInvocation invocation, String resultCode) {
        System.out.println(“放回的逻辑视图是:”+resultCode);
    }
}
```

批注 [U23]: 必须实现 PreResultListener 接口

批注 [U24]: 就是 execute 的返回值

然后再在拦截器里面调用 invocation.addPreResultListener(new MyListener());

监听器是在这个拦截器完成别的拦截器之后调用的

5，struts2 的标签库 相比 struts1，struts2 提供了大量的标签，且更加强大 ，且主要有，1 UI 用户界面标签（html 页面显示） 2 非 UI 主要是数据访问，逻辑控制的标签 3 Ajax 支持的标签

6，Struts2 的（客户端和服务端）校验更为方便更容易国际化处理

7，国际化处理：

8，支持文件的上传和下载

Struts1 和 Struts2 的区别和对比:

Action 类:
• Struts1 要求 Action 类继承一个抽象基类。Struts1 的一个普遍问题是使用抽象类编程而不是接口。
• Struts 2 Action 类可以实现一个 Action 接口，也可实现其他接口，使可选和定制的服务成为可能。Struts2 提供一个 ActionSupport 基类去实现常用的接口。Action 接口不是必须的,任何有 execute 标识的 POJO 对象都可以用作 Struts2 的 Action 对象。

线程模式:
• Struts1 Action 是单例模式并且必须是线程安全的，因为仅有 Action 的一个实例来处理所有的请求。单例策略限制了 Struts1 Action 能作的事，并且要在开发时特别小心。Action 资源必须是线程安全的或同步的。
• Struts2 Action 对象为每一个请求产生一个实例，因此没有线程安全问题。（实际上，servlet 容器给每个请求产生许多可丢弃的对象，并且不会导致性能和垃圾回收问题）

Servlet 依赖:
• Struts1 Action 依赖于 Servlet API ,因为当一个 Action 被调用时 HttpServletRequest 和 HttpServletResponse 被传递给 execute 方法。
• Struts 2 Action 不依赖于容器，允许 Action 脱离容器单独被测试。如果需要，Struts2 Action 仍然可以访问初始的 request 和 response。但是，其他的元素减少或者消除了直接访问 HttpServetRequest 和 HttpServletResponse 的必要性。

可测性:
• 测试 Struts1 Action 的一个主要问题是 execute 方法暴露了 servlet API（这使得测试要依赖于容器）。一个第三方扩展－－Struts TestCase－－提供了一套 Struts1 的模拟对象（来进行测试）。
• Struts 2 Action 可以通过初始化、设置属性、调用方法来测试，“依赖注入”支持也使测试更容易。

捕获输入:
• Struts1 使用 ActionForm 对象捕获输入。所有的 ActionForm 必须继承一个基类。因为其他 JavaBean 不能用作 ActionForm，开发者经 常创建多余的类捕获输入。动态 Bean（DynaBeans）可以作为创建传统 ActionForm 的选择，但是，开发者可能是在重新描述(创建)已经存 在的 JavaBean（仍然会导致有冗余的 javabean）。
• Struts 2 直接使用 Action 属性作为输入属性，消除了对第二个输入对象的需求。输入属性可能是有自己(子)属性的 rich 对象类型。Action 属性能够通过 web 页面上的 taglibs 访问。Struts2 也支持 ActionForm 模式。rich 对象类型，包括业务对象，能够用作输入/输出对象。这种 ModelDriven 特性简化了 taglib 对 POJO 输入对象的引用。

表达式语言:
• Struts1 整合了 JSTL，因此使用 JSTL EL。这种 EL 有基本对象图遍历，但是对集合和索引属性的支持很弱。
• Struts2 可以使用 JSTL，但是也支持一个更强大和灵活的表达式语言－－"Object Graph Notation Language" (OGNL).

绑定值到页面（view）:
• Struts 1 使用标准 JSP 机制把对象绑定到页面中来访问。
• Struts 2 使用 "ValueStack"技术，使 taglib 能够访问值而不需要把你的页面（view）和对象绑定起来。ValueStack 策略允许通过一系列名称相同但类型不同的属性重用页面（view）。

类型转换:
• Struts 1 ActionForm 属性通常都是 String 类型。Struts1 使用 Commons-Beanutils 进行类型转换。每个类一个转换器，对每一个实例来说是不可配置的。
• Struts2 使用 OGNL 进行类型转换。提供基本和常用对象的转换器。

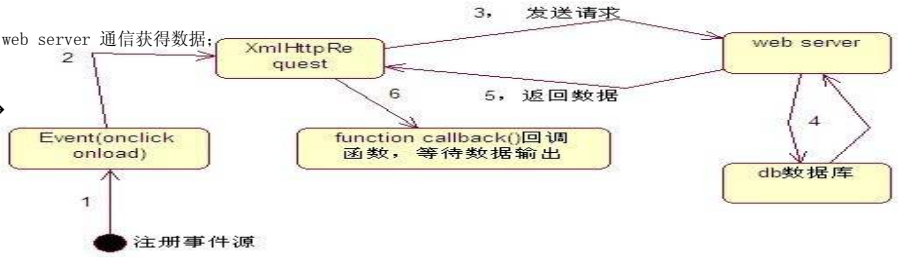
校验:
• Struts 1 支持在 ActionForm 的 validate 方法中手动校验，或者通过 Commons Validator 的扩展来校验。同一个类可以有不同的校验内容，但不能校验子对象。
• Struts2 支持通过 validate 方法和 XWork 校验框架来进行校验。XWork 校验框架使用为属性类类型定义的校验和内容校验，来支持 chain 校验子属性

Action 执行的控制:
• Struts1 支持每一个模块有单独的 Request Processors（生命周期），但是模块中的所有 Action 必须共享相同的生命周期。
• Struts2 支持通过拦截器堆栈(Interceptor Stacks)为每一个 Action 创建不同的生命周期。堆栈能够根据需要和不同的 Action 一起使用

Ajax 部分

1 定义：多种应用异步通信，能够数据的局部刷新，使用户有不间断的体验
ajax 的四项技术：
javascript, css, dom, XMLHttpRequest

2 四者之间的关系：
1. javascript 创建 XMLHttpRequest 对象并用它与 web server 通信获得数据;
2. javascript 使用 dom 的 api 将数据更新到页面;
3. javascript 将 css 应用到 HTML 的元素上
Ajax 的传输步骤---------->



3 ajax 的编程步骤:
1. 创建 XMLHttpRequest 对象 xhr;
2. 使用 xhr 的 open 函数打开资源; open("GET or POST" , "传向的页面"+如果是 GET 要加参数(不用加/));
3. 使用 xhr 的 onreadystatechange 属性注册处理应答的回调函数的**句柄**;(为什么只传句柄？如果传 display()的话相当于传入的是函数值，
而传入 display()的话是将整个函数传给它，当有变化时交个这个函数来处理传入 display()还会出错？)
4. (在使用 POST 方法使用)使用 xhr 的 setRequestHeader 设置请求头。通常设置 content-type
请求头，可能的值是: application/x-www-form-urlencoded 和 text/xml;
5. 使用 xhr 的 send 方法发送请求;
6. 编写回调函数处理应答：在此函数里通过 xhr 的 **readyState** 属性判断通信是否结束(等于 4 结束)；然后再通过 xhr 的 status 属性判断 web server 是否正确处理应答(等于 200 正确)，如果正确处理应答，应答的文本存放在 xhr 的 responseText 属性中，应答是 xml 再将生成的 xml 文档放在 xhr 的 responseXML 中 传 XML 文档只能用 POST 方法传

res.getCharactorEncoding();可获得 res 的字符编码
res.setCharactorEncoding("UTF-8");

用 DOM api 解析 XML 文档的步骤:
1. 创建 DocumentBuilderFactory:
DocumentBu ilderFactory dbf = DocumentBuilderFactory.newInstance();
2. (可选) 设置 dbf 的属性:
设置合法性检测： dbf.setValidating(true);
设置处理名字空间： dbf.setNamespaceAware(true);
3.创建 DocumentBuilder:
DocumentBu ilder db = dbf.newDocumentBuilder();
4a.解析 XML 文档:
Document doc = db.parse(xmlResource);
4b.生成 XML 文档:
Document doc = db.newDocument();

4 XMLHTTPRequest 的属性和方法介绍
方法属性：
open(string method, string url, boolean asynch, string username, string password)：post 还是 get，url 地址，同步还是异步 后面三个参数是可选的
void send(content)：
string getAllResponseHeaders()
void setRequestHeader(string header, string value)：这个方法为 HTTP 请求中一个给定的首部设置值。它有两个参数，第一个串表示要设置的首部，第二个串表示要在首部中放置的值。需要说明，这个方法必须在调用 open()之后才能调用。
string getResponseHeader(string header)：
onreadystatechange ：每个状态改变时都会触发这个事件处理器，通常会调用一个 JavaScript 函数、回调函数
readyState：请求的状态。有 5 个可取值：0 = 未初始化，1 = 正在加载，2 = 已加载，3 = 交互中，4 = 完成
responseText：服务器的响应，表示为一个串
responseXML：服务器的响应，表示为 XML。这个对象可以解析为一个 DOM 对象
statusText：HTTP 状态码的相应文本（OK 或 Not Found（未找到）等等）

5 kettasAjax 封装 XMLHttpRequest 四个方法说明:
1.kettasAjax.doGetText(url, textHandler, async):
用 GET 方法发出请求到 url, 返回的文本 responseText 作为回调函数 textHandler 的参数。
textHandler 的签名类似 function(txt).
2.kettasAjax.doGetXml(url, xmlHandler, async):
用 GET 方法发出请求到 url, 返回的 XML Document 也就是 responseXML 作为回调函数

xmlHandler 的参数。xmlHandler 的签名类似 function(doc).
3.kettasAjax.doPostText(url, textHandler, body, async):
用 POST 方法将请求体 body 发送到 url, 返回的文本 responseText 作为回调函数 textHandler
的参数。textHandler 的签名类似 function(txt).
4.kettasAjax.doPostXml(url, xmlHandler, body, async):
用 POST 方法将请求体 body 发送到 url, 返回的 XML Document 也就是 responseXML 作为
回调函数 xmlHandler 的参数。xmlHandler 的签名类似 function(doc).

6 AJAX 全称为"Asynchronous JavaScript and XML"(异步 JavaScript 和 XML)，是指一种创建交互式网页应用的网页开发技术。它容许网页的部分刷新，比起传统的整个页面刷新，传输的数据量大大减少，这样就显著提高了用户浏览 web 页面的速度。

1) 获取 ajax 核心对象 XMLHttpRequest 的标准 JavaScript 函数 :

```
function createXhr(){
    // 判断是 IE 浏览器还是其他浏览器，然后返回对象
    if(window.ActiveXObject){
        return new ActiveXObject("Microsoft.XMLHTTP");
    }else if(window.XMLHttpRequest){
        return new XMLHttpRequest();
    }else{ throw new Error("Does not ajax programming");}
}
```

注 : 下面的笔记中，都用 createXhr()这个函数获取 XMLHttpRequest 对象

2) 实例 : 在网页中验证电子邮箱输入的格式

(1) 网页文件

```
<title>Validate Email Format</title>
<!-- createXhr()以及其他工具函数都在 utils.js 中 -->
<script type="text/javascript" src="js/utils.js"></script>

<script type="text/javascript">
    //以下是标准的步骤，要牢记
    var xhr = null;
    function validateEmail(){
        xhr = createXhr();// 获取 XMLHttpRequest 对象
        // 要提交到服务器的请求，escape 函数用于除去字符串两端空格，$函数根据 id 获得对象(此方法在
            util.js 中)  vm 是 servlet 的访问地址
        var url = "vm?mail=" + escape($("mail").value);
        /* 指定发送的方式、接受请求的页面(或 servlet)、是否采用异步回调
         * 参数也可以是("GET", url, true)
         * true 是默认的，故可省略不写，表示采用异步回调
         */
        xhr.open("GET", url);
        // 指定异步回调函数
        xhr.onreadystatechange = displayStatus;

        // 若采用 POST 方式提交请求，这里就必须发送请求体，这里是 GET 方式，故为 NULL
        //请求体的设置 xhr.setRequestHeader("Content-Type","application/x-www-form-urlencoded");
        xhr.send(null);
    }
    //异步回调函数的实现
    function displayStatus(){
        // readyState 是服务端状态， 0 代表无状态，1 代表建立连接，2 准备发送数据，3 正在发送数据 ，
            等于 4 表明对请求响应完毕，该返回客户端的数据都返回了
        if(xhr.readyState == 4){
            // status 是客户端状态，等于 200 表明数据已经存在于缓存，可以直接拿来使用了
                400 url 出错，500 内部出错
            if(xhr.status == 200){
                var status = $("mailFormatStatus");
                // 获得服务器返回的文本数据
                var mailStatus = xhr.responseText;
                if(mailStatus == "true"){
                    status.innerHTML = "valid email format";
                }else{ status.innerHTML = "invalid email format"; }
            }else{// 获得客户端状态值以及对应的描述性文本
                errorHandler(xhr.status, xhr.statusText);}
        }else{ status.innerHTML = "please wait..."; }}
</script>
```

```
</head>
<body>
    <h2>Please enter an email address to validate</h2>
    <input type="text" id="mail" onblur="validateEmail()"/><br/>
    <div id="mailFormatStatus"></div>
</body>
</html>
```

(2) Servlet 文件 ValidateEmailFormatServlet.java 片段

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    // 注意这里不是"text/html" 因为我们只需要返回纯文本字符串即可 故用"text/plain"
    response.setContentType("text/plain;charset=gbk");
    String mail = request.getParameter("mail");
    String status = "false";
    if(mail != null){
        Pattern p = Pattern.compile("\\S+@\\S+");
        Matcher m = p.matcher(mail);
        if(m.matches()) status = "true";
    }
    PrintWriter out = response.getWriter();
    out.print(status);
    out.flush();
    //这里关闭与否 应该无关痛痒
    out.close();
}
```

(3) web.xml 片段

```
<servlet>
    <description>
    </description>
    <display-name>
    ValidateEmailFormatServlet</display-name>
    <servlet-name>ValidateEmailFormatServlet</servlet-name>
    <servlet-class>
    com.kettas.ajax.xhr.day1.ValidateEmailFormatServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ValidateEmailFormatServlet</servlet-name>
    <url-pattern>/vm</url-pattern>
</servlet-mapping>
```

3) 向服务器请求获得 xml 文件(对象)

本例中，得到从服务器返回的 xml 对象，然后将数据填充到表格

```
<title>Online super store</title>
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript">
    var xhr = null;
    function initStore(){
        xhr = createXhr();
        xhr.open("GET", "xmlListProduct.jsp");
        xhr.onreadystatechange = display;
        xhr.send(null);
    }

    function display(){
        if(xhr.readyState == 4){
            if(xhr.status == 200){
                fillStoreTable(xhr.responseXML);
            }else{
                errorHandler(xhr.status, xhr.statusText);
            }}}
    function fillStoreTable(productsRoot){
        // 将 xml 数据，填充表格，具体过程略
        ...
    }
</script>
</head>
<body onload="initStore();">
<table id="storeTable">
```

```
        <tbody id="storeBody">
        </tbody>
    </table>
    </html>

    xmlListProduct.jsp 文件内容如下 :
        <!-- 注意，客户端需要的是 xml 文件，故这里是"text/xml" -->
        <%@page language="java" contentType="text/xml; charset=ISO-8859-1"
            pageEncoding="ISO-8859-1"%>
        <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
        <jsp:useBean id="store" class="com.kettas.ajax.xhr.day1.SuperStore" scope="application"/>
        <products>
            <c:forEach var="product" items="${store.products}">
              <product>
                    <id>${product.id }</id>
                    <name>${product.name }</name>
                    <price>${product.price}</price>
              </product>
            </c:forEach>
        </products>
```

7 封装 ajax 的调用步骤 :

获得 XMLHttpRequest 对象，然后向服务器发送请求，最后对服务器返回的数据进行处理。这个过程千篇一律，每次按照相同的步骤书写，故将它们封装起来是当务之急。这样的话，调用起来会方便得多。

封装的具体实现如下所示(十分经典的 javascript 代码，掌握并牢记) :

```
        function Request(l){
            var xhr = createXhr();// 私有属性 XMLHttpRequest 对象
            var listener = l; // 私有属性 此对象负责对服务端响应作反应
            // 保险起见 若初始化时没有赋予对象 则赋予一个新创建的对象
            if(!listener) listener = new ResponseListener();
            // 公有属性 请求体内容的类型 默认的表示发送的是表单数据
            // 若发送的是 xml 类型数据 则 this.contentType = "text/xml";
            // 实际上 这个是无关痛痒的 都用默认的类型即可
            this.contentType = "application/x-www-form-urlencoded";
            // 公有方法 以 GET 方式发送请求
            this.doGet = function(url,async){
                sendRequest("GET", url, null, async);
            }
            // 公有方法 以 POST 方式发送请求
            this.doPost = function(url, requestBody, async){
                sendRequest("POST", url, requestBody, async);
            }
            // 私有方法 向服务器发送请求
            function sendRequest(method, url, requestBody, async){
                if(async == undefined) async = true; // 如果没有说明是否采用异步回调 则采用异步回调
                      // 如果 async 有值 并且 async != ( false || 0 || null ) 则采用异步回调
                    else if(async) async = true;
                     else async = false; // 其他情况( async = false || 0 || null ) 则不采用异步回调
                     // 指定发送的方式、接受请求的页面(或 servlet)、是否采用异步回调
                     xhr.open(method, url, async);
                     if(async) xhr.onreadystatechange = callback; // 指定异步回调函数
                    // 设置请求体的内容类 contentType =application/x-www-form-urlencoded"
                     xhr.setRequestHeader("Content-Type", contentType);
                     xhr.send(requestBody); // 发送请求
                     // 若没有采用异步回调 则执行此函数
                     if(!async) listener.complete(
                           xhr.status, xhr.statusText, xhr.responseText, xhr.responseXML
                     );
            }
            // 私有函数 异步回调函数的实现
            function callback(){
                switch(xhr.readyState){
                     case 0: listener.uninitialized(); break;
                     case 1: listener.loading(); break;
                     case 2: listener.loaded(); break;
                     case 3: listener.interactive(); break;
                     case 4: listener.complete(
                           xhr.status, xhr.statusText, xhr.responseText, xhr.responseXML
```

```
                ); break;
            }}
        // 私有函数
        function createXhr(){
            if(window.ActiveXObject){
                return new ActiveXObject("Microsoft.XMLHTTP");
            }else if(window.XMLHttpRequest){
                return new XMLHttpRequest();
            }else{
                throw new Error("Does not ajax programming");
            }}}
// 此对象处理服务端响应
function ResponseListener(){
        this.uninitialized = function(){}
        this.loading = function(){}
        this.loaded = function(){}
        this.interactive = function(){}
        // 最后一个响应状态 响应完毕
        this.complete = function(status, statusText, responseText, responseXML){}
/* 继承上面的对象
 * 对于其它四个状态 一般是不需要处理响应的 故这里不予理会
 * 只是覆盖响应完毕后的处理方法
 */
function ResponseAdapter(){
        this.handleText = function(text){}
        this.handleXml = function(xml){}
        this.handleError = function(status, statusText){
            alert("Error: " + status + " " + statusText);
        }
        // 覆盖父类的方法
        this.complete = function(status, statusText, responseText, responseXML){
            if(status == 200){
                this.handleText(responseText);
                this.handleXml(responseXML);
            }else{
                this.handleError(status, statusText);
            }
        }
}
// 注意 javascript 的继承机制
ResponseAdapter.prototype = new ResponseListener();
ResponseAdapter.prototype.constructor = ResponseAdapter;
// 对以上部分实现功能的组织调用类
if(!kettasAjax) var kettasAjax = {};
// 采用 GET 方式发送请求 对返回的文本数据进行处理
if(!kettasAjax.getText){
        kettasAjax.getText = function(url, handleText, async){
            var l = new ResponseAdapter();
            l.handleText = handleText;
            var req = new Request(l);
            req.doGet(url, async);
        }}
// 采用 GET 方式发送请求 对返回的 XML 数据进行处理
if(!kettasAjax.getXml){
        kettasAjax.getXml = function(url, handleXml, async){
            var l = new ResponseAdapter();
            l.handleXml = handleXml;
            var req = new Request(l);
            req.doGet(url, async);
        }
}
// 采用 POST 方式发送请求 对返回的文本数据进行处理
if(!kettasAjax.postText){
        kettasAjax.postText = function(url, requestBody,handleText, async){
            var l = new ResponseAdapter();
            l.handleText = handleText;
            var req = new Request(l);
            req.doPost(url,requestBody, async);
        }
```

```
}
// 采用 POST 方式发送请求 对返回的 XML 数据进行处理
if(!kettasAjax.postXml){
      kettasAjax.postXml = function(url, requestBody, handleXml, async){
            var l = new ResponseAdapter();
            l.handleText = handleXml;
            var req = new Request(l);
            req.doPost(url, requestBody, async);
      }
}
```

注 ： 以后就使用这个 kettasAjax 对象，会方便异常

8 采用 kettasAjax 对象，采用不同的方式向服务器发送不同类型的数据

(1) html 文件如下 ：

```
<title>Send Request Parameters</title>
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript" src="js/kettasAjax.js"></script>
<script type="text/javascript">
      function handleForm(httpMethod){
            if(httpMethod == "GET"){
                  // 采用 GET 方式向服务器发送表单数据
                  kettasAjax.getText("echo.jsp?" + getQueryString(), display);
            }else{
                  // 采用 POST 方式向服务器发送表单数据
                  kettasAjax.postText("echo.jsp", getQueryString(), display);
            }}
      // 显示服务器返回的数据
      function display(txt){
            $("response").innerHTML = txt;
      }
      // 获得表单数据的字符串组织形式
      function getQueryString(){
            var name = escape(getEavById("name"));
            var age = escape(getEavById("age"));
            var password = escape(getEavById("password"));
            var queryStr = "name=" + name +"&age=" + age + "&password=" + password;
            return queryStr;
      }
      // 采用 POST 方式向服务器发送 xml 数据
      // 由于发送的是 xml 形式字符串 故服务端不能以 getParameter 的方式读取 要以 readLine 方式读取
      function handleXmlForm(){
            kettasAjax.postText("handleXml", getXmlFromForm(), display);
      }
      // 获得表单数据的 xml 字符串表现形式
      function getXmlFromForm(){
            var name = getEavById("name");
            var age = getEavById("age");
            var password = getEavById("password");
            var xmlStr = "<params>"
            + "<name>" + name + "</name>"
            + "<age>" + age + "</age>"
            + "<password>" + password + "</password>"
            + "</params>"
            return xmlStr;
      }
</script>
</head>
<body>
      <form action="#">
       Name: <input type="text" id="name"/><br/>
       Age: <input type="text" id="age"/><br/>
       Password: <input type="password" id="password"/><br/>
      </form>
      <button onclick="handleForm('GET')">Get</button>
      <button onclick="handleForm('POST')">Post</button>
      <button onclick="handleXmlForm()">Send Parameters as XML</button><br/>
      <div id="response"></div>
</body>
</html>
```

(2) echo.jsp 内容如下 :

```
Http method: ${pageContext.request.method}, and parameters
are name: ${param.name}, age: ${param.age}, password: ${param.password}
```

(3) Servlet 文件 HandleXmlServlet 片段如下 :

```
protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
      response.setContentType("text/plain");
      StringBuilder sb = new StringBuilder();
      // 由于客户端发送的是 xml 形式的字符串 故要获得 BufferedReader 对象用于读取
      BufferedReader reader = request.getReader();
      PrintWriter out = response.getWriter();
      String line = null;
      // 可能客户端是采用多行发送的 故不能只读取一行了事
      while ((line = reader.readLine()) != null)
            sb.append(line);
      // 解析 xml 数据
      DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
      try {
            DocumentBuilder db = dbf.newDocumentBuilder();
            // 由于不存在.xml 文件 故只能以内存字节流的方式初始化
            Document doc = db.parse(
                  new ByteArrayInputStream(sb.toString().getBytes())
            );
            String name = getElementData(doc, "name");
            String age = getElementData(doc, "age");
            String password = getElementData(doc, "password");
            out.print("Parameters are name: " + name
             + " age: " + age
             + " password: " + password
            );
      } catch (Exception e) {
            e.printStackTrace();
            out.print("error");
      }
      out.close();
}
```

9 页面部分更新，只更新表格部分

(1) html 文件的实现 :

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Online super store</title>
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript" src="js/kettasAjax.js"></script>
<script type="text/javascript">
      function initStore(){
            kettasAjax.getText("listProduct.jsp", display);
      }
      function display(txt){
            $("store").innerHTML = txt;
      }
</script>
</head>
<body onload="initStore();">
      <div id="store"></div>
</body>
</html>
```

(2) listProduct.jsp 内容如下 :

```
<%@ page language="java" contentType="text/html; charset=GBK"
      pageEncoding="GBK"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<jsp:useBean id="store" class="com.kettas.ajax.xhr.day1.SuperStore" scope="application"/>
<table border="1">
  <tbody>
    <c:forEach var="product" items="${store.products}">
      <tr>
```

```
                <td>${product.id }</td>
                <td>${product.name }</td>
                <td>${product.price}</td>
            </tr>
          </c:forEach>
        </tbody>
      </table>
```

10 于 Ajax 中的 Json

Json 是一种线上协议，轻量级的 xml 一种文体格式

Json 的功能，简单的说，就是实现字符串和对象之间的转换。要使用其功能，在客户端，要引入 json.js 文件，在服务器端，则要引入 json.jar 这个包。

Json 对象和数组 {}大括号代表一个对象,,【】代表数组

(1) Json 在客户端的应用实例 ：

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>JSON test</title>
<script type="text/javascript" src="js/json.js"></script>
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript">
    // 创建一个对象，注意格式 {}
    var product = {
        id: 1,  name: "core java",  price: 100      };
    // 创建一个对象字符串，注意格式
    var s = '{"id": 1, "name": "George", "age":30}';
    // 实现对象和字符串之间的互换 只须 stringify 和 parse 这两个方法即可
    function display(){
        // 将对象转换为字符串，发送给服务器处理
        byId("json").innerHTML = JSON.stringify(product);
        // 将字符串转换为对象，把服务端的字符串组装成对象
        var student = JSON.parse(s);
        alert(student.id + "\t" + student.name + "\t" + student.age); }
</script>
</head>
<body onload="display()"><h2>
    <div id="json"></div>
    </h2>
</body>
</html>
```

(2) Json 在客户端和服务器端同时应用 ：

1 客户端 html 文件 ：

```
<html><head>
<title>Query Price</title>
<script type="text/javascript" src="js/json.js"></script>
<script type="text/javascript" src="js/kettasAjax.js"></script>
<script type="text/javascript" src="js/utils.js"></script>
<script type="text/javascript">
    function query(){
        // obj4form 是一个工具函数 根据表单 id 将用户输入表单的数据封装为一个对象
        var car = obj4form("carInfo");
        // 直接向服务器发送对象字符串
//"queryPrice"是访问 servlet 地址，JSON.stringify(car)发送对象字符串，display 为服务器返回内容
        kettasAjax.postText("queryPrice", JSON.stringify(car), display);
    }
    function display(txt){
        // 将从服务器返回的对象字符串转换为对象
        var ret = JSON.parse(txt);
        var model = ret.model;
        var price = ret.price;
        $("result").innerHTML = model + " " + price
    }
</script>
</head>
<body>
    <h2>Please enter car information</h2>
    <!-- 利用输入的汽车 id 和生产厂商信息 得以查询汽车的价格 -->
    <form action=# id="carInfo">
        Id: <input type="text" id="id"/><br/>
```

```
                Make: <input type="text" id="make"/><br/>
            </form>
            <button onclick="query()">Query Price</button>
            <div id="result"></div>
        </body>
        </html>
```

2 Servlet 文件 QueryPriceServlet 的片段 :

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/plain");
        BufferedReader reader = request.getReader();//得到客户端的字符串
        PrintWriter out = response.getWriter();
        StringBuilder sb = new StringBuilder();
        String str = null;
        while((str = reader.readLine()) != null)
            sb.append(str);
        try {// 将客户端发送的对象字符串转化为 JSONObject 对象
            JSONObject jsonObj = new JSONObject(sb.toString());
            // 获得对象中的数据 注意格式
            int id = jsonObj.getInt("id");
            String make = jsonObj.getString("make");
            // 不用 make.equals("Audi") 避免了 make == null 的情况下抛异常 小技巧
            if(id == 1 && "Audi".equals(make)){
                // 给客户端回应一个对象字符串 注意格式// 首先创建一个新的 JSONObject 对象
                jsonObj = new JSONObject();
                // 向这个对象中填充数据
                jsonObj.put("model", "A8");
                jsonObj.put("price", 860000);
                // 将对象转化为对象字符串 回发给客户端
                // 这个对象字符串的格式为 : '{"model" : "A8" , "price" : 860000}'
                out.print(jsonObj.toString());
            }
        } catch (Exception e) {
            e.printStackTrace();  throw new ServletException(e);
        }finally{  out.close();
        }}
```

(3) 在服务器端将 java 中的实体 bean、collections、array 以及 map 对象转换为对象字符串 :

要实现这个功能，可以借助内部工具类 **JsonUtil.java**，其中的几个方法是很有用的，如 :

```
public static JSONObject bean2Json(Object bean),
public static JSONObject map2Json(Map map),
public static JSONArray array2Jarr(Object value),
public static JSONArray collection2Json(Object value)
```

测试类片段如下 :

```
public class TestJsonUtil {
    public static void main(String[] args) {
        SuperStore ss = new SuperStore();
        // ss.getProducts()方法返回一个商品对象的集合
        JSONArray jarr = JsonUtil.collection2Json(ss.getProducts());
        System.out.println(jarr.toString());
    }
}
```

11 关于 Ajax 中的 dwr 框架 :

有了 dwr 框架，操作 html 页面感觉就是在直接操作 java 代码。实际上，还是通过服务器端执行的。

要使用此框架提供的功能，必须引入 **dwr.jar** 这个包。

思想：在 web 上的 servlet 动态生成 javascript 代码

(1) 最简单的实例

java 类文件 :

```
package com.kettas.ajax.dwr.day4;
public class Hello {
    public String sayHello(String user){
        try{
            Thread.sleep(2000);
        }catch(Exception e){
        }
        return "Hello, " + user;
    }
}
```

配置文件 dwr.xml(与 web.xml 处于同一个位置) :

```
<dwr>
    <allow>
        <!-- javascript="hello"指定客户端对象名称为 hello
             creator="new"指定 java 对象的创建方式
             scope="application"指定 java 对象的放置位置 -->
        <create javascript="hello" creator="new" scope="application">
            <param name="class" value="com.kettas.ajax.dwr.day4.Hello"></param>
        </create>
    </allow>
</dwr>
```

客户端 html 文件的内容 :

```
<!-- 服务器动态生成的 js 文件 内有 hello 对象 -->
<script type="text/javascript" src="dwr/interface/hello.js"></script>
<!-- dwr 引擎包 -->
<script type="text/javascript" src="dwr/engine.js"></script>
<!-- 引入 dwr 提供的工具包 -->
<script type="text/javascript" src="dwr/util.js"></script>
<script type="text/javascript">
    // element.style.visibility = "hidden";
    // element.style.visibility = "";
    // CSS 内容 对象的隐藏属性
    /*
    // 这样写的话 仅此函数就可以了
    function sayHello(){
        // 直接给参数赋内部函数
        hello.sayHello($("user").value, function(txt){
            dwr.util.setValue("helloStr", txt);
        });
    }*/
    /*
    // 也可以这么写 最简便
    function sayHello(){
     // hello 是在 dwr.xml 配置的, hello.sayHello 相当调用服务端的方法, ($("user").value 参数,
     display 回调方法, 返回值就是 display (text) 中的 text 值
    hello.sayHello($("user").value, display);
    }
    */
    // 这么写 是最完整的
    function sayHello(){
        //dwr.util.useLoadingMessage("Wait a minute...");
        // 给参数赋一个对象
        var cb = {
            callback: display, // 指定回调函数
            timeout: 500, // 指定等待时间 如果超出这个时间 则出错
            errorHandler: display, // 指定处理错误的回调函数
            // ???
            name: "George"
        };
        // 在客户端调用方法 始终比服务端对应方法要多一个参数
        // 这个参数最终的实现是一个回调函数
        // 这个回调函数的参数就是服务端方法返回的对象(在客户端是 javascript 对象)
        hello.sayHello($("user").value, cb);
    }
    function display(txt){
        // 调用工具包中的工具函数 为容器添加文本
        dwr.util.setValue("helloStr", txt);
    }
</script>
</head>
<body>
    <h2 id="h2">Please enter a user who you want to say hello to:</h2>
    User: <input type="text" id="user"/><button onclick="sayHello()">Say Hello</button><br/>
    <div id="helloStr"></div>
</body>
</html>
```

在 web.xml 文件中不妨加上如下内容作为测试用 :

```
    <servlet>
        <servlet-name>dwr</servlet-name>
```

```
            <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
            <init-param>
                <param-name>debug</param-name>
                <param-value>true</param-value>
            </init-param>
        </servlet>
        <servlet-mapping>
            <servlet-name>dwr</servlet-name>
            <url-pattern>/dwr/*</url-pattern>
        </servlet-mapping>
```

(2) 实例 : 人员管理

Person 实体类 : Integer id; String name; String address; float salary;

人员管理类 PersonMgmt :

```
package com.kettas.ajax.dwr.day5;
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
public class PersonMgmt {
    private Map<Integer, Person> people = new HashMap<Integer, Person>();
    private static int count = 10;
    public PersonMgmt() {
        Person fred = new Person("Fred", "1 Red Street", 2, 100000.0f);
        Person jim = new Person("Jim", "42 Brown Lane", 3, 20000.0f);
        Person shiela = new Person("Shiela", "12 Yellow Road", 4, 3000000.0f);
        people.put(new Integer(fred.getId()), fred);
        people.put(new Integer(jim.getId()), jim);
        people.put(new Integer(shiela.getId()), shiela);
    }
    public Collection<Person> getPeople() {
        return people.values();
    }
    public Person getPerson(Integer id) {
        return people.get(id);
    }
    public void add(Person p) {
    //如果 p 中没有值时执行
        if (p.getId() == -1)
            p.setId(count++);
            people.put(p.getId(), p);
        }
    public void remove(Person p) {
        people.remove(p.getId());
    }
}
```

配置文件 dwr.xml :

```
<dwr>
    <allow>
        <create javascript="personMgmt" creator="new" scope="session">
            <param name="class" value="com.kettas.ajax.dwr.day5.PersonMgmt"></param>
        </create>
        <!-- 由于客户端传入的是 javascript 的 Person 对象
              故这里要声明对应的服务器端 java 的 Person 对象
              否则服务端将无法将对象从客户端类型转换为服务端类型
            <convert》转换器, 对象到字符串之间的转换 -->
        <convert match="com.kettas.ajax.dwr.day5.Person" converter="bean"></convert>
    </allow>
</dwr>
```

客户端 html 文件内容 :

```
<title>Person Management</title>
<script type="text/javascript" src="dwr/engine.js"></script>
<script type="text/javascript" src="dwr/util.js"></script>
<script type="text/javascript" src="dwr/interface/personMgmt.js"></script>
<script type="text/javascript">
    // 初始化显示人员信息
    function init(){
        personMgmt.getPeople(fillTable);
    }
    // 函数数组
    var peoplebodyCellFuncs = [
```

```
    // 函数参数为人员对象 返回值即我们要在单元格中显示的内容
    // 此数组的长度即表主体显示内容的列数
    function(person) { return person.name;},
    function(person) { return person.address;},
    function(person) { return person.salary;},
    // 在第四列添加两个操作按钮
    function(person) {
        var div = document.createElement("div");
        div.appendChild(createBtn("personId", person.id, prepareEdit, "Edit"));
        div.appendChild(createBtn("personId", person.id, removePerson, "Remove"));
        return div;
    }
];
// 编辑人员信息
function prepareEdit(){
    var personId = this.personId; // 获得此人 id 注意这里 this 的使用
    personMgmt.getPerson(personId, fillFields) // 调用服务端方法 获得此人对象
}
// 填充人员信息编辑表
function fillFields(person){
    // person 对象如下 : {id : 1, name : "yinkui", address : "xf", salary : 50000}
    // 相当于执行 : document.getElementById("id").value = 1
    //              document.getElementById("name").value = "yinkui" ...
    dwr.util.setValues(person);
}
// 删除一个人
function removePerson(){
    var personId = this.personId; // 获得此人 id 注意这里 this 的使用
    // 由于对应服务端 java 方法中 删除一个人只需要此人 id
    // 故 创建 person 对象时只需要赋予 id
    // 这样传到服务端 创建的 java 的 Person 对象顶多是其他属性为 null
    var person = {
        id: personId
    };
    personMgmt.remove(person, init); // 调用服务端方法删除此人 然后刷新人员信息表的显示
}
// 创建按钮的函数
function createBtn(attrName, attrValue, onclick, label){
    var btn = document.createElement("button");
    btn[attrName] = attrValue; // 按钮对象中存放对应人员 id 值
    btn.onclick = onclick; // 为按钮添加事件
    btn.appendChild(document.createTextNode(label)); // 为按钮添加显示内容
    return btn;
}
// 填充人员信息表
// 其中 people 对象是从服务端返回的人员对象数组
// 其格式为[{id : 1, name : "yinkui", address : "xf", salary : 50000}, {...}, ...]
function fillTable(people){
    dwr.util.removeAllRows("peoplebody");// 调用工具函数 清空表主体中的所有行
    // 调用工具函数 通过人员对象数组 为表主体添加行
    // 其中 peoplebodyCellFuncs 是一个函数数组
    dwr.util.addRows("peoplebody", people, peoplebodyCellFuncs);
    clearPerson();// 初始化人员信息编辑表的内容
}
// 添加新人员
function writePerson(){
    var person = {
        id: -1,
        name: null,
        address: null,
        salary: null
    };
    // 注意此工具函数的使用
    // 其作用与 dwr.util.setValues({...})恰恰相反
    // 相当于执行 : person.id = document.getElementById("id").value
    //    person.name = document.getElementById("name").value
    dwr.util.getValues(person);
    personMgmt.add(person, init); // 调用服务端方法 然后刷新人员信息表的显示
}
```

```
                    // 初始化显示人员信息编辑表
                    function clearPerson(){
                        // 注意此工具函数的使用 参数为对象
                        // 相当于执行 : document.getElementById("id").value = -1
                        //              document.getElementById("name").value = null ...
                        dwr.util.setValues(
                            {     id: -1,          name: null,
                                 salary: null,          address: null
                            }
                        );
}
            </script>
            </head>
            <body onload="init()">
            <h1>Dynamically Editing a Table</h1>
            <h2>People Management</h2>
            <!-- 人员信息列表 -->
            <table border="1">
                <!-- 表头 -->
                <thead>
                    <tr> th>Name</th>
                        <th>Address</th>
                        <th>Salary</th>
                        <th colspan="2">Actions</th></tr>
                </thead>
                <!-- 表的主体 -->
                <tbody id="peoplebody"></tbody>
            </table>
            <h4>Edit Person</h4>
            <!-- 对人员信息进行编辑 或是添加新人员 -->
            <table>
                <tr><td>ID:</td>
                    <td><span id="id">-1</span></td></tr>
                <tr><td>Name:</td>
                    <td><input id="name" type="text" /></td></tr>
                <tr><td>Salary:</td>
                    <td><input id="salary" type="text" /></td></tr>
                <tr><td>Address:</td>
                    <td><input type="text" id="address" /></td></tr>
                <tr><td colspan="2" align="right">
                    <input type="button" value="Save" onclick="writePerson()" />
                    <input type="button" value="Clear" onclick="clearPerson()" /></td></tr>
            </table></body></html>
```

12，关于 DOJO 的框架：

含义：Dojo 是一个非常强大的面向对象的 JavaScript 的工具箱, 建议读者能够去补充一下 JavaScript 下如何使用 OO 进行编程的, 这对于你以后阅读 Dojo Source 有很大的用处

作用：①处理浏览器的不兼容问题，可以方便建立互动的互动用户见面。②更容易统一页面风格。③封装与服务端的通信，动态从服务器下载 javascript

开发流程：①把 dojo 工具包考到 js 文件下

②引用 dojo 的启动代码

```
<script type="text/javascript" src="/yourpath/dojo.js" />
```

③声明你所要用到的包

```
<script type="text/javascript">
  dojo.require("dojo.math");引入数学方法
  dojo.require("dojo.io.*"); 引入 io
  dojo.require("dojo.event.*");  引入事件
  dojo.require("dojo.widget.*");引入页面
   function init(){  初始化
        var btn = dojo.widget.byId("helloBtn");
        dojo.event.connect(btn, "onClick", "sayHello");事件源，产生关联，时间方法
   }
   function sayHello(){
        dojo.io.bind(
          { url: "sayHello.jsp",处理事件的 url
            handler: callback,  回调函数
            formNode: "myForm" 表单内容
         });}
   function callback(type, data, evt){
        document.getElementById("result").innerHTML
```

```
            = data;
      }

      dojo.addOnLoad(init);加入事件
    </script>
    </head>
    <body>
      <h2>DOJO Test</h2>
      <form action="#" id="myForm">
           User: <input type="text" name="user" />
      </form>
      <button dojoType="Button" widgetId="helloBtn">DOJO Say Hello</button>
      <div id="result"></div>
    </body>
    </html>
```

13，关于 Ext 框架

参考项目文档

14 补充：Dom 部分 创建标签 var td=document.createElement(“td”); td.apperdchild(childName)

javascript 的高级部分

1，javascript 的对象和继承 由于 javascript 是采取 **prototype 机制的“伪继承”**，prototype 必须显示引用“基类”对象，所以注定了 javascript 只能实现“弱继承”，或者叫做“对象继承”

注意这里的次序关系需要很明确，prototype 的赋值必须在对象构造之外。

```
 function classA()
{  classA.prototype.name=”xxx”;属性
  classA.prototype.methodA = function() {...}这是一个对象方法
  classA.methodA = function(){...} //这是一个类方法相当于 java 的 static 方法
 }
 function classB()
{  classB.prototype.methodA = function(){...}
 }
classB.prototype = new classA();

//注意这里声明 B 继承自 A 出现在 function classB()函数体之外，并且先于
//classB 对象的构造被执行。
```

私有成员的设置

```
 Var name=“zhang”;
 this.getName=function(){return name;}
 this.setName=function(myname){name=myname; }
```

Spring

一：Ioc

1 含义：为解决企业应用开发的复杂性而创建的开源框架，用基本的 javaBean 来完成 EJB 的事情 从大小和开销方向 spring 都是轻量级的

2，用途

① Ioc 容器可以将对象之间的依赖关系交由 spring 管理，进行控制
② AOP:方便进行面向切面的编程，是 oop 的扩展，想加什么功能直接加
③ 能够集成各种优秀的框架，struts hibernate 等

3，spring 组成内容

4 准备配置工作

①下载 SpringFramework 的最新版本，并解压缩到指定目录。
在 IDE 中新建一个项目，并将 Spring.jar 将其相关类库加入项目
② 配置文件 bean.xml
③在 classpath 创建日志输出文件。log4j.properties
④org.springframework.beans 及 org.springframework.context 包是 Spring IoC 容器的基础

5 Spring 基础语义

1）IoC （Inversion of Control）=**DI** （Dependency Injection）控制反转和依赖注入
它是一种基于接口的编程，bean 由容器创建在需要的时候拿来用即可，主要是采用反射来实现，其核心组建就是 BeanFactory 但实际开发常用 XmlBeanFactory

2）依赖注入的几种实现类型

Type1 设值注入：通过类的 setter 方法完成依赖关系的设置，就是给 bean 类中属性加 set 方法
Type3 构造子注入：即通过构造函数完成依赖关系的设

```
public class DIByConstructor {
private final DataSource dataSource;
private final String message;
public DIByConstructor(DataSource ds, String msg) {
this.dataSource = ds;
this.message = msg;
}}
```

3)几种依赖注入模式的对比总结

Type2 设值注入的优势

1. 对于习惯了传统JavaBean开发的程序员而言，通过setter方法设定依赖关系显得更加直观，更加自然。
2. 如果依赖关系（或继承关系）较为复杂，那么Type3模式的构造函数也会相当庞大（我们需要在构造函数中设定所有依赖关系），此时Type2模式往往更为简洁。
3. 对于某些第三方类库而言，可能要求我们的组件必须提供一个默认的构造函数（如Struts中的Action），此时Type3类型的依赖注入机制就体现出其局限性，难以完成我们期望的功能。

Type3 构造子注入的优势：

1. “在构造期即创建一个完整、合法的对象”，对于这条Java设计原则，Type3无疑是最好的响应者。
2. 避免了繁琐的setter方法的编写，所有依赖关系均在构造函数中设定，依赖关系集中呈现，更加易读。
3. 由于没有setter方法，依赖关系在构造时由容器一次性设定，因此组件在被创建之后即处于相对“不变”的稳定状态，无需担心上层代码在调用过程中执行setter方法对组件依赖关系产生破坏，特别是对于Singleton模式的组件而言，这可能对整个系统产生重大的影响。
4. 同样，由于关联关系仅在构造函数中表达，只有组件创建者需要关心组件内部的依赖关系。对调用者而言，组件中的依赖关系处于黑盒之中。对上层屏蔽不必要的信息，也为系统的层次清晰性提供了保证。
5. 通过构造子注入，意味着我们可以在构造函数中决定依赖关系的注入顺序，对于一个大量依赖外部服务的组件而言，依赖关系的获得顺序可能非常重要，比如某个依赖关系注入的先决条件是组件的DataSource及相关资源已经被设定。

理论上，以Type3类型为主，辅之以Type2类型机制作为补充，可以达到最好的依赖注入效果，不过对于基于Spring Framework开发的应用而言，Type2使用更加广泛。

5）bean.xml配置文件

Bean Factory，顾名思义，负责创建并维护Bean实例。

Bean Factory负责根据配置文件创建Bean实例，可以配置的项目有：

1. Bean属性值及依赖关系（对其他Bean的引用）
2. Bean创建模式（是否Singleton模式，即是否只针对指定类维持全局唯一的实例）
3. Bean初始化和销毁方法
4. Bean的依赖关系

6)XmlBeanFactory两中注入方式的配置

①property-------→set方法的注入配置

```
<p:bean id=" hello"  class=" com.kettas.HelloIFImp" >
   <p:property name=" user"  value=" xxx" ></p:property>
</p:bean>
```

②constructor---------→构造方法的注入配置

```
<p:bean id="hello2" class="com.kettas.spring.ioc.day1.HelloIFImpl">
   <p:constructor-arg index=" 0"  value="world"></p:constructor-arg>
   <p:constructor-arg type="java.lang.String"" ref="calendar"></p:constructor-arg>
</p:bean>
```

说明： index=" 0"构造方法第一个参数，用index可以稍微减少冗余，但是它更容易出错且不如type属性可读性高。你应该仅在构造函数中有参数冲突时使用index。

7) 依赖的目标类型分成三种形式

1) 基本类型+String

```
   <value>data</value>类型自动转化
```

2) 对其他bean 的引用

```
      <ref bean="target"/>
```

3) 集合类型　　list props set map

list set properties配置类似

```
  <p:property name="intList">
     <p:list>
                  <p:value>1</p:value>
                  <p:value>2</p:value>
      </p:list>
  </p:property>
   <p:property name="objMap">
        <p:map>
           <p:entry>
              <p:key>
                   <p:value>1</p:value>
               </p:key>
              <p:ref local="hello2"/>
           </p:entry>
        </p:map>
  </p:property>
  <p:property name="pros">
        <p:props>
             <p:prop key="1">red</p:prop>
             <p:prop key="2">green</p:prop>
        </p:props>
  </p:property>
```

Xml配置文件属性的说明

```
<bean id="TheAction" (1) class="net.xiaxin.spring.qs.UpperAction" (2) singleton="true" (3)
       init-method="init" (4)destroy-method="cleanup" (5)depends-on="ActionManager" (6) >
       <property…>
</bean>
```

(1) id

Java Bean在BeanFactory中的唯一标识，代码中通过BeanFactory获取JavaBean实例时需以此作为索引名称。

(2) class Java Bean 类名 即真正接口的实现类

(3) singleton bean的作用域（创建模式（prototype还是singleton））

单例（**Singleton**）模式，如果设为“true”，只维护此Java Bean的一个实例，反之，如果设为“false”， BeanFactory每次都将创建一个新的实例返回。默认为true

实现方式是第一次getBean时放入Map中保存，第二次再用时直接在Map中拿，类名为key，实例为value。Bean的其他作用域还有**prototype**：原型模式：在获取prototype定义的bean时都产生新的实例，其生命周期由客户端维护。**Session** 对每次HTTPsession中都回产生一个新的实例。**Global session** 仅在使用portletcontext的时候才有效，**常用的是singleton和prototype**

(4) init-method

初始化方法，此方法将在BeanFactory创建JavaBean实例之后，在向应用层返回引用之前执行。一般用于一些资源的初始化工作。在javaBean中创建init方法，再添加属性init-method=“init”就行

(5) destroy-method

销毁方法。此方法将在BeanFactory销毁的时候执行，一般用于资源释放。与init用法类似

(6) depends-on

Bean依赖关系。一般情况下无需设定。Spring会根据情况组织各个依赖关系的构建工作（这里示例中的depends-on属性非必须）。

只有某些特殊情况下，如JavaBean中的某些静态变量需要进行初始化（这是一种Bad Smell，应该在设计上应该避免）。通过depends-on指定其依赖关系可保证在此Bean加载之前，首先对depends-on所指定的资源进行加载。

(7) <value>

通过<value/>节点可指定属性值。BeanFactory将自动根据Java Bean对应的属性类型加以匹配。

下面的”desc”属性提供了一个null值的设定示例。注意<value></value>代表一个空字符串，如果需要将属性值设定为null，必须使用<null/>节点。

(8) <ref>指定了属性对BeanFactory中其他Bean的引用关系。

8）使用抽象bean　定义抽象类Abstract=“true”抽象bean不能实例化，一个类可以创建多个bean

抽象bean的配置和一般bean的配置基本一样只是在增加了Abstract=“true”抽象bean是一个bean的模板，容器会忽略抽象bean的定义，不会实例化抽象bean，故不能通过getBean（）显示的获得抽象bean的实例也不

能将抽象bean注入其他bean的依赖属性。

抽象bean的配置和继承

通过Abstract属性配置抽象bean

```
<bean id=" fatherTemple" class=" abstractClass" abstract=" true" >
 <!—注入属性→
 <property name=" name" ref=" xxx" />
 </bean>
 <!—通过parent属性定义子bean→
 <bean id=" childTemple" parent=" fatherTemple" >
  <property name=" name2" ref=" yyyy" /> -定义自己的属性
 </bean>
```

说明：子bean配置可以增加新的配置信息，并可以定义新的配置覆盖父类的定义

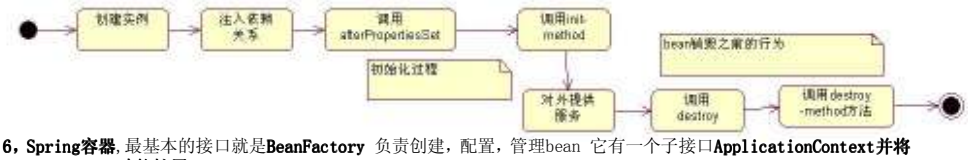
子类和父类中至少有一个class属性否则不知道实现类，父类的class可以不写

9）bean在容器上的生命周期

初始化两种方法 1使用init-method属性指定那个方法在bean依赖关系设置好后自动执行

2实现initializingBean接口 实现该接口必须实现void afterPropertiesSet（）throws Exception那么就不用设置init-method方法了，注意：最好使用init-method方法，减少代码的侵入性，如果两种方法都实现则先实现接口再init方法（一般写入日志文件）

销毁两种方法和初始化一样也有两种方法：1，destroy-method和1实现DisposableBean接口



6，Spring容器,最基本的接口就是**BeanFactory** 负责创建，配置，管理bean 它有一个子接口**ApplicationContext并将功能扩展。**

理论上bean装配可以从任何资源获得，包括属性文件，关系数据库等，但xml是最常用的XmlBeanFactory ，
ClassPathXmlApplicationContext ， FileSystemXmlApplicationContext ，
XmlWebApplicationContext应用系统配置源。Spring中的几种容器都支持使用xml装配bean，包括：
XmlBeanFactory ， ClassPathXmlApplicationContext ，FileSystemXmlApplicationContext ，
XmlWebApplicationContext

BeanFactory接口包含如下的基本方法：

Boolean containsBean(String name)：判断Spring容器是否包含id为name的bean定义。
Object getBean(String name)：返回容器id为name的bean.
Object getBean(String name, Class requiredType)：返回容器中id为name,并且类型为requiredType的bean.
Class getType(String name)：返回容器中id为name的bean的类型.

ApplicationContext有三个实现类实现资源访问

FileSystemXmlApplicationContext：基于文件系统的xml配置文件创建**ApplicationContext，**
ClassPathXmlApplicationContext ：以类加载路径下的xml的配置文件创建**ApplicationContext（更为常用）**
XmlWebApplicationContext：对使用servletContextResource进行资源访问
获得容器的应用的方式

```
①InputStream is = new FileInputStream("bean.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
或者BeanFactory factory = new XmlBeanFactory(
                              new ClassPathResource("classpath：bean.xml"))
 Action action = (Action) factory.getBean("TheAction");
②ApplicationContext bf=new ClassPathXmlApplicationContext("classpath：bean.xml")
Action action = (Action) factory.getBean("TheAction");
③ApplicationContext bf=new FileSystemXmlApplicationContext (“classpath：bean.xml”)
```

第一种**BeanFactory**启动快（启动是不创建实体，到用时才创建实体），
第二种**ApplicationContext**运行快（在加载时就创建好实体）更为常用，继承BeanFactory

5)工厂bean和bean工厂

FactoryBean（工厂bean）：是bean的加工工厂，是对已知Bean的加工，是一个接口，要实现三个方法：

① Object getObject()可以对bean进行加工添加功能
②Class getObjectType()③Boolean isSingleton()
Bf.getBean(“ab”)只是得到MyFactory.getObject()的object对象 所以最后要强转。

Beanfactory bean工厂 就是生产bean的工厂，注入：

由于Spring IoC容器以框架的方式提供了工厂方法的功能，并以透明的方式给开发者，不过在一些遗留系统或第三方类库中，我们还会碰到工厂方法，这时用户可以使用Sping使用工厂方法注入的方式进行配置。

静态工厂方法：

很多工厂类方法都是静态的，这意味着用户在无须创建工厂类实例的情况就可以调用工厂类方法。因此静态工厂方法比非静态工厂方法的调用
更加方便。我们将carFactory类的getCar()方法调**整为静态**的然后再Spring配置如下：

```
<bean id=" car" class =" carFactory" factory-method=" getCar" />
```

用户直接通过class属性指定工厂类， 然后在通过factory-method指定对应的静态工厂方法创建bean。
如果静态工厂方法需要参数则用<p:constructor-arg index=" 1" value="calendar"></p:constructor-arg>传入

实例工厂方法:
有些工厂是非静态的，即必须是**实例化工厂类**才能调用工厂方法。
下面我们实例化一个工厂类CarFactory类来为Car类提供实例。

```
package com.car;
public class CarFactory
{ public Car getCar(){return new Car();}}
```

工厂类负责创建一个或多个目标类实例，工厂类方法一般以接口或抽象类变量的形式返回目标类。工厂类对外屏蔽了目标类的实例化步骤。调
用甚至不知道如何具体的目标类是什么。
下面我们在Spring 配置文件中进行配置

```
<!--工厂Bean生成目标Bean-->
<bean id=" carFactory"  class=" com.CarFactory" />
<!--工厂Bean目标Bean-->
<bean id=" car"  factory-bean=" carFactory"  factory-method=" getCar" />
```

factory-bean=" carFactory" 指定了工厂类Bean，factory-method=" getCar" 指定了工厂类Bean创建该Bean的工厂方法。
和静态工厂类似如果工厂方法需要参数则用

```
<p:constructor-arg index=" 0" value="calendar"></p:constructor-arg>传入
```

7，使用ApplicationContext ApplicationContext覆盖了BeanFactory的所有功能，并提供了更多的特，容器创建时就创建了singleton Bean
相对BeanFactory而言，ApplicationContext提供了以下扩展功能：

1. 国际化支持：继承MessageSource接口，提供国际化支持
2. 资源访问：支持对文件和URL的访问。
3. 事件传播：事件传播特性为系统中状态改变时的检测提供了良好支持。
4. 多实例加载：可以在同一个应用中加载多个Context实例，即加多个配置文件

1，国际化处理的步骤

1） **写相应的国家资源文件**如：ApplicationResources_zh.properties

注意字符的转化类似struts的国际化

2） bean.xml 的配置

```
<p:bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
  <p:property name="basename" value="com.kettas.spring.ioc.day3.ApplicationResource" />
  </p:bean>
  <p:bean id="msc" class="com.kettas.spring.ioc.day3.MessageSourceConsumer" />
  </p:beans>
```

批注 [U25]: 注意：这里声明了一个名为messageSource的Bean(注意对于Message定义，Bean ID必须为messageSource，这是目前Spring的编码规约)

3）实现类MessageSourceConsumer
具体的实现类implements MessageSourceAware。**注入messageSource ms**
得到字符：String name = ms.getMessage("name", null, Locale.CHINA);name是资源文件的key值
Locale.CHINA是中文，Locale.ENGLISH英文

2. 资源访问：即外部资源文件的获取;资源文件
两种引入外部资源的方式

```
①<!-- <p:bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
<p:property name="location" value="com/kettas/spring/ioc/day3/jdbc.properties"></p:property>
     </p:bean> -->
```

②，引入解析：xmlns:context="http://www.springframework.org/schema/context"

```
  <context:property-placeholder location="com/kettas/spring/ioc/day3/jdbc.properties"/>
  使用<p:property name="driverClassName" value="${jdbc.driver}"></p:property>
```

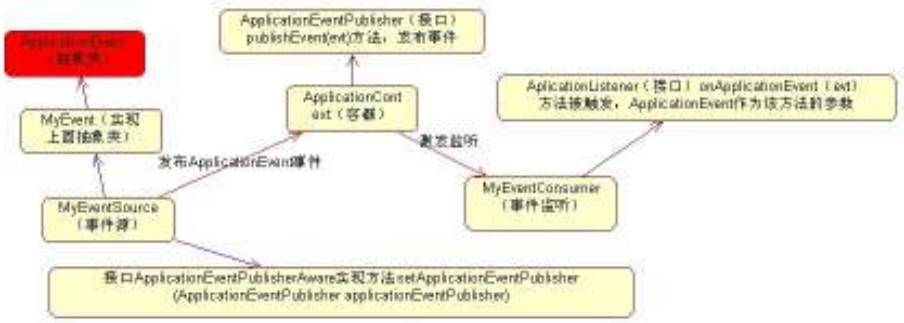
jdbc.driver是资源的key值
其它：ApplicationContext.getResource方法提供了对资源文件访问支持，如：

```
Resource rs = ctx.getResource("classpath:config.properties");
File file = rs.getFile();
```

3. 事件传播：事件机制是观察者模式的实现
事件框架两个重要的成员：

1） ApplicationEvent：容器事件,必须由ApplicationContext发布
2） ApplicationListener：监听器：可有其他任何监听器bean担任
3） ApplicationContext是事件源必须由java程序显示的触发

1)事件流程：



2)代码实例:

1， 事件源.

```
public class LogEventSource implements
                    ApplicationEventPublisherAware
{
  private ApplicationEventPublisher publisher;
  public void setApplicationEventPublisher(
                ApplicationEventPublisher publisher){
   this.publisher = publisher;
     }
  public void fireEvent(){
     LogEvent evt = new LogEvent(this, "Test message");
     publisher.publishEvent(evt);
    }
    }
```

2， 事件监听，

```
public class Logger implements ApplicationListener {
      private Log logger =
                    LogFactory.getLog(Logger.class);
       public void onApplicationEvent(
                          ApplicationEvent evt) {
          logger.info("Event type: " +
                    evt.getClass().getName());
          if(evt instanceof LogEvent){
         logger.info(((LogEvent)evt).getMessage());
           }}
}
```

3)配置文件
<p:bean id="les" class="com.kettas.spring.ioc.day3.LogEventSource"> 有相应的事件方法
</p:bean>
<p:bean class="com.kettas.spring.ioc.day3.Logger"></p:bean> 处理事件的后台
</p:beans>
说明：LogEventSourc有相应的事件方法**publisher.publishEvent(evt)主动触发容器事件**；Logge处理事件的后台

4. 多实例加载

BeanPostProcessor bean后处理器 实现**BeanPostProcessor**接口 对bean实例进一步增加功能，实现两个方法
processBeforeInitialization(Object bean，String name)方法（该方法的第一个参数是将进行后处理的实例bean，name是该bean的名字）和ProcessaAfterInitialization(Object bean，String name).在init（）或destory之前做一些处理.Spring的工具类就是通过bean的后处理器完成的。

BeanFactoryPostProcessor 容器后处理器：实现接口**BeanFactoryPostProcessor只负责处理容器本身 ，实现的方法是 postProcessBeanFactory（ConfigurableListableBeanFactory beanFactory）**参加资源的引入和获取，可以修改bean工厂bean的定义相当一个再配置的过程。类似BeanPostProcessor，ApplicationContext可以自动检测到容器中的容器后处理器，但是BeanFacory必须手动调用。

5，web中如何使用spring
1)，加入相应的jar包
2)，Web.xml的配置
<listener>
<listener-class> org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
或者
（ <servlet><servlet-name>context</servlet-name>
<servlet-class> org.springframework.web.context.ContextLoaderServlet</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet> ）
通过以上配置，Web容器会默认自动加载/WEB-INF/applicationContext.xml初始化
ApplicationContext实例，如果需要指定配置文件位置，**可通过context-param加以指定**：
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>**/WEB-INF/myApplicationContext.xml**</param-value>
</context-param>
3) Servlet中应用
WebApplicationContext wac= WebApplicationUtils.getRequiredWebApplication(Context(getServletContext()));
HelloIf h=(HelloIf)wac.getBean(“hello”);

6,BeanFactoryLocator工厂的工厂，主要的两个实现类ContextSingletonBeanFactoryLocator和SingletonBeanFactoryLocator
BeanFactoryLocator bfl = SingletonBeanFactoryLocator.getInstance();
BeanFactoryReference bfr = facLoc.useBeanFactory("il8b");
// BeanFactory fac = bfr.getFactory();
MessageSourceConsumer msc=(MessageSourceConsumer)bfr.getFactory().getBean(“xxxx”);
配置文件只能是beanRefFactory.xml且放在根目录下
<p:bean id=”il8n” class=”org.springframework.context.support.ClassPathXmlApplicationContext”>
<p:constructor-arg><p:value>spring/il8n.xml</p:value></p:constuctor-arg>
</p:bean> 应用另外一个配置文件即另外一个beanFactory

7，.让 spring 完成自动装配 Autowiring 解决<ref>标签为 javaBean 注入时难以维护而实现的

下面是几种autowire type的说明：
- byname ： 试图在容器中寻找和需要自动装配的**属性名相同**的bean或id，如果没有找到相应的bean，则这个属性未被装配上。配置文件中的id/name中查找
- byType ： 试图在容器中寻找一个与需要自动装配的**属性类型相同**的bean或id，如果没有找到，则该属性未被装配上。相当set方法注入
- constructor ： 试图在容器中寻找与需要自动装配的bean的构**造函数参数一致**的一个或多个bean，如果没找到则抛出异常。构造方法注入
- autodetect ： 首先尝试使用constructor来**自动装配**，然后再使用byType方式。

Dependeney_cheching 依赖检查 一般和自动装配配套使用 四种类型：name，simple，object ，all

缺点：spring 不一定能很准确的找到 javaBean 的依赖对象，大型应用一般不用，且配置文件可读性差

8，BeanFactoryAware和BeanNameAware
实现 BeanFactoryAware 接口的 bean 可以直接访问 Spring 容器，被容器创建以后，它会拥有一个指向 Spring 容器的引用。BeanFactoryAware 接口只有一个方法void setBeanFactory(BeanFactorybeanFactory)。配置和一般的bean一样

如果某个 bean 需要访问配置文件中本身的 id 属性，则可以使用 BeanNameAware 接口，该接口提供了回调本身的能力。实现该接口的 bean，能访问到本身的 id 属性。该接口提供一个方法:void setBeanName(String name)。

9，spring2.5标签的使用（新特性）对属性，方法的注入 减少配置文件是工作量
1）属性，方法，构造函数的标签注入
1）@Autowired @Autowired按byType自动注入 是对属性的注入，按照类型匹配原则（set和constructors）
2）@Resource(name="target")：@Resource默认按 byName自动注入罢了。@Resource有两个属性是比较重要的，分别是name和type，Spring将@Resource注解的name属性解析为bean的名字，而type属性则解析为bean的类型。所以如果使用name属性，则使用byName的自动注入策略，而使用type属性时则使用byType自动注入策略。如果既不指定name也不指定type属性，这时将通过反射机制使用byName自动注入策略。**-->常用**

3）bean lifecycle ：@PostConstruct--→init注入 @PreDestroy ----→destory方法注入
4）ClassPath 类路径扫描，就是注入bean
2）Bean的标签注入spring2.5为我们引入了组件自动扫描机制，他可以在类路径底下寻找标注了**@Component、@Service、@Controller、@Repository**注解的类，并把这些类纳入进spring容器中管理。它的作用和在xml文件中使用bean节点配置组件是一样的。要使用自动扫描机制，我们需要打开以下配置信息:
@Service用于标注业务层组件bean、@Service("studentBiz")。 @Controller用于标注控制层组件（如struts中的action）、@Repository用于标注数据访问组件，即DAO组件。@Repository("dao")
而@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，即就是一般的bean。

二，AOP相关术语

1

▲AOP是OOP的延续，是Aspect Oriented Programming的缩写，意思是面向切面的编程。并不是全部的AOP框架都是一样的。他们连接点模型的功能可能有强弱之分，有些可以字段，方法，构造函数级别的，有些只能是方法的比如spring aop 最主要的三种aop框架：AspectJ Jboss AOP Spring Aop 前面两种都可以对字段方法构造函数的支持。Sping和AspectJ有大量的协作
▲Aop添加的主要功能有：事务管理，安全，日志，检查，锁 等
▲Spring对Aop支持的4种情况：
★经典的基于代理的Aop（各个版本的spring） ★@AspectJ注解驱动的切面（spring 2.0）
★纯POJO切面（spring 2.0） ★注入式AspectJ切面（各个版本的spring）

2 名词解释：
☆关注点 (Concern)：关注点就是我们要考察或解决的问题。如订单的处理，用户的验证、用户日志记录等都属于关注点。
关注点中的核心关注点 (Core Concerns) ，是指系统中的核心功能，即真正的商业逻辑。如在一个电子商务系统中，订单处理、客户管理、库存及物流管理都是属于系统中的核心关注点。
还有一种关注点叫横切关注点 (Crosscutting Concerns) ，他们分散在每个各个模块中解决同一样的问题，跨越多个模块。如用户验证、日志管理、事务处理、数据缓存都属于横切关注点。
在 AOP 的编程方法中，主要在于对关注点的提起及抽象 。我们可以把一个复杂的系统看作是由多个关注点来有机组合来实现，一个典型的系统可能会包括几个方面的关注点，如核心业务逻辑、性能、数据存储、日志、授权、安全、线程及错误检查等，另外还有开发过程中的关注点，如易维护、易扩展等。
☆切面 (Aspect)：切面是通知和切入点的结合，通知和写入点定义了切面的全部内容—它的功能，在何时何地完成功能。在Spring 2.0 AOP中，切面可以使用通用类（基于模式的风格XML Schema 的风格） 或者在普通类中以 @Aspect 注解（@AspectJ风格）来实现。
下面的例子是基于 xml schema 风格来定义一个 Aspect(红字部分) ：

```
<aop:aspect id="aspectDemo" ref="aspectBean">
< aop:pointcut id = "myPointcut" expression = "execution(* package1.Foo.handle*(..)" />
< aop:before pointcut -ref = "myPointcut" method = "doLog" />
</aop:aspect >
```

这个定义的意思是：每执行到 package1.Foo 类的以 handle 开头的方法前，就会先执行 aspectBean 的 doLog 方法

☆连接点 (Join point)：连接点就是在程序执行过程中某个特定的点，比如某方法调用的时候或者处理异常的时候。这个点可以是一个方法、一个属性、构造函数、类静态初始化块，甚至一条语句。切面代码可以通过这些点插入到程序的一般流程之中，从而添加新的行为。 **而对于 SPRING 2.0 AOP 来说他是基于动态代理的**，故
只支持方法连接点，这个一定要记住～！**每一个方法都可以看成为一个连接点**，(AspectJ和Jboss可以提供其他aop的实现如，字段构造函数等）只有被纳入某个Point Cut的 JointPoint 才有可能被 Advice 。 通过声明一个org.aspectj.lang.JoinPoint 类型的参数可以使通知（Advice）的主体部分获得连接点信息。JoinPoint 与CutPoint 之间的关系见下面的CutPoint 的讲解

☆切入点 (Pointcut)：如果说通知定义了切面的“什么”和“何时”那么切入点就定义了“何地”。切入点指一个或多个连接点，可以理解成连接电点的集合 .我们通常使用明确的类和方法或是利用正则表达式定义这些切入点。 Advice 是通过 Pointcut 来连接和介入进你的 JointPoint 的。
比如在前面的例子中，定义了
< aop:pointcut id = "myPointcut" expression = "execution(* package1.Foo.handle*(..)" /> 那个类的那个方法使用
那么这就是定义了一个PointCut ，该Pointcut 表示“在package1.Foo类所有以handle开头的方法”
假设package1.Foo类里类似于：

```
Public class Foo {
   public handleUpload(){..}
   public handleReadFile(){..}
}
```

那么handleUpload 是一个JointPoint ，handleReadFile 也是一个Joint，那么上面定义的id=”myPointcut” 的PointCut 则是这两个JointPoint 的集合
☆通知 (Advice)：通知定义了切面是什么，以及何时使用，去了描述切面要完成的工作，通知还要解决何时执行这个工作的问题。它应该在某个方法之前调用还是之后调用，或者抛出一个异常。故通知有各种类型Advice。定义了切面中的实际逻辑（即实现 ），比如日志的写入的实际代码。换一种说法Advice 是指在定义好的切入点处，所要执行的程序代码 。
通知有以下几种：
§前置通知(Before advice) ：在切入点匹配的方法执行之前运行使用@Before 注解来声明。implements MethodBeforeAdvice 实现的方法是public void before(Method method, Object[] args, Object target)
§返回后通知 (After returning advice) ：在切入点匹配的方法返回的时候执行。使用 @AfterReturning 注解来声明
§抛出后通知 (After throwing advice) ：在切入点匹配的方法执行时抛出异常的时候运行。使用 @AfterThrowing 注解来声明
§后通知 (After (finally) advice) ：不论切入点匹配的方法是正常结束的，还是抛出异常结束的，在它结束后（finally）后通知（After (finally) advice）都会运行。使用 @After 注解来声明。这个通知必须做好处理正常返 回和异常返回两种情况。通常用来释放资源。
§环绕通知 (Around Advice) ：环绕通知既在切入点匹配的方法执行之前又在执行之后运行。并且，它可以决定这个方法在什么时候执行，如何执行，甚至是否执行。在环绕通知中，除了可以自由添加需要的横切功能以外，还需要负责主动调用连接点 (通过 proceed) 来执行激活连接点的程序 。 请尽量使用最简单的满足你需求的通知。(比如如果前置通知也可以适用的情况下，就不要使用环绕通知)
§环绕通知使用 @Around 注解来声明。而且该通知对应的方法的第一个参数必须是 ProceedingJoinPoint 类型 。在通知体内（即

通知的具体方法内），调用 ProceedingJoinPoint 的 proceed() 方法来执行连接点方法 。

☆引入（Introduction）：引入是指给一个现有类添加方法或字段属性，引入还可以在不改变现有类代码的情况下，让现有的 Java 类实现新的接口 （以及一个对应的实现 ）。相对于 Advice 可以动态改变程序的功能或流程来说，引入（Introduction）则用来改变一个类的静态结构 。比如你可以使用一个引入来使bean实现 IsModified 接口，以便简化缓存机制。举例来说，我们可以创建一个Audiable通知类，记录对象在最后一次被修改是时的状态，这很简单，只要一个方法，setLastModified(Date)，和一个实例变量来保存这个状态，这个新的方法和实例变量然后就可以引入到现在的类，从而在不修改他们的情况下让他们有新的行为和状态。

☆目标对象（ Target Object ）被一个或者多个切面（aspect）所通知（advise）的对象。也有人把它叫做 被通知（advised） 对象。 既然Spring AOP是通过运行时代理实现的，这个对象永远是一个 被代理（proxied ）对象。

☆AOP 代理（ AOP Proxy ）“代理”是向目标对象应用通知之后被创建的对象，对客户端来说目标对象和代理对象是一样的。AOP框架创建的对象，用来实现切面契约（aspect contract）（包括通知方法执行等功能）。 在Spring中，**AOP生成被代理的类有两种**:如果目标对象实现的是一个**接口**则是用JDK的 java.lang.reflect.Proxy动态代理;如果不是接口则用**CGLIB**库生成目标类的子类在创建这个子类的时候spring织入通知,并且把这个子类的调用委托给子类，这样就要注意两点，1 最好是用接口实现代理，cglib只是实现没有接口也可以通知的情况。2，被标**记final的**方法和类不能通知因为无法创建子类。。

☆织入（ Weaving ）把切面（aspect）连接到其它的应用程序类型或者对象上，并创建一个被通知（advised）的对象，这样一个行为就叫做Weaving。 这些可以在编译时（例如使用AspectJ 编译器），类加载时和运行时完成。 Spring 和其他纯 Java AOP 框架一样，在运行时完成织入 。

其实织入的方式有3种:

1、编译器织入： 切面在目标类编译时植入，这需要特殊的编译器。AspectJ 主要就是是使用这种织入方式 。

2、类加载器织入： －切面在目标类加载到JVM的时候进行织入，这需要特殊的ClassLoader。它可以在目标类被加载到程序之前增强类的字节代码。比如AspectWerkz （ 已并入 AspecJ ） 及 JBoss 就使用这种方式 。

3、运行时织入： －即在 java运行的过程中，使用Java提供代理来实现织入。根据代理产生方式的不同，运行时织入又可以进一步分为J2SE动态代理及动态字节码生成两种方式。由于J2SE动态代理只能代理接口，因此，需要借助于一些动态字节码生成器来实现对类的动态代理。大多数 AOP 实现都是采用这种运行时织入的方式，包括 Spring 。

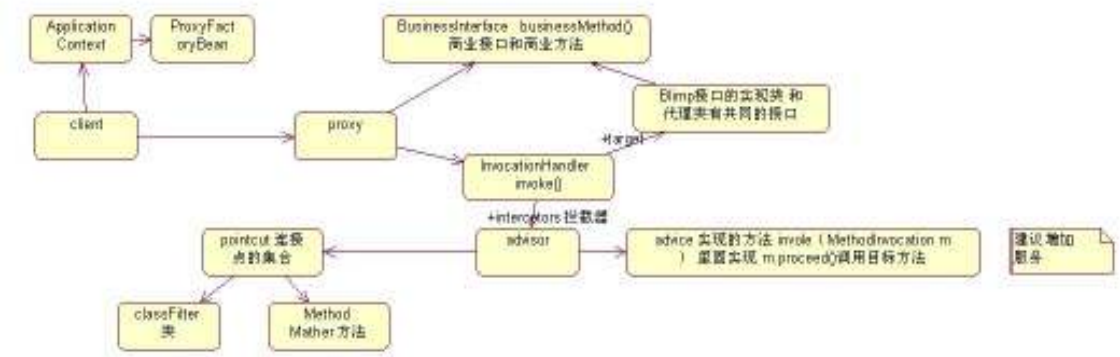
3：AOP的工作模式

代理主要有静态代理和动态代理

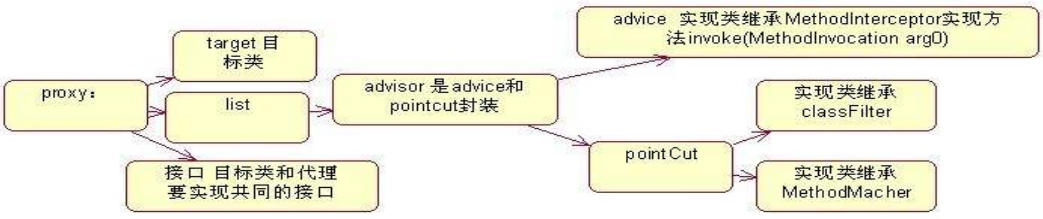
静态代理：在代理中实现接口并创建实现类对象，在对实现类的方法增加功能（不常用）

动态代理：实现implements InvocationHandler接口。实现方法

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable{
   System.out.println("=========");
        Object o=method.invoke(this.ins, args);
        System.out.println("---------");
        return o;
}
```



流程图：在配置文件的配置



配置文件代码：

```
        <!-- pointcut definition -->
    <p:bean id="cf" class="com.kettas.spring.aop.day4.MyClassFilter">
       <p:property name="classes">
            <p:set>
                 <p:value>com.kettas.spring.ioc.day1.HelloIF</p:value>   ---
            </p:set>
       </p:property>
    </p:bean>
    <p:bean id="mm" class="com.kettas.spring.aop.day4.MyMethodMatcher">
       <p:property name="methodNames">
            <p:set>
                 <p:value>sayHello</p:value>
            </p:set>
       </p:property>
    </p:bean>
    <p:bean id="pc" class="com.kettas.spring.aop.day4.MyPointcut">
       <p:property name="classFilter" ref="cf"></p:property>
       <p:property name="methodMatcher" ref="mm"></p:property>
    </p:bean>

    <!--advice   要继承implements MethodInterceptor-->
    <p:bean id="timingAdvice" class="com.kettas.spring.aop.day4.TimingInterceptor"></p:bean>

    <!-- advisor  把通知和切入点结合在一起- 最好给代理增加功能->
    <p:bean id="timingAdvisor" class="org.springframework.aop.support.DefaultPointcutAdvisor">
       <p:property name="advice" ref="timingAdvice"></p:property>
       <p:property name="pointcut" ref="pc"></p:property>
    </p:bean>

    <!—target 目标 -->
    <p:bean id="helloTarget" class="com.kettas.spring.ioc.day1.HelloIFImpl">
       <p:property name="cal">
            <p:bean class="java.util.GregorianCalendar"></p:bean>
       </p:property>
       <p:property name="user" value="world"></p:property>
    </p:bean>
    <!-- proxy -->
    <p:bean id="hello" class="org.springframework.aop.framework.ProxyFactoryBean">
       <p:property name="target" ref="helloTarget"></p:property>
       <p:property name="interceptorNames">
            <p:list>
                 <p:idref bean="timingAdvisor"/>   增加一种服务
                  <p:idref bean="xxxAdvisor"/>   增加另一种服务
            </p:list>
       </p:property>
       <p:property name="proxyInterfaces">    要和目标类实现共同的接口
            <p:value>com.kettas.spring.ioc.day1.HelloIF</p:value>
       </p:property>
    </p:bean>
</p:beans>
```

简化配置：有可能只是目标类不一样，其他的都是一样的。解决每一个目标类都需要一个复杂的代理过程配置的问题，可以简化配置的问题 抽象ProxyFactoyBean的方法 如：

```
       <!--  抽象proxy --定义抽象的类，只是没有目标类，其他的通知和接口都一样>
   <p:bean id="helloBase" class="org.springframework.aop.framework.ProxyFactoryBean" abstract=“true” >
       <p:property name="interceptorNames">
            <p:list>
                 <p:idref bean="timingAdvisor"/>   增加一种服务
                  <p:idref bean="xxxAdvisor"/>   增加另一种服务
            </p:list>
       </p:property>
       <p:property name="proxyInterfaces">    要和目标类实现共同的接口
            <p:value>com.kettas.spring.ioc.day1.HelloIF</p:value>
       </p:property>
    </p:bean>
</p:beans>
```

真正的代理

```
    <!—target 目标 继承抽象方法  只用写目标类就可以了 -->

    <!-- proxy -->
    <p:bean id="hello" parent=”  helloBase” >
       <p:property name="target" ref="helloTarget"></p:property>
    </p:bean>
```

4：AOP的自动代理

Spring的aop机制提供两类方式实现类代理。**一种是单个代理，一种是自动代理**。

单个代理通过**ProxyFactoryBean**来实现（就如上面的配置），

自动代理：自动代理能够让切面定义来决定那个bean需要代理，不需要我们为特定的bean明确的创建代理从而提供一个更完整

的aop实现 通过**BeanNameAutoProxyCreator**或者 **DefaultAdvisorAutoProxyCreator**实现。

◆采用单个代理方式 （费时费力,配置复杂臃肿）

下面就采用自动代理

实现代理bean的两种方式：

1,“基于Spring上下文的里声明的通知者bean的基本自动代理”：通知者的切点表达式用于决定哪个bean和那些方法需要被代理

2,”基于@AspectJ注释驱动切面的自动代理”：切面里包含的通知里指定的切点将用于选择哪个bean和哪个方法要被代理

第一种：<！——自动代理增加此行，容器会自动根据通知要匹配的切入点，为包含切入点的类创建 代理。注意这个bean没有id，因为永远都不会直接引用它——>

<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/> feedom.net

第二种 自动代理@AspectJ切面

然而**aspectJ提供可一种基于jdk1.5注解技术的方式**，使得配置文件更少，更方便。能够把POJO类注释为切面这通常称为@AspectJ.我们利用@AspectJ注释，我们不需要声明任何额外的类或Bean就可以把POJO转换成一个切面例如：

@Aspect 定义切面不再是普通的POJO了 在POJO类中加注释

```
public class AspectJMixAspect {
   private Log logger = LogFactory.getLog(AspectJMixAspect.class);
   @Pointcut("execution(* *..HelloIF.*(..)) || execution(* *..TestBeanIF.*(..))")定义切入点那些类的那些方法添加
   public void allMethods() {    }
   @Pointcut("execution(* *..TestBeanIF.toDate(..)) && args(dateStr)")
   public void toDate(String dateStr) {
   }
   @Around("allMethods()") 环绕通知
   public Object timing(ProceedingJoinPoint pjp) throws Throwable {
      long begin = System.currentTimeMillis();
      Object ret = pjp.proceed();
      long end = System.currentTimeMillis();
      String methodName = pjp.getSignature().getName();
      String targetClass = pjp.getTarget().getClass().getName();
      logger.info("Around advice: It took " + (end - begin) + "ms to call "
                                    + methodName + " on " + targetClass);
                        return ret;
                        }
   @Before("allMethods()")
   public void logBefore(JoinPoint jp) {
     logger.info("Before advice: " + jp.getSignature().toLongString());
   }
   @AfterReturning(value="toDate(dateStr)", returning = "date", argNames = "date, dateStr")
   public void afterReturning(Date date, String dateStr) {
      logger.info("call method toDate(" + dateStr + ") and return " + date);
   }
   @AfterThrowing(value="toDate(dateStr)", throwing="ex", argNames="dateStr, ex")
   public void afterThrowing(String dateStr, ParseException ex){方法名任意但参数要和argNames=""中的参数顺序一样,
   }
}
```

配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<p:beans xmlns:aop="http://www.springframework.org/schema/aop"
                  xmlns:p="http://www.springframework.org/schema/beans"
                  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xsi:schemaLocation="http://www.springframework.org/schema/beans
                  http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                  http://www.springframework.org/schema/aop
                  http://www.springframework.org/schema/aop/spring-aop-2.5.xsd ">
                  <!-- target -->
                  <p:bean id="hello" class="com.kettas.spring.ioc.day1.HelloIFImpl">
                     <p:property name="cal">
                          <p:bean class="java.util.GregorianCalendar"></p:bean>
                     </p:property>
                     <p:property name="user" value="world"></p:property>
                  </p:bean>
                  <p:bean id="testBean" class="com.kettas.spring.aop.day5.TestBean">
                     <p:property name="pattern" value="yyyy-MM-dd"></p:property>
                  </p:bean>
                  <!-- apsect bean -->
                  <p:bean id="aspectJAspect" class="com.kettas.spring.aop.day5.AspectJMixAspect"></p:bean>
            <aop:aspectj-autoproxy></aop:aspectj-autoproxy> 声明自动代理bean需要命名空
                                                            间:aop="http://www.springframework.org/schema/aop"
</p:beans>
```

5，定义纯粹的POJO切面 不在普通的bean中加注释，而是在配置文件中配置

```
<!-- target -->
<bean xxx ></bean>
←-Pojo bean-->
<bean id =” aspectJAspect” class=” com.kettas.spring.aop.day5.AspectJMixAspect” />
```

```
<aop:config>
  <aop:aspect ref=" aspectJAspect" > 将aspectJAspect定义为切面   下面定义加方法的类和方法
 <aop:pointcut id="allMethods" expression="execution(* *..HelloIF.*(..)) or execution(* *..TestBeanIF.*(..))"/>
  <aop:advisor pointcut-ref="allMethods" advice-ref="timingAdvice" />
  <aop:advisor pointcut-ref="allMethods" before-method=“logBefore" />
  <aop:advisor pointcut-ref="allMethods" advice-ref="logAfterAdvice" /> 引入外部的通知
  <aop:advisor pointcut="execution(* *..TestBeanIF.toDate(..))" advice-ref="logThrowingAdvice" />
</aop:config>
```

14，注入AspectJ切面

为什么要用AspectJ：AspectJ提供了Spring AOP很多不能实现的多种切点类型（比如属性，构造方法切入，由于不能实现构造方法的切入spring aop就不能实现对象创建过程的通知）

AspectJ是一个代码生成工具（Code Generator）。AspectJ有自己的语法编译工具，编译的结果是Java Class文件，运行的时候，classpath需要包含AspectJ的一个jar文件。AspectJ是AOP最早成熟的Java实现，它稍微扩展了一下Java语言，增加了一些Keyword等

```
public aspect TestAspectJ {
    public TestAspectJ();
    public pointcut writeOperations():
        execution(public boolean Worker.createData()) ||
        execution(public boolean Worker.updateData()) ||
        execution(public boolean AnotherWorker.updateData()) ;
    before() : writeOperations() {
        XXXXXX;  advice body
    }
    after() : writeOperations() {
       XXXX;// advice body
    }
}
```

配置文件

```
<bean class=" xxx/TeatAspectJ"  factory-method=" aspectof" >
  <property name="" ref="" /></bean>
```

说明机制：这个<bean>和其他的bean并没有太多区别，只是多了 factory-nmthod属性 要想获得切面的实例，就必须使用factory-method来调用 aspectof（）方法，而不是调用TestAspectJ的构造方法，简单来说Spring不使用《bean》声明来创建TestAspectJ实例，它已经被AspectJ运行时创建了，Spring通过aspectof（）工厂方法获得切面的引用，然后利用<bean>元素对她执行属性注入

上述代码关键点是pointcut，意味切入点或触发点，那么在那些条件下该点会触发呢？是后面红字标识的一些情况，在执行Worker的createData()方法，Worker的update方法等时触发

Pointcut类似触发器，是事件Event发生源，一旦pointcut被触发，将会产生相应的动作Action，这部分Action称为Advice。

Advice在AspectJ有三种：before、 after、Around之分，上述aspect Lock代码中使用了Advice的两种before和after。

所以AOP有两个基本的术语：Pointcut和Advice。你可以用事件机制的Event和Action来类比理解它们

其中advice部分又有：

Interceptor - 解释器并没有在AspectJ出现，在使用JDK动态代理API实现的AOP框架中使用，解释有方法调用或对象构造或者字段访问等事件，是调用者和被调用者之间的纽带，综合了Decorator/代理模式甚至职责链等模式。

Introduction - 修改一个类，以增加字段、方法或构造或者执行新的接口，包括Mixin实现。

Spring2.5 aop新特性

在配置文件中的配置：就和**POJO**中的配置一样

在java类中的配置:就和**自动代理类**配置一样 ,只是在配置的时候注意加上<aop:aspectj-autoproxy></aop:aspectj-autoproxy>

注意： execution(* *..HelloIF.*(..)))的含义 任意权限（*），任意返回值（*），任意包下（..），类名（HelloIF），任意方法（*），任意参数（..）。 星号（*）代表之间没有逗号和点号，（..）代表可以有逗号和点号

5， **切入点指示符**：前面的execution就是一个切入点指示符。Spring AOP还支持的切入点指示符有

Within：配置特定的连接点 。 this：Aop代理必须是指定类型的实例 如this(org.crazyit.service.AccountService)匹配实现了AccountService接口的所有连接点。 Target:和this的表达类似。实现限定目标对象必须是指定类型的实例。

Bean：匹配实例内方法执行的连接点 bean（tradeService）可以用* 通配符

三：Data Access 数据库的模板操作

1，好处：支持事务的管理 2有统一的异常处理 RuntimeException 3 Support各种数据库

2，数据源的配置：在spring中数据源的配置有

★有JDBC驱动程序定义的数据源 ：在org.springframework.jdbc.datesource包有两个实现类 DriverManagerDateSouce。每个请求时都建立一个连接 或者SingleConnectionDataSource 配置api中的set方法，用户名，密码，url driver

★由JNDI查询的数据源 ：在tomcate配置数据源，还有Jbose等都可以 通过jndi引用就可以了

```
<bean id=" dataSource"  class="…jndi.jndiObjectFactory" >
  <property name=" jndi"  value=" /jdbc/rantzDataSource" />
    <property name=" resourceRef"  value=" true" />
```

★连接池的数据源:Spring 没有提供数据源连接池，但是DBCP项目提供了一个，下载相关的jar到lib中 配置类似第一种JDBC驱动的配置

3 JDBC ：我们知道，Spring JDBC的主要目标是为了简化JDBC的编程，方便我们构建健壮的应用程序。这里，它的一个基本设计理念，就是将JDBC编程中变化的和不变化的分开。 在JDBC中，什么是变化的？毫无疑问，SQL语句是变化的。那什么是不变化的？正确的使用JDBC的方式是不变化的。使用JDBC模板
提供了三个模板：JdbcTemplate 最基本的jdbc模板 使用索引参数查询数据库
NamedParameterJdbcTemplate ， 查询时把值绑定到SQL里的命名参数而不是使用索引参数
SimpleJdbcTemplate利用java5的特性，比如自动装箱，通用，可变参数列表来简化JDBC模板的使用

以JdbcTemplate为例子说明：在已经配置datasource的情况下就能使用 JdbcTemplate有相关的查询，插入，等方法

在xml中配置模板JdbcTemplate bean

```
<bean id=" jdbcTemplate" class=" org.springframework.jdbc.core.JdbcTemplate" >
    < property name=" dataSource" ref=" dataSource" >
  </bean>
```

在dao层的bean中**加入JdbcTemplate引用**就可以了，这样就可以使用它来访问数据库

```
如：class JdbcDao{
        Private JdbcTemplate jdbcTemplate;
         //setJdbcTemplate 方法注入
        Public User getUserById(long id){
          List users=jdbcTemple.query(sql,new Object[]{Long.valueOf(id)}),new RowMapper(){
                        Public Object mapRow(ResultSet rs,int rowNum) throws SQLException
                             User user=new User();
                              user.setId(rs.getInt(1)); ….;
                               return user;
                        }
         Return users.size()>0?(user)users.get(0):null;
        }
    }
```

说明：一个字符串，包含用于从数据库里选择数据的 SQL 语句。
一个 Object 数组，包含与查询里索引参数绑定的值。Sql几个问号就绑定几个值
一个 RowMapper 对象，它从ResultSet 里提取数值并构造一个域对象。封装对象返回结果

然后像下面的设置一样注入模板JdbcTemplate属性就可以了

```
<bean id=" jdbcDao" class=" …/JdbcTemplate" >
    <property name=" JdbcTemplate" ref=" jdbcTemplate" ></bean>
```

JdbcDaoSupport

注意：可以使用Spring 对JDBC 的DAO 支持类 JdbcDaoSupport
思想： 全部 DAO 对象创建一个通用父类，在其中设置JdbcTemplate 属性，然后让全部DAO 继承这个类，使用父类的JdbcTemplate 进行数据访问，这样可以减少配置量。Spring 的**JdbcDaoSupport** 就是用于编写基于JDBC 的DAO 类的基类，我们只需让自己的DAO 类继承它即可 例如上面的

```
class JdbcDao extends JdbcDaoSupport{
      Public User getUserById(long id){
        List users=getJdbcTemple().query(sql,new Object[]{Long.valueOf(id)}),new RowMapper(){
                      Public Object mapRow(ResultSet rs,int rowNum) throws SQLException
                           User user=new User();
                            user.setId(rs.getInt(1)); ….;
                             return user;
                      }
       Return users.size()>0?(user)users.get(0):null;
      }
  }
```

在配置文件与配置不继承JdbcDaoSupport 的DAO 没有什么区别

```
<bean id=" jdbcDao" class=" …/JdbcTemplate" >
    <property name=" JdbcTemplate" ref=" jdbcTemplate" ></bean>
```

4 在Spring 里集成Hibernate

1）下载相关的Hibernante的jar包到lib文件 注意最好是3.0版本以上

2）使用Hibernate 模板 HibernateTemplate 有各种增删改查的方法

▲与 Hibernate 进行交互的主要接口是org.hibernate.Session。这个Session 接口提供了基本的数据访问功能，比如从数据库保存、更新、删除和加载对象。获得 Hibernate Session 对象引用的标准方式是实现。Hibernate的SessionFactory 接口。SessionFactory负责打开、关闭和管理Hibernate Session，以及其他一些功能。

3）为了让事情简单一些，Spring 提供了HibernateDaoSupport，用法类似JdbcDaoSupport，它能够让我们把会话工厂Bean直接装配到DAO 类

两种获得session的配置 bean.xml

```
<!-- 自己加载 hibernate -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName">
      <value>oracle.jdbc.driver.OracleDriver</value>
    </property>
    <property name="url">
      <value>jdbc:oracle:thin:@localhost:1521:XE</value>
    </property>
    <property name="password">
      <value>kettas</value>
    </property>
    <property name="username">
       <value>kettas</value>
     </property>
```

批注 [U26]: 配置数据源，和前面的配置类似

```
  </bean>
  <bean id="factory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
      <property name="dataSource">
          <ref  bean="dataSource"/>
      </property>
      <property name="mappingResources">
         <list>
           <value>com/kettas/entity/shoppingcart.hbm.xml</value>
         </list>
      </property>
      <property name="hibernateProperties">
         <props>
             <prop key="hibernate.dialect">org.hibernate.dialect.Oracle9Dialect</prop>
             <prop key="hibernate.show_sql">true</prop>
             <prop key="hibernate.format_sql">true</prop>
         </props>
      </property>
  </bean>
  <!-- use 外面的 hibernate.cfg.xml
  <bean id="factory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
          <property name="configLocation" value="classpath:hibernate.cfg.xml"></property>
     </bean>-->

<bean  id="DaoSupport" class="com.kettas.dao.impl.Hibernate2DaoSupportImpl">
     <property name="sessionFactory">
       <ref bean="factory"/>
     </property>
  </bean>

  <bean id="template"
     class="org.springframework.orm.hibernate3.HibernateTemplate">
       <property name="sessionFactory">  模板注入sessionFactory
          <ref bean="factory" />
     </property>
  </bean>

HibernateTemplate中常用的方法：getHibernateTemplate().delete(o); update() 。。。。;
                                super.getHibernateTemplate().executeFind(
                                 new HibernateCallback() {  回调使用hibenate的方法
                                      public Object doInHibernate(Session session)
                                                throws HibernateException, SQLException { 。。。。}
```

3， 使用Hibernate 3 上下文会话

如果使用 Hibernate 2，那么就只能使用HibernateTemplate。
Hibernate 上下文会话的主要好处在于把DAO 实现与Spring 解耦。
Hibernate 上下文的主要缺点是它们抛出Hibernate 特有的异常。虽然HibernateException 是运行时异常，但这个异常体系是Hibernate 特有的，而且不像Spring 存留异常体系那样与ORM 无关，这可能会在程序向不同ORM 迁移时产生问题。
HibernateTemplate 的缺点是具有一定的侵入性，HibernateDaoSupport)，HibernateRantDao 类都被耦合到Spring API。
我们还有另外一种办法。Hibernate 3 引入的上下文会话可以管理每事务一个会话，也就不需要让HibernateTemplate 来确保这种情况了。所以，我们可以在DAO 里装配一个Hibernate SessionFactory 来取代HibernateTemplate

```
public class HibernateRantDao implements RantDao {
   private SessionFactory sessionFactory;
   public void setSessionFactory(SessionFactory sessionFactory) {
       this.sessionFactory = sessionFactory; }
   public void saveRant(Rant rant) {
   sessionFactory.getCurrentSession().saveOrUpdate(rant);
}}

<bean id="rantDao" class="com.roadrantz.dao.hibernate.HibernateRantDao">
     <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

4， JPA 是个基于POJO 的存留机制，不仅从Hibernate 和Java 数据对象（JDO）吸取了观念，还适当地混合了Java 5 的注解功能。用法类似Hibernate。基于JPA 的程序使用EntityManagerFactory 的一个实现来获取EntityManager 的实例，他用entityManagerFactory进行管理，相当sessionFactory。可以在persistence.xml,类似hibernate的hibernate.cfg.xml配置打他source等资源，也可以直接在spring的配置文件配置（使用见EJB）

在spring中的配置

```
   <bean id="entityManagerFactory"
   class="org.springframework.orm.jpa.
       □    LocalEntityManagerFactoryBean">
   <property name="persistenceUnitName" value="rantzPU" />
   </bean>
   <bean id="jpaTemplate" class="org.springframework.orm.jpa.JpaTemplate">
            <property name="entityManagerFactory" ref="entityManagerFactory" />
```

批注 [U27]: 映射文件 ，并且可以配置多个

批注 [U28]: Hibernate 的配置文件，数据库的方言一定要配置 hibernate.dialect

批注 [U29]: Hibernate2DaoSupportImpl extends HibernateDaoSupport 这样就不需要HibernateTemplate 属性，而是使用 getHibernateTemplate() 方法获得由 HibernateDaoSupport 创建的 HibernateTemplate，HibernateDaoSupport 需要一个Hibernate SessionFactory，这样它才能在内部生成一个 HibernateTemplate。所以我们要把sessionFactory Bean 装配到 Hibernate2DaoSupportImpl 的sessionFactory 属性

批注 [U30]: persistence.xml 中 <persistence-unit name="rantzPU"> persistenceUnitName 属性的值就是 persistence.xml 里存留单元的名称。

</bean>
获得jpa的模板，用法类似Hibernate，就不多说，Ejb中再讨论

四：事务管理

1，**事务**（见hibernate的ACID）：Spring和EJB一样，不仅提供对程序控制事务管理的支持（**手动事务**），也对提供声明式事务管理的支持（**容器管理事务**），但是Spring对程序控制事务管理的支持与EJB很不一样。EJB的事务管理和Java Transaction API(JPA)密不可分。和Ejb不同的是Spring采用的是一种回调机制，把真实的事务从事务代码中抽象出来，那么Spring就不需要JPA的实现。选择手动事务还是容器管理，就是在细微控制和简便操作之间做出选择。想精确控制事务就可以选择手动事务，不用那么精确就可以容器管理事务。

事务管理器：不管你是在bean中代码编写事务还是用切面（aspect aop）那样声明事务，都需要Spring的事务管理器连接特定平台的事务实现，每一种访问形式都有一个事务管理器。比如：

```
jdbc.datasource.DataSourceTransactionManager：jdbc连接的事务管理，iBATIS也支持
orm.hibernate3. HibernateTransactionManager  ：hibernate3的事务支持
orm.jpa.JpaTransactionManager ：jpa的事务支持
orm.jdo.JdoTransactionManager   ：Jdo事务管理支持
```

这些事务管理器分别充当了某个特定的事务实现门面，这样你就只要和Spring的事务打交道，而不用关心实际上的事务是怎么实现的（门面模式）

各种事务管理器的配置，以**Hibernate 3**为例：

```
<bean id="transactionManager" class="org.springframework.
   □ orm.hibernate3.HibernateTransactionManager">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

批注 [U31]: 这 sessionFactory 属性必须被装配为一个 Hibernate 的 SessionFactory 。name=“sessionFactory”是固定的。HibernateTransactionManager 把事务管理委托给一个当前 Hibernate 会话中检索到的 org.hibernate.Transaction 对象，当事务成功则调用 Transaction.commit(),失败则调用 Transaction.rollback().

JDBC事务管理

```
<bean id="transactionManager" class="org.springframework.jdbc.
  □ datasource.DataSourceTransactionManager">  --->  DataSourceTransactionManager调用Connection来管理事务
  <property name="dataSource" ref="dataSource"/>
</bean>
```

4，在spring中手动编写事务

利用事务模板TransactionTemplate来手动添加事务

```
public void addRant(Rant rant) {
    transactionTemplate.execute(->transactionTemplate是注入transactionManager得到的
   new TransactionCallback() {-> TransactionCallback()只有一个方法实现doInTransaction，用一个匿名内部类实现
         public Object doInTransaction(TransactionStatus ts) {  ------>在事务内执行
               try {
                 rantDao.saveRant(rant);
               } catch (Exception e) {
                  ts.setRollbackOnly();------------------>出现异常就回滚
              }return null;
                }
}
```

配置文件

```
<bean id="rantService"
class="com.roadrantz.service.RantServiceImpl">
…
<property name="transactionTemplate ">
   <bean class="org.springframework.transaction.support.
          □ TransactionTemplate">
       <property name="transactionManager"
           ref="transactionManager" />
     </bean>
  </property>
</bean>
```

批注 [U32]: transactionTemplate 是注入 transactionManager 得到的

5，声明式事务

可以把事务想成一个切面，那么就可以用事务性边界包裹Biz层的方法，然后注入事务

Spring提供了三种在配置文件声明事务性边界的方式：★常用的Spring aop代理 bean来支持事务。★但在Spring 2中增加了两种新的方式：简单的XML声明（xml-declared）事务和★注释驱动事务

1）代理事务：声明式事务管理通过使用Spring的TransactionProxyFactoryBean代理POJO来完成。TransactionProxyFactoryBean是ProxyFactoryBean的一个特化，他知道如何通过事务性边界包裹一个POJO的方法来代理他们。

```
<bean id="rantService" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="target" ref="rantServiceTarget" />       -->装配事务目标，相当给biz层的方法加事务
<property name="proxyInterfaces" value="com.roadrantz.service.RantService" />
<property name="transactionManager" ref="transactionManager" />   -->提供适当的事务管理器
 <property name="transactionAttributes">
```

批注 [U33]: 声明什么方法将在一个事务内运行，以及相关的事务参数每一个<prop>的 key 是方法，可以有*通配符。Value 是事务传播行为：

```
        <props>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_SUPPORTS,readOnly</prop>
        </props>
</property>
</bean>
```

事务传播行为：

PROPAGATION_REQUIRED ：当前方法必须有一个事务，有事务则运行该事务，没有则开始新的事务。--->最常用

PROPAGATION_MANDATORY：该方法必须有事务，没有事务则抛出异常

PROPAGATION_NESTED ：该方法运行在嵌套事务中。如果封装事务不存在则就像第一种PROPAGATION_REQUIRED

PROPAGATION_NEVER ：该方法不能有事务，有事务则抛出异常。

PROPAGATION_NOT_SUPPORTED：该方法不能有事务，如果有事务，则将该方法在运行期间挂起。

PROPAGATION_REQUIRES_NEW：方法必须运行在事务里，

PROPAGATION_SUPPORTS：表示当前方法不需要事务性上下文，但是如果有一个事务已经在运行的话，他可以在这个事务里运行。

PROPAGATION, ISOLATION, readOnly, -Exception, +Exception

(传播行为) （隔离级别 可选） （事务只读 可选） （回滚规则 可选）

可以创建事务模板简化配置 ：建立事务的抽象声明

```
<bean id="TXServiceTemplate" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
      abstract="true" >
<property name="transactionManager" ref="transactionManager" />   -->提供适当的事务管理器
 <property name="transactionAttributes">
      <props>
          <prop key="add*">PROPAGATION_REQUIRED</prop>
          <prop key="*">PROPAGATION_SUPPORTS,readOnly</prop>
      </props>
</property>
</bean>
<bean id=" ranBiz"  parent=" TXServiceTemplate" >
  <property name="proxyInterfaces" value="com.roadrantz.service.RantService" />
 <property name="transactionManager" ref="transactionManager" />   -->提供适当的事务管理器
</bean>
```

2）在Spring2.0声明事务 <tx>上面的方法会导致配置很臃肿，下面就是更简单的配置

在Spring2.0中专门为声明事务提供了一些新的标签 tx名称空间下

```
xmlns:tx=http://www.springframework.org/schema/tx
xmlns:aop="http://www.springframework.org/schema/aop"
    http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
<tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
        <tx:method name="get*" read-only="true"/>
        <tx:method name="create*" />
        <tx:method name="join*"/>
    </tx:attributes>
</tx:advice>

<aop:config>
     <aop:advisor pointcut="execution(* *..Roster.*(..))" advice-ref="txAdvice"/>
</aop:config>
```

3）定义注释驱动的事务，@Transactional可以在源代码中注释来进一步简化配置

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
@Service("roster")
public class RosterImpl implements Rosterpublic
  @Transactional                ---------------  -------->方法层面的事务
Public  Player createPlayer(Player p) {
     playerDao.save(p);
     return p;
}

<context:component-scan
      base-package="com.kettas.spring.dao.day5.roster.dao,com.kettas.spring.dao.day5.roster.biz">
</context:component-scan>
```

<tx:annotation-driven/> 自动搜索@Transactional的bean 然后把事务通知告诉它。

五：Spring的MVC

目前比较好的MVC,老牌的有Struts、Webwork。新兴的MVC 框架有Spring MVC、Tapestry、JSF等。这些大多是著名团队的作品，另外还有一些边缘团队的作品，也相当出色，如Dinamica、VRaptor等。

六：Spring的安全机制 Spring Security：它提供全面的安全性解决方案，同时在Web请求和方法调用处理身份确认和授权，利用依赖注入和aop技术。主要名词：

批注 [U34]: 抽象事务，其他biz 类可以继承，从而简化配置

批注 [U35]: 声明什么方法将在一个事务内运行，以及相关的事务参数每一个<prop>的key 是方法，可以有*通配符。Value 是事务传播行为：

批注 [U36]: 继承上面的抽象

批注 [U37]: Aop 空间也应该包含在内，新的声明事务要依赖一些新的 Aop 元素

批注 [U38]: 定义通知，事务参数在一个<tx:atrributes>中定义，他包含一个或多个<tx:method>方法 txManager 是上面已经定义的事务管理器。

批注 [U39]: 定义通知器，告知那些类那些方法使用 advice-ref="txAdvice"定义的事务通知

批注 [U40]: 这是类层面的事务注入，说明下面所有的方法都支持事务，且是只读的。该事务标签也可以注释接口，那么该接口的实现方法都支持事务。

1安全拦截器：相当应用的一把锁，能够阻止对应用程序中保护资源的访问（通常是用户名和密码 正确才能打开锁）
2 认证管理器：通过用户名和密码来做到这点的，负责确定用户的身份。是由
3：访问决策管理器：根据你的身份来确定你是否拥有对资源的访问，即授权管理
4：运行身份管理器：运行身份管理器可以用来使用另一个身份替换你的身份，从而允许你访问应用程序内部更深处的受保护对象。
5：调用后管理器：访问结束后还可以返回取得相关资源，其他安全管理器组件在受保护资源被访问之前实施某种形式的安全措施强制执行，而调用后管理器则是在受保护资源被访问之后执行安全措施。

认证管理器是由org.acegisecurity. AuthenticationManager 接口定义的
ProviderManager 是认证管理器的一个实现，它将验证身份的责任委托给一个或多个认证提供者
DaoAuthenticationProvider 是一个简单的认证提供者，它使用数据存取对象（DAO）来从关系数据库中检索用户Spring Security 支持通过LdapAuthenticationProvider 根据LDAP 进行身份验证，
LdapAuthenticationProvider 是一个知道如何根据LDAP 仓库查看用户凭证的认证提供信息（包括用户的密码）
身份验证只是Spring Security 安全保护机制中的第一步。一旦Spring Security 弄清用户的身份后，它必须决定是否允许用户访问由它保护的资源
Spring Security 对Web 安全性的支持大量地依赖于Servlet 过滤器。这些过滤器拦截进入请求，并且在你的应用程序处理该请求之前进行某些安全处理。Spring Security 提供有若干个过滤器，它们能够拦截Servlet 请求，并将这些请求转给认证和访问决策管理器处理，从而增强安全性。

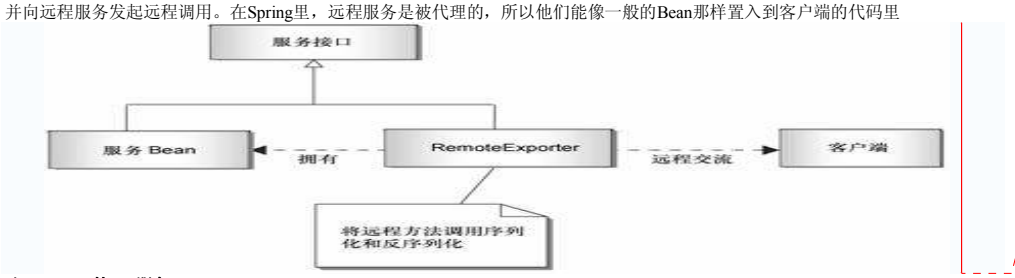
七：Spring的远程调用：Spring远程支持是由普通（Spring）POJO实现的，这使得开发具有远程访问功能的服务变得相当容易
四种远程调用技术：
◆ 远程方法调用（RMI） ◆ Caucho的Hessian和Burlap
◆Spring自己的Http invoker ◆使用SOAP和JAX-RPC的web Services
Spring对上面的远程服务调用都有支持
Spring远程调用支持6种不同的RPC模式：远程方法调用（RMI）、Caucho的Hessian和Burlap、Spring自己的HTTP invoker、EJB和使用JAX-RPC 的Web Services。表6.1概括地论述了每个模式，并简略讨论它们在不同情况下的用处。

表6.1 Spring远程调用所支持的RPC模式

RPC模式	在何种情况下有用
远程方法调用（RMI）	不考虑网络限制（如防火墙）时，访问/公开基于Java的服务
Hessian或 Burlap	考虑网络限制时，通过HTTP访问/公开基于Java的服务
HTTP invoker	考虑网络限制时，访问/公开基于Spring的服务
EJB	访问用EJB实现的遗留的J2EE系统
JAX-RPC	访问Web Services

远程方法调用（RMI）。通过使用 RmiProxyFactoryBean 和 RmiServiceExporter，Spring同时支持传统的RMI（使用java.rmi.Remote接口和java.rmi.RemoteException）和通过RMI调用器实现的透明远程调用（支持任何Java接口）。
Spring的HTTP调用器。Spring提供了一种特殊的允许通过HTTP进行Java串行化的远程调用策略，支持任意Java接口（就像RMI调用器）。相对应的支持类是 HttpInvokerProxyFactoryBean 和 HttpInvokerServiceExporter。
Hessian。通过 HessianProxyFactoryBean 和 HessianServiceExporter，可以使用Caucho提供的基于HTTP的轻量级二进制协议来透明地暴露服务。
Burlap。 Burlap是Caucho的另外一个子项目，可以作为Hessian基于XML的替代方案。Spring提供了诸如 BurlapProxyFactoryBean 和 BurlapServiceExporter 的支持类。
JAX RPC。Spring通过JAX-RPC为远程Web服务提供支持。

客户端发起对代理的调用，好像是代理提供了这些服务的功能一样。代理代表客户端和远程服务交流。它处理连接的具体情况，并向远程服务发起远程调用。在Spring里，远程服务是被代理的，所以他们能像一般的Bean那样置入到客户端的代码里



批注 [U41]: 代理和服务实现相同的接口，用代理调用服务的方法，而客户端调用代理的方法就像调用服务端的方法一样。客户端没有察觉。

八：Spring的Web服务：
Spring支持：使用JAX-RPC暴露服务 访问Web服务
除了上面所说的支持方法，你还可以用XFire xfire.codehaus.org 来暴露你的服务。XFire是一个轻量级的SOAP库，目前在Codehaus开发。
使用JAXI-RPC暴露服务
Spring对JAX-RPC Servlet的端点实现有个方便的基类 – ServletEndpointSupport。为暴露我们的Account服务，我们继承了Spring的ServletEndpointSupport类来实现业务逻辑，这里通常把调用委托给业务层。
访问服务：
Spring有两个工厂bean用来创建Web服务代理，LocalJaxRpcServiceFactoryBean 和 JaxRpcPortProxyFactoryBean。前者只返回一个JAX-RPT服务类供我们使用。后者是一个全功能的版本，可以返回一个实现我们业务服务接口的代理。本例中，我们使用后者来为前面段落中暴露的AccountService端点创建一个代理。你将看到Spring对Web服务提供了极好的支持，只需要很少的代码 – 大多数都是通过类似下面的Spring配置文件：

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
  <property name="serviceInterface" value="example.RemoteAccountService"/>
  <property name="wsdlDocumentUrl" value="http://localhost:8080/account/services/accountService?WSDL"/>
  <property name="namespaceUri" value="http://localhost:8080/account/services/accountService"/>
```

```
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountPort"/>
  </bean>
```

XFire是一个Codehaus提供的轻量级SOAP库。在写作这个文档时（2005年3月）XFire还处于开发阶段。虽然Spring提供了稳定的支持，但是在未来应该会加入更多特性。暴露XFire是通过XFire自身带的context，这个context将和RemoteExporter风格的bean相结合，后者需要被加入到在你的WebApplicationContext中。

九：Spring的消息 Java Message Service （JMS）：

前面的RMI,Hesian ， Burlap，HTTP invoker web服务的间通信，都是程序之同步,即当客户端调用远程方法时必须在方法完成之后才能继续执行。

JMS是一种异步消息传递的标准API，客户端不需要等待服务端处理消息，客户端发送消息，会继续执行，这是因为客户端假设服务端最终能够收到并处理这条消息。类似Ajax

发送消息 使用JMS的模板JsmTemplate 接收消息

十：Spring和EJB的整合：

Spring有两种方法提供对EJB的支持：

Spring能让你在Spring的配置文件里，把EJB作为Bean来声明。这样，把EJB引用置入到其他Bean的属性里就成为可能了，好像EJB就是另一个POJO。

Spring能让你写EJB，让EJB成为Spring配置的Bean的代理的工作。

Spring提供了两个代理工厂Bean，来代理EJB的访问：

LocalStatelessSessionProxyFactoryBean——用来访问本地EJB（EJB和它的客户端在同一个容器中）。

SimpleRemoteStatelessSessionProxyFactoryBean——用来访问远程EJB（EJB和它的客户端在独立的容器中）。

十一：访问企业服务：

SSH和Ajax的整合

一：准备工作

把Hibernate的jar包 ， spring的jar 包 ， Struts的jar包 ajax中的jar包比如dwr。

二：配置文件说明

Web.xml

```
  <listener>
    <listener-class>
          org.springframework.web.context.ContextLoaderListener
     </listener-class>
  </listener>
  <context-param>
          <param-name>contextConfigLocation</param-name>
          <param-value>/WEB-INF/classes/com/kettas/config/springBean.xml</param-value>
  </context-param>
<servlet>
<servlet-name>action</servlet-name>
   <servlet-class> org.apache.struts.action.ActionServlet</servlet-class>
   <init-param>
        <param-name>config</param-name>
        <param-value>/WEB-INF/struts-config.xml</param-value>
   </init-param>
   <init-param>
      <param-name>debug</param-name>
      <param-value>3</param-value>
  </init-param>
  <init-param>
      <param-name>detail</param-name>
      <param-value>3</param-value>
   </init-param>
<load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping>
     <servlet-name>action</servlet-name>
     <url-pattern>*.do</url-pattern>
</servlet-mapping>

  <servlet>
    <servlet-name>dwr</servlet-name>
    <servlet-class>org.directwebremoting.servlet.DwrServlet</servlet-class>
    <init-param>
       <param-name>debug</param-name>
       <param-value>true</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>
```

struts-config.xml

```
 <form-beans>
       <form-bean name="dynaActionForm"
             type="org.apache.struts.action.DynaActionForm">
            <!-- admin formBean -->
           <form-property name="loginName" type="java.lang.String" />
        </form-bean>
</form-beans>
<global-exceptions />
<global-forwards />
<action-mappings>
        <action path="/adminLogin" name="dynaActionForm" parameter="adminLogin"
                  type="org.springframework.web.struts.DelegatingActionProxy">
            <forward name="ok" path="/admin/admin.html" redirect="true" />
            <forward name="error" path="/error.jsp" />
         </action>
</action-mappings>
<!--message-resources
      parameter="com.yourcompany.struts.ApplicationResources" /-->
<plug-in
        className="org.springframework.web.struts.ContextLoaderPlugIn">
       <set-property property="contextConfigLocation"
                    value="/WEB-INF/classes/beans.xml" />
</plug-in>
```

批注 [U42]: 用监听器引入spring的配置文件，也可以在struts-config.xml中以<plug-in>方式注入

批注 [U43]: 引入 Struts 的控制器，并且引入配置文件,并初始化属性

批注 [U44]: 对 dwr 的映射

批注 [U45]: 真正的实现类由 Spring 代理，故在 spring.xml 配置文件中有相关的注入，见下面的 spring 配置

批注 [U46]: 注入 spring 的配置文件

Spring.xml

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
     <property name="configLocation" value="classpath:hibernate.cfg.xml"></property>
</bean>
<bean id="template" class="org.springframework.orm.hibernate3.HibernateTemplate">
     <property name="sessionFactory"><ref local="sessionFactory" /></property>

</bean>
<!--Dao####################################################################################-->
<bean id="daoSupport" class="com.kettas.upp02.dao.impl.DaoSupport">
       <property name="sessionFactory"><ref bean="sessionFactory" /></property>
</bean>
<bean id="accountDao" class="com.kettas.upp02.dao.impl.AccountDaoImpl" parent="daoSupport">
</bean>
<!--Biz####################################################################################-->
<bean id="accountBiz" class="com.kettas.upp02.biz.impl.AccountBizImpl">
        <property name="accountDao"><ref bean="accountDao" /></property>
        <property name="subAccountDao"><ref bean="subAccountDao" /> </property>
</bean>
<!--BizProxy###############################################################################-->
<!--
      <bean id="accountBizProxy"
              class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
              <property name="target" ref="accountBiz"></property>
              <property name="transactionManager" ref="transactionManager"></property>
               <property name="transactionAttributes">
                  <props>
                      <prop key="*">PROPAGATION_REQUIRED</prop>
                      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
                 </props>
               </property>
</bean>
-->
<!-- ############################################################################ -->
<!-- transaction  切面注入事物-->
<bean id="transaction" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
      <property name="sessionFactory"> <ref bean="sessionFactory" /></property>
</bean>
<!-- interceptor -->
<bean id="interceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
       <property name="transactionManager"><ref bean="transaction" /></property>
       <property name="transactionAttributes">
           <props>
                 <prop key="*">PROPAGATION_REQUIRED</prop>
                 <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
           </props>
       </property>
</bean>
<!-- BeanNameAutoProxyCreator -->
<bean id="auto" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
        <property name="beanNames">
                <list><value>*Biz</value></list>
        </property>
       <property name="interceptorNames">
                <list><value>interceptor</value></list>
        </property>
</bean>

<!-- AdminAction~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~-->
<bean name="/adminLogin,/createAdmin"
      class="com.kettas.upp02.web.background.AdminAction">
                      <property name="adminBiz" ref="adminBiz" />
                      <property name="roleBiz" ref="roleBiz" />
                      <property name="powerBiz" ref="powerBiz" />
</bean>
```

批注 [U47]: 外面的 hibernate.cfg.xml

批注 [U48]: 定义一个 dao 父类，方便其他 dao 的继承

批注 [U49]: 注入 dao 层的类

批注 [U50]: 这种代理的方式注入事务很麻烦，下面有一种更简单的，自动注入方式

批注 [U51]: 自动注入的方式，方便配置文件，有两个属性，beanNames 和 interceptorNames 并且都可以注入多个类和切面

批注 [U52]: 和 struts 的路径一样，下面的属性是 biz 层的注入。

dwr.xml的配置

```
<dwr>
      <allow>
            <create javascript="schoolBiz" creator="spring" scope="application">
                  <param name="beanName" value="schoolBiz"></param>
                 <!--<include method="getSchoolById" ></include>-->
            </create>
```

批注 [U53]: 表示是 spring 创建的。name="beanName"是固定的。schoolBiz 就是 spring 注入的 id 值

```
            </create>
            <!-- 自己定义的实体 -->
            <convert match="com.kettas.entity.School" converter="bean">
               <!-- <param name="exclude" value="students"></param>  不会显示的属性，避免在加载时，类的关联属性无法
                   加载 -->
            </convert>
      </allow>
</dwr>
```

总结

1,action中利用request.getParameter("*",x)可以获得jsp也没传过来的参数

2,jsp也没中加<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>后

<bean:parameter id="cid" name="companyId" />获得参数，使用方法：${cid}

<bean:define id="c" name="company" ></bean:define>获得对象,使用同上

3,hibernate中：一对多关系中，hibernate默认是懒的初始化（lazy=true），这样当你查询一的一方时，它不会级联查询多的一方，这样单的一方就没办法使用保存在类中多的一方的属性

如果想使用就必须在多的一方中加上lazy=false，如：

<many-to-one name="company" column="company_id" lazy="false"></many-to-one>

当多的一方是通过外键指向单的一方主键时：默认的

4,jsp 页面中实现国际化方法：

<%@taglib uri="http://struts.apache.org/tags-html" prefix="html"%>

<html:messages id="error" name="errorMessages">

${error}

</html:messages> 其中 name 为配置文件中写的

<%@taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>

<bean:message key="login.title"/> key 中为配置文件写的

5,页面编码问题：

通常可以通过写过滤器来解决，但是写了过滤器之后，还可能出现页面乱码，url 不支持中文传输，所有通常在传中文时最好用 post 方法（表单实现，get 是用 url 实现），如果要用 get 也可以在 tomcat 中 conf.server.xml 文件中找到修改端口的地方加上 URIEncoding="编码方式"即可。

6,级联删除：

如果想删除 A 表，但是 B 表的一个外键指向 A 表，关系维护在 B 表中，这时要向删除 A 表中的数据，首先要查出 B 表中指向 A 表的数据，然后把 B 表中的外键设为空，在删除 A 表中数据。

如果两个表都维护关系，写级联就 OK

7,通过 hibernate 配置文件自动创建表：

（1）写好实体，和详细的 xml 影射文件

（2）在 hibernate 配置文件中添加

<!-- 此属性可以自动生成 sql 语句，自动创建表

<property name="hbm2ddl.auto">create</property>

-->

（3）然后在 main 方法中执行：ApplicationContext ctx = new ClassPathXmlApplicationContext("applicationContext.xml");

加载后即可生成 sql 语句。

8,利用 PowerDesigner 通过代码生成关系图：

file--->reverse engineer--->datebase--->弹出对话框

输入名字，选择数据库，单击确定

添加.sql 文件后单击确定后可以生产图。

9,以下代码很有效：实现了分页查询所有对象的模板 // <T>定义泛型

```
public <T> List<T> selectAllObject(Class<T> clazz,Page page) {
            String hql0 = "select count(*) from " + clazz.getSimpleName();
            String hql = "from " + clazz.getSimpleName();
            List<T> Objects = new ArrayList<T>();
            //  query()是分页实现方法
            for(Object obj : this.query(hql0, null, hql, null, page)){
                  Objects.add((T) obj);
            }
            return Objects;
}
```

实现用对象或对象的属性实现分页查询：

/**实现 HibernateCallback（）方法所传的参数必须是 final 的，

*final T exampleEntity : 传过来的实体对象 final String propertyName : 实体对象的属性名字

*final Object startData, final Object endData : 通过时间查询时需要 final Page page : 分页时需要

```
*/

import org.hibernate.Criteria;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.criterion.Criterion;
import org.hibernate.criterion.Example;
import org.hibernate.criterion.Expression;
import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

protected <T> List<T> selectByExampleEntity(final T exampleEntity,final String propertyName,
      final Object startData,final Object endData,final Page page) {
      Object object = getHibernateTemplate().executeFind(new HibernateCallback() {
            public Object doInHibernate(Session session) throws HibernateException, SQLException {
                  Example example = Example.create(exampleEntity); //通过实例创建 Example
                        //通过 session 获得标准
                  Criteria criteria = session.createCriteria(exampleEntity.getClass());
                              //如果以下这 3 个参数都不为空的时候执行
                              //propertyName 是两个时间对应的列的名字，两个时间是同一列的不同结果
                  if(propertyName != null && startDate != null && endDate != null){
                        Criterion c = Expression.between(propertyName, startDate, endDate);
                        criteria.add((Criterion) c);
                  }
                  criteria.add(example);
                  page.setOrderCount(new Long(criteria.list().size()));
                  criteria.setFirstResult((page.getPageNumber()-1)*page.getMax());
                  criteria.setMaxResults(page.getMax());
                  return criteria.list();
            }
      });
      return (List<T>) object;
}
```

10, 在 hibernate.hbm.xml 文件配置中指定 lazy="false"的话，固然可以在查询该对象的时候将其关系对象一并查询出来，只是影响全局，或许我们只是想在某些情况下顺带查询关系对象，在不需要的时候则浪费了查询时间以及内存空间。为了缩小影响范围，我们可以不指定 lazy="false"，只是在 web.xml 文件中，配置一个 spring 提供的过滤器，如此便可在一次连接的范围内扩大事务，即使事务结束、Session 被关闭，再获取关系属性一样可行(school.getStudents() ...)，使用如下所示 :

```
<filter>
      <filter-name>OpenSessionInViewFilter</filter-name>
      <filter-class>org.springframework.orm.hibernate3.support.OpenSessionInViewFilter</filter-class>
       <init-param>
          <param-name>singleSession</param-name>
          <param-value>true</param-value>
       </init-param>
       <init-param>
                <param-name>sessionFactoryBeanName</param-name>
                <!-- 对应 spring.xml 中的 id -->
                <param-value>sessionFactory</param-value>
       </init-param>
       </filter>
       <filter-mapping>
          <filter-name>OpenSessionInViewFilter</filter-name>
          <!-- 指定要扩大事务的连接(一次请求) -->
          <url-pattern>*.do</url-pattern>
       </filter-mapping>
```

11,Oracle 数据库：

当你修改了自己的计算机名称后，就会出现无法登陆 Oracle 主页等其它状况;
找到你数据库的安装目录：D:\..\oracle\product\10.2.0\server\NETWORK\ADMIN 下边有两个文件
listener.ora 和 tnsnames.ora,打开它们并把其中的主机名（HOST）改为现在的主机名就 OK

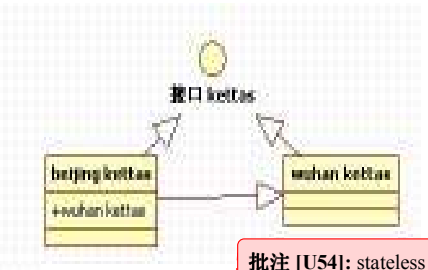
当两个表的列相同时（个数，名称，约束）：
union 连接两个 select 语句可以将两个表的数据加在一起（去掉重复），union all 显示所有
eg: select id,name from student1 union (all) select id,name from student2 ;

12,日期和字符串之间相互转换：

```
Date d = new SimpleDateFormat("yyyy-mm-dd").parse("1234-25-30");
```

```
System.out.println(d);
String str = new SimpleDateFormat("yyyy-mm-dd").format(d);
System.out.print(str);
```

EJB



1、J2EE 是什么？

J2EE 是 Sun 公司提出的多层(multi-diered),分布式(distributed),基于组件(component-base)的企业级应用模型(enterpriese application model).在这样的一个应用系统中，可按照功能划分为不同的组件，这些组件又可在不同计算机上，并且处于相应的层次(tier)中。所属层次包括客户层(clietn tier)组件,web 层和组件,Business 层和组件,企业信息系统(EIS)层。

JAVA EE=**Components**（规范—servlet/jsp/ejb/jsf）+**Contanous**（容器→WEB 容器/EJB 容器/spring 容器）+**MVC 模式+service（服务）+多层次**

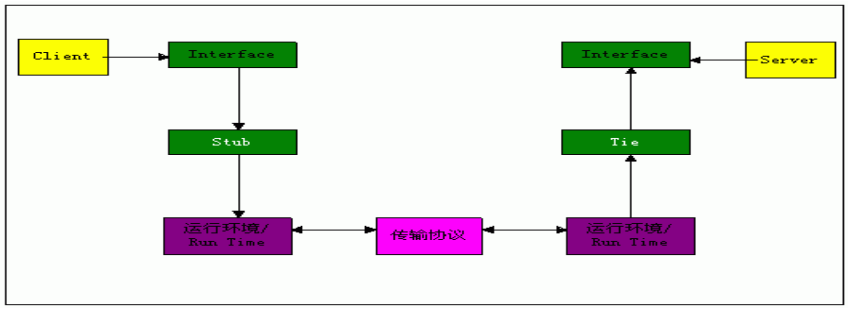
在大型的应用中一般会有多个 web 服务器和 app 应用，这样可以解决，1，多台机器负载均衡，server 集群（类似多个 Tomcat）2，app 调用 web 的方法，远程方法调用，3，透明的错误切换（从 web1 切换到 web2 客户端不能察觉，其实是 web2 有 web1 的备份）4 db 数据共享

2，Enterprice JavaBeans 3.0 是一个用于分布式业务应用的标准服务端组件模型。采用 Enterprice JavaBeans 架构编写的应用是可伸的、事务性的、多用户安全的。可以一次编写这些应用，然后部署在任何支持 Enterprice JavaBeans 规范的服务器平台，如 jboss、weblogic。
Enterprise JavaBean（EJB）定义了三种企业 Bean，分别是会话 Bean（**Session Bean**），实体 Bean（**Entity Bean**）和消息驱动 Bean（**MessageDriven Bean**）

3， EJB 的开发流程：①开发 ejb ②组装应用 ③ 部署 ④ 管理 ⑤ 开发工具

4，**Session Bean**：Session Bean 用于实现业务逻辑，它分为有状态 bean 和无状态 bean。每当客户端请求时，容器就会选择一个 Session Bean 来为客户端服务,session bean 是一个与其他系统打交道的边界 bean，响应客户端的调用（生命较短）。Session Bean 可以直接访问数据库，但更多时候，它会通过 Entity Bean 实现数据访问。

1）EJB 实现远程调用的机制：代理



注意：客户端和服务端**都实现一样的接口，**在代理的实现类保存目标类的引用,**这就叫代理. 如果没有共同的接口，那就是委托**。客户端调用服务端的应用，其实调用的是 **Stub（桩）**。

2,Ejb 的编写：

①商业接口和 bean：注意区别本地接口和远程接口，在同一个虚拟机的是本地接口，不是在同一个虚拟机的是远程接口（远程接口消耗资源） 如 本地 local 远程 stateless（mappedName=“ejb/datecom”）

②，本地接口@Local 远程接口@Remote
例子：

```
@Local
public interface LocalHello{
  String sayHello(String name);
}
```

```
import javax.ejb.Remote;
@Remote
```

批注 [U54]: stateless 是无状态的，stateful 有状态

批注 [U55]: Jndi的名字。因为客户端需要通过**JNDI** 查找**EJB,**那么**JNDI** 是什么？JNDI(The Java Naming and Directory Interface，Java 命名和目录接口) 是一组在Java 应用中访问命名和目录服务的API。为开发人员提供了查找和访问各种命名和目录服务的通用、统一的方式。借助于JNDI 提供的接口，能够通过名字定位用户、机器、网络、对象服务等。
命名服务：就像DNS 一样，通过命名服务器提供服务，大部分的J2EE 服务器都含有命名服务器。
目录服务：一种简化的RDBMS 系统，通过目录具有的属性保存一些简单的信息。目录服务通过目录服务器实现，比如微软ACTIVE DIRECTORY 等。

批注 [U56]: 在运行一个含有main 方法的 java 程序时，会对应的启动一个**虚拟机**。一个web 容器（tomcat...）或是一个 EJB 容器（jboss...）同样都对应着一个 java 虚拟机。那么，在这些虚拟机之间的方法调用就是远程方法调用，而无须考虑这些虚拟机是否在同一台机器上。

批注 [U57]: 远程接口

```
interface RemoteHello{
  String sayHello(String name);
}

import javax.ejb.Stateless;
@Stateless(mappedName=" ejb/helloEjb ")
@remote(RemoteHello.class)
public class StatelessHello implements RemoteHello, LocalHello {
  public  String sayHello(String name) {
       return "hello"+name;
  }
}
```

调用 session bean

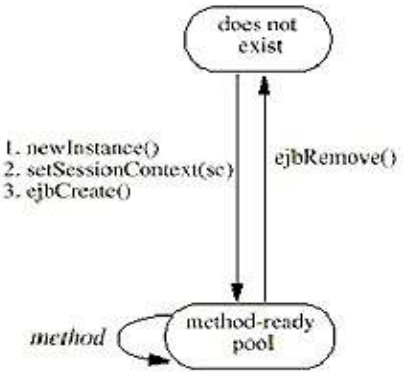
```
    Context ctx = new InitialContext();
    HelloRemote   hr = (HelloRemote)ctx.lookup("ejb/helloEjb");
```

3，无状态 session Bean 主要用来实现单次使用的服务，该服务能被启用许多次，但是由于无状态会话 Bean 并不保留任何有关状态的信息，其效果是每次调用提供单独的使用。在很多情况下，无状态会话 Bean 提供可重用的单次使用服务。无状态就意味着共享，多客户端，可以构建 pooling 池

注意：单例 bean 一定是无状态的。通过配置文件 sun-ejb-jar.xml(netbeans 环境 glassfish 服务器)来配置这个池。

配置文件举例如下 ：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE sun-ejb-jar PUBLIC "-//Sun Microsystems,
      Inc.//DTD Application Server 9.0 EJB 3.0//EN"
      "http://www.sun.com/software/appserver/dtds/sun-ejb-jar_3_0-0.dtd">
<sun-ejb-jar>
  <enterprise-beans>
      <ejb>
          <ejb-name>SllcBean</ejb-name><!-- 指定 ejb 的名字 -->
          <!-- 对池的配置 -->
          <bean-pool>
             <steady-pool-size>2</steady-pool-size>    <!-- 稳定状态下 池中 ejb 数量 -->
             <resize-quantity>2</resize-quantity>      <!-- 创建 ejb 的步长 一次新创建两个 -->
             <max-pool-size>6</max-pool-size>          <!-- 池中容许的 ejb 最大数目 -->
          </bean-pool>
      </ejb>
  </enterprise-beans>
</sun-ejb-jar>
```



2）**无状态 session bean 的生命周期**

1 构造方法(只能无参)
2 属性的依赖注入
3 @PostConstruct 标记的方法
4 商业方法

不存在---------->存在(在 Stateless Bean Pool 中被管理)---

(被销毁 时间容器决定 销毁前执行@PreDestroy 标记的方法)

---------->不存在

由不存在到存在，并非在用户调用此 ejb 时才转换，容器对 ejb 的加载分为懒汉式和非懒汉式，若为后者，则在 ejb 容器启动时就完成了对 ejb 的加载，就是说容器启动后 ejb 就是存在的状态了。

有状态的 session bean 有状态 Bean 是一个可以维持自身状态的会话 Bean。每个用户都有自己的一个实例，在用户的生存期内，StatefulSession Bean 保持了用户的信息，即“有状态”；一旦用户灭亡（调用结束或实例结束），Stateful Session Bean 的生命期也告结束。即每个用户最初都会得到一个初始的 Stateful Session Bean。Stateful Session Bean 的开发步骤与 Stateless Session Bean 的开发步骤相同。**Stateful Session Bean 跟踪用户的状态，一个 bean 只能被一个用户所使用。此类 bean 不是管理在 bean pool 中，是比较消耗资源的，一般的时候要尽量避免使用。**

4， 。

1）容器保存信息（java ee 接口的保存）

注意：把常用的 session bean 放在内存中，把不常用的放在外部，只有用到时才调用。

2） 继承 **SessionSynchronization** 接口实现事务同步。

支持事务，实现同步接口，用来保存事务在失败时可以回复原来的状态。

afterBegin()容器刚启动时，容器用于保护现场的方法。

beforeCompletion ()

批注 [U58]: // mappedName="ejb/sllc" 表明此 ejb 被创建后 挂在 JNDI 服务器的名为"ejb/sllc"的节点上 context.lookup("ejb/sllc") 方法即可取得此 ejb 的引用(不是 ejb 的本身)

批注 [U59]: 继承远程接口,也可以用标签@remote（xxx.class）

批注 [U60]: 采用 jndi 来调用资源

批注 [U61]: pooling 池的配置

批注 [U62]: 有状态的 stateful session bean 必须实现 Serialiazble 接口，只有这样容器才能在他们不再使用的时候，序列化存储他们的状态

afterCompletion（）在事务提交时被容器调用，如果参数是 false 则表示事务失败，此时要处理状态的恢复。

3） 例子和无状态的差不多，只是用@stateful 表示

```
 // 有状态会话 bean 挂到 JNDI 服务器名为"ejb/teller"的节点上
@Stateful(mappedName = "ejb/teller")
 // mappedName = "jdbc/pointbase", type = DataSource.class 指定别名的真实值 对应的数据类型
 // 倘若不在这里指定的话 就得在配置文件中说明了
@Resource(name = "jdbc/tellerDB", mappedName = "jdbc/pointbase", type = DataSource.class)
 // 这里实现 SessionSynchronization 接口，是位了防止数据库操作失败而可能造成的用户数据不准确
public class TellerBean implements TellerRemote, TellerLocal, SessionSynchronization {
    private String name;
    private int cash, cashBak;
    // transient 关键字表明此对象无需保存到外部存储中去
    private transient Connection con;
    // 使用依赖注入
    @Resource
    private SessionContext sctx;
    // 放入外部存储前执行
    @PrePassivate
    // bean 被销毁前执行
    @PreDestroy
    public void passivate() {
        try {
           // bean 被放入外部存储前或销毁前 将连接归还连接池
            con.close();
        } catch (SQLException ex) {
            Logger.getLogger(TellerBean.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // 从外部存储取回后执行
    @PostActivate
    // 构造完毕后执行
    @PostConstruct
    public void activate() {
        try {
            Context ctx = new InitialContext();
            // 数据源实际 JNDI 节点名为"jdbc/oracle" 这里采用的是别名
            // "java:comp/env"起始，说明采用的是别名
            // "jdbc/tellerDB"对应值在类开头已注入
            DataSource ds = (DataSource) ctx.lookup("java:comp/env/jdbc/tellerDB");
            con = ds.getConnection();
        } catch (Exception e) {
            throw new EJBException(e);
        }
    }

    public void login(String name, int cash) {
        this.name = name;
        this.cash = cash;
    }

    public void openAccount(String accountno, String name, int balance) {
        cash += balance;
        PreparedStatement stm = null;
        try {
            stm = con.prepareStatement("insert into bank_account(accountno, name, balance) values (?, ?, ?)");
            stm.setString(1, accountno);
            stm.setString(2, name);
            stm.setInt(3, balance);
            stm.executeUpdate();
        } catch (SQLException se) {
           // EJB 容器只有在捕捉到 RunTime 异常后才会认为事务执行失败
            // 故这里调用 SessionContext 的 setRollbackOnly 方法 告诉容器 事务失败 要回滚
            sctx.setRollbackOnly();
            se.printStackTrace();
        } finally {
            if (stm != null) {
                try {
                    stm.close();
```

批注 [U63]: /* 为了测试几个标签，这里采用 Connection 作为实例变量* 这里本不应该采用 Connection 作为实例变量，应当如下这么注入 DataSource 从 JNDI 服务器上找到名为"jdbc/oracle"的节点，此节点挂着的正是已配置好的 DataSource
注意 @Resource 注入时 mappedName 不能用别名 只能是节点真实值
@Resource(mappedName="jdbc/oracle")
private DataSource ds;

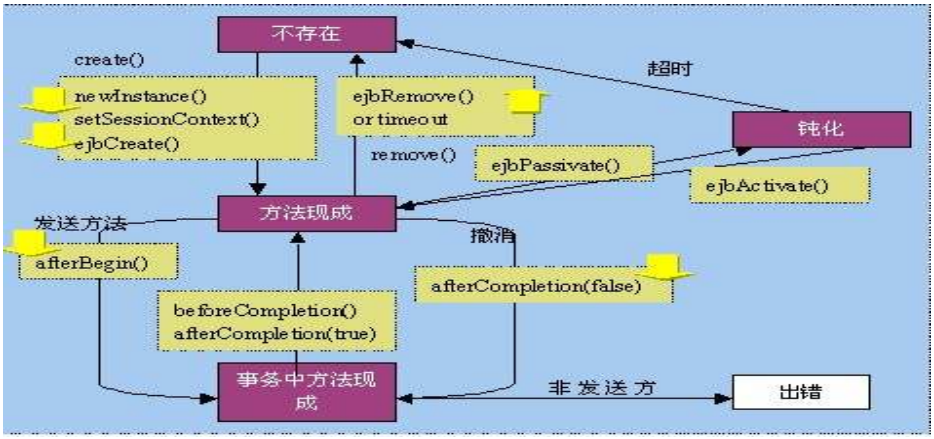
```
                } catch (SQLException se) {
                }
            }
        }
    }

    public void deposit(String accountno, int amt) {
        cash += amt;
        PreparedStatement stm = null;
        try {
            stm = con.prepareStatement("update bank_account set balance=balance+? where accountno=?");
            stm.setInt(1, amt);
            stm.setString(2, accountno);
            stm.executeUpdate();
        } catch (SQLException se) {
            sctx.setRollbackOnly();
            se.printStackTrace();
        } finally {
            if (stm != null) {
                try {
                    stm.close();
                } catch (SQLException se) {
                }
            }
        }
    }

    public int withdraw(String accountno, int amt) {
        return 0;
    }

    public int getBalance(String accountno) {
        PreparedStatement stm = null;
        ResultSet rs = null;
        try {
            stm = con.prepareStatement("select balance from bank_account where accountno=?");
            stm.setString(1, accountno);
            rs = stm.executeQuery();
            if (rs.next()) {
                return rs.getInt(1);
            }
            return -1;
        } catch (SQLException se) {
            sctx.setRollbackOnly();
            return -1;
        }
    }

    /********** SessionSynchronization 接口提供的三个方法 **********/
    // 事务开始后立刻执行
    public void afterBegin() throws EJBException, RemoteException {
        cashBak = cash; // 将 cash 中数据拷贝一份到 cashBak 中 以备 cash 数据出错时没有备份
    }
    // 事务提交前执行
    public void beforeCompletion() throws EJBException, RemoteException {
    }
    // 事务提交后执行 boolean 的真假代表事务执行的成败
    public void afterCompletion(boolean committed) throws EJBException, RemoteException {
        if(!committed) cash = cashBak; // 若 cash 数据出错 则恢复
    }

    // 执行@Remove 标签标记的方法 告诉 EJB 容器 此 bean 可以被销毁了
    // 执行此方法后 然后会执行@PreDestroy 方法 接着销毁此 bean 至此 此 bean 生命结束
    @Remove
    public int logout() {
        return cash;
    }
}
```

单元测试 测试类 : 测试方法如下(要使用的包为 JUnit 4.1) 单元测试。

```
package net.kettas.ejb3.test.day2;
import javax.naming.Context;
```

```
import javax.naming.InitialContext;
import net.kettas.ejb3.day2.TellerRemote;
import org.junit.Assert;
import org.junit.Test;
public class TellerEJBJUnitTest {
   @Test
   public void tellerOpers() throws Exception {
        Context ctx = new InitialContext();
        TellerRemote tr = (TellerRemote)ctx.lookup("ejb/teller");
        Assert.assertNotNull(tr);
        tr.login("george", 100000);
        tr.openAccount("312", "abc", 10000);
        Assert.assertEquals(10000, tr.getBalance("312"));
        tr.deposit("312", 2000);
        Assert.assertEquals(12000, tr.getBalance("312"));
        Assert.assertEquals(112000, tr.logout());
    }
}
```

4)有状态的 session bean 的生命周期



5， **Stateless Session Bean 与 Stateful Session Bean 的区别**。这两种 Session Bean 都可以将系统逻辑放在方法之中执行，不同的是 Stateful Session Bean 可以记录呼叫者的状态，因此一个使用者会有自己的一个实例。Stateless Session Bean 虽然也是逻辑组件，但是他却不负责记录使用者状态，也就是说当使用者呼叫 Stateless Session Bean 的时候，EJB 容器并不会寻找特定的 Stateless Session Bean 的实体来执行这个 method。换言之，很可能数个使用者在执行某个 Stateless Session Bean 的 methods 时，会是同一个 Bean 的实例在执行。从内存方面来看，Stateful Session Bean 与 Stateless Session Bean 比较，Stateful Session Bean 会消耗 J2EE Server 较多的内存，然而 Stateful Session Bean 的优势却在于他可以维持使用者的状态。

6 Invoke 拦截器 @ArounedInvoke 方法参数

实现步骤：①拦截器是增加功能的 ejb 的构造方法无参数

```
public class LogInterceptor {
private static Logger logger = Logger.getLogger("LogInterceptor");
 @PostConstruct
public void construct(InvocationContext ic) {
    logger.info("@PostConstruct: " + ic.getTarget().getClass().getSimpleName());
}
@PreDestroy
public void destroy(InvocationContext ic) {
    logger.info("@PreDestroy: " + ic.getTarget().getClass().getSimpleName());
}
@AroundInvoke
public Object invoke(InvocationContext ic) throws Throwable{
    logger.info("before calling " + ic.getMethod().getName());
    Object ret = ic.proceed();
    logger.info("after calling " + ic.getMethod().getName());
    return ret;
}
}
```

②在 session bean 中调用拦截器

```
@Stateless(mappedName="ejb/sllc")
@Interceptors(LogInterceptor.class)
public class SllcBean implements SllcRemote, SllcLocal {
  private static Logger logger =
          Logger.getLogger("SllcBean");
  private SessionContext sctx;
  public SllcBean(){
```

批注 [U64]: // 构造后执行 InvocationContext 对象记录着被拦截的 ejb 的信息,这里此对象是被注入的 注意不能作为实例变量采用@Resource 注入 因为具有"层面"的特征 倘若在被拦截的 ejb 中也含有 @PostConstruct 标记的方法 则 ejb 的方法会被覆盖而不会被执行

批注 [U65]: 在商业方法前后添加功能。

批注 [U66]: 直接用标签 **@Interceptors(LogInterceptor.class)就可以调拦截器**

```
    logger.info("Constructor: SllcBean");
  }
  @Resource
  public void setContext(SessionContext sctx){
    logger.info("Dependency injection: setContext");
    this.sctx = sctx;
  }
  @PostConstruct
  public void construct(){
    logger.info("@PostConstruct: construct");
  }
  @PreDestroy
  public void destroy(){
    logger.info("@PreDestroy: destroy");
  }
    public void test() {
        logger.info("just test.");
    }
}
```

批注 [U67]: 注意：对于一个无状态的 session bean 必须提供一个无参的构造方法。

批注 [U68]: Session bean 的生命周期的标签，见下面的 session bean 的生命周期

7，**生命周期的标签：** Session Bean 的生命周期

EJB 容器创建和管理 session bean 实例,有些时候，你可能需要定制 session bean 的管理过程。例如，你可能想在创建 session bean 实例的时候初始化字段变量,或在 bean 实例被销毁的时候关掉外部资源。上述这些，你都可能通过在 bean 类中定义生命周期的回调方法来实现。这些方法将会被容器在生命周期的不同阶段调用（如：创建或销毁时）。通过使有下面所列的注释，EJB 3.0 允许你将任何方法指定为回调方法。这不同于 EJB 2.1，EJB 2.1 中，所有的回调方法必须实现，即使是空的。EJB 3.0 中，bean 可以有任意数量，任意名字的回调方法。

- **@PostConstruct**：当 bean 对象完成实例化后，使用了这个注释的方法会被立即调用。这个注释同时适用于有状态和无状态的会话 bean。
- **@PreDestroy**：使用这个注释的方法会在容器从它的对象池中销毁一个无用的或者过期的 bean 实例之前调用。这个注释同时适用于有状态和无状态的会话 bean。
- **@PrePassivate**：当一个有状态的 session bean 实例空闲过长的时间，容器将会钝化(passivate)它，并把它的状态保存在缓存当中。使用这个注释的方法会在容器钝化 bean 实例之前调用。这个注释适用于有状态的会话 bean。当钝化后，又经过一段时间该 bean 仍然没有被操作，容器将会把它从存储介质中删除。以后，任何针对该 bean 方法的调用容器都会抛出例外。
- **@PostActivate**：当客户端再次使用已经被钝化的有状态 session bean 时，新的实例被创建，状态被恢复。使用此注释的 session bean 会在 bean 的激活完成时调用。这个注释只适用于有状态的会话 bean。
- **@Init**：这个注释指定了有状态 session bean 初始化的方法。它区别于@PostConstruct 注释在于：多个@Init 注释方法可以同时存在于有状态 session bean 中，但每个 bean 实例只会有一个@Init 注释的方法会被调用。这取决于 bean 是如何创建的（细节请看 EJB 3.0 规范）。@PostConstruct 在@Init 之后被调用。

另一个有用的生命周期方法注释是@Remove，特别是对于有状态 session bean。当应用通过存根对象调用使用了 **@Remove** 注释的方法时，容器就知道在该方法执行完毕后，要把 bean 实例从对象池中移走。

8， 在 EJB 中依赖注入(dependency injection)， 类似 spring 的注入

EJB 3.0，对任何 POJO,提供了一个简单的和优雅的方法来解藕服务对象和资源。使用@EJB 注释，你可以将 EJB 存根对象注入到任何 EJB 3.0 容器管理的 POJO 中。如果注释用在一个属性变量上，容器将会在它被第一次访问之前赋值给它。依赖注入只工作在本地命名服务中，因此你不能注入远程服务器的对象。

例子：

```
@Stateless(mappedName=" ejb/ InjectionBean ")
@Remote (Injection.class)
public class InjectionBean implements Injection {
   @EJB (beanName="HelloWorldBean")
   //@EJB (mappedName="HelloWorldBean/remote")
   HelloWorld helloworld;
   public String SayHello() {
     return helloworld.SayHello("注入者");
     }
   @Resource(mappedName = "java:/DefaultMySqlDS")
    DataSource myDb;
   @EJB (beanName="HelloWorldBean")
   public void setHelloworld(HelloWorld helloworld) {
       this.helloworld = helloworld;
       Connection conn = myDb.getConnection();..... 数据库
       } }
```

批注 [U69]: @EJB 注释的 beanName 属性指定 EJB 的名称(如果没有设置过 @Stateless 或@Stateful 的 name 属性，默认为不带包名的类名)，他的另一个属性 mappedName 指定 EJB 的全局 JNDI 名。

批注 [U70]: @EJB 注释只能注入 EJB 存根对象，除@EJB 注释之外，EJB 3.0 也支持 **@Resource** 注释来注入来自 JNDI 的任何资源,比如数据库的资源，先配置连接池 DefaultMySqlDS 连接池的名字

批注 [U71]: @EJB 注释如果被用在 JavaBean 风格的 setter 方法上时，容器会在属性第一次使用之前，自动地用正确的参数调用 bean 的 setter 方法

8，定时服务(Timer Service)

定时服务用作在一段特定的时间后执行某段程序，估计各位在不同的场合中已经使用过。下面就直接介绍 EJB3.0

定时服务的开发过程。定时服务的开发过程与会话 Bean 的开发过程大致相同，但比会话 Bean 多了几个操作，那就是使用容器对象 SessionContext 创建定时器，并使用@Timeout 注释声明定时器方法。

```
@Stateless
@Remote (TimerService.class)
public class TimerServiceBean implements TimerService {
   private int count = 1;
   private @Resource SessionContext ctx;
   public void scheduleTimer(long milliseconds){
   count = 1;
   ctx.getTimerService().createTimer(new Date(new Date().getTime() +
   milliseconds),milliseconds, "大家好，这是我的第一个定时器");
   }
@Timeout
public void timeoutHandler(Timer timer) {
   System.out.println("----------------------");
   System.out.println("定时器事件发生,传进的参数为: " + timer.getInfo());
   System.out.println("----------------------");
   if (count>=5){
   timer.cancel();//如果定时器触发 5 次，便终止定时器
   }
   count++;
   }
}
```

下面是 TimerServiceBean 的 Remote 业务接口

TimerService.java

```
package com.foshanshop.ejb3;
public interface TimerService {
   public void scheduleTimer(long milliseconds);
}
```

通过依赖注入@Resource SessionContext ctx，我们获得 SessionContext 对象，调用 ctx.getTimerService().createTimer (Date arg0, long arg1, Serializable arg2)方法创建定时器，三个参数的含义如下：

Date arg0 定时器启动时间，如果传入时间小于现在时间，定时器会立刻启动。

long arg1 间隔多长时间后再次触发定时事件。单位：毫秒

Serializable arg2 你需要传给定时器的参数，该参数必须实现 Serializable 接口。

当定时器创建完成后，我们还需声明定时器方法。定时器方法的声明很简单，只需在方法上面加入@Timeout 注释，另外定时器方法必须遵守如下格式：

```
void XXX(Timer timer)
```

在定时事件发生时，此方法将被执行。

9, 安全服务(SECURITY SERVICE)

使用 Java 验证和授权服务（JAAS）可以很好地解决在应用程序变得越来越复杂时，安全需求也会变得很复杂的问题，你可以用它来管理应用程序的安全性。JAAS 具有两个特性：验证（Authentication）和授权（authorization），认证是完成用户名和密码的匹配校验；授权是决定用户可以访问哪些资源，授权是基于角色的

1）开发的第一步是定义安全域，安全域的定义有两种方法

第一种方法：通过 Jboss 发布文件(jboss.xml)进行定义

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss>
  <!-- Bug in EJB3 of JBoss 4.0.4 GA
   <security-domain>java:/jaas/other</security-domain>
   -->
    <security-domain>other</security-domain>
    <unauthenticated-principal>AnonymousUser</unauthenticated-principal>
    </jboss>
```

第二种方法：通过@SecurityDomain 注释进行定义，注释代码片断如下：

```
@Stateless
@Remote ({SecurityAccess.class})
@SecurityDomain("other")
public class SecurityAccessBean implements SecurityAccess{}
```

2）定义好安全域之后，因为我们使用 Jboss 默认的安全域”other”，所以必须使用 users.propertes 和 roles.properties 存储用户名/密码及用户角色。现在开发的第二步就是定义用户名，密码及用户的角色。用户名和密码定义在 users.propertes 文件，用户所属角色定义在 roles.properties 文件。以下是这两个文件的具体配置：

users.propertes（定义了本例使用的三个用户）

```
lihuoming=123456
zhangfeng=111111
wuxiao=123
```

roles.properties（定义了三个用户所具有的角色，其中用户 lihuoming 具有三种角色）

```
lihuoming=AdminUser,DepartmentUser,CooperateUser
zhangfeng=DepartmentUser
wuxiao=CooperateUser
```

批注 [U72]: jboss.xml(本例使用 Jboss 默认的安全域”other”)。 jboss.xml 必须打进 Jar 文件的 META-INF 目录。

批注 [U73]: 由于使用的是 Jboss 安全注释，程序采用了硬编码，不利于日后迁移到其他J2EE 服务器（如:WebLogic），所以作者不建议使用这种方法定义安全域

以上两个文件必须存放于类路径下。在进行用户验证时，Jboss 容器会自动寻找这两个文件。

3）开发的第三步就是为业务方法定义访问角色。本例定义了三个方法：AdminUserMethod(), DepartmentUserMethod(),AnonymousUserMethod()，第一个方法只允许具有 AdminUser 角色的用户访问，第二个方法只允许具有 DepartmentUser 角色的用户访问，第三个方法允许所有角色的用户访问。下面是 Session Bean 代码。

```
@Stateless
@Remote (SecurityAccess.class)
public class SecurityAccessBean implements SecurityAccess{
   @RolesAllowed({"AdminUser"})
   public String AdminUserMethod() {
      return "具有管理员角色的用户才可以访问 AdminUserMethod()方法";
   }
   @RolesAllowed({"DepartmentUser"})
   public String DepartmentUserMethod() {
      return "具有事业部门角色的用户才可以访问 DepartmentUserMethod()方法";
   }
   @PermitAll
   public String AnonymousUserMethod() {
     return "任何角色的用户都可以访问 AnonymousUserMethod()方法，注：用户必须存在
     users.properties 文件哦";
   }
}
```

批注 [U74]: @RolesAllowed 注释定义允许访问方法的角色列表，如角色为多个，可以用逗号分隔

批注 [U75]: 。@PermitAll 注释定义所有的角色都可以访问此方法。

自定义安全域

把用户名/密码及角色存放在 users.propertes 和 roles.properties 文件，不便于日后的管理。大多数情况下我们都希望把用户名/密码及角色存放在数据库中。为此，我们需要自定义安全域。他采用数据库存储用户名及角色（两个表）下面的例子定义了一个名为 foshanshop 的安全域，安全域在[jboss 安装目录]/server/default/conf/login-config.xml 文件中定义，本例配置片断如下：

```
<!-- ..................... foshanshop login configuration .....................-->
<application-policy name="foshanshop">
<authentication>
<login-module code="org.jboss.security.auth.spi.DatabaseServerLoginModule"
flag="required">
<module-option name="dsJndiName">java:/DefaultMySqlDS</module-option>
<module-option name="principalsQuery">
     select password from sys_user where name=?
</module-option>
<module-option name="rolesQuery">
     select rolename,'Roles' from sys_userrole where username=?
</module-option>
<module-option name = "unauthenticatedIdentity">AnonymousUser</module-option>
</login-module>
</authentication>
</application-policy>
```

上面使用了 Jboss 数据库登录模块(org.jboss.security.auth.spi.DatabaseServerLoginModule)，他的 dsJndiName 属性(数据源 JNDI 名）使用 DefaultMySqlDS 数据源(本教程自定义的数据源，关于数据源的配置请参考前面章节：JBoss 数据源的配置)，**principalsQuery** 属性定义 Jboss 通过给定的用户名如何获得密码，**rolesQuery** 属性定义 Jboss 通过给定的用户名如何获得角色列表，**注意:SQL 中的'Roles'常量字段不能去掉**。**unauthenticatedIdentity** 属性允许匿名用户(不提供用户名及密码)访问。“foshanshop”安全域使用的 sys_user 和 sys_userrole 表是自定义表，实际项目开发中你可以使用别的表名。sys_user 表必须含有用户名及密码两个字段，字段类型为字符型，至于字符长度视你的应用而定。sys_userrole 表必须含有用户名及角色两个字段，字段类型为字符型，字符长度也视你的应用而定。

批注 [U76]: JPA（java persistence Api）是对 Hibernate、TopLink(GlassFish 默认的持久化框架)等的一次再封装，屏蔽了这些持久化框架之间的差异。在创建实体的时候，会提示选择持久化框架(hibernate、toplink...)以及数据源，这里选择在控制台 . 中配置好了的 JNDI 管理着的数据源。

4,实体 Bean:实体是 JPA 中用来做持久化操作的 Java 类.从名字上我们就能猜到,实体 bean 代表真实物体的数据,在 JDBC+JavaBean 编程中，通常把 JDBC 查询的结果信息存入 JavaBean，然后供后面程序进行处理。在这里你可以把实体 Bean 看作是用来存放数据的 JavaBean。但比普通 JavaBean 多了一个功能，实体 bean 除了担负起存放数据的角色，还要负责跟数据库表进行对象与关系映射（O/R Mapping），通过标注或 xml 表明实体和表的关系，下图就说明了这一映射：

批注 [U77]: Javabean 的特点：是普通的 java，实现 Seialization 接口 2 提供没有参数的构造方法 3 属性为私有，且提供 set 和 get 方法 4 最好覆盖 equal 和 hashset 方法 5 利用的观察者模式。

1）ORMapping 的四项基本原则：类型-->表　　对象--->行　　关系--->外键　　属性----->列

2）Java Persistence API 定义了一种方法，可以将常规的普通 Java 对象（**有时被称作 POJO**）映射到数据库。这些普通 Java 对象被称作 entity bean。除了是用 Java Persistence 元数据将其映射到数据库外，entity bean 与其他 Java 类没有任何区别。事实上，创建一个 Entity Bean 对象相当于新建一条记录，删除一个 Entity Bean 会同时从数据库中删除对应记录，修改一个 Entity Bean 时，容器会自动将 Entity Bean 的状态和数据库同步。
Java Persistence API 还定义了一**种查询语言（JPQL）**，具有与 SQL 相类似的特征，只不过做了裁减，以便处理 Java 对象而非原始的关系 schema。

批注 [U78]: 实体 Bean 可分为 Bean 管理的持续性（BMP）和容器管理的持续性（CMP）两种

3）一个实体 Bean 应用由实体类和 persistence.xml 文件组成。persistence.xml 文件在 Jar 文件的 META-INF 目录。persistence.xml 文件指定实体 Bean 使用的数据源及 EntityManager 对象的默认行为。persistence.xml 文件的配置说明如下：

```
<persistence-unit name="StudentMgmtPU" transaction-type="JTA">
    <!-- 持久化框架的提供者 -->
  <provider>oracle.toplink.essentials.PersistenceProvider</provider>
  <!-- 已经在 JNDI 上配置好了的数据源 -->
  <jta-data-source>jdbc/mysql</jta-data-source>
  <mapping-file/> xml 的映射文件，而在 javaee 中一般用标注
  <properties>
    <!-- 在部署的时候创建表 在取消部署的时候删除表 -->
    <property name="toplink.ddl-generation" value="drop-and-create-tables"/>
  </properties>
 </persistence-unit>
</persistence>
```

若不采用 glassfish 默认的 ToopLink 作为持久化框架，采用 Hibernate + 另外配置的数据源
则此时的配置文件格式如下 ：

```
<persistence >
  <persistence-unit name="TestPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <class>test.NewEntity</class>
    <properties>
      <property name="hibernate.connection.username" value="zhangweifeng "/>
      <property name="hibernate.connection.driver_class" value="oracle.jdbc.driver.OracleDriver"/>
      <property name="hibernate.connection.password" value="123"/>
      <property name="hibernate.connection.url" value="jdbc:oracle:thin:@localhost:1521:XE"/>
      <property name="hibernate.cache.provider_class" value="org.hibernate.cache.NoCacheProvider"/>
    </properties>
  </persistence-unit>
</persistence>
```

4） hibernate 和 jpa 的相似之处。

Hibernate	jpa
Configuration	persistence
SessionFactory	EntityManagerFactory
Session	EntityManager
save ()	persist ()
update ()	merge ()
delete ()	remove ()
get () /load()	find()/getRefernce()
Transaction	EntityTransaction
配置文件：xxx.cfg.xml	persistence.xml

5)jpa 的常用标注例子

```
@Entity([name="Adm"])// @Entity 指明这是需要持久化的实体 若加上 name 属性，则以后在操作实体时 用 Adm 取代 Admin
@Table(name="ADMINS")// 指定数据库中与本实体对应的表 若去掉此标注 则默认与实体名一致
@EntityListeners(EntityLcLoggerListener.class) // 指定监听器 监听对实体的数据库操作
// 建立命名查询，以后根据 name 即可得到对应的 Query
// 如 : Query q = entityManager.createNamedQuery("findByName");
@NamedQueries({
   @NamedQuery(name="findByName", query="select a from Admin a where a.name=:name"),
   @NamedQuery(name="findAll", query="select a from Admin a")
})
public class Admin implements Serializable {
    private static final long serialVersionUID = 1L;
    // ID 的生成由 JPA 自动管理
    // 若持久化框架是 hibernate 数据库为 oracle 则自动生成名为"hibernate_sequence"的序列
    // 若持久化框架是 toplink 数据库为 oracle 则自动创建一个 ID 号生成表
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Basic(optional=false)              // 指定在内存中即对数据进行检查 是否满足条件 无需在数据库中才进行检查
    @Column(length=16,nullable=false,unique=true) // 指定字段属性以及约束
    private String name;

    @Basic(optional=false)
    @Column(length=8,nullable=false)
```

批注 [U79]: 相当于 hibernate 的配置文件

批注 [U80]: 注：实体bean 需要在网络上传送时必须实现 Serializable 接口，否则将引发 java.io.InvalidClassException 例外。

批注 [U81]: @javax.persistence.Column 注释定义了将成员属性映射到关系表中的哪一列和该列的一些结构信息(如列名是否唯一，是否允许为空，是否允许更新等)，他的属性介绍如下：

- name: 映射的列名。如：映射Person 表的PersonName 列，可以在name 属性的 getName 方法上面加入 @Column(name = "PersonName")，如果不指定映射列名，容器将属性名称作为默认的映射列名。
- unique: 是否唯一
- nullable: 是否允许为空
- length: 对于字符型列，length 属性指定列的最大字符长度
- insertable: 是否允许插入
- updatable: 是否允许更新
- columnDefinition: 定义建表时创建此列的DDL
- secondaryTable: 从表名。如果此列不建在主表上(默认建在主表)，该属性定义该列所在从表的名字。

```
    private String password;

    @Column(length=64)
    private String address;

    @Temporal(javax.persistence.TemporalType.DATE) // 由于这里是 util 包下面的 Date 类型 故要变换一下
    @Column(name="EXPIRE_DATE")// 指定对应的字段名
    private Date expireDate;

    // 乐观锁时的 version 字段
    @Version
    private int version;

    public Admin() {
    }
    public Admin(String name, String password, String address) {
        this.name = name;
        this.password = password;
        this.address = address;
    }
    // GET and SET 方法
      ...
}
```

注意：@Id 注释指定 id 属性为表的主键，它可以有多种生成方式：

- TABLE：容器指定用底层的数据表确保唯一。例子代码如下：@TableGenerator(name="Person_GENERATOR",//为该生成方式取个名称
  table="Person_IDGenerator",//生成 ID 的表
  pkColumnName="PRIMARY_KEY_COLUMN",//主键列的名称
  valueColumnName="VALUE_COLUMN",//存放生成 ID 值的列的名称
  pkColumnValue="personid",//主键列的值(定位某条记录)
  allocationSize=1)//递增值

```
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="Person_GENERATOR")
private Integer id;
```

- SEQUENCE：使用数据库的 SEQUENCE 列来保证唯一(Oralce 数据库通过序列来生成唯一 ID)，例子代码如下：

```
@SequenceGenerator(name="Person_SEQUENCE",    //为该生成方式取个名称
sequenceName="Person_SEQ")                    //sequence 的名称(如果不存在，会自动生成)
public class Person implements Serializable{
@Id
@GeneratedValue(strategy=GenerationType.SEQUENCE, generator="Person_SEQ")
private Integer id;
```

- IDENTITY：使用数据库的 INDENTIT 列来保证唯一(像 mysql，sqlserver 数据库通过自增长来生成唯一 ID)
- **AUTO：** 由容器挑选一个合适的方式来保证唯一(由容器决定采用何种方式生成唯一主键，hibernate 会根据数据库类型选择适合的生成方式，相反 toplink 就不是很近人情)
- NONE：容器不负责主键的生成，由调用程序来完成。

@GeneratedValue 注释定义了标识字段的生成方式,本例 personid 的值由 MySQL 数据库自动生成。

注意：其他属性标签

1 如果不想让一些成员属性映射成数据库字段，我们可以使用@Transient 注释进行标注，属性将不会被持久化成数据库字段

2，@Lob 注释用作映射这些大数据类型，当属性的类型为 byte[]，Byte[]或 java.io.Serializable 时，@Lob 注释将映射为数据库的 Blob 类型，当属性的类型为 char[]，Character[]或 java.lang.String 时，@Lob 注释将映射为数据库的 Clob 类型

3，属性进行延时加载，这时我们需要用到@Basic 注释

Dao 层的编码。操作 Admin 的 EJB ：

```
@Stateless(mappedName="ejb/adminDao")
public class AdminDaoBean implements AdminDaoRemote, AdminDaoLocal {
      // 注入 EntityManager 对象(相当于 hibernate 中的 Session 对象)
      // 注意这里不是用@Resource 注入
      /**
       * 在测试的时候，可以这么获得 EntityManager 对象 :
       *  public static void main(String[] args) {
       *          // "TestPU"即为配置文件中<persistence-unit/>标签 name 属性对应的值
       *      EntityManagerFactory emf = Persistence.createEntityManagerFactory("TestPU");
       *      EntityManager em = emf.createEntityManager();
       *      EntityTransaction et = em.getTransaction();
       *      et.begin();
```

批注 [U82]: 通过配置文件的名字 TestPU 获得 EntityManagerFactory

批注 [U83]: 开始事务

```
     *       NewEntity s = new NewEntity("123", "11", 18, "111");
     *       em.persist(s);
     *       System.out.println("Id: " + s.getId());
     *       et.commit();
     *          ...
     *  }
     */
    @PersistenceContext
    private EntityManager em;

    public Integer save(Admin admin) {
        em.persist(admin);
        return admin.getId();
    }

    public Admin update(Admin admin) {
        return em.merge(admin);
    }

    public Admin get(Integer id) {
        return em.find(Admin.class, id);
    }

    public void delete(Integer id) {
        Admin admin = get(id);
        em.remove(admin);
    }

    public Admin getByName(String name) {
        Query q = em.createNamedQuery("findByName");
        // a.name=:name ---> a.name=?
        // Query q = em.createQuery("select a from Admin a where a.name=:name");
        q.setParameter("name", name);
        return (Admin)q.getSingleResult();
    }

    public java.util.Set<Admin> getAll() {
        Query q = em.createNamedQuery("findAll");
        // Query q = em.createQuery("select a from Admin a");
        return new HashSet<Admin>(q.getResultList());
    }
}
```

批注 [U84]: EntityManager 是由EJB 容器自动地管理和配置的，不需要用户自己创建，他用作操作实体Bean。它可以持久化环境，相当于hibernate的一级case。
如果persistence.xml 文件中配置了多个不同的持久化内容。在注入EntityManager 对象时必须指定持久化名称，可以通过@PersistenceContext 注释的unitName 属性进行指定，例：
@PersistenceContext(unitName=" TestPU ")
EntityManager em;

批注 [U85]: 在实体中已经标注，直接调用名字就可以，可以从用

注意：持久化实体管理器 EntityManager

1, find()或 getReference(),Person person = em.find(Person.class,id); Person person = em.getReference (Person.class, 1);
当在数据库中没有找到记录时，getReference()和 find()是有区别的，find()方法会返回 null，而 getReference()方法会抛出 javax.persistence.EntityNotFoundException 例外，另外 getReference()方法不保证实体 Bean 已被初始化。
如果传递进 getReference()或 find()方法的参数不是实体 Bean，都会引发 IllegalArgumentException 例外

2, 添加 persist(),em.persist(person);

3, EntityManager.flush()更新实体，当实体正在被容器管理时,你可以调用实体的 set 方法对数据进行修改,在容器决定 flush 时，更新的数据才会同步到数据库。如果你希望修改后的数据实时同步到数据库，你可以 flush()。将实体的改变立刻刷新到数据库中

4, 合并 Merge(),merge ()方法是在实体 Bean 已经脱离了 EntityManager 的管理时使用，当容器决定 flush 时，数据将会同步到数据库中，

5, 删除 Remove()

6, 执行 JPQL 操作 createQuery(),通过 EntityManager 的 createQuery()或 createNamedQuery()方法创建一个 Query 对象
(Query query = em.createQuery("select p from Person p where p. name=' 黎明' "); List result = query.getResultList();)

7, 执行 SQL 操作 createNativeQuery(),注意这里操作的是 SQL 语句，并非 JPQL，千万别搞晕了。
Query query = em.createNativeQuery("select * from person", Person.class); List result = query.getResultList();

8, **刷新实体 refresh()**,通过 refresh()方法刷新实体，**容器会把数据库中的新值重写进实体**。这种情况一般发生在你获取了实体之后，有人更新了数据库中的记录，这时你需要得到最新的数据

7）事务管理服务

1>.EJB 声明式事务(也叫容器管理事务)
由 EJB 容器来负责管理事务,EJB 的 Bean 类不需要直接编写事务的代码,通过标注或配置文件来声明事务特性获得容器提供的事务。有两种方式：①一般性的事务管理（只管理一个事务，事务结束后就关闭，一个事物范围内）
②可扩展的事务管理（可以管理多个事务，但是只要一个事物出错，就可能导致与数据库的数据不同步，只能是有状态的 bean 使用，一般很少用）

```
@PersistenceContext
protected EntityManager em;
@TransactionAttribute(TransactionAttributeType.REQUIRED)
```

批注 [U86]: 无状态的 Bean，一般事务管理。
如果没有指定参数，@TransactionAttribute 注释使用 REQUIRED 作为默认参数。

```
public void ModifyProductName(String newname, boolean error) throws Exception {
   Query query = em.createQuery("select p from Product p");
   List result = query.getResultList();
   if (result!=null){
   for(int i=0;i<result.size();i++){
   Product product = (Product)result.get(i);
   product.setName(newname+ i);
   em.merge(product);
}
if (error && result.size()>0) throw new TransException ("抛出应用例外");
}
}
}
```

事务的传播行为

EJB 的 Bean 对待客户端事务的行为
共分为六种:

1. REQUIRED：方法在一个事务中执行，如果调用的方法已经在一个事务中，则使用该事务，否则将创建一个新的事务。
2. MANDATORY：如果运行于事务中的客户调用了该方法，方法在客户的事务中执行。如果客户没有关联到事务中，容器就会抛出 TransactionRequiredException。如果企业 bean 方法必须用客户事务则采用 Mandatory 属性。
3. REQUIRESNEW:方法将在一个新的事务中执行，如果调用的方法已经在一个事务中，则暂停旧的事务。在调用结束后恢复旧的事务。
4. SUPPORTS:如果方法在一个事务中被调用，则使用该事务，否则不使用事务。
5. NOT_SUPPORTED：如果方法在一个事务中被调用，容器会在调用之前中止该事务。在调用结束后，容器会恢复客户事务。如果客户没有关联到一个事务中，容器不会在运行入该方法前启动一个新的事务。用 NotSupported 属性标识不需要事务的方法。因为事务会带来更高的性能支出，所以这个属性可以提高性能。
6. Never：如果在一个事务中调用该方法，容器会抛出 RemoteException。如果客户没有关联到一个事务中，容器不会在运行入该方法前启动一个新的事务。

标注
@TransactionManagement
用在类之前,用来告诉容器现在使用 BMT/CMT
*TransactionManagementType.BEAN
*TransactionManagementType.CONTAINER(默认)
@TransactionAttribute
用在方法之前,用来声明事务传播行为

8）Entity 的生命周期和状态

在 EJB3 中定义了四种 Entity 的状态:
1. 新实体(new)。Entity 由应用产生，和 EJB3 Persistence 运行环境没有联系，也没有唯一的标示符(Identity)。
2. 持久化实体(managed)。新实体和 EJB3 Persistence 运行环境产生关联（通过 persist(), merge()等方法），在 EJB3 Persistence 运行环境中存在和被管理，标志是在 EJB3 Persistence 运行环境中有一个唯一的标示(Identity)。
3. 分离的实体(detached)。Entity 有唯一标示符，但它的标示符不被 EJB3 Persistence 运行环境管理，同样的该 Entity 也不被 EJB3 Persistence 运行环境管理。
4. 删除的实体(removed)。Entity 被 remove()方法删除，对应的纪录将会在当前事务提交的时候从数据库中删除

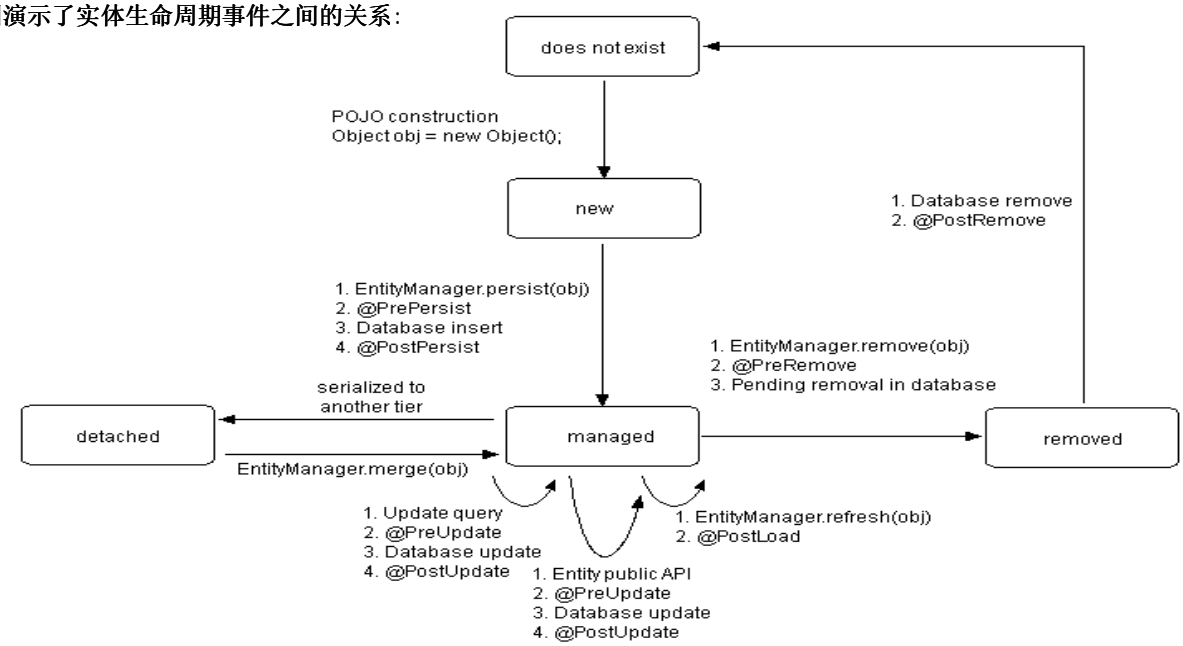
持久化规范允许你在实体类中实现回调方法，当这些事件发生时将会通知你的实体对象。当然你也可以使用一个外部类去拦截这些事件，这个外部类称作实体监听者。通过@EntityListeners 注释绑定到实体 Bean。

批注 [U87]: 见 5)jpa 的常用标注例子

生命周期回调事件

如果需要在生命周期事件期间执行自定义逻辑，请使用以下生命周期事件注释关联生命周期事件与回调方法，EJB 3.0 允许你将任何方法指定为回调方法。这些方法将会被容器在实体生命周期的不同阶段调用。

下图演示了实体生命周期事件之间的关系：



@PostLoad 事件在下列情况触发

1. 执行 EntityManager.find( )或 getreference( )方法载入一个实体后
2. 执行 JPQL 查询过后
3. EntityManager.refresh( )方法被调用后

@PrePersist 和@PostPersist 事件在实体对象插入到数据库的过程中发生，@PrePersist 事件在调用 EntityManager.persist( )方法后立刻发生，级联保存也会发生此事件，此时的数据还没有真实插入进数据库。

@PostPersist 事件在数据已经插入进数据库后发生。

@PreUpdate 和@PostUpdate 事件的触发由更新实体引起，@PreUpdate 事件在实体的状态同步到数据库之前触发，此时的数据还没有真实更新到数据库。@PostUpdate 事件在实体的状态同步到数据库后触发，同步在事务提交时发生。

@PreRemove 和@PostRemove 事件的触发由删除实体引起，@PreRemove 事件在实体从数据库删除之前触发，即调用了 EntityManager.remove（）方法或者级联删除时发生，此时的数据还没有真实从数据库中删除。

@PostRemove 事件在实体已经从数据库中删除后触发。

批注 [U88]: 作者对@PostPersist 和@PostUpdate 事件触发的时机有点怀疑，文档上说@PostPersist 事件在数据真实插入进数据库后发生，但作者测试的结果是：在数据还没有真实插入进数据库时，此事件就触发了。@PostUpdate 事件的情况一样

6.12.2 在外部类中实现回调---------最好覆盖 toString（）方法

Entity listeners（实体监听者）用作拦截实体回调事件，通过@javax.persistence.EntityListeners 注释可以把他们绑定到一个实体类。在 Entity listeners 类里，你可以指定一个方法拦截实体的某个事件，所指定的方法必须带有一个 Object 参数及返回值为 void，格式如下：void <MethodName>(Object)

监听器 ： 一般可以用作日志管理，对 Entity 生命周期回调。

```
public class EntityLcLoggerListener {
    private static Logger logger = Logger.getLogger("EnityLcLoggerListener");
    @PrePersist
    public void persist(Object e) {
        logger.info("before saving " + e);
    }
    @PostPersist
    public void persisted(Object e) {
        logger.info("after saving " + e);
    }
    @PreUpdate
```

```
    public void update(Object e) {
        logger.info("before updating " + e);
    }

    @PostUpdate
    public void updateed(Object e) {
        logger.info("after updating " + e);
    }
    @PreRemove
    public void remove(Object e) {
        logger.info("before removing " + e);
    }
    @PostRemove
    public void removed(Object e) {
        logger.info("after removing " + e);
    }
    @PostLoad
    public void loaded(Object e) {
        logger.info("after loading " + e);
    }
```

6.12.3 在 Entity 类中实现回调

除了外部类可以实现生命周期事件的回调，你也可以把回调方法写在 Entity 类中。要注意：直接写在 Entity 类中的**回调方法不需带任何参数**，格式如下：void <MethodName>()

EntityLifecycle.java 程序片断

```
@Entity
@Table(name = "EntityLifecycle")
public class EntityLifecycle implements Serializable{
@PostLoad
public void postLoad() {
System.out.println("载入了实体 Bean{" + this.getClass().getName( ) + "}");}
```

9）JPA 的高级部分

9.1 多态，继承

因为关系数据库的表之间不存在继承关系，Entity 提供三种基本的继承映射策略:

每个类分层结构一张表（一张大表）(table per class hierarchy)

每个子类一张表(table per subclass)

每个具体类一张表(table per concrete class)

9.1.1 每个类分层结构一张表（一张大表）(table per class hierarchy) 需要一个字段来区分子类

```
@Entity
    1 单表对应整个类层次结构 也就是一张超级大表的形式 有继承关系 查询效率比较高
        但是属性多的话 表会变得很宽 同时子类型字段不容许为空 */
    @Table(name="PAYMENTS")// 指定对应的表名
    @Inheritance(strategy=InheritanceType.SINGLE_TABLE) //指定形式为一张大表
      由于是一张大表 name 属性指定实体类型字段名  discriminatorType 属性指定此字段值类型
    @DiscriminatorColumn(name="PAYMENT_TYPE",discriminatorType=DiscriminatorType.INTEGER)

    2，在子类中的实体 bean 中标注@DiscriminatorValue (“car”)
```

该策略的优点：

SINGLE_TABLE 映射策略在所有继承策略中是最简单的，同时也是执行效率最高的。他仅需对一个表进行管理及操作，持久化引擎在载入 entiry 或多态连接时不需要进行任何的关联，联合或子查询，因为所有数据都存储在一个表。

该策略的缺点：

这种策略最大的一个缺点是需要对关系数据模型进行非常规设计，在数据库表中加入额外的区分各个子类的字段，此外，不能为所有子类的属性对应的字段定义 not null 约束，此策略的关系数据模型完全不支持对象的继承关系。

选择原则：查询性能要求高，子类属性不是非常多时，优先选择该策略。

9.1.2 每个子类一张表(table per subclass)

这种映射方式为每个类创建一个表。在每个类对应的表中只需包含和这个类本身的属性对应的字段，子类对应的表参照父类对应的表。

1，**父类的标注 ：**

```
@Entity
@Inheritance(strategy=InheritanceType.JOINED)
@Table(name="Vehicle")
```

2，子类中：

```
@Entity
@Table(name="Car")
@PrimaryKeyJoinColumn(name="CarID")//把主键对应的列名更改为 CarID
```

批注 [U89]: 类似 hibernate 的映射

批注 [U90]: 需要把@javax.persistence.Inheritance 注释的strategy属性设置为InheritanceType.SINGLE_TABLE。除非你要改变子类的映射策略，否则@Inheritance 注释只能放在继承层次的基类。通过鉴别字段的值，持久化引擎可以区分出各个类，并且知道每个类对应那些字段。鉴别字段通过@javax.persistence.DiscriminatorColumn 注释进行定义，name 属性定义鉴别字段的列名，discriminatorType 属性定义鉴别字段的类型（可选值有：String, Char, Integer），如果鉴别字段的类型为String或Char，可以用length 属性定义其长度。@DiscriminatorValue 注释为继承关系中的每个类定义鉴别值，如果不指定鉴别值，默认采用类名。

批注 [U91]: 自己定义的区分列，不写的话默认是 Dtype STRING 类型

批注 [U92]: 子类没有对应的表，故不要 id，子类的区别字段就是“car”

该策略的优点：
这种映射方式支持多态关联和多态查询，而且符合关系数据模型的常规设计规则。在这种策略中你可以对子类的属性对应的字段定义 not null 约束。
该策略的缺点：
它的查询性能不如上面介绍的映射策略。在这种映射策略下，必须通过表的内连接或左外连接来实现多态查询和多态关联。
选择原则：子类属性非常多，需要对子类某些属性对应的字段进行 not null 约束，且对性能要求不是很严格时，优先选择该策略。

9.1.3 每个具体类一张表(table per concrete class)

这种映射方式为每个类创建一个表。在每个类对应的表中包含和这个类所有属性（包括从超类继承的属性）对应的字段。要使用每个具体类一张表(table per concrete class)策略，需要把@javax.persistence.Inheritance 注释的 strategy 属性
设置为 InheritanceType.TABLE_PER_CLASS。

```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
@Table(name="Vehicle")
```

注意：一旦使用这种策略就意味着你不能使用 AUTO generator 和 IDENTITY generator，即主键值不能采用数据库自动生成.
该策略的优点：
在这种策略中你可以对子类的属性对应的字段定义 not null 约束。
该策略的缺点：
不符合关系数据模型的常规设计规则，每个表中都存在属于基类的多余的字段。同时，为了支持策略的映射，持久化管理者需要决定使用什么方法，一种方法是在 entity 载入或多态关联时，容器使用多次查询去实现，这种方法需要对数据库做几次来往查询，非常影响执行效率。另一种方法是容器通过使用 SQLUNIOU 查询来实现这种策略。
选择原则：除非你的现实情况必须使用这种策略，一般情况下不要选择。

9.2 JTA 对实体对应关系的处理 ：

one --- one ：

Order 类 ：

```
@Entity(name="TOrder")
@Table(name="ORDERS")
public class Order implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;

    // 指定关系属性的对应关系 以及级联策略
    @OneToOne(cascade={CascadeType.ALL})
    @JoinColumn(name="SID")
    private Shipment shipment;

    public Order() {
    }
    public Order(String name) {
        this.name = name;
    }
      // GET and SET
      ...
}
```

Shipment 类 ：

```
@Entity
@Table(name="SHIPMENTS")
public class Shipment implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String city;
    private String zipcode;

    @OneToOne(mappedBy="shipment")
```

批注 [U93]: 关系所有者拥有外键 ID 由它维护关系
倘若去掉这一标记 则默认为 ：
@JoinColumn(name="shipment_id")

批注 [U94]: 一对一关联可能是双向的，在双向关联中，有且仅有一端是作为主体(owner)端存在的.
主体端负责维护联接列(即更新). 对于不需要维护这种关系的从表则通过 mappedBy 属性进行声明 mappedBy 的值指向主体的关联属性.

```
    private Order order;

    public Shipment() {
    }

    public Shipment(String city, String zipcode) {
        this.city = city;
        this.zipcode = zipcode;
    }
// GET and SET
     ...
}
```

League (1) --- (n) Team --- (n) Player

```
League :
@Entity
@Table(name="LEAGUES")
public class League implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(length=7)
    private Long id;
    @Basic(optional=false)
    @Column(length=64,unique=true,nullable=false)
    private String name;
    @Basic(optional=false)
    @Column(length=32,nullable=false)
    private String sport;
    // 若加上下面代码 则为双向关联
    // mappedBy="league"表明这一端非主体 无外键 ID 不维护关系 主体端对应关系属性名为"league"
    // fetch=FetchType.EAGER 指定采用热切查询 即将关系对象一并查询出来
    // 在默认的情况下 若关系对象为 Set 集合的话 采用的是懒加载 若非集合 则为非懒加载
    // @OneToMany(mappedBy="league", fetch=FetchType.EAGER, cascade={...})
    // private Set<Team> teams = new HashSet<Team>();
    public League() { }
    public League(String name, String sport) { this.name = name; this.sport = sport;
    }// GET and SET
      ...
}
Team :
@Entity
@Table(name="TEAMS")
public class Team implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(length=7)
    private Long id;
    @Basic(optional=false)
    @Column(length=64,unique=true,nullable=false)
    private String name;
    @Column(length=64)
    private String city;
    // 主体方 拥有外键 ID 维护关系  省略的标记为 : @JoinColumn(name="league_id")
    @ManyToOne(cascade={CascadeType.MERGE,CascadeType.PERSIST, CascadeType.REFRESH})
    private League league;
    // 若加上下面代码 则为双向关联
    // mappedBy="teams"表明这一端非主体 无外键 ID 不维护关系 主体端对应关系属性名为"teams"
    // fetch=FetchType.EAGER 指定采用热切查询 即将关系对象一并查询出来
    // @ManyToMany(mappedBy="teams", fetch=FetchType.EAGER, cascade={...})
    // private Set<Player> players = new HashSet<Player>();
    public Team() {
    }
    public Team(String name, String city) {
        this.name = name;
        this.city = city;
    }// GET and SET
         ...
}
```

```
Player :
@Entity
@Table(name="PLAYERS")
public class Player implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(length=7)
    private Long id;
    @Basic(optional=false)
    @Column(length=32, nullable=false)
    private String name;
    @Column(length=16)
    private String position;
    @Column(precision=12,scale=2) // 指定精度
    private BigDecimal salary;

    // 主体方 由于拥有的关系属性是一个集合 故不可能把外键 ID 建在本表中
    // 此时 就必须建中间表了
    @ManyToMany(fetch=FetchType.EAGER,cascade={CascadeType.MERGE,CascadeType.PERSIST, CascadeType.REFRESH})
    // 配置中间表
    @JoinTable(
      name="P_T_MAP",
        joinColumns=@JoinColumn(name="PID"),
        inverseJoinColumns=@JoinColumn(name="TID")
    )
    private Set<Team> teams = new HashSet<Team>();
    public Player() {}
    public Player(String name, String position, BigDecimal salary) {
        this.name = name;
        this.position = position;
        this.salary = salary;
    }// GET and SET
      ...
    public void joinTeam(Team team){ teams.add(team); }
    public void leaveTeam(Team team){ teams.remove(team);  }
}
```

8 整合了 struts1 的一个企业级应用 ：

(1) EJB 应用部分 ： 这里的 MVC 模式中 负责 Dao 操作的 EJB 应用部分显然是 M 一层

[1] 三个实体 League Team Player 以及他们之间的关系，在 JPA 中已说明

[2] 负责处理各个实体的 Dao

由于一些方法是每个 Dao 都得具备的 所以 做成模版接口 ：

```
public interface GenericDao<E, ID extends Serializable> {
    E save(E e);
    E get(ID id);
    E update(E e);
    void delete(ID id);
    Set<E> getAll();
}
```

模版接口的实现 ：

```
public abstract class GenericDaoJPAImpl<E, ID extends Serializable>implements GenericDao<E, ID>{
    abstract protected EntityManager getEntityManager();
    private Class<E> entityClass;
    private String entityName;
    public GenericDaoJPAImpl(){
      entityClass=
        (Class<E>)((ParameterizedType)getClass().getGenericSuperclass()).getActualTypeArguments()[0];
        Entity e = entityClass.getAnnotation(Entity.class);
        if(e != null){ entityName = e.name();}
        if(entityName == null){  entityName = entityClass.getSimpleName();}
    }
    public void setEntityName(String entityName){ this.entityName = entityName; }
    public E save(E e) { getEntityManager().persist(e); return e;
    }
    public E get(ID id) { E e = getEntityManager().find(entityClass, id);  return e;   }
    public E update(E e) { return getEntityManager().merge(e); }
```

批注 [U95]: 由于这里不是 EJB 故不能采用如此注入的方式获得 EntityManager 对象 @PersistenceContext private EntityManager em; 于是 这里采用了抽象方法的方式 由子类(EJB)实现抽象方法 注入获取 EntityManager 对象

```
    public void delete(ID id) {  E e = get(id); getEntityManager().remove(e); }
    public Set<E> getAll(){
        Query q = getEntityManager().createQuery("select e from " + entityName + " e");
        return new HashSet<E>(q.getResultList());
    }
}
```

由于 EJBDao 不会直接暴露给 C 层调用 而是通过一个"门面"把这些 EJBDao 聚合起来 C 层通过这个门面来调用 Dao 中的方法 也是因为这样 EJBDao 只需要提供本地接口(供"门面"本地调用) 无需提供远程接口

LeagueDao 本地接口 :

```
@Local
public interface LeagueDao extends GenericDao<League, Long>{
}
```

LeagueDao 实现 :

```
package net.kettas.roster.dao;
@Stateless
public class LeagueDaoBean extends GenericDaoJPAImpl<League, Long> implements LeagueDao{
    @PersistenceContext   // 注入 EntityManager 对象
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() { return em; }
}
```

TeamDao 本地接口 :

```
@Local
public interface TeamDao extends GenericDao<Team, Long>{
    Set<Team> getByLeague(Integer leagueId);
}
```

TeamDao 实现 :

```
@Stateless
public class TeamDaoBean extends GenericDaoJPAImpl<Team, Long> implements TeamDao{
    @PersistenceContext
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() { return em;
    }
    public Set<Team> getByLeague(Integer leagueId) {
       Query q = em.createQuery("select t from Team t where t.league.id=:leagueId");
       return new HashSet<Team>(q.getResultList());
    }
}
```

PlayDao 本地接口 :

```
@Local
public interface PlayerDao extends GenericDao<Player, Long> {
    Set<Player> getPlayersOfTeam(Integer teamId);
    Set<Player> getPlayersByPosition(String position);
    Set<Player> getPlayersHigherSalary(String name);
    Set<Player> getPlayersBySalaryRange(BigDecimal high, BigDecimal low);
    Set<Player> getPlayersBySport(String sport);
    Set<Player> getPlayersByCity(String city);
    Set<Player> getPlayersNotOnTeam();
    Set<String> getSportsOfPlayer(Integer playerId);
}
```

PlayDao 实现 :

```
@Stateless
public class PlayerDaoBean extends GenericDaoJPAImpl<Player, Long> implements PlayerDao {
    @PersistenceContext
    private EntityManager em;
    @Override
    protected EntityManager getEntityManager() {
        return em;
    }

    public Set<Player> getPlayersOfTeam(Integer teamId) {
        Query q = em.createQuery("select p from Player p join p.teams t where t.id=:teamId");
        q.setParameter("teamId", teamId);
```

```
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersByPosition(String position) {
        Query q = em.createQuery("select p from Player p where p.position=:position");
        q.setParameter("position", position);
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersHigherSalary(String name) {
  Query q = em.createQuery("select p from Player p, Player p1 where p.salary > p1.salary and p1.name=:name");
        q.setParameter("name", name);
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersBySalaryRange(BigDecimal high, BigDecimal low) {
        Query q = em.createQuery("select p from Player p where p.salary between :high and :low");
        q.setParameter("high", high).setParameter("low", low);
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersBySport(String sport) {
        Query q = em.createQuery("select p from Player p join p.teams t where t.league.sport=:sport");
        q.setParameter("sport", sport);
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersByCity(String city) {
        Query q = em.createQuery("select p from Player p join p.teams t where t.city=:city");
        q.setParameter("city", city);
        return new HashSet<Player>(q.getResultList());
    }

    public Set<Player> getPlayersNotOnTeam() {
        Query q = em.createQuery("select p from Player p where p.teams is empty");
        return new HashSet<Player>(q.getResultList());
    }

    public Set<String> getSportsOfPlayer(Integer playerId) {
  Query q = em.createQuery("select l.sport from Player p join p.teams t join t.league l where p.id=:playerId");
        q.setParameter("playerId", playerId);
        return new HashSet<String>(q.getResultList());
    }
}
```

批注 [U96]: 设置参数

[3] 负责把这些 EJBDao 聚合到一起供 C 层(Struts)调用的"门面", 显然门面应该同时实现
本地和远程接口 :

本地接口 :

```
package net.kettas.roster;
@Local
public interface RosterLocal {
    League createLeague(League league);
    Team createTeam(Long leagueId, Team team);
    Player createPlayer(Player player);
    void joinTeam(Long playerId, Long teamId);
    void leaveTeam(Long playerId, Long teamId);
    Set<Player> getPlayersBySport(String sport);
}
```

远程接口略

"门面"的实现 :

```
@Stateless(mappedName = "ejb/roster")
public class RosterBean implements RosterRemote, RosterLocal {
      // 注入所有 EJBDao
    @EJB(beanName = "LeagueDaoBean")
    private LeagueDao leagueDao;
```

```
    @EJB(beanName = "TeamDaoBean")
    private TeamDao teamDao;
    @EJB(beanName = "PlayerDaoBean")
    private PlayerDao playerDao;

    public League createLeague(League league) {
        League l = leagueDao.save(league);
        return l;
    }
    public Team createTeam(Long leagueId, Team team) {
        Team t = teamDao.save(team);
        League l = leagueDao.get(leagueId);
        t.setLeague(l);
        return t;
    }
    public Player createPlayer(Player player) {
        Player p = playerDao.save(player);
        return p;
    }
    public void joinTeam(Long playerId, Long teamId) {
        Player p = playerDao.get(playerId);
        Team t = teamDao.get(teamId);
        p.joinTeam(t);
    }
    public void leaveTeam(Long playerId, Long teamId) {
        Player p = playerDao.get(playerId);
        Team t = teamDao.get(teamId);
        p.leaveTeam(t);
    }

    public Set<Player> getPlayersBySport(String sport) {
        Set<Player> players = playerDao.getPlayersBySport(sport);
        Set<Player> ps = new HashSet<Player>();
        for(Player p : players){
            Player ptmp = new Player(p.getName(),p.getPosition(),p.getSalary());
            ptmp.setId(p.getId());
            for(Team t : p.getTeams()){
                Team ttmp = new Team(t.getName(),t.getCity());
                ttmp.setId(t.getId());
                League l = new League(t.getLeague().getName(), t.getLeague().getSport());
                l.setId(t.getLeague().getId());
                ttmp.setLeague(l);
                ptmp.joinTeam(ttmp);
            }
            ps.add(ptmp);
        }
        return ps;
    }

    // 其他方法
    ...
}
```

(2) Web 应用部分(将 EJB 应用部分导入其中 就像导 JAR 包一般)

[1] 由于 Action 中不能通过@EJB(beanName="")的方式注入 EJB 应用中的"门面"，故这里必须提供**一些工具类来获取"门面"**，实现间接访问 EJBDao

此类定义一个 Map，在查找 JNDI 上的对象时，先看看 Map 中是否已经含有此对象，若有则直接返回无需重新去 JNDI 上查找，节省时间和资源。若无，则先将查找出来的对象放入 Map 中， 然后返回

// 了解此工具类的作用

```
public class CachingServiceLocator {
    private InitialContext ic;
    private Map cache;
    private static CachingServiceLocator me;

    static {
        try {
            me = new CachingServiceLocator();
        } catch(NamingException se) {
            throw new RuntimeException(se);
```

```
    }
}

private CachingServiceLocator() throws NamingException  {
    ic = new InitialContext();
    cache = Collections.synchronizedMap(new HashMap());
}
public static CachingServiceLocator getInstance() {  return me; }
private Object lookup(String jndiName) throws NamingException {
    Object cachedObj = cache.get(jndiName);
    if (cachedObj == null) {
        cachedObj = ic.lookup(jndiName);
        cache.put(jndiName, cachedObj);
    }
    return cachedObj;
}
/**
 * will get the ejb Local home factory. If this ejb home factory has already been
 * clients need to cast to the type of EJBHome they desire
 *
 * @return the EJB Home corresponding to the homeName
 */
public EJBLocalHome getLocalHome(String jndiHomeName) throws NamingException {
    return (EJBLocalHome) lookup(jndiHomeName);
}

/**
 * will get the ejb Remote home factory. If this ejb home factory has already been
 * clients need to cast to the type of EJBHome they desire
 * @return the EJB Home corresponding to the homeName
 */
public EJBHome getRemoteHome(String jndiHomeName, Class className) throws NamingException {
    Object objref = lookup(jndiHomeName);
    return (EJBHome) PortableRemoteObject.narrow(objref, className);
}

/**
 * This method helps in obtaining the topic factory
 * @return the factory for the factory to get topic connections from
 */
public ConnectionFactory getConnectionFactory(String connFactoryName) throws NamingException {
    return (ConnectionFactory) lookup(connFactoryName);
}

/**
 * This method obtains the topc itself for a caller
 * @return the Topic Destination to send messages to
 */
public Destination getDestination(String destName) throws NamingException {
    return (Destination) lookup(destName);
}

/**
 * This method obtains the datasource
 * @return the DataSource corresponding to the name parameter
 */
public DataSource getDataSource(String dataSourceName) throws NamingException {
    return (DataSource) lookup(dataSourceName);
}

/**
 * This method obtains the mail session
 * @return the Session corresponding to the name parameter
 */
public Session getSession(String sessionName) throws NamingException {
    return (Session) lookup(sessionName);
}
/**
 * @return the URL value corresponding
```

```
     * to the env entry name.
     */
    public URL getUrl(String envName) throws NamingException {
        return (URL) lookup(envName);
    }

    /**
     * @return the boolean value corresponding
     * to the env entry such as SEND_CONFIRMATION_MAIL property.
     */
    public boolean getBoolean(String envName) throws NamingException {
        Boolean bool = (Boolean) lookup(envName);
        return bool.booleanValue();
    }

    /**
     * @return the String value corresponding
     * to the env entry name.
     */
    public String getString(String envName) throws NamingException {
        return (String) lookup(envName);
    }

    public Object getEjb3BI(String jndiName) throws NamingException{
        return lookup(jndiName);
    }
}
```

异常类 :

```
package net.kettas.roster;
import javax.naming.NamingException;
class RosterException extends RuntimeException{
    public RosterException(NamingException ex) {
        super(ex);
    }
}
```

通过上面的工具类，查找指定 jndi 名的 EJB :

```
public class RosterUtils {
      // 这里的 JNDI 引用名是是在 web.xml 中配置的
    public static final String ROSTER_LOCAL = "java:comp/env/ejb/rosterLocal";
    public static final String ROSTER_REMOTE = "java:comp/env/ejb/rosterRemote";
    public static RosterLocal getLocal(){
        try {
            return (RosterLocal) CachingServiceLocator.getInstance().getEjb3BI(ROSTER_LOCAL);
        } catch (NamingException ex) {
            Logger.getLogger(RosterUtils.class.getName()).log(Level.SEVERE, null, ex);
            throw new RosterException(ex);
        }
    }

    public static RosterRemote getRemote(){
         try {
            return (RosterRemote) CachingServiceLocator.getInstance().getEjb3BI(ROSTER_REMOTE);
        } catch (NamingException ex) {
            Logger.getLogger(RosterUtils.class.getName()).log(Level.SEVERE, null, ex);
            throw new RosterException(ex);
        }
    }
}
```

[2] 配置文件 :

web.xml :

```
<web-app >
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
        <init-param>
```

```
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>debug</param-name>
            <param-value>2</param-value>
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>2</param-value>
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>action</servlet-name>
        <url-pattern>*.do</url-pattern>
    </servlet-mapping>

    <session-config>
        <!-- 在配置文件中单位是分钟 若在程序里面 那就是秒了 -->
        <session-timeout> 30</session-timeout>
    </session-config>

    <!-- EJB 的 JNDI 引用名在这里配置 -->
    <ejb-ref>
        <ejb-ref-name>ejb/rosterRemote</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <home/>
        <remote>net.kettas.roster.RosterRemote</remote>
        <ejb-link>RosterBean</ejb-link>
    </ejb-ref>
    <ejb-local-ref>
        <ejb-ref-name>ejb/rosterLocal</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <local-home/>
        <local>net.kettas.roster.RosterLocal</local>
        <ejb-link>RosterBean</ejb-link>
    </ejb-local-ref>
</web-app>
```

struts-config.xml :

```
<struts-config>
    <form-beans>
        <form-bean name="LeagueForm" type="net.kettas.roster.form.LeagueForm"/>
    </form-beans>
    <action-mappings>
       <action   input="/leagueInfo.jsp"   name="LeagueForm"   parameter="create"   path="/create"
        scope="request" type="net.kettas.roster.action.LeagueAction" validate="false">
            <forward name="success" path="/displayLeague.jsp"/>
        </action>
    </action-mappings>

    <!-- 资源文件 -->
    <message-resources parameter="net/kettas/roster/ApplicationResource"/>
</struts-config>
```

[3] **Action :**

```
public class LeagueAction extends MappingDispatchAction {
    /* forward name="success" path="" */
    private final static String SUCCESS = "success";
    public ActionForward create(ActionMapping mapping, ActionForm  form,
            HttpServletRequest request, HttpServletResponse response)
            throws Exception {
        League l = ((LeagueForm)form).getLeague();

        l = RosterUtils.getLocal().createLeague(l); // 通过工具类 取得"门面" 调用其中方法
        ((LeagueForm)form).setLeague(l);
        return mapping.findForward(SUCCESS);
    }
```

```
}
```

ActionForm :

```
public class LeagueForm extends org.apache.struts.action.ActionForm {
    private League league = new League();
    public Long getId(){      return league.getId();}
    public void setId(Long id){   league.setId(id);   }
    public String getName(){   return league.getName();}
    public void setName(String name){  league.setName(name);  }
    public String getSport(){    return league.getSport();}
    public void setSport(String sport){   league.setSport(sport);  }
    public League getLeague() {   return league; }
    public void setLeague(League league) {   this.league = league; }
    }
```

[4] 前端页面 :

创建 League leagueInfo.jsp :

```
<%@page contentType="text/html" pageEncoding="GBK"%>
<html>
    <body>
        <h2>Please enter a league to add</h2>
        <form action="create.do">
            Name:<input type="text" name="name"><br/>
            Sport:<input type="text" name="sport"><br/>
            <input type="submit" value="Create a League">
        </form>
    </body>
</html>
```

显示 League 信息 displayLeague.jsp :

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
    <body>
        <h2>League Information is</h2>
           <!-- requestScope 是 jstl 的隐含对象，代表 request 范围内可用的变量，
                       和 jsp 中使用 request 基本一样 -->
              ID: ${requestScope.LeagueForm.league.id}<br/>
              Name: ${requestScope.LeagueForm.league.name}<br/>
              Sport: ${requestScope.LeagueForm.league.sport}<br/>
        <a href="leagueInfo.jsp">Add a league</a>
    </body>
</html>
```

持久化的高级应用：1 大量的更新和删除

1） 级联操作
2） 乐观锁的版本不会更新，verson 不会自动更新
3） 持久化不会同步

2，Join 连接操作

1） 隐式连接 person.name 点号连接
2） 显示连接：A 内连接（以左为主，但是右不能为空）B,左连接 （以左为主，右可以为空）
C,JOIN FETCH 实例化查询 （参考 hibernate）

注意，最好不用 from T1,T2 这种方式查询 会产生笛卡尔连接降低查询性能（比如 10 乘以 10 的查询次数）
最好用 from T1 join T2 连接 子查询可以用 join 代替

3，查询时可以构建新对象，select new person() from xxxx ------写一个实体有相应的构造方法

Java 消息服务（Java Message Service，简称 JMS）是企业级消息传递系统，紧密集成于 Jboss Server 平台之中。
企业消息传递系统使得应用程序能够通过消息的交换与其他系统之间进行通信。

消息组成
消息传递系统的中心就是消息。一条 Message 分为三个组成部分：
• **头（header）是个标准字段集**，客户机和供应商都用它来标识和路由消息。
JMSMessageID: 标识提供者发送的每一条消息, 发送过程中由提供者设置
JMSDestination: 消息发送的 Destination, 由提供者设置
JMSDeliveryMode: 包括 DeliveryMode.PERSISTENT(被且只被传输一次)和
DeliveryMode.NON_PERSISTENT(最多被传输一次)
JMSTimestamp: 提供者发送消息的时间, 由提供者设置
JMSExpiration: 消息失效的时间, 是发送方法的生存时间和当前时间值的和, 0 表明消息不会过期
JMSPriority: 由提供者设置, 0 最低, 9 最高
JMSCorrelationID: 用来链接响应消息和请求消息, 由发送消息的 JMS 程序设置
JMSReplyTo: 请求程序用它来指出回复消息应发送的地方
JMSType: JMS 程序用来指出消息的类型
JMSRedelivered: 消息被过早的发送给了 JMS 程序, 程序不知道消息的接受者是谁

• **属性（property）支持把可选头字段添加到消息**。如果您的应用程序需要不使用标准头字段对消息编目和分类，
您就可以添加一个属性到消息以实现这个编目和分类。提供 set<Type>Property(...) 和 get<Type>Property(...) 方法
以设置和获取各种 Java 类型的属性，包括 Object。JMS 定义了一个供应商选择提供的标准属性集。
JMSXUserID: 发送消息的用户的身份
JMSXAppID: 发送消息的应用程序的身份
JMSXDeliveryCount: 尝试发送消息的次数
JMSXGroupID: 该消息所属的消息组的身份
JMSXGroupSeq: 该消息在消息组中的序号
JMSXProducerTxID: 生成该消息的事物的身份
JMSXConsumerTxID: 使用该消息的事物的身份
JMSXRcvTimestamp: JMS 将消息发送给客户的时间

• **消息的主体（body）包含要发送给接收应用程序的内容。**每个消息接口特定于它所支持的内容类型。
JMS 为不同类型的内容提供了它们各自的消息类型，但是所有消息都派生自 Message 接口。
• StreamMessage：包含 Java 基本数值流，用标准流操作来顺序的填充和读取。
• MapMessage：包含一组名/值对；名称为 string 类型，而值为 Java 的基本类型。
• TextMessage：包含一个 String。
• ObjectMessage：包含一个 Serializable Java 对象；能使用 JDK 的集合类。
• BytesMessage：包含未解释字节流: 编码主体以匹配现存的消息格式。
消息的传递模型：
JMS 支持两种消息传递模型：点对点（point-to-point，简称 PTP）和发布/订阅（publish/subscribe，简称 pub/sub）。
这两种消息传递模型非常相似，只有以下区别：
PTP 消息传递模型规定了一条消息只能传递给一个接收方。
Pub/sub 消息传递模型允许一条消息传递给多个接收方。
一个队列可以关联多个队列发送方和接收方，但一条消息仅传递给一个队列接收方。如果多个队列接收方正在监
听队列上的消息，jboss JMS 将根据“先来者优先”的原则确定由哪个队列接收方接收下一条消息。如果没有队
列接收方在监听队列，消息将保留在队列中，直至队列接收方连接队列为止。

2. 发布/订阅消息传递
通过发布/订阅(pub/sub) 消息传递模型，应用程序能够将一条消息发送到多个应用程序。Pub/sub 消息传递应用
程序可通过订阅主题来发送和接收消息。主题发布者（生成器）可向特定主题发送消息。主题订阅者（使用者）
从特定主题获取消息。
与 PTP 消息传递模型不同，pub/sub 消息传递模型允许多个主题订阅者接收同一条消息。JMS 一直保留消息，
直至所有主题订阅者都收到消息为止。
上面两种消息传递模型里，我们都需要定义消息发送者和接收者，消息发送者把消息发送到 Jboss JMS 某个
Destination，而消息接收者从 Jboss JMS 的某个 Destination 里获取消息。消息接收者可以同步或异步接收消息，
一般而言，异步消息接收者的执行和伸缩性都优于同步消息接收者

4,**消息驱动 Bean**(MDB)：是设计用来专门处理基于消息请求的组件。它能够收发异步 JMS 消息，并能够轻易地与
其他 EJB 交互。它特别适合用于当一个业务执行的时间很长，而执行结果无需实时向用户反馈的这样一个场合。参考 spring 的消息
消息发送者 ：

```
public class Main {
    @Resource(mappedName="jms/qcf")
    private static QueueConnectionFactory qcf; // 得到会话连接工厂
    @Resource(mappedName="jms/q")// 消息发送的目的地
```

```
    private static Queue q;
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args){
       QueueConnection con = null;
       QueueSession ses = null;
       QueueSender sender = null;

       try{
           con = qcf.createQueueConnection();  // 得到会话连接
           ses = con.createQueueSession(true, QueueSession.AUTO_ACKNOWLEDGE); // 根据会话连接创建一个会话
           sender = ses.createSender(q);  // 根据会话以及消息发送目的地创建消息发送者
           Message msg = ses.createTextMessage("My first JMS program.");  // 根据会话创建消息
           sender.send(msg); // 向目的地发送消息
           ses.commit();   // 提交会话
       }catch(JMSException e){
            try {
                ses.rollback();
            } catch (JMSException ex) {
                Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
            }
           e.printStackTrace();
       }finally{
           try{ if(sender != null) sender.close();
              }catch(JMSException e){e.printStackTrace();}
           try{ if(ses != null) ses.close();
           }catch(JMSException e){e.printStackTrace();}
           try{ if(con != null) con.close();
           }catch(JMSException e){e.printStackTrace();}
       }
    }
}
```

消息接收者 ：

```
public class Main {
    @Resource(mappedName="jms/qcf")
    private static QueueConnectionFactory qcf;
    @Resource(mappedName="jms/q")
    private static Queue q;

    public static void main(String[] args) {
        QueueConnection con = null;
        QueueSession ses = null;
        QueueReceiver receiver = null;

        try{
            con = qcf.createQueueConnection();
            ses = con.createQueueSession(true, QueueSession.AUTO_ACKNOWLEDGE);
            receiver = ses.createReceiver(q);
            con.start();
            TextMessage msg = (TextMessage)receiver.receive();
            System.out.println(msg.getText());
            ses.commit();
        }catch(JMSException e){
            e.printStackTrace();
            try {
                ses.rollback();
            } catch (JMSException ex) {
                Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
            }
        }finally{
            try{
                if(receiver != null) receiver.close();
            }catch(JMSException e){e.printStackTrace();}
             try{
                if(ses != null) ses.close();
            }catch(JMSException e){e.printStackTrace();}
```

```
                try{
                   if(con != null) con.close();
                }catch(JMSException e){e.printStackTrace();}
            }
        }
    }
```

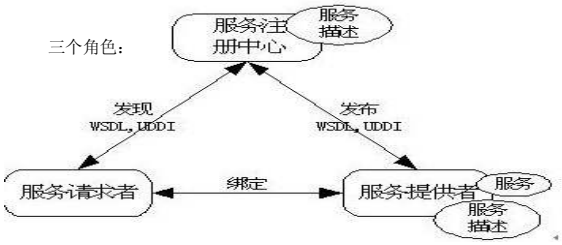
9，web 服务

Web Service：1 程序间以一种标准的方式通讯（和程序的开发语言，运行的操作系统，硬件，网络无关）
2，用 SOAP ,WSDL 等通讯，以 xml 文档进行数据的交换的网络应用程序

技术	接口	线上协议	名字服务
Web Servive	WSDL	SOAP	UDDI
EJB	java	RMI IIOP	JNDI

Web Service 特点 ： 跨平台、跨语言、跨 Internet，最底层是由 XML + HTTP 实现的。

三个角色：



实现一个完整的 Web 服务包括以下步骤：

◆ Web 服务提供者设计实现 Web 服务，并将调试正确后的 Web 服务通过 Web 服务中介者发布，并在 UDDI 注册中心注册 ； （发布）
◆ Web 服务请求者向 Web 服务中介者请求特定的服务，中介者根据请求查询 UDDI 注册中心，为请求者寻找满足请求的服务(其实现是通过服务注册器（uddi）来获得 wsdl 文件，再使用 wsdl 文档将请求绑定到 SOAP 调用服务提供者的服务)； （发现）
◆ Web 服务中介者向 Web 服务请求者返回满足条件的 Web 服务描述信息，该描述信息用 WSDL 写成，各种支持 Web 服务的机器都能阅读；（发现）
◆ 利用从 Web 服务中介者返回的描述信息生成相应的 SOAP 消息，发送给 Web 服务提供者，以实现 Web 服务的调用；（绑定）
◆ Web 服务提供者按 SOAP 消息执行相应的 Web 服务，并将服务结果返回给 Web 服务请求者。（绑定）

WSDL （本质是 xml 文档，web 服务的描述语言）
包括三个部分 ：①抽象(定义类型、消息...） 指定服务内容（方法参数，返回值，数据类型）

```
 | <types><!--定义数据类型与元素-->
 | - <message name="HelloSoapIn"></message>
 |   <portType name="HelloWorldSoap"><!--定义 web 服务的抽象接口,由 binding 和 service 元素来实现-->
      <operation name="Hello"><!--声明每一个 operation 元素要用一个或多个消息定义来定义它的输入，输出以及错误
          <input message="s0:HelloSoapIn" /> <!--传递到 web 服务的有效负载-->
          <output message="s0:HelloSoapOut" /> <!--传递给客户的有效负载-->
       </operation>
  </portType>
```

②绑定(生成 SOAP)

```
<binding name="HelloWorldSoap" type="s0:HelloWorldSoap"><!--将一个抽象 portType 映射到一组具体协议（如 soap 和
      http),消息传递样式（rpc 或文档)以及编码样式-->
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document" /> <!--指定了传输 soap 消息的
      internet 协议以及其操作的错误消息传递样式（rpc 或文档）-->
- <operation name="Hello">
  <soap:operation soapAction="http://workshop.bea.com/HelloWorld/Hello" style="document" /> <!--为具体的操作指
      定了消息传递样式（rpc 或文档）-->
- <input>
  <soap:body use="literal" /> <!--use 元素取单值-->
  </input>
- <output>
  <soap:body use="literal" />
  </output>
  </operation>
  </binding>
```

批注 [U97]: <!--描述 web 服务使用的消息的有效负载，即直接发送到 web 服务或直接从 web 服务接受的消息 <part name="parameters" element="s0:Hello" /> <!--可以使用 rpc 样式，也可以使用文档样式，本 part 使用的是文档岩石-->

批注 [U98]: 指名①中的参数，方法是放在 soap 的 head 中还是放在 body 中

③实现(指定访问名)

```
<service name="HelloWorld"><!--包含一个或多个 port 元素，其中每个 port 元素表示一个不同的 web 服务-->
  <documentation>A very simple web service.</documentation>
  -   <port name="HelloWorldSoap" binding="s0:HelloWorldSoap"><!--将 URL 赋给了一个特定 bing--->
        <soap:address location="http://localhost:7001/WebServices/HelloWorld.jws" />
    </port>
  </service>
```

批注 [U99]: 指定访问的地址

SOAP，简单对象访问协议（Simple Object Access Protocol，SOAP）实际上是一种 Web 服务技术，但 Web 服务中客户机和服务器之间的数据交换格式是通过灵活的 XML 模式实现的。SOAP 的结构 ： 相当于一封信，header 是信封，body 是信体

实例 ：

Web 服务端(新建 Web 应用) ：

```
package testWS;
@WebService(
      // 在客户端生成一个名为"UserOper"的接口，此接口正是对应这个类
      name = "UserOper",
      // 在客户端生成一个名为"UserOperService"的工具类
      // 此工具类中含有一个 "get" + portName 的方法，如这里为 getUserOperService()
      // 此方法返回对当前类的远程引用，给人的感觉就是返回当前类
      serviceName = "UserOperService",
      // 最好以大写字母开头
      portName = "UserOperPort",
      // ???
      targetNamespace = ""
)
@Resource(name = "yinkui", mappedName = "jdbc/mysql", type = DataSource.class)
public class UserMgmt {
    // @Resource(mappedName="jdbc/mysql")
    // private DataSource ds;

    // 指定方法为 Web 服务方法
    @WebMethod
    public void removeUser(int id){
        Connection conn = null;
        PreparedStatement ps = null;
        try{
            Context con = new InitialContext();
            DataSource ds = (DataSource)con.lookup("java:comp/env/yinkui");
            conn = ds.getConnection();
            ps = conn.prepareStatement("delete from user_info where user_id = " + id);
            ps.executeUpdate();
        }catch(Exception ex){
            ex.printStackTrace();
        }finally{
            try{ conn.close(); }catch(Exception ex){ }
        }
    }
}
```

根据服务端生成 Web 客户端(测试时新建 Java 应用) ：

在生成 Web 客户端时，工具自动为我们生成了大量的代码，使用起来很方便

```
package webclient;
// 这些包和接口以及类什么的都是自动生成的 直接引用
public class Main {
    public static void main(String[] args) {
      // 注意这里的写法
        UserOper u = new UserOperService().getUserOperPort();
        u.removeUser(100006);
    }
}
```

EJB 补充说明：1 单元测试---jnuit Assert 工具类

2 命令模式， 测试先行，先写测试，根据测试的要求再写具体的实现----先要结果在给过程

3 应用服务器有那些?

BEA WebLogic Server，IBM WebSphere Application Server，Oracle9i Application Server，jBoss，Tomcat

116、应用服务器与 WEB SERVER 的区别?

应用服务器：Weblogic、Tomcat、Jboss。。。。。

WEB SERVER：IIS、 Apache

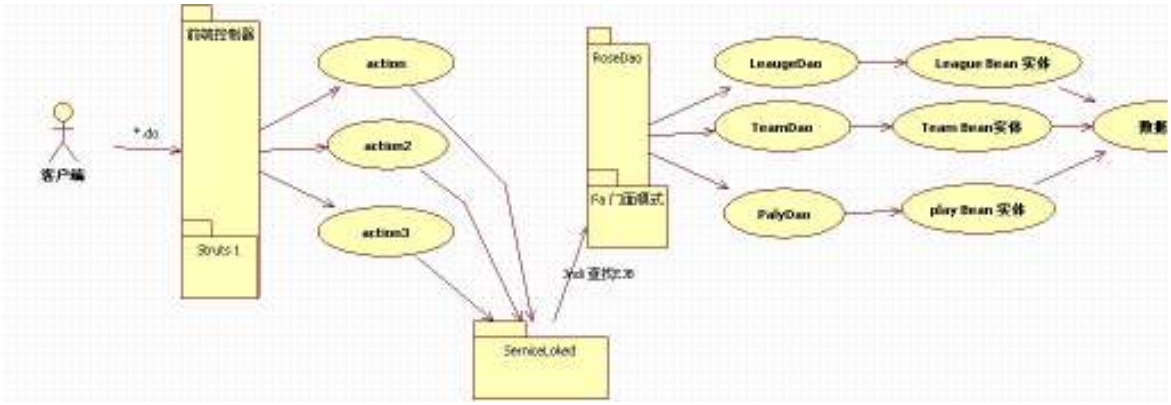
企业及应用 打包(.ear) 一个完整的企业应用包含 EJB 模块和 WEB 模块，在发布企业应用时，我们需要把它打成*.ear 文件，在打包前我们必须配置 application.xml 文件，该文件存放于打包后的 META-INF 目录。我们在 application.xml 文件中需要指定 EJB 模块和 WEB 模块的信息，

```
        | → WEB APP (.war)
.ear    |→  EJB      .jar
        |→Resouce 资源适配器（.rar）
```

一般在 NetBean 和 elicsape 都有自动的打包工具

```
ear 应用根目录
| -- ejb3.jar   (你的 EJB 模块)
| -- web.war   (你的 WEB 模块)
| -- META-INF
        | -- MANIFEST.MF (如果使用工具打包，该文件由工具自动生成)
        | -- application.xml
```

Java ee 的设计方案　两个集中连个发散



PL/SQL

PL/SQL 是 Procedure Language & Structured Query Language 的缩写。ORACLE 的 SQL 是支持 ANSI(American national Standards Institute)和 ISO92 (International Standards Organization)标准的产品。PL/SQL 是对 SQL 语言存储过程语言的扩展。从 ORACLE6 以后，ORACLE 的 RDBMS 附带了 PL/SQL。它现在已经成为一种过程处理语言，简称 PL/SQL。目前的 PL/SQL 包括两部分，一部分是数据库引擎部分；另一部分是可嵌入到许多产品（如 C 语言，JAVA 语言等）工具中的独立引擎。可以将这两部分称为：数据库 PL/SQL 和工具 PL/SQL。两者的编程非常相似。都具有编程结构、语法和逻辑机制。工具 PL/SQL 另外还增加了用于支持工具（如 ORACLE Forms）的句法，如：在窗体上设置按钮等。本章主要介绍数据库 PL/SQL 内容。

§1.2.1 PL/SQL 的好处

§1.2.1.1 有利于客户/服务器环境应用的运行

对于客户/服务器环境来说，真正的瓶颈是网络上。无论网络多快，只要客户端与服务器进行大量的数据交换。应用运行的效率自然就回受到影响。如果使用 PL/SQL 进行编程，将这种具有大量数据处理的应用放在服务器端来执行。自然就省去了数据在网上的传输时间。

PL/SQL 程序由三个块组成，即声明部分、执行部分、异常处理部分。

PL/SQL 块的结构如下：

```
DECLARE
      /* 声明部分: 在此声明 PL/SQL 用到的变量,类型及游标，以及局部的存储过程和函数 */
BEGIN
      /* 执行部分: 过程及 SQL 语句 , 即程序的主要部分 */

EXCEPTION
      /* 执行异常部分: 错误处理 */
END;
```

其中 执行部分是必须的。

学习的流程：数据类型变量---》流程控制---》存储过程----》函数----》包-----》 job

PL/SQL day1

```
--设置数据库输出，默认为关闭，每次新打开窗口都要从新设置
set serveroutput on

--调用    包    函数     参数
execute dbms_output.put_line('hello world');
--或者用 call 调用,相当于 java 的调试程序打桩
call dbms_output.put_line('hello world');

--pl 语句块是 pl/sql 里最小的编程块，其中可以再嵌套 begin end
begin
      dbms_output.put_line('Hello World');
      dbms_output.put_line('2*3='||(2*3));
      dbms_output.put_line('what''s');
end;

--如何声明变量,所有变量必须再 declare 中声明，程序中不允许声明
--没有初始化的变量默认值为 null,屏幕上 null 是看不见的，命名习惯：一般变量以 v_开头
--注意 number 也能存小数，最长 38 位，所以以后建议整数都用 binary_integer 存
--long 是字符类型;boolean 类型不能打印
--标准变量类型：数字，字符，时间，布尔
declare
      v_number1 number ;
      v_number2 number(3,2) ;
      v_number3 binary_integer :=1;
      v_name varchar2(20) :='kettas';
      v_date date :=sysdate;
      v_long long :='ni hao';
      v_b boolean := true;
begin
      if (v_number1 is null) then
            dbms_output.put_line( 'hello');
      end if;
      dbms_output.put_line(v_number1);
      dbms_output.put_line(v_number2);
      dbms_output.put_line(v_number3);
      dbms_output.put_line(v_name);
      dbms_output.put_line(v_date);
      dbms_output.put_line(v_long);
    --dbms_output.put_line(v_b);
end;
```

--组合类型：record ，table

```
--record 类型最常用，声明的时候可以加 not null，但必须给初始值
--如果 record 类型一致可以互相赋值，如果类型不同，里面的字段恰好相同，不能互相赋值
declare
      type t_first is record(
            id number,
            name varchar2(20)
      );
      v_first t_first;
begin
      v_first.id:=1;
      v_first.name:='sx';
      dbms_output.put_line(v_first.id);
      dbms_output.put_line(v_first.name);
end;
```

--考虑一下，orcale 中赋值是拷贝方式还是引用方式？

```
declare
      v_number1 number:=1;
      v_number2 number;
      type t_first is record(
            id number,
            name varchar2(20)
      );
      v_first t_first;
      v_second t_first;
begin
      v_number2 := v_number1;
      v_number1:=2;
      dbms_output.put_line('number1'||v_number1);
      dbms_output.put_line(v_number2);
      v_first.id:=1;
      v_first.name:='sx';
      v_second := v_first;
      v_first.id:=2;
      v_first.name:='kettas';
      dbms_output.put_line(v_first.id);
      dbms_output.put_line(v_first.name);
      dbms_output.put_line(v_second.id);
      dbms_output.put_line(v_second.name);
end;

--table 类型，相当于 java 中的 map，就是一个可变长的数组
--key 必须是整数(可以是负数)，value 可以是标量，也可以是 record
--可以不按顺序赋值，但必须先赋值后使用
declare
      type t_tb is table of varchar2(20) index by binary_integer;
      v_tb t_tb;
begin
      v_tb(100):= 'hello';
      v_tb(98):='world';
      dbms_output.put_line(v_tb(100));
      dbms_output.put_line(v_tb(98));
end;

declare
      type t_rd is record(id number,name varchar2(20));
      type t_tb is table of t_rd index by binary_integer;
      v_tb2 t_tb;
begin
      v_tb2(100).id:=1;
      v_tb2(100).name:='hello';
      dbms_output.put_line(v_tb2(100).id);
      dbms_output.put_line(v_tb2(100).name);
end;
```

-- %type 和 %rowtype 以及如何从数据库把数据取回来

```
create table student(
```

```
      id number,
      name varchar2(20),
      age number(3,0)
)
insert into student(id,name,age) values(1,'sx',25);
--查找一个字段的变量
declare
      v_name varchar2(20);
      v_name2 student.name%type;
begin
            select name into v_name2 from student where rownum=1;
            dbms_output.put_line(v_name2);
end;

--查找一个类型的变量，推荐用*
declare
      v_student student%rowtype;
begin
      select * into v_student from student where rownum=1;
      dbms_output.put_line(v_student.id||' '||v_student.name||' '||v_student.age);
end;

--也可以按字段查找，但是字段顺序必须一样，不推荐这样做
declare
      v_student student%rowtype;
begin
      select id,name,age into v_student from student where rownum=1;
      dbms_output.put_line(v_student.id||' '||v_student.name||' '||v_student.age);
end;

--注意：insert,update,delete,select 都可以，create table，drop table 不行
--dql,dml,和流程控制可以在 pl/sql 里用，ddl 不行
declare
      v_name student.name%type :='shixiang';
begin
      insert into student (id,name,age) values( 2,v_name,26);
end;

declare
      v_name student.name%type :='shixiang';
begin
      update student set name = v_name where id=1;
end;

--变量的可见空间
declare
      v_i1 binary_integer :=1;
begin
      declare
            v_i2 binary_integer:=2;
      begin
            dbms_output.put_line(v_i1);
            dbms_output.put_line(v_i2);
      end;

      --dbms_output.put_line(v_i1);
      --dbms_output.put_line(v_i2);
end;


-----下午课程：流程控制
--if 判断
declare
      v_b boolean :=true;
begin
      if  v_b then
            dbms_output.put_line('ok');
      end if;
end;
```

```
--if else
declare
      v_b boolean :=false;
begin
      if v_b then
            dbms_output.put_line('ok');
      else
            dbms_output.put_line('false');
      end if;
end;

--if elsif else
declare
      v_name varchar2(20):='sx';
begin
      if v_name='0701' then
            dbms_output.put_line('0701');
      elsif v_name ='sx' then
            dbms_output.put_line('sx');
      else
            dbms_output.put_line('false');
      end if;
end;

--loop 循环,注意退出 exit 是退出循环，不是退出整个代码块
declare
      v_i binary_integer := 0;
begin
      loop
            if v_i >10 then
                  exit;
            end if;
            v_i := v_i+1;
            dbms_output.put_line('hehe');
      end loop;
            dbms_output.put_line('over');
end;

--更简单的写法
declare
      v_i binary_integer := 0;
begin
      loop
            exit when v_i>30;
            v_i := v_i+1;
            dbms_output.put_line('hehe');
      end loop;
            dbms_output.put_line('over');
end;

--while 循环
declare
      v_i binary_integer:=0;
begin
      while  v_i<30    loop
            dbms_output.put_line('hello' || v_i);
            v_i := v_i+1;
      end loop;
      dbms_output.put_line('over');
end;

--for 循环,注意不需要声明变量
begin
      for v_i in 0..30 loop
            dbms_output.put_line('hello'||v_i);
      end loop;
            dbms_output.put_line('over');
end;
```

```
--练习 1 用循环往 student 里插入 30 条记录
--要求：id 为 0,1,2...
--              name 为 kettas0,kettas1,kettas2...
--              age 为 11,12,13...
--练习 2，假设不知道数据库里的数据规则和数量,
--把所有的 student 数据打印到终端

begin
     delete from student;
     for v_i in 0..30 loop
          insert into student(id,name,age) values(v_i,'kettas'||v_i,    10+v_i);
     end loop;
end;
```

--rownum 是伪列，在表里没有

```
--数据库是先执行 from student 遍历 student 表,如果没有 where 条件过滤,则先做成一个结果集,然后再看 select
--后面的条件挑出合适的字段形成最后的结果集,如果有 where 条件,则不符合条件的就会从第一个结果集中删除,
--后面的数据继续加进来判断。所以如果直接写 rownum=2，或者 rownum>10 这样的语句就查不出数据。
--可以用一个子查询解决
select rownum,id from student where rownum=2;
declare
     v_number binary_integer;
     v_student student%rowtype;
begin
     select count(*) into v_number from student;
     for i in 1..v_number loop
          select id,name,age into v_student
          from
               (select rownum rn,id,name,age from student)
          where rn=i;
          dbms_output.put_line('id: '||v_student.id||' name:'||v_student.name);
     end loop;
end;

--异常的定义使用
begin
     dbms_output.put_line(1/0);
exception
     when others then
          dbms_output.put_line('error');
end;

declare
     e_myException exception;
begin
     dbms_output.put_line('hello');
     raise e_myException;--raise 抛出异常
     dbms_output.put_line('world');
     dbms_output.put_line(1/0);
exception
     when e_myException then
          dbms_output.put_line(sqlcode);--当前会话执行状态，错误编码
          dbms_output.put_line(sqlerrm);--当前错误信息
          dbms_output.put_line('my error');
     when others then
          dbms_output.put_line('error');
end;

--error 表,每次有异常可以加到表里,相当于 log 日志
--id,code,errm,information,create_date
--目前先这么写，明天学存储过程后就再改成一个过程，用的时候只需要调用即可
drop table error;
create table error(
     id number,
     code number,
     errm varchar2(4000),
     information varchar2(4000),
     create_date date
```

```
)
drop sequence my_key;
create sequence my_key;
insert into error(id,code,errm,information,create_date)
      values(my_key.nextVal,1,'xxx','xxxxx',sysdate);

declare
      e_myException exception;
begin
      dbms_output.put_line('hello');
      raise e_myException;
      dbms_output.put_line('world');
      dbms_output.put_line(1/0);
exception
      when e_myException then
            declare
                  v_code binary_integer;
                  v_errm varchar2(4000);
            begin
                  v_code:=sqlcode;
                  v_errm:=sqlerrm;
                  insert into error(id,code,errm,information,create_date)
                        values(my_key.nextVal,v_code,v_errm,'e_myException',sysdate);
            end;
      when others then
            dbms_output.put_line('error');
end;
```

--cursor 游标(结果集)用于提取多行数据

```
--定义后不会有数据,使用后才有
--一旦游标被打开,就无法再次打开(可以先关闭,再打开)
declare
      cursor c_student is
            select * from student;
begin
      open c_student;
      close c_student;
end;

--第 2 种游标的定义方式,用变量控制结果集的数量
declare
      v_id binary_integer ;
      cursor c_student is
            select * from student where id>v_id;
begin
      v_id:=10;
      open c_student;
      close c_student;
end;

--第 3 种游标的定义方式,带参数的游标,用的最多
declare
      cursor c_student(v_id binary_integer) is
            select * from student where id>v_id;
begin
      open c_student(10);
      close c_student;
end;

--游标的使用,一定别忘了关游标
declare
      v_student student%rowtype;
      cursor c_student(v_id binary_integer) is
            select * from student where id>v_id;
begin
      open c_student(10);
      fetch c_student into v_student;
      close c_student;
      dbms_output.put_line(v_student.name);
```

```
end;

--如何遍历游标 fetch
--游标的属性 %found,%notfound,%isopen,%rowcount
--%found:若前面的 fetch 语句返回一行数据,则%found 返回 true,如果对未打开的游标使用则报 ORA-1001 异常
--%notfound:与%found 行为相反码          ••••••••••••••••••
--%isopen:判断游标是否打开
--%rowcount:当前游标的指针位移量,到目前位置游标所检索的数据行的个数,若未打开就引用,返回 ORA-1001

--loop 方式遍历游标
declare
      v_student student%rowtype;
      cursor c_student(v_id binary_integer) is
            select * from student where id>v_id;
begin
      open c_student(10);
      loop
            fetch c_student into v_student;
            exit when c_student%notfound;
            dbms_output.put_line('name: ' || v_student.name);
      end loop;
      close c_student;
end;

--while 循环遍历游标,注意,第一次游标刚打开就 fetch,%found 为 null,进不去循环
--如何解决:while nvl(c_student%found,true) loop
declare
      v_student student%rowtype;
      cursor c_student(v_id binary_integer) is
            select * from student where id>v_id;
begin
      open c_student(10);
      while c_student%found is null or c_student%found loop
            fetch c_student into v_student;
            dbms_output.put_line('name: ' ||v_student.name);
      end loop;
      close c_student;
end;

--for 循环遍历,最简单,用的最多,不需要声明 v_student、打开关闭游标、fetch。
declare
      cursor c_student(v_id binary_integer) is
            select * from student where id>v_id;
begin
      for v_student in c_student(10) loop
            dbms_output.put_line('name: '||v_student.name);
      end loop;
end;

--goto 例子,不推荐使用 goto,会使程序结构变乱
declare
      i binary_integer :=0;
begin
      if i=0 then goto hello;end if;
      <<hello>>
      begin
            dbms_output.put_line('hello');
            goto over;
      end;
      <<world>>
      begin
            dbms_output.put_line('world');
            goto over;
      end;
      <<over>>
            dbms_output.put_line('over');
end;
```

--存储过程:把匿名块存储下来

```
--匿名块运行后不会在数据库留下
declare
      v_content   varchar2(4000):='hello';
begin
      dbms_output.put_line(v_content);
end;

--创建或修改一个过程,sx_println(形式参数,声明类型即可),as 可以替换成 is
--变量可以声明再 as 和 begin 之间
create or  replace     procedure  sx_println(v_content varchar2)
as
begin
      dbms_output.put_line(v_content);
end;

--调用有参过程
execute     sx_println('sx');
call sx_println('sx');
begin
                  sx_println('sx');
end;

--删除一个过程
drop procedure sx_println;
--查看数据库里的过程
select * from    user_procedures;
desc user_procedures;
select object_name,procedure_name from user_procedures;

--procedure 里可以调用其他的 procedure
create or  replace     procedure  say_hello
is
begin
           sx_println('hello');
end;

--调用无参过程的方式: execute     say_hello; call  say_hello(); begin say_hello;     end;

--输入参数 in,输入参数不能进行赋值,默认不写就是 in, v_name in  varchar2 中的 varchar2 不能设置长度,以对方传过来的为主。
--out 只负责赋值, 外界对其赋值不起作用, 存储返回的结果。
--in out 既可以当输入页可以当输出。
--存储过程没有重载,这个有参的 say_hello 会替代上面的无参 say_hello
create or  replace     procedure  say_hello(v_name in    varchar2)
as
begin
      --v_name :='a';
      sx_println('hello      '||v_name);
end;

--调用有参的存储过程的两种方式
begin
      --say_hello('shixiang');
      say_hello(v_name=>'sx');
end;

--多个参数的存储过程
create or  replace     procedure  say_hello
      (v_first_name     in varchar2,v_last_name      in varchar2)
as
begin
      sx_println('hello      '||v_first_name||'.'||v_last_name);
end;

--调用多个参数的两种方式
begin
      say_hello('cksd0701','shixiang');
end;
```

```
--用指定形参名的方式调用可以不按顺序赋值
begin
      say_hello(v_last_name=>'sx',v_first_name=>'0701');
end;

--out 输出参数,用于利用存储过程给一个或多个变量赋值,类似于返回值,
create or  replace    procedure  say_hello
      (v_name    in varchar2,v_content  out   varchar2)
begin
      v_content:='hello     '     || v_name;
end;

--调用
declare
      v_con varchar2(200);
      v_in varchar2(20):='cksd0702';
begin
      say_hello(v_in,v_con);
      sx_println(v_con);
end;

--in out 参数,既赋值,又取值
create or  replace    procedure  say_hello(v_name in    out   varchar2)
as
begin
      v_name:=v_name||'     hi';
end;
--调用
declare
      v_inout    varchar2(20):='sx';
begin
      say_hello(v_inout);
      sx_println(v_inout);
end;

--缺省参数 default 为默认值
create or  replace    procedure  say_hello(v_name varchar2   default    'a',v_content    varchar2 default 'hello')
as
begin
      sx_println(v_name||' '||v_content);
end;

--调用,用指名型参名的方式调用更好 , 指明参数赋值，避免多个参数的顺序的问题。
begin
            say_hello();
end;
begin
      say_hello(v_name=>'sx');
end;

begin
      say_hello(v_content=>'hi');
end;
```

--function 函数

```
--过程和函数都以编译后的形式存放在数据库中，函数可以没有参数也可以有多个参数并有一个返回值。
--过程有零个或多个参数，没有返回值。函数和过程都可以通过参数列表接收或返回零个或多个值，函数和过程的主要区别不在于返回值
--而在于他们的调用方式。过程是作为一个独立执行语句调用的,函数以合法的表达式的方式调用.
--传入和返回的参数的
create or  replace    function func(v_name in     varchar2)
return varchar2
is
begin
      return(v_name||' hello');
end;
--调用
declare
      v_name varchar2(20);
```

```
begin
      v_name:=func('shixiang');
      sx_println(v_name);
end;

--out 参数的函数
create or  replace    function func(v_name in     varchar2,v_content out varchar2)
return varchar2
is
begin
      v_content:=v_name||' hello';
      return v_content;
end;
--调用 相当调用引用
declare
      v_name varchar2(20):='ccccccccccc';
      v_name1     varchar2(20):='zzzzzzzzz';
begin
      v_name1:=func('shixiang',v_name);
      v_name:='ccccccccccc';

      sx_println(v_name1);
end;

--in out 参数
create or  replace    function func(v_name in     out   varchar2)
return varchar2
is
begin
      v_name:=v_name||'      hello';
      return 's';
end;
--调用
declare
      v_inout     varchar2(20):='sx';
      v_ret varchar2(20);
begin
      v_ret:=func(v_inout);
      sx_println(v_inout);
      sx_println(v_ret);
end;
```

--存储过程调函数，函数调用存储过程

```
create or replace procedure print(v_name varchar2,v_res out varchar2)
as
      v_test varchar2(30);
      v_s varchar2(30);
begin
      --dbms_output.put_line('==========');
      v_test:=callPrint(v_name,v_res);
      if(v_test=v_s) then
            dbms_output.put_line('equlas');
      end if;
      v_res:=v_test||'zzzzzzzz';
end;

create or replace function callPrint(v_name varchar2,v_res out varchar2)
return varchar2
as
begin
      --print();
      --return 'sss';
      v_res:=v_name||'hello';
      return v_res;
end;

declare
      v_test varchar2(30);
begin
```

```
      v_test:=callPrint();
      dbms_output.put_line(v_test);
end;

declare
      v_print varchar2(30);
begin
      print('zhangsan',v_print);
      dbms_output.put_line(v_print);
end;
```

--子程序:没有 CREATE OR REPLACE 关键字,子程序的定义放在程序语句块的声明部分
--子程序只能被该程序语句块使用

```
declare
      procedure  println(v_name varchar2)
      as
      begin
            dbms_output.put_line(v_name);
      end;
begin
      println('sx');
end;

declare
      v_name1    varchar2(20);
      function f_func(v_name in   varchar2)
      return varchar2
      is
      begin
            return(v_name||' hello');
      end;
begin
      v_name1:=f_func('sx');
      sx_println(v_name1);
end;

--删除函数
drop function   func;
```

--下午:包,触发器
--和 java 的包类似,可以把函数,过程,变量等相关对象定义在一个包下,便于管理,例如:跟产品相关的包,用户相关的包,订单相关的**包**
--包里可以有过程,函数,变量,类型,异常,游标
--包只能存储再数据库中,不能是本地的
--包分两部分:包头(各种声明),包体(各种实现),可以分开编译

```
--package  先申明后在 body 中实现具体的实现
create or  replace    package    sx
as
      procedure   testP(v_name    varchar2);
      --包里面可以发生重载,但要注意参数大类型必须不同
      procedure   testP(v_name    binary_integer);
      function   testF(v_name varchar2) return    binary_integer;
      type t_student_tb     is table ofstudent%rowtype
            index by binary_integer;
      type t_student_re     is record(
            id binary_integer,
            name varchar2(20)
      );
      v_id binary_integer:=1;
      cursor c_student is   select * from   student;
end  sx;
--使用包
declare
      v_length binary_integer;
      v_student_tb sx.t_student_tb;
begin
      v_student_tb(1).id:=1;
```

```
      v_student_tb(1).name:='sx';
      v_student_tb(1).age:=25;
      sx.v_id    :=1;
      for   v_student  in sx.c_student  loop
            dbms_output.put_line(v_student.id);
      end   loop;
      sx.testP('hello');
      --sx.testP1('hello');
      v_length :=sx.testF('hehe');
      dbms_output.put_line(v_length);
end;
--看看 v_id 的值,实际上数据库为每一个会话(线程)保存一个变量的缓存
declare
      v_student_tb sx.t_student_tb;
begin
      dbms_output.put_line(sx.v_id);
end;

--包体里也可以写变量类型函数过程等,如果包头里没有声明,包体里有,相当与私有,只能在包体内部使用
create or  replace    package    body sx
as
      procedure   testP(v_name    varchar2)
      is
      begin
            dbms_output.put_line(v_name);
      end;

      procedure   testP1(v_name varchar2)
      is
      begin
            dbms_output.put_line(v_name);
      end;

      procedure   testP(v_name    binary_integer)
      is
      begin
            dbms_output.put_line(v_name);
      end;

      function   testF(v_name varchar2) return    binary_integer
      is
      begin
            return length(v_name);
      end;
end   sx;
```

--触发器(有点像监听器，在操作数据的前后的动作)

主要针对 insert ，update，delete 操作 两种类型：before，after（表级，行级）

```
drop table test_sx;
create table test_sx(
      id number,
      name varchar2(20),
      age   number
);
```

--创建表级触发器：只会在动作前后操作，而不是在插入每条数据就触发。

```
--before insert
create or  replace    trigger    trg_test1
     before insert    on test_sx
declare
      v_name varchar2(20);
begin
      sx_println('before insert   test table');
end;

--after    insert
create or  replace    trigger    trg_test2
after insert on  test_sx
declare
```

```
      v_name varchar2(20);
begin
      sx_println('after     insert testtable');
end;

--before update
create or  replace    trigger    trg_test3
before update    on test_sx
declare
      v_name varchar2(20);
begin
      sx_println('before update   test table');
end;

--after    update
create or  replace    trigger    trg_test4
after update on  test_sx
declare
      v_name varchar2(20);
begin
      sx_println('after     update testtable');
end;

--before delete
create or  replace    trigger    trg_test5
before delete    on test_sx
declare
      v_name varchar2(20);
begin
      sx_println('before delete   test table');
end;

--after    delete
create or  replace    trigger    trg_test6
after delete on  test_sx
declare
      v_name varchar2(20);
begin
      sx_println('after     delete testtable');
end;


--表级(语句级)触发器,只会在动作前后触发,而不是每插入一条数据就触发
insert intotest_sx    select * from    test_sx;


--行级触发器，只是在表级触发器加 for each    row
--before insert
create or  replace    trigger    trg_test7
before insert    on test_sx for each   row
begin
      sx_println('before insert   test row');
end;

--after    insert
create or  replace    trigger    trg_test8
after insert on  test_sx    for  each row
begin
      sx_println('after     insert testrow');
end;

--:new 行触发器的一个特殊 record 变量,指当前操作行对象
create or  replace    trigger    trg_test7
before insert    on test_sx for each   row
begin
      sx_println('before insert   test row new id: '||:new.id);
end;

create or  replace    trigger    trg_test8
```

```
after insert on  test_sx    for  each row
begin
     sx_println('after     insert testrow  new  id:'|| :new.id);
end;
```

--可以写触发条件

```
create or  replace    trigger    trg_test7
before insert    on test_sx for each    row  when(new.id>10)
begin
     sx_println('before insert   test row new id: '||:new.id);
     sx_println('before insert   test row old id: '||:old.id);
end;
```

--:old 指原对象,对于 update 来说要更新谁,谁就是 old

```
--对于 insert 来说,old 为空
--对于 delete 来说,new 为空
create or  replace    trigger    trg_test9
before delete    on test_sx for each    row
begin
     insert intotest_sx    values(32,'sss',24);
end;

create or  replace    trigger    trg_test10
afters delete    on test_sx for each    row
declare
     v_name varchar2(20);
begin
     insert intotest_sx    values(32,'sss',24);
end;


--触发器功能强大,但是不易于维护,所以不适合做非常复杂的逻辑
---一般用于维护数据的完整性
--共有 12 种触发器类型(不全面)
--三个语句(insert/update/delete)
--两种类型(before/after)
--两种级别(表级/行级)
---一个表最多可以定义 12 种触发器
--触发器的限制:触发器的主体是 pl/sql 语句块,所有能出现在 pl/sql 中的语句再触发器主体中都是合法的限制.不应该使用事务控
制语句 commit,rollback,savepoint,
--由触发器调用的审核过程与函数都不能使用事务控制语句,不能声明任何 long 或者 long raw 变量,可以访问的表有限
--行级触发器无法读                    变化表:被 dml 语句正在修改的表,或者对自定义触发器的表进行 select 操作
                                                --限制表:跟变化表有外键关联的表.

--dbms_job 包:允许调度 pl/sql 语句块,让它再指定时间自动运行,类似于 unix 的 cron

drop table testJob;
create table testJob(
     create_datedate
);

create or  replace    procedure  doJob
as
begin
     insert intotestJob    values(sysdate);
end;
```

--创建 JOB,submit 的 4 个参数:job 号,过程名,当前运行时间,下一次运行时间

```
declare
     v_job binary_integer;
begin
     dbms_job.submit(v_job,'doJob;',sysdate,'sysdate+1/1440');
     sx_println(v_job);
end;

--查看 job
desc user_jobs;
select job,what  from user_jobs;
```

--手动运行 job

```
begin
      dbms_job.run(21);
end;
```

--删除 JOB

```
begin
      dbms_job.remove(21);
end;
```

```
--相关的几个 job 操作
--修改要执行的操作:dbms_job.what(jobno,what);
--修改下次执行时间:dbms_job.next_date(job,next_date);
--修改间隔时间:              dbms_job.interval(job,interval);
--停止 job:                        dbms_job.broken(job,broken,nextdate);
```

--动态 sql

```
declare
      v_day varchar2(20);
begin
      select to_char(sysdate,'dd') into v_day from dual;
      if v_day='01' then
            insert into test1 values(1,'xx');
      elsif v_day='02' then
            insert into test2 values(1,'xx');
      elsif v_day='03' then
            insert into test3 values(1,'xx');
      ...
end;

declare
      v_day varchar2(20);
begin
      execute immediate 'insert into test'|| v_day||'values(....)';
end;

--oracle8i 以前只能用 dbms_sql 实现动态 sql,比较麻烦
--动态 sql 虽然方便但是效率很低,慎用.
```

最后用 java 调用

```
public class TestMain3 {
      public static void main(String[] args) throws Exception{
            Class.forName("oracle.jdbc.driver.OracleDriver");
            String url="jdbc:oracle:thin:@localhost:1521:XE";
            Connection con=DriverManager.getConnection(url,"sunying","sunying");
            CallableStatement c=con.prepareCall("{call pro(?,?)}");
            c.setInt(1,10);
            c.registerOutParameter(2,oracle.jdbc.OracleTypes.CURSOR);
            c.execute();
            ResultSet rs=(ResultSet)c.getObject(2);
            while(rs.next()){
                  System.out.println(rs.getInt(1)+"====="+rs.getString(2)+"===="+rs.getInt(3));

            }

            c.close();
            con.close();

      }

}

create or replace package sun
as type sun_cur is REF CURSOR;
```

```
end sun;

create or replace procedure pro(v_cur out sun.sun_cur)
as
begin
      open v_cur for select * from student;
end;
```

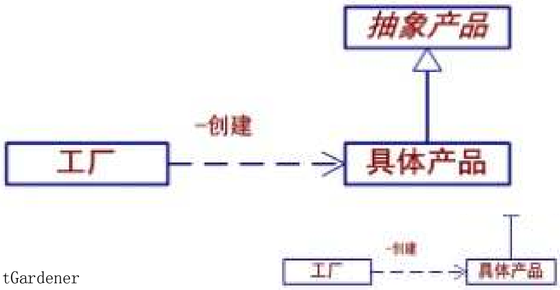
Java 与模式

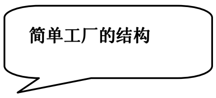
1 工厂模式专门负责将大量有共同接口的类实例化。工厂模式可以动态决定将哪一个类实例化，不必事先知道每次要实例化哪一个类。工厂模式有以下几种形态：

（1）简单工厂 …… rn)。

（2）工厂 …… 造子（Virtual Constructor 模式；

（3）抽象 ……





1）简单工 …… Static Factory Method）模式。

简单工厂模 …… 类可以根据传入的参量决定创建

出哪一种产 ……

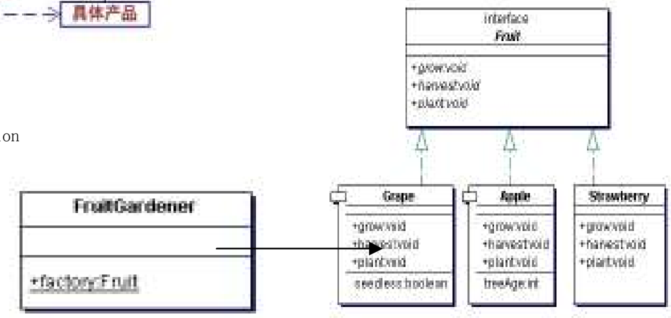
实例：有三 ……

葡萄 Grape

草莓 Strawberry

苹果 Apple

```
public class FruitGardener
{
/**
* 静态工厂方法
*/
public static Fruit factory(String which) throws BadFruitException
  { if (which.equalsIgnoreCase("apple")){
      return new Apple();
   }
else if (which.equalsIgnoreCase("strawberry")){
   return new Strawberry();
   }
else if (which.equalsIgnoreCase("grape")){
  return new Grape();
} else{ throw new BadFruitException("Bad fruit request");
}
}
}
```

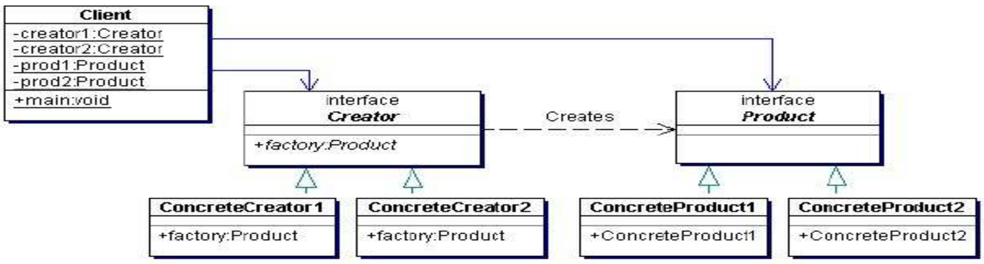


在使用时，客户端只需调用 FruitGardener 的静态方法 factory()即可

2）工厂方法（Factory Method）模式工厂方法模式是类的创建模式，又叫做虚拟构造子（Virtual Constructor）模式或者多态性工厂（Polymorphic Factory）模式。工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点

批注 [U100]: 简单工厂模式的缺点是对“开-闭”原则的支持不够，因为如果有新的产品加入到系统中去，就需要修改工厂类，将必要的逻辑加入到工厂类中

首先，在工厂方法模式中，核心的工厂类不再负责所有的产品的创建，而是将具体创建的工作交给子类去做。这个核心类则摇身一变，成为了一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。这种进一步抽象化的结果，使这种工厂方法模式可以用来允许系统在不修改具体工厂角色的情况下引进新的产品，这一特点无疑使得工厂模式具有超过简单工厂模式的优越性。



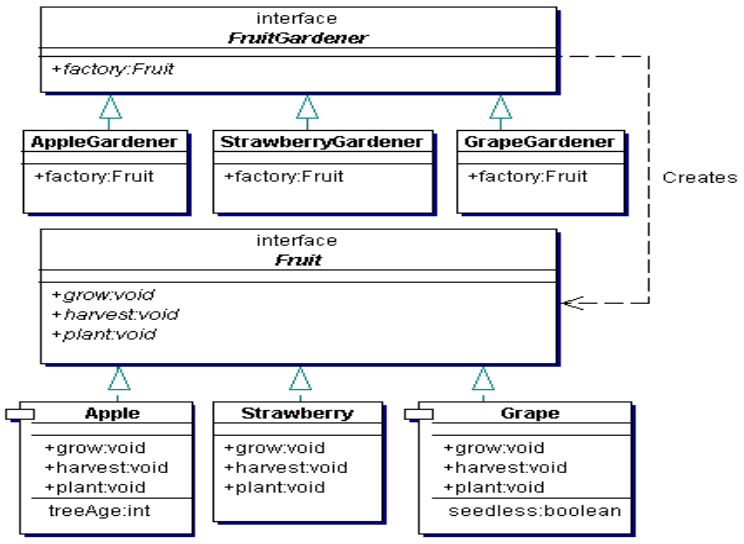
1：抽象工厂角色 Creator 类的源代码

```
package com.javapatterns.factorymethod;
public interface Creator{
/*** 工厂方法*/
public Product factory();
}
```

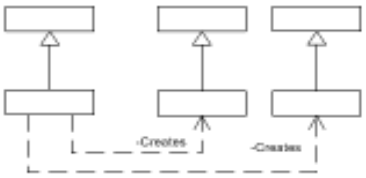
抽象产品角色 Product 类的源代码

```
package com.javapatterns.factorymethod;
public interface Product
```

```
{
}
```
3：具体工厂角色 ConcreteCreator1 类的源代码
```
package com.javapatterns.factorymethod;
public class ConcreteCreator1 implements Creator
{
/*** 工厂方法*/
public Product factory(){
   return new ConcreteProduct1();
}
}
```
具体工厂角色 ConcreteCreator2 类的源代码
```
package com.javapatterns.factorymethod;
public class ConcreteCreator2 implements Creator{
   /*** 工厂方法*/
public Product factory(){
   return new ConcreteProduct2();
}
}
```
具体产品角色 ConcreteProduct1 类的源代码
```
package com.javapatterns.factorymethod;
public class ConcreteProduct1 implements Product{
  public ConcreteProduct1(){
//do something
}
}
```
具体产品角色 ConcreteProduct2 类的源代码
```
package com.javapatterns.factorymethod;
public class ConcreteProduct2 implements Product{
   public ConcreteProduct2(){
   //do something
   }
}
```
客户端角色 Client 类的源代码
```
package com.javapatterns.factorymethod;
public class Client{
   private static Creator creator1, creator2;
   private static Product prod1, prod2;
public static void main(String[] args) {
   creator1 = new ConcreteCreator1();
   prod1 = creator1.factory();
   creator2 = new ConcreteCreator2();
   prod2 = creator2.factory();
}
}
```
注意：工厂方法模式在农场系统中的实现图



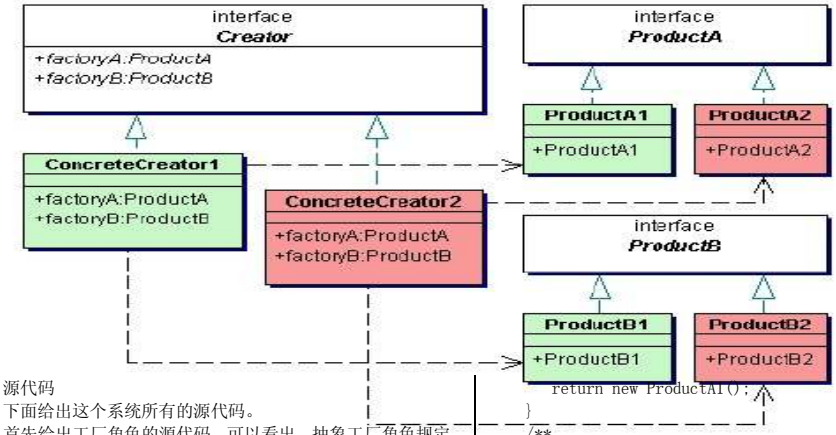
3）抽象工厂（Abstract Factory）模式抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。它是工厂方法模式的进一步推广。抽象工厂模式的简略类图如下所示。



★左边的等级结构代表工厂等级结构，右边的两个等级结构分别代表两个不同的产品的等级结构。抽象工厂模式可以向客户端提供一个接口，使得客户端在不必指定产品的具体类型的情况下，创建多个产品族中的产品对象。这就是抽象工厂模式的用意。

实例

采用抽象工厂模式设计出的系统类图如下所示。



源代码

下面给出这个系统所有的源代码。

首先给出工厂角色的源代码，可以看出，抽象工厂角色规定出两个工厂方法，分别提供两个不同等级结构的产品对象。

代码清单 1：抽象产品角色的源代码

```
package com.javapatterns.abstractfactory;
public interface Creator{
/**
* 产品等级结构 A 的工厂方法
*/
public ProductA factoryA();
/**
* 产品等级结构 B 的工厂方法
*/
public ProductB factoryB();
}
```

具体工厂类 ConcreteCreator1 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator1 implements Creator
{
/**
* 产品等级结构 A 的工厂方法
*/
public ProductA factoryA(){
   return new ProductA1();
}
/**
* 产品等级结构 B 的工厂方法
*/
public ProductB factoryB(){
  return new ProductB1();
  }
}
```

代码清单 3：具体工厂类 ConcreteCreator2 的源代码

```
package com.javapatterns.abstractfactory;
public class ConcreteCreator2 implements Creator{
/**
* 产品等级结构 A 的工厂方法
*/
public ProductA factoryA(){
   return new ProductA1();
}
/**
* 产品等级结构 B 的工厂方法
*/
public ProductB factoryB(){
   return new ProductB1();
 }
}
```

代码清单 4：具体产品类 ProductA 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductA
{
}
```

代码清单 5：具体产品类 ProductA1 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA1 implements ProductA{
   public ProductA1(){}
}
```

代码清单 6：具体产品类 ProductA2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductA2 implements ProductA{
  public ProductA2(){  }
}
```

代码清单 7：抽象产品角色 ProductB 的源代码

```
package com.javapatterns.abstractfactory;
public interface ProductB{
}
```

代码清单 8：具体产品类 ProductB1 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductB1 implements ProductB{
   /*** 构造子*/
   public ProductB1(){    }
}
```

代码清单 9：具体产品类 ProductB2 的源代码

```
package com.javapatterns.abstractfactory;
public class ProductB2 implements ProductB{
/*** 构造子*/
  public ProductB2(){}
}
```

工厂模式的相关应用：

★spring 中的大量使用工厂模式。

★在 SAX2 库中，XMLReaderFactory 类使用了简单工厂模式，用来创建产品类 XMLReader 的实例。

★用工具类 java.text.DateFormat 或其子类来格式化一个本地日期或者时间

★Java 集合是 Java 1.2 版提出来的。多个对象聚在一起形成的总体称之为集合（Aggregate），集合对象是能够包容一组对象的容器对象。所有的 Java 集合都实现 java.util.Collection 接口，这个接口规定所有的 Java 聚集必须提供一个 iterator()方法，返还一个 Iterator 类型的对象一 个具体的 Java 聚集对象会通过这个 iterator()方法接口返还一个具体的 Iterator 类。可以看出，这个 iterator()方法就是一个工厂方法。

★在微软公司所提倡的COM（Component Object Model）技术架构中， 工厂方法模式起着关键的作用。

★在 EJB 技术架构中，工厂方法模式也起着关键的作用

★JMS（Java Messaging Service） 技术架构中的工厂方法模式

2，单例（Singleton）模式，单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例。这个类称为单例类。单例模式有以下三个的特点：

★ 单例类只可有一个实例。

★单例类必须自己创建自己这惟一的实例。

★ 单例类必须给所有其他对象提供这一实例。

注意：虽然单例模式中的单例类被限定只能有一个实例，但是单例模式和单例类可以很容易被推广到任意且有限多个实例的情况，这时候称它为多例模式（Multiton Pattern）和多例类（Multiton Class），

一个例子：Windows 回收站

在整个视窗系统中，回收站只能有一个实例，整个系统都使用这个惟一的实例，而且回收站自行提供自己的实例。因此，回收站是单例模式的应用。

单列模式的实现主要有两种：饿汉式单例类和懒汉式单例类

1）饿汉式单例类

代码清单 1：饿汉式单例类

```
public class EagerSingleton{
   private static final EagerSingleton m_instance = new EagerSingleton();
/*** 私有的默认构造子*/
private EagerSingleton() { }
/*** 静态工厂方法*/
public static EagerSingleton getInstance(){
  return m_instance;
}
}
```

读者可以看出，在这个类被加载时，静态变量 m_instance 会被初始化，此时类的私有构造子会被调用。这时候，单例类的惟一实例就被创建出来了。

Java 语言中单例类的一个最重要的特点是类的构造子是私有的，从而避免外界利用构造子直接创建出任意多的实例。值得指出的是，由于构造子是私有的，因此，此类不能被继承。

2）懒汉式单例类。与饿汉式单例类相同之处是，类的构造子是私有的。与饿汉式单例类不同的是，懒汉式单例类在第一次被引用时将自己实例化。如果加载器是静态的，那么在懒汉式单例类被加载时不会将自己实例化

代码清单 2：懒汉式单例类

```
package com.javapatterns.singleton.demos;
public class LazySingleton{
  private static LazySingleton  m_instance = null;   加载时并不创建对象，而是在需要的时候在创建
/**
* 私有的默认构造子，保证外界无法直接实例化
*/
private LazySingleton() { }
/*** 静态工厂方法，返还此类的惟一实例*/
  synchronized public static LazySingleton
   getInstance(){
     if (m_instance == null)
       { m_instance = new LazySingleton();
   }
   return m_instance;
}
}
```

读者可能会注意到，在上面给出懒汉式单例类实现里对静态工厂方法使用了同步化，以处理多线程环境。有些设计师在这里建议使用所谓的“双重检查成例”。必须指出的是，“双重检查成例”不可以在 Java 语言中使用。

3）登记式单例类

登记式单例类是 GoF 为了克服饿汉式单例类及懒汉式单例类均不可继承的缺点而设计的

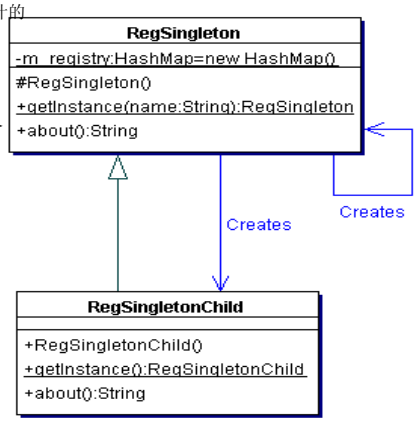
这样做的缺点

由于子类必须允许父类以构造子调用产生实例，因此，它的构造子必须是公开的。

这样一来，就等于允许了以这样方式产生实例而不在父类的登记中。

这是登记式单例类的一个缺点。

GoF 曾指出，由于父类的实例必须存在才可能有子类的实例，这在有些情况下是一个浪费。 这是登记式单例类的另一个缺点。



3,多例模式

所谓的多例模式（Multiton Pattern），实际上就是单例模式的自然推广。作为对象的创建模式，多例模式或多例类有以下的特点：

（1）多例类可有多个实例。

（2）多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

1)有上限多例模式。一个实例数目有上限的多例类已经把实例的上限当做逻辑的一部分，并建造到了多例类的内部，这种多例模式叫做有上限多例模式。

比如每一麻将牌局都需要两个色子，因此色子就应当是双态类。这里就以这个系统为例，说明多例模式的结构。色子的类图如下所示。

下面就是多例类 Die（色子）的源代码。

代码清单 3：多例类的源代码

```
package com.javapatterns.multilingual.dice;
import java.util.Random;
import java.util.Date;
public class Die{
   private static Die die1 = new Die();
   private static Die die2 = new Die();
/**
* 私有的构造子保证外界无法
* 直接将此类实例化
*/
private Die(){   }
/*** 工厂方法*/
public static Die getInstance(int whichOne) {
    if (whichOne == 1) {
         return die1;
    }
   else{
      return die2;
     }
}/*** 掷色子，返还一个在 1~6 之间的* 随机数*/
public synchronized int dice(){
   Date d = new Date();
   Random r = new Random( d.getTime() );
   int value = r.nextInt();
   value = Math.abs(value);
   value = value % 6;
   value += 1;
   return value;
}
```

}代码清单 4：客户端的源代码

```
package com.javapatterns.multilingual.dice;
public class Client{
   private static Die die1, die2;
   public static void main(String[] args) {
      die1 = Die.getInstance(1);
      die2 = Die.getInstance(2);
```

```
    die1.dice();
    die2.dice();
}
}
```

2)无上限多例模式，由于没有上限的多例类对实例的数目是没有限制的，因此，虽然这种多例模式是单例模式的推广，但是这种多例类并不一定能够回到单例类。

应用：序列键生成器与单例及多例模式

4,原始模式，通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的办法创建出更多同类型的对象。这就是原始模型模式的用意。Java 语言的构件模型直接支持原始模型模式。所有的 JavaBean 都继承自 java.lang.Object，而 Object 类提供一个 clone()方法，可以将一个 JavaBean 对象复制一份；但是这个 JavaBean 必须实现一个标示接口 Cloneable 表明这个 JavaBean 支持复制。如果一个对象没有实现这个接口而调用 clone()方法，Java 编译器会抛出 CloneNotSupportedException 异常。

5 建造模式，建造模式的定义为：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。可以将建造模式的精髓概括为：将构造复杂对象的过程和对象的部件解耦。这是对降低耦合、提高可复用性精神的一种贯彻。其实这种精神贯彻在 GOF 几乎所有的设计模式中。

2）各类的说明如下：

i）抽象建造者（Builder）角色：给出一个抽象接口，以规范产品对象的各个组成成分的构造；

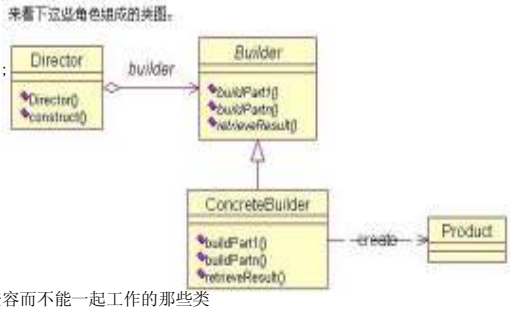
ii）具体建造者（ConcretBuilder）角色：它在应用程序的调用下创建产品的实例。完成任务为；

a） 实现抽象建造者 Builder 接口，给出一步步完成创建产品实例的操作；

b） 在创建完成后，提供产品的实例。

iii）导演者（Director 角色）：调用具体建造者角色以创建产品对象；

iv）产品（Product）角色：建造中的复杂对象。一般情况下，一个系统不止一个产品类。



Java 中的应用：javamail 中使用建造者模式

6，适配器模式

目的：将一个类的接口转换成客户希望的另外一个接口。A d a p t e r 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。适配器的两种模式：类的适配器模式和对象的适配器模式

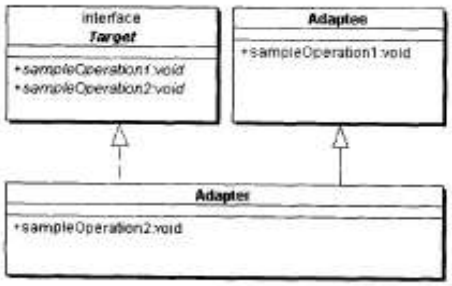
1）类的适配器模式，类的适配器模式把适配的类的 API 转换成目标类的 API。

目标(Target)角色：这就是所期待得到的接口。

源(Adaptee)角色：现有需要适配的接口。

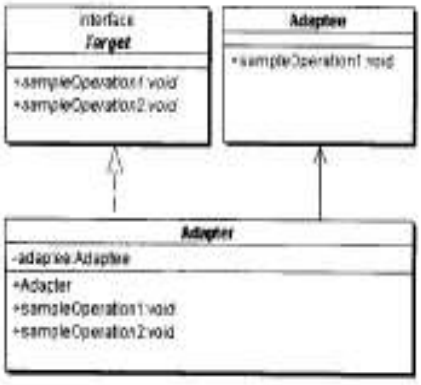
适配器(Adapter)角色：适配器类是本模式的核心。适配器把源接口转换成目标接口。

显然这一角色不可以是接口，而必须是具体类。




```
/*** 定义 Client 使用的与特定领域相关的接口  */
  public interface Target {
      void sampleOperation1();
      void sampleOperation2();
  }
  /*** 定义一个已经存在的接口，这个接口需要适配*/
  public class Adaptee {
      public void sampleOperation1() {
          // ......
      }
  }
  /*** 对 Adaptee 与 Target 接口进行适配*/
  public class Adapter extends Adaptee implements Target {
      public void sampleOperation2() {
          // ......
      }
  }
```

2）对象的适配器，与类的适配器模式一样，对象适配器模式把适配的类的 API 转换成为目标类的 API，与类的适配器模式不同的是，对象的适配器模式不是使用继承关系连接到 Adaptee 类，而是使用委派关系连接到 Adaptee 类。示意代码如下：




```
/** * 定义 Client 使用的与特定领域相关的接口*/
public interface Target {
    void sampleOperation1();
    void sampleOperation2();
}
/** * 定义一个已经存在的接口，这个接口需要适配*/
public class Adaptee {
    public void sampleOperation1() {
        // ......
```

```
    }
}
/** * 对 Adaptee 与 Target 接口进行适配 */
public class Adapter implements Target {
    private Adaptee adaptee;
    public Adapter(Adaptee adaptee) {
        super();
        this.adaptee = adaptee;
    }
    public void sampleOperation1() {
        adaptee.sampleOperation1();
    }
    public void sampleOperation2() {
        // ......
    }
}
```

类适配器模式和对象适配器模式的异同：Target 接口和 Adaptee 类都相同，不同的是类适配器的 Adapter 继承 Adaptee 实现 Target，对象适配器的 Adapter 实现 Target 聚集 Adaptee。

应用：1JDBC 驱动软件与适配器模式.JDBC 给出一个客户端通用的界面，每个数据库引擎的 JDBC 驱动软件都是一个介于 JDBC 接口和数据库引擎接口之间的适配器软件

2 JDBC/ODBC 如果没有合适的 jdbc 软件驱动,用户也可以用 ODBC 驱动软件把 JDBC 通过一个 JDBC/ODBC 桥梁软件与 ODBC 连接起来从而达到连接数据库的目的，JDBC 的库不可能和 ODBC 的库有相同的接口，因此使用适配器模式将 ODBC 的 api 接口改为 JDBC 的接口，因此 JDBC/ODBC 桥梁是适配器模式的应用。

3，XMLProperties 使用了适配器模式

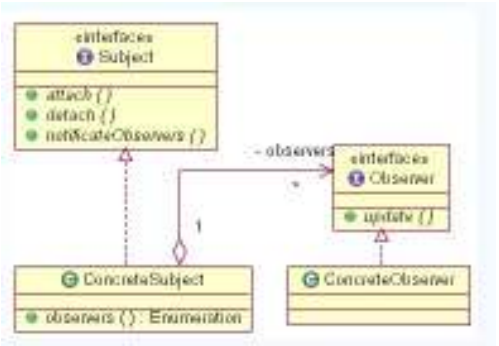
6， 观察者模式，观察者模式又叫作发布-订阅模式(publish-subscribe)，模型-视图(model-view)模式，源-监听者(source-listener)模式，或者从属者(dependents)模观察者模式定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生改变时，会通知所有的观察者对象，使他们能够自动更新自己。

这种观察模式在理解上相当于报社与订阅者之间的关系。出版者相当于“主题 Subuject”，订阅者相当于“Observer”。主题对象管理某些数据，当主题内的数据改变时就会通知观察者，观察者（已经订阅的/ 注册了的）就会收到更新。

发布者叫事件源，订阅者叫事件监听器，在 java 里时间用类代表

java.util 包内包含最基本的 Observer 接口与 Observable 类，如果使用该内置的支持，就只需要写一个类去扩展（继承）Observable，并告诉它何时该通知观察者， 一切就完成了，剩下的 API 会帮你做。你可以根据需要编写具体的观察者的类，在这个类中定义 update()方法，去实现 Observer 接口。

```
public class Observable {
   private boolean changed = false;
   private Vector obs;
  //创建被观察者时就创建一个它持有的观察者列表，
  //注意，这个列表是需要同步的。
   public Observable() {
      obs = new Vector();
   }
/**  * 添加观察者到观察者列表中去*/
public synchronized void addObserver(Observer o) {
    if (o == null)
        throw new NullPointerException();
if (!obs.contains(o)) {
    obs.addElement(o);
}
}
```




```
/** * 删除一个观察者 */
public synchronized void deleteObserver(Observer o) {
    obs.removeElement(o);
}

/**  * 通知操作，即被观察者发生变化，通知对应的观察者进行事先设定的操作，不传参数的通知方法 */
public void notifyObservers() {
     notifyObservers(null);
}
/**  * 与上面的那个通知方法不同的是，这个方法接受一个参数，这个参数一直传到观察者里，以供观察者使用*/
public void notifyObservers(Object arg) {
```

```
        Object[] arrLocal;
    synchronized (this) {
        if (!changed)
                return;
            arrLocal = obs.toArray();
            clearChanged();
        }

        for (int i = arrLocal.length-1; i>=0; i--)
            ((Observer)arrLocal[i]).update(this, arg);
    }
}


public interface Observer {
    void update(Observable o, Object arg);
}
}
```

这是一个接口，接口中就只有一个方法，update，方法中有两个参数，Observable 和一个 object，第一个参数就是被观察的对象，而第二个参数就得看业务需求了，需要什么就传进去什么。我们自己的观察者类必须实现这个方法，这样在被观察者调用 notifyObservers 操作时被观察者所持有的所有观察者都会执行 update 操作了（当然如果你 override 这个方法，你甚至可以指定何种情况下只执行某种 observer 了，是不是比较像责任链模式了）。

首先让我们来实现一个发送邮件的观察者：

Java 代码

```
public class MailObserver implements Observer{
    /**   这个类取名为 MailObserver，顾名思义，她是一个用来发送邮件的观察者 */
    public void update(Observable o, Object arg) {
        System.out.println("发送邮件的观察者已经被执行");
    }
}
```

接下来让我们再来实现被观察者，示例如下：

Java 代码

```
public class Subject extends Observable{
    /**   * 业务方法，一旦执行某个操作，则通知观察者  */
    public void doBusiness(){
        if (true) {
            super.setChanged();
        }
        notifyObservers("现在还没有的参数");
    }


    public static void main(String [] args) {
        //创建一个被观察者
        Subject subject = new Subject();

        //创建两个观察者
        Observer mailObserver = new MailObserver();
        Observer jmsObserver = new JMSObserver();

        //把两个观察者加到被观察者列表中
        subject.addObserver(mailObserver);
        subject.addObserver(jmsObserver);

        //执行业务操作
        subject.doBusiness();
    }
}
```

观察者模式与 AWT 中的事件处理。Java1.0 的事件处理机制是建立在责任链模式的基础之上的，但是这种不能满足打应用系统的需求，在 java1.1 后改为建立在观察者模式之上的以事件的委派为特征的委派事件模型（Delegation Event Model DEM），这种 DEM 的机制不仅在 AWT 中使用还在 Swing 中使用

在 AWT 中的观察者模式 DEM 的结构（三要素）

1）事件源对象 ，一个类要成为事件源不需要实现任何接口和继承任何类，但是一个事件源要保持一个事件监听器的列表。调用 addXXXListener（）方法增加一个监听器，调用 removeXXXListener（）方法删除一个监听器。不同的事件就要不同的监听器

2）事件对象，每一个事件都有一个事件对象与他对应，所有的 AWT 中的事件对象都是从 java.util.EventObject 继承而来的
常用的有 ActionEvent MourseEvent
3）事件监听器对象，当事件发生时被调用的对象，一个对象要想成为一个事件监听对象，必须要实现事件监听接口，AWT 中的事件接口都是 java.util.EventListener

实例

```
import java.awt.Frame;
import java.awt.event.MouseListener;
public class ConcreteSubject extends Frame{
    private static MouseListener m;
    public ConcreteSubject(){}
  public static void main(String[] argv)
    {
        ConcreteSubject s = new ConcreteSubject();
         m = new ConcreteListener();
        s.setBounds(100, 100, 100 , 100);
        s.addMouseListener(m);
         s.show();
}}
```

```
public class ConcreteListener
            implements MouseListener{
  ConcreteListener(){    }
  public void mouseClicked(MouseEvent e)  {
        System.out.println(e.getWhen());
  }
    public void mousePressed(MouseEvent e){}
    public void mouseReleased(MouseEvent e){}
    public void mouseEntered(MouseEvent e){}
    public void mouseExited(MouseEvent e){}
}
```

7， 代理模式

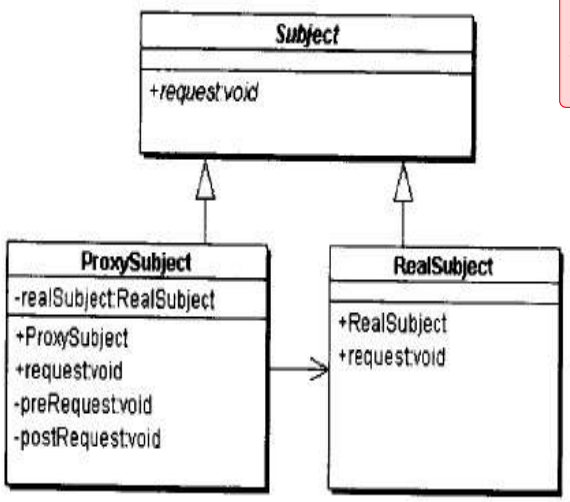
1）代理模式的作用是：为其他对象提供一种代理以控制对这个对象的访问。在某些情况下，一个客户不想或者不能直接引用另一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。利用中间对象起到加强对象功能的模式还有，装饰模式，适配器模式

代理模式有以下几种：远程代理（EJB Spring 的远程代理） 虚拟代理 保护代理 同步代理 防火墙代理 智能代理

抽象角色：声明真实对象和代理对象的共同接口；没有共同的接口那就是委托

2）代理模式的结构

```
  abstract public class Subject{
      abstract public void request();
}
package com.javapatterns.proxy;
public class RealSubject extends Subject {
      public RealSubject(){ }
      public void request(){
            System.out.println("From real subject.");
      }
}
public class ProxySubject extends Subject {
    private RealSubject realSubject;
    public ProxySubject(){}
    public void request(){
        preRequest();
      if( realSubject == null ) {
         realSubject = new RealSubject();
      }
        realSubject.request();
        postRequest();
      }
    private void preRequest(){
        //something you want to do before requesting
    }
    private void postRequest(){
        //something you want to do after requesting
    }
}
public class Client{
      private static Subject subject;
    static public void main(String[] args){
            subject = new ProxySubject();
            subject.request();
            subject.request();
      }
}
```

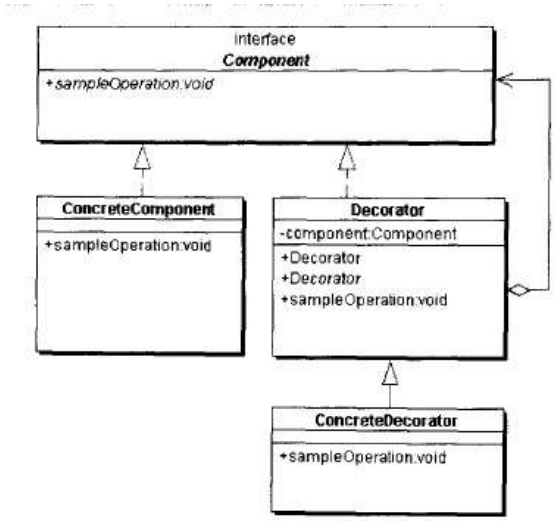


批注 [U101]: 应用：在 ejb 和 spring 的大量使用，java2.0 对代理的支持：自 jdk1.3 以来 java 通过 java.lang.reflect 提供三个直接的代理模式：Proxy ，InvocationHandler ，Method

8， 装饰模式又叫包装模式，装饰模式是对对象功能增强时，平时使用继承的一种替代方案

模式结构和代码

```
public interface Component{
    void sampleOperation();
}

public class Decorator implements Component{
     public Decorator(Component component){
       //        super();
        this.component = component;
    }
    public Decorator() {}
    public void sampleOperation(){
        component.sampleOperation();
    }
    private Component component;
}

public class ConcreteComponent implements Component{
    public void sampleOperation(){
        // Write your code here
    }
}

public class ConcreteDecorator extends Decorator{
    public void sampleOperation() {
        super.sampleOperation();
    }
}
```



装饰模式的电信创建模式

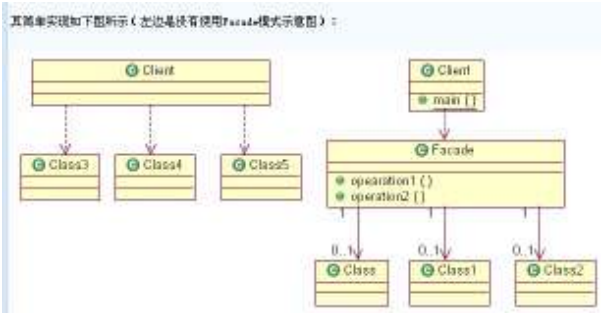
```
   new Decorator1(
                new Decorator2(
                        new ConcreteComponent()));  所以装饰模式常常也叫包裹模式
```

注意： 1,InputStreamReader 是把 InputStream 包装起来，把 InputStreamReader 的 API 转换成 Reader 的 api，所以它是适配器模式，而不是桥梁模式。只是相当一个桥梁。

2,在 java.io 中的 BufferedReader 是一个装饰类，也可以看成半个适配器模式(因为它提供了一个 readLine（）新方法)一个装饰类实现的新方法越多，就离装饰类越远

3，在 java.io 中充满了装饰模式和适配器模式。

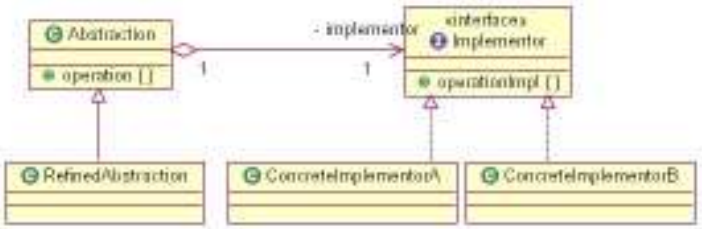
9，门面模式：外部与一个子系统的通信必须通过一个统一的门面对象进行，这就是门面模式。一般而言，Facade 模式是为了降低子系统之间，客户端与实现化层之间的依赖性。当在构建一个层次化的系统时，也可以同过使用 Facade 模式定义系统中每一层的入口，从而简化层与层之间的依赖关系。



应用：struts 的 action 就是相当一个门面，参考 ejb java ee 的结构

10，桥梁模式。桥梁模式的用意是把抽象化与实现化脱耦。脱耦是说把抽象和实现之间的耦合解脱，或者说把强关联变成弱关联。桥梁模式的脱耦指的就是把抽象和实现之间的继承/实现关系变成组合/聚合关系。从而可以使两者可以相对独立的变化。这就是桥梁模式的本意。

结构图如下



桥梁模式的关键是找出抽象化角色和具体化角色。典型应用是 JDBC 驱动器的应用。

11，不变模式

不变模式可增强对象的强壮性(robustness)。不变模式允许多个对象共享某一对象，降低对该对象进行并发访问的同步化开销。如果需要修改一个不变对象的状态，就需要建立一个新的同类型对象，并在创建时将这个新的状态存储在新对象里。

不变模式之设计一个类。一个类的内部状态创建后，在整个生命期内都不会发生变化时，这个类被称为不变类。这种使用不变类的做法叫作不便模式。

弱不变模式：一个类的实例状态不可改变，但是子类的实例具有可能会变化的状态。对象没有任何方法可以修改对象的状态 所有的属性都应该是私有的

强不变模式：一个类的实例不可改变，子类的实例也不可改变，满足弱不变模式还要满足

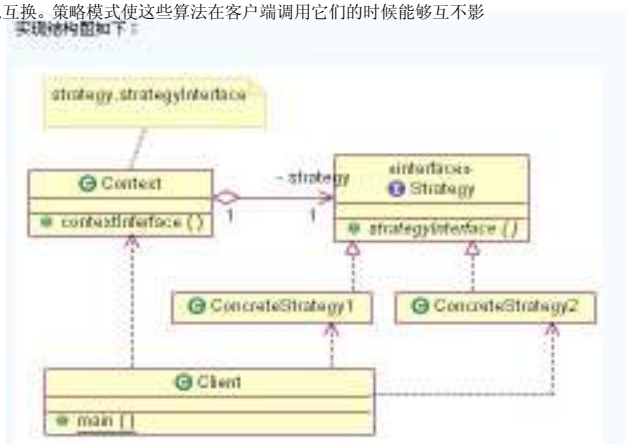
1 类所有的方法都应当是 final，这样这个类的子类不能够换掉此类的方法

2 这个类本身就是 final 的，不存在子类

在 java 中的应用：最著名的就是 String 类，它是一个强不变类，

12，策略模式。策略模式（Strategy Pattern）中体现了两个非常基本的面向对象设计的基本原则：封装变化的概念；编程中使用接口，而不是对接口实现。策略模式的定义如下：

定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。策略模式使这些算法在客户端调用它们的时候能够互不影响地变化。



策略模式的好处在于你可以动态的改变对象的行为。

```
public class Context{
    public void contextInterface(){
        strategy.strategyInterface();
    }
    private Strategy strategy;
}

abstract public class Strategy{
    public abstract void strategyInterface();
}

public class ConcreteStrategy1 extends Strategy{
    public void strategyInterface(){
     //write you algorithm code here
     //写自己独特的算法
    }
}
```

从中可以看出，策略模式并不负责安排哪种情况应用哪种算法，
需要客户端来指定。策略模式只能同时应用一种策略。

在 Java 中 BorderLayout，排序算法等地方都应用了 Strategy 模式。

可以应用在不同商品的打折算法不一样上

13，模板模式（Template Method）：是类的行为模式，准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类来实现剩下的逻辑方法，不同的子类可以有不同的实现形式，从而对剩余的逻辑有不同的实现，这就是模板模式的用意。
代码实例：

```
abstract public class AbstractClass{
    public void TemplateMethod() {   //先实现一个模板方法，这是子类都要用的方法
        doOperation1();
        doOperation2();
        doOperation3();
    }
    protected abstract void doOperation1();//定义抽象方法，需要不同的子类自己完成
    protected abstract void doOperation2();
    private final void doOperation3(){
        //do something
    }
}
public class ConcreteClass extends AbstractClass {  //继承抽象类，实现抽象方法
    public void doOperation1() {
        //write your code here
        System.out.println("doOperation1();");
    }
    public void doOperation2() {
        //The following should not happen:
        //doOperation3();
        //write your code here
        System.out.println("doOperation2();");
    }
}
```

在 java 中这是常用的技巧，可以解决代码复用的问题
在 java 中的 web 系统中 HttpServlet 就是建立在模板模式的基础之上的。HttpServlet 提供了一个 service（）方法，这个方法调用 7 个 do 方法中的一个或几个，完成客户端的处理，那么 service（）就是一个模板，其他的 do 方法就是就是基本方法

其他模式还有：责任链模式，命令模式，状态模式，解释器模式，调停者模式，这就不多讲了